

34th Annual Symposium on Combinatorial Pattern Matching

CPM 2023, June 26–28, 2023, Marne-la-Vallée, France

Edited by


Laurent Bulteau
Zsuzsanna Lipták



Editors

Laurent Bulteau 

LIGM, CNRS, Université Gustave Eiffel, Marne-la-vallée, France
laurent.bulteau@univ-eiffel.fr

Zsuzsanna Lipták 

University of Verona, Italy
zsuzsanna.liptak@univr.it

ACM Classification 2012

Theory of computation → Design and analysis of algorithms; Theory of computation → Pattern matching;
Theory of computation → Data compression; Mathematics of computing → Discrete mathematics;
Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatoric
problems; Applied computing → Computational biology

ISBN 978-3-95977-276-1

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/978-3-95977-276-1>.

Publication date

June, 2023

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>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work
under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.CPM.2023.0

ISBN 978-3-95977-276-1

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB and Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

To all men and women whose passion are algorithms, data structures, and
combinatorics on strings.

■ Contents

Preface	
<i>Laurent Bulteau and Zsuzsanna Lipták</i>	0:ix
Program Committee	
.....	0:xi
External Reviewers	
.....	0:xiii
List of Authors	
.....	0:xv–0:xvi

Regular Papers

Trie-Compressed Adaptive Set Intersection	
<i>Diego Arroyuelo and Juan Pablo Castillo</i>	1:1–1:19
Approximation Algorithms for the Longest Run Subsequence Problem	
<i>Yuichi Asahiro, Hiroshi Eto, Mingyang Gong, Jesper Jansson, Guohui Lin,</i> <i>Eiji Miyano, Hirotaka Ono, and Shunichi Tanaka</i>	2:1–2:12
Optimal LZ-End Parsing Is Hard	
<i>Hideo Bannai, Mitsuru Funakoshi, Kazuhiro Kurita, Yuto Nakashima,</i> <i>Kazuhisa Seto, and Takeaki Uno</i>	3:1–3:11
Sliding Window String Indexing in Streams	
<i>Philip Bille, Johannes Fischer, Inge Li Gørtz, Max Rishøj Pedersen, and</i> <i>Tord Joakim Stordalen</i>	4:1–4:18
Faster Algorithms for Computing the Hairpin Completion Distance and Minimum Ancestor	
<i>Itai Boneh, Dvir Fried, Adrian Miclăuş, and Alexandru Popa</i>	5:1–5:18
On Distances Between Words with Parameters	
<i>Pierre Bourhis, Aaron Boussidan, and Philippe Gambette</i>	6:1–6:23
Parameterized Algorithms for String Matching to DAGs: Funnels and Beyond	
<i>Manuel Cáceres</i>	7:1–7:19
Optimal Near-Linear Space Heaviest Induced Ancestors	
<i>Panagiotis Charalampopoulos, Bartłomiej Dudek, Paweł Gawrychowski, and</i> <i>Karol Pokorski</i>	8:1–8:18
From Bit-Parallelism to Quantum String Matching for Labelled Graphs	
<i>Massimo Equi, Arianne Meijer-van de Griend, and Veli Mäkinen</i>	9:1–9:20
On the Impact of Morphisms on BWT-Runs	
<i>Gabriele Fici, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina</i>	10:1–10:18

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications <i>Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba</i>	11:1–11:20
Compressed Indexing for Consecutive Occurrences <i>Paweł Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner</i>	12:1–12:22
Order-Preserving Squares in Strings <i>Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau</i>	13:1–13:19
MUL-Tree Pruning for Consistency and Compatibility <i>Christopher Hampson, Daniel J. Harvey, Costas S. Iliopoulos, Jesper Jansson, Zara Lim, and Wing-Kin Sung</i>	14:1–14:18
Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String <i>Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba</i>	15:1–15:15
Faster Prefix-Sorting Algorithms for Deterministic Finite Automata <i>Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza</i>	16:1–16:16
Encoding Hard String Problems with Answer Set Programming <i>Dominik Köppl</i>	17:1–17:21
On the Complexity of Parameterized Local Search for the Maximum Parsimony Problem <i>Christian Komusiewicz, Simone Linz, Nils Morawietz, and Jannik Schestag</i>	18:1–18:18
String Factorization via Prefix Free Families <i>Matan Kraus, Moshe Lewenstein, Alexandru Popa, Ely Porat, and Yonathan Sadia</i>	19:1–19:10
Improving the Sensitivity of MinHash Through Hash-Value Analysis <i>Gregory Kucherov and Steven Skiena</i>	20:1–20:12
Suffix-Prefix Queries on a Dictionary <i>Grigorios Loukides, Solon P. Pissis, Sharma V. Thankachan, and Wiktor Zuba</i> ..	21:1–21:20
Merging Sorted Lists of Similar Strings <i>Gene Myers</i>	22:1–22:15
PalFM-Index: FM-Index for Palindrome Pattern Matching <i>Shinya Nagashita and Tomohiro I</i>	23:1–23:15
Computing MEMs on Repetitive Text Collections <i>Gonzalo Navarro</i>	24:1–24:17
L-Systems for Measuring Repetitiveness <i>Gonzalo Navarro and Cristian Urbina</i>	25:1–25:17
MONI Can Find k -MEMs <i>Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie</i> ..	26:1–26:14

■ 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 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023) held on June 26–28, 2023 in Marne-la-Vallée, France. The conference program includes 26 contributed papers and three invited talks, by

- Olgica Milenkovic (University of Illinois Urbana-Champaign, USA),
- Tatiana Starikovskaya (École Normale Supérieure, France), and
- Virginia Vassilevska Williams (Massachusetts Institute of Technology, USA).

For the fifth time, CPM includes the “Highlights of CPM” special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this fifth edition we selected as highlight papers “Separating words and trace reconstruction”, by Zachary Chase, presented at STOC 2021, and “Approximating Dynamic Time Warping Distance Between Run-Length Encoded Strings”, by Zoe Xi and William Kuszmaul, presented at ESA 2022. The conference was preceded by a one-day student summer school taught by Karel Břinda (Inria/IRISA Rennes) and Panagiotis Charalampopoulos (Birkbeck, University of London), and organized at the École Normale Supérieure in Paris by Gabriel Bathie, Paweł Gawrychowski, Garance Gourdel and Tatiana Starikovskaya.

The contributed papers were selected out of 44 submissions, corresponding to an acceptance ratio of 59%. 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, Copenhagen (on-line), Wrocław and Prague. From 1992 to the 2015 meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, the CPM proceedings have appeared in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM 2016), 78 (CPM 2017), 105 (CPM 2018), 128 (CPM 2019), 161 (CPM 2020), 191 (CPM 2021) and 223 (CPM 2022). 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.

Laurent Bulteau and Zsuzsanna Lipták



■ Program Committee

Golnaz Badkobeh
Goldsmiths, University of London, UK

Hideo Bannai
Tokyo Medical and Dental University, Japan

Marie-Pierre Béal
University Gustave Eiffel, France

Giulia Bernardini
University of Trieste, Italy

Paola Bonizzoni
University of Milano-Bicocca, Italy

Laurent Bulteau (co-chair)
CNRS – University Gustave Eiffel, France

Johannes Fischer
TU Dortmund, Germany

Travis Gagie
Dalhousie University, Canada

Pawel Gawrychowski
University of Wrocław, Poland

Jan Holub
Czech Technical University in Prague, Czech
Republic

Wing-Kai Hon
National Tsing Hua University, Taiwan

Dominik Kempa
Stony Brook University, USA

Tomasz Kociumaka
Max Planck Institute for Informatics,
Germany

Avivit Levy
Shenkar College of Engineering, Design and
Art, Israel

Moshe Lewenstein
Bar Ilan University, Israel

Zsuzsanna Lipták (co-chair)
University of Verona, Italy

Yuto Nakashima
Kyushu University, Japan

Gonzalo Navarro
University of Chile, Chile

Enno Ohlebusch
University of Ulm, Germany

Svetlana Puzynina
Saint Petersburg State University, Russia

Jakub Radoszewski
University of Warsaw, Poland

Rajeev Raman
University of Leicester, UK

Giovanna Rosone
University of Pisa, Italy

Leena Salmela
University of Helsinki, Finland

Joe Sawada
University of Guelph, Canada

Marinella Sciortino
University of Palermo, Italy

Florian Sikora
Université Paris-Dauphine, PSL University,
France

Teresa Anna Steiner
Technical University of Denmark, Denmark

Sharma V. Thankachan
North Carolina State University, USA

Oren Weimann
University of Haifa, Israel

Petra Wolf
University of Bergen, Norway

Wiktór Zuba
CWI, Netherlands

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ External Reviewers

Emmanuel Arrighi

Nathalie Aubrun

Guillaume Blin

Manuel Cáceres

Giuseppa Castiglione

Julien Clément

Sami Davies

Joel Day

Gianluca Della Vedova

Diego Diaz

Riccardo Dondi

Jonas Ellert

Estéban Gabory

Daniel Gabric

Martin Gebser

Samah Ghazawi

Daniel Gibney

Daniel Gibney

Shay Golan

Garance Gourdel

Roberto Grossi

Veronica Guerrini

Wojciech Janczewski

Mark Jones

Saira Karim

Steven Kelk

Dominik Köppl

Dmitry Kosolobov

Thierry Lecroq

Florin Manea

Nils Morawietz

Nicola Prezza

Narad Rampersad

Raffaella Rizzi

Šimon Schierreich

Markus L. Schmid

Tatiana Starikovskaya

Tord Stordalen

Michelle Sweering

Alexander Tiskin

Jacobo Toran

Rossano Venturini

Hilde Verbeek

Tomasz Walen

Dennis Wong

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ List of Authors

Diego Arroyuelo
Yuichi Asahiro
Hideo Bannai
Philip Bille
Itai Boneh
Pierre Bourhis
Aaron Boussidan
Manuel Cáceres
Juan Pablo Castillo
Panagiotis Charalampopoulos
Bartłomiej Dudek
Massimo Equi
Hiroshi Eto
Gabriele Fici
Johannes Fischer
Dvir Fried
Mitsuru Funakoshi
Estéban Gabory
Travis Gagie
Philippe Gambette
Pawel Gawrychowski
Samah Ghazawi
Mingyang Gong
Inge Li Gørtz
Garance Gourdel
Christopher Hampson
Daniel J. Harvey
Tomohiro I
Costas S. Iliopoulos
Jesper Jansson
Sana Kashgouli
Sung-Hwan Kim
Tomasz Kociumaka
Christian Komusiewicz
Dominik Köppl
Matan Kraus
Gregory Kucherov
Kazuhiro Kurita
Gad M. Landau
Moshe Lewenstein
Zara Lim
Guohui Lin
Simone Linz
Grigorios Loukides
Veli Mäkinen
Arianne Meijer van de Griend
Adrian Miclăuş
Eiji Miyano
Nils Morawietz
Moses Njagi Mwaniki
Eugene Myers
Shinya Nagashita
Yuto Nakashima
Gonzalo Navarro
Francisco Olivares
Hirotaka Ono
Max Rishøj Pedersen
Nadia Pisanti
Solon Pissis
Karol Pokorski
Alexandru Popa
Ely Porat

0:xvi **Authors**

Nicola Prezza

Jakub Radoszewski

Giuseppe Romana

Wojciech Rytter

Yonathan Sadia

Jannik Schestag

Marinella Sciortino

Kazuhisa Seto

Ardavan Shahrabi Farahani

Steven Skiena

Tatiana Starikovskaya

Teresa Anna Steiner

Tord Joakim Stordalen

Wing-Kin Sung

Michelle Sweering

Shunichi Tanaka

Igor Tatarnikov

Sharma V. Thankachan

Takeaki Uno

Cristian Urbina


Tomasz Waleń

Wiktor Zuba

Trie-Compressed Adaptive Set Intersection

Diego Arroyuelo  

Departamento de Informática, Universidad Técnica Federico Santa María, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Juan Pablo Castillo 

Departamento de Informática, Universidad Técnica Federico Santa María, Santiago, Chile
Millennium Institute for Foundational Research on Data, Santiago, Chile

Abstract

We introduce space- and time-efficient algorithms and data structures for the offline set intersection problem. We show that a sorted integer set $S \subseteq [0..u)$ of n elements can be represented using compressed space while supporting k -way intersections in adaptive $O(k\delta \lg(u/\delta))$ time, δ being the alternation measure introduced by Barbay and Kenyon. Our experimental results suggest that our approaches are competitive in practice, outperforming the most efficient alternatives (Partitioned Elias-Fano indexes, Roaring Bitmaps, and Recursive Universe Partitioning (RUP)) in several scenarios, offering in general relevant space-time trade-offs.

2012 ACM Subject Classification Theory of computation \rightarrow Data compression; Theory of computation \rightarrow Design and analysis of algorithms; Theory of computation \rightarrow Data structures and algorithms for data management; Information systems \rightarrow Information retrieval query processing

Keywords and phrases Set intersection problem, Adaptive Algorithms, Compressed and compact data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.1

Supplementary Material *Software*: <https://github.com/jpcastillo/compressed-binary-tries> archived at `swh:1:dir:4ec1b85ad1d97fa10c648a3ad1ad2366c0cafb5c`

Funding This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile (both authors).

Acknowledgements We thank Gonzalo Navarro, Cristian Riveros, Adrián Gómez-Brandón, and Francesco Tosoni for enlightening comments, suggestions, and discussions about this work. We also thank the anonymous reviewers whose thorough reviews helped us to improve this paper.

1 Introduction

Sets are one of the most fundamental mathematical concepts related to the storage of data. Operations such as set intersections, unions, and differences are key for querying them. E.g., the use of logical AND and OR operators in web search engines translate into intersections and unions, respectively. Representing sets to support their basic operations efficiently has been a major concern since many decades ago [4]. In several applications, such as query processing in information retrieval (IR) [15] and database management systems (DBMS) [23], sets are known in advance to queries, hence data structures can be built to speed up query processing. With this motivation, in this paper we focus on the following problem.

THE OFFLINE SET INTERSECTION PROBLEM, OSIP

Input: A family $\mathcal{S} = \{S_1, \dots, S_N\}$ of N sorted integer sets over universe $[0..u)$, with $|S_i| = n_i$.

Task : To preprocess family \mathcal{S} to efficiently support query instances of the form $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, which ask to compute $\mathcal{I}(\mathcal{Q}) = \bigcap_{i \in \mathcal{Q}} S_i$.



© Diego Arroyuelo and Juan Pablo Castillo;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 1; pp. 1:1–1:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We assume $u = 2^k$ in this paper, for $k \geq 0$. Unless explicitly otherwise stated, we also assume $\lg x = \lceil \lg_2 x \rceil$ and $\lg 0 = 0$. Typical applications of this problem include the efficient support of join operations in DBMS [23, 51], query processing using inverted indexes in IR [15, 52], and computational biology [33], among others. Building a data structure to speed up intersections, however, increases the space usage. Today, data-intensive applications encourage not only time- but also space-efficient solutions [7]. Being able to process big datasets entirely in main memory is the main motivation. Compact, succinct, and compressed data structures are important to achieve this [41]. We study here compressed data structures to efficiently support the OSIP. We assume the word RAM model of computation with word size $w = \Theta(\lg u)$. Arithmetic, logic, and bitwise operations, as well as accesses to w -bit memory cells, take $O(1)$ time.

The literature on this problem is vast. For the online version of the problem, where sets to be intersected are given at query time – so there is no time to preprocess them – algorithms like the ones by Baeza-Yates [9], Demaine et al. [20], and Barbay and Kenyon [12] are among the most efficient and well-known approaches. In particular, the two latter algorithms are adaptive, meaning that they are able to perform faster on “easier” query instances. The algorithm by Barbay and Kenyon runs in optimal $O(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ time, where δ is the so-called alternation measure that quantifies the query difficulty [12]. The algorithm by Demaine et al. [20] has running time $O(k\delta \lg(n/\delta))$, for $n = \sum_{i \in \mathcal{Q}} n_i$, which is optimal when $\max_{i \in \mathcal{Q}} \{\lg n_i\} = O(\min_{i \in \mathcal{Q}} \{\lg n_i\})$ [11]. These algorithms require sets to be stored in plain form, e.g. using a sorted array or a B-tree [20], requiring $\Theta(mw)$ bits of space, for $m = \sum_{i=1}^N n_i$. This can be excessive when dealing with large databases.

For the OSIP, we have the extensive literature on inverted indexes [52, 55, 15, 46], whose main focus is on practical space-efficient set representations supporting intersections. Approaches like Optimized PForDelta [53], Roaring Bitmaps [36], SIMD-BP128 [35], and Recursive Universe Partitioning [45] shine in practical scenarios, yet without appealing theoretical guarantees of space usage and intersection computation time. Another relevant approach on these lines is Partitioned Elias-Fano (PEF) [43], able to exploit the distribution and clustering of set elements to improve space usage. Barbay and Kenyon’s algorithm can be implemented on PEF, taking $O(\delta \sum_{i \in \mathcal{Q}} \lg(u/n_i))$ time. Regarding space usage, there is no known bound (although it performs well in practice). On a more theoretical track, Bille et al. [14] introduce a data structure that uses $O(mw)$ bits of space and supports intersections in $O(n \lg^2(w)/w + k|\mathcal{I}(\mathcal{Q})|)$ time. Cohen and Porat [18] data structure also uses $O(mw)$ bits of space and allows one to compute the intersection between any two sets in \mathcal{S} in $O(\sqrt{N}|\mathcal{I}(\mathcal{Q})| + |\mathcal{I}(\mathcal{Q})|)$ time. Besides using linear space, this approach only works for pair-wise intersections (and is hard to efficiently extend to multiway intersections). Finally, Ding and Konig [21] introduce a data structure able to compute intersections in $O(n/\sqrt{w} + k|\mathcal{I}(\mathcal{Q})|)$ expected time, and uses linear $O(m)$ space. The space can be improved in practice to use about 1.88 times the space of an Elias γ/δ compressed inverted index [21], yet with no theoretical guarantees. Later, Gagie et al. [26] showed that wavelet trees [28] can support intersections in $O(k\delta \lg(u/\delta))$ time, using *uncompressed* $mw(1 + o(1))$ bits of space.

In this paper we show that $O(k\delta \lg(u/\delta))$ intersection time using compressed space is possible. In particular, (1) in Section 3 we revisit a classic (and neglected) algorithm by Trabb-Pardo [50] (former Knuth’s student) to prove that its running time is actually $O(k\delta \lg(u/\delta))$ – so it is likely the first adaptive intersection algorithm that ever existed; (2) in Section 4 we show that Trabb-Pardo’s algorithm can be implemented in compressed space, yielding an adaptive and compressed set intersection algorithm; (3) in Section 5 we show how to exploit the presence of runs of successive elements, typical in some applications [8],

to formally improve both space usage of input sets and the intersection computation time by introducing an intersection algorithm that runs in time $O(k\xi \lg(u/\xi))$, where $\xi \leq \delta$ is an adaptability measure we introduce; and (4) in Sections 6 and 7 we implement our proposals and show preliminary experimental results that indicate that our approaches are appealing not only in theory, but also in practice, outperforming the most competitive state-of-the-art approaches in some practical inverted-index datasets we use in our tests. Overall, we conclude that both theoretical guarantees and practicality can be achieved with a single approach, which is a step forward in bridging the gap between theory and practice in this important line of research.

2 Preliminaries and Related Work

2.1 Operations rank and select

The following operations on a sorted integer set S are of interest:

- $\text{rank}(S, x)$: for $x \in [0..u)$, yields $|\{y \in S, y \leq x\}|$.
- $\text{select}(S, j)$: for $1 \leq j \leq |S|$, yields $x \in S$ s.t. $\text{rank}(S, x) = j$.

A set S can be alternatively described using its *characteristic bit vector* (cbv, for short) $C_S[0..u)$, such that $C_S[x] = \mathbf{1}$ if $x \in S$, $C_S[x] = \mathbf{0}$ otherwise. On a cbv C_S we define:

- $C_S.\text{rank}_1(x)$: for $x \in [0..u)$, yields the number of $\mathbf{1}$ s in $C_S[0..x]$.
- $C_S.\text{select}_1(k)$: for $1 \leq j \leq |S|$, yields the smallest position $0 \leq x < u$ s.t. $C_S.\text{rank}_1(x) = j$.

Notice that $\text{rank}(S, x) \equiv C_S.\text{rank}_1(x)$ and $\text{select}(S, j) \equiv C_S.\text{select}_1(j)$.

2.2 Set Compression Measures

A *compression measure* quantifies the amount of bits needed to encode data using a particular compression model. For an integer universe $U = [0..u)$, let $C^{(n)} \subseteq 2^U$, $n \in U$, denote the class of all sets $S \subseteq U$ such that $|S| = n$. We assume $S = \{x_1, \dots, x_n\}$, for $0 \leq x_1 \leq \dots \leq x_n < u$. As $|C^{(n)}| = \binom{u}{n}$, in the worst case one needs at least $\mathcal{B}(n, u) = \lceil \lg \binom{u}{n} \rceil$ bits to encode a set $S \in C^{(n)}$. If $n \ll u$, $\mathcal{B}(n, u) = n \lg(u/n) + n \lg e - O(\lg u)$ bits (using Stirling for $n!$). Notice $\mathcal{B}(n, u)$ is a worst-case lower bound: some sets in $C^{(n)}$ can be encoded using less bits, as we shall see.

2.2.1 The $\text{gap}(S)$ Compression Measure

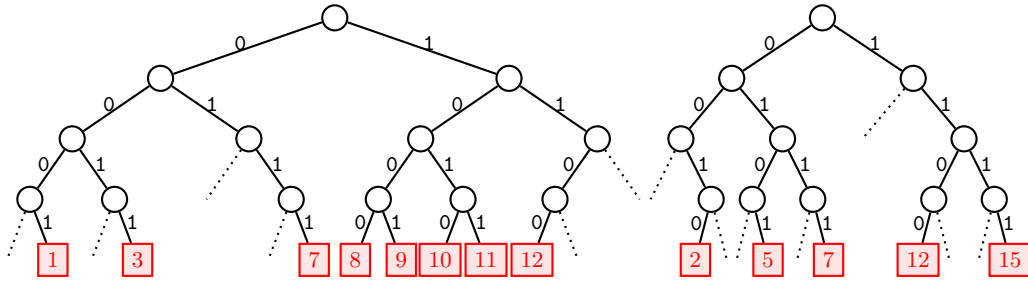
Let us denote $g_1 = x_1$ and, for $i = 2, \dots, n$, $g_i = x_i - x_{i-1} - 1$. Thus, in the gap model we have $C_S[0..u) = \mathbf{0}^{g_1} \mathbf{1} \mathbf{0}^{g_2} \mathbf{1} \dots \mathbf{0}^{g_n} \mathbf{1}$ (assuming wlog that C_S ends with $\mathbf{1}$). Then, we define $\text{gap}(S) = \sum_{i=1}^n (\lceil \lg g_i \rceil + 1)$, as the amount of bits required to represent S provided we encode the sequence of gaps $\mathcal{G} = \langle g_1, \dots, g_n \rangle$, using $\lceil \lg g_i \rceil + 1$ bits per gap. Although this measure is not achievable, it exploits the variation in the gaps between consecutive set elements: the closer the elements, the smallest this measure is. It holds that $\text{gap}(S) \leq n \lg \frac{u}{n}$, with equality only when $g_i = \frac{u}{n}$ (for $i = 1, \dots, n$). This is a measure traditionally used in applications like inverted-index compression in information retrieval [15] and databases [52].

2.2.2 The $\text{rle}(S)$ Compression Measure

When set elements tend to be clustered into runs of successive elements, a (usually) better way to model its cbv is $C_S[0..u) = \mathbf{0}^{z_1} \mathbf{1}^{\ell_1} \mathbf{0}^{z_2} \mathbf{1}^{\ell_2} \dots \mathbf{0}^{z_r} \mathbf{1}^{\ell_r}$, where the sequences $\mathcal{Z} = \langle z_1, \dots, z_r \rangle$ and $\mathcal{O} = \langle \ell_1, \dots, \ell_r \rangle$ are the lengths of the alternating $\mathbf{0}/\mathbf{1}$ -runs in C_S (assume wlog that C_S begins with $\mathbf{0}$ and ends with $\mathbf{1}$). Then, $\text{rle}(S) = \sum_{i=1}^r (\lceil \lg(z_i - 1) \rceil + 1) + \sum_{i=1}^r (\lceil \lg(\ell_i - 1) \rceil + 1)$. Unfortunately, $\text{gap}(S)$ and $\text{rle}(S)$ are not comparable measures. If $n < u/2$, it holds that $\text{rle}(S) < \mathcal{B}(n, u) + n + O(1)$ [24].

2.2.3 The $\text{trie}(S)$ Compression Measure

Let us consider now representing a set $S \in C^{(n)}$ using a binary trie denoted $\text{bintrie}(S)$, where the $\ell = \lceil \lg u \rceil$ -bit binary encoding of every element is added. Each internal node in $\text{bintrie}(S)$ has two children, the left one corresponding to bit **0** and the right one to bit **1**. The external nodes of $\text{bintrie}(S)$ have no children, as usual. In our case, we distinguish two kinds of external nodes. A *void external node* is one whose depth is either $d < \ell$, or alternatively $d = \ell$ yet it represents no element in S . A *valid external node* (or, simply, *external node*, or alternatively a *leaf*), on the other hand, is one whose depth is exactly ℓ and corresponds to an element in S . Thus, $\text{bintrie}(S)$ has $|S|$ valid external nodes, all at depth ℓ . For a leaf v corresponding to element $x_i \in S$, the root-to- v path is hence labeled with the binary encoding of x_i . This approach has been used for representing sets since at least the late 70s by Trabb-Pardo [50]. Consider the example sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$, and $S_2 = \{2, 5, 7, 12, 15\}$ over universe $[0..16)$, that we shall use as running examples. Figure 1 shows the corresponding tries $\text{bintrie}(S_1)$ and $\text{bintrie}(S_2)$, with external nodes shown as squares and void external nodes with dotted lines. Interestingly, the following compression



■ **Figure 1** Binary tries $\text{bintrie}(S_1)$ and $\text{bintrie}(S_2)$ encoding sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$. Square nodes at depth 4 in the tries correspond to set elements, whereas dotted lines indicate void external nodes.

measure can be derived from this representation [29]. Given two bit strings x and y of ℓ bits each, let $x \ominus y$ denote the bit string obtained after removing the longest common prefix among x and y from x . For instance, for $x = 0110100$ and $y = 0111011$, we have $x \ominus y = 0100$. The prefix omission method by Klein and Shapira [31] represents a sorted set S as a binary sequence $\mathcal{T} = \langle x_1; x_2 \ominus x_1; \dots; x_n \ominus x_{n-1} \rangle$. If we denote $|x_i \ominus x_{i-1}|$ the length of bit string $x_i \ominus x_{i-1}$, then the whole sequence uses

$$\text{trie}(S) = |x_1| + \sum_{i=2}^n |x_i \ominus x_{i-1}|.$$

It turns out that $\text{trie}(S)$ is the number of edges in $\text{bintrie}(S)$ [29]. Notice that $\text{trie}(S)$ decreases as longer trie paths are shared among set elements: consider two integers x and y , the trie represents their longest common prefix just once (then saving space), and then represents both $x \ominus y$ and $y \ominus x$. Extreme cases are as follows: (1) All set elements form a single run of consecutive elements, which maximizes the number of trie edges shared among set elements, hence minimizing the space usage; and (2) The n elements are uniformly distributed within $[0..u)$ (i.e., the gap between successive elements is $g_i = u/n$), which minimizes the number of trie edges shared among elements, and hence maximizes space usage. Notice this is similar to the case that maximizes the $\text{gap}(S)$ measure.

► **Definition 1.** We say that a node v in $\text{bintrie}(S)$ covers all leaves that descend from it. In such a case, we call v a cover node of the corresponding leaves.

The following lemma summarizes several results that shall be important for our work:

► **Lemma 2** ([26], Lemmas 1–5). *For $\text{bintrie}(S)$, the following results hold:*

1. *Any contiguous range of L leaves in $\text{bintrie}(S)$ is covered by $O(\lg L)$ nodes.*
2. *Any set of r nodes in $\text{bintrie}(S)$ has $O(r \lg \frac{u}{r})$ ancestors.*
3. *Any set of r nodes in $\text{bintrie}(S)$ minimally covering a contiguous range of leaves in the trie has $O(r + \lg u)$ ancestors.*
4. *Any set of r nodes in $\text{bintrie}(S)$ minimally covering L contiguous leaves has $O(\lg u + r \lg \frac{L}{r})$ ancestors.*

► **Definition 3.** *Given a set $S = \{x_1, \dots, x_n\} \subseteq [0..u)$, let $S + a$, for $a \in [0..u)$, denote a shifted version of S : $S + a = \{(x_1 + a) \bmod u, (x_2 + a) \bmod u, \dots, (x_n + a) \bmod u\}$.*

The following result is relevant for our proposal:

► **Lemma 4** ([29], Section 2). *Given a set $S \subseteq [0..u)$ of n elements, it holds that:*

1. $\text{trie}(S) \leq \min \{2\text{gap}(S), n \lg(u/n) + 2n - 2\}$.
2. $\exists a \in [0..u)$, such that $\text{trie}(S + a) \leq \text{gap}(S) + 2n - 2$.
3. $\text{trie}(S + a) \leq \text{gap}(S) + 2n - 2$ on average over all values of $a \in [0..u)$.

2.3 Adaptive Set Intersection Algorithms

An adaptive algorithm is one whose running time is a function not only of the instance size (as usual), but also of a difficulty measure of the instance. In this way, “easy” instances are solved faster than “difficult” ones, allowing for a more refined analysis than typical worst-case approaches. For the set intersection problem, algorithms by Demaine, López-Ortiz, and Munro [20] and by Barbay and Kenyon [12] are the most important adaptive approaches. To analyze adaptive intersection algorithms, Demaine et al. and Barbay and Kenyon agree in that any algorithm that computes $\mathcal{I}(\mathcal{Q})$ must show a certificate [12] or proof [20] to prove that the intersection is correct. That is, that any element in $\mathcal{I}(\mathcal{Q})$ belongs to the k sets S_{i_1}, \dots, S_{i_k} , and no element in the intersection has been left out of the result. Then, the analysis determines the size of a certificate (or proof) and the time it takes to compute them. In particular, Barbay and Kenyon [12] *partition certificates* are defined as follows.

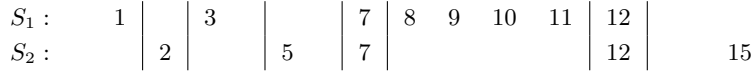
► **Definition 5.** *Given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, a partition certificate is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P}_{\text{BK}}(\mathcal{Q}) = \{I_1, I_2, \dots, I_p\}$, such that:*

1. $\forall x \in \mathcal{I}(\mathcal{Q}), [x..x] \in \mathcal{P}_{\text{BK}}(\mathcal{Q})$;
2. $\forall x \notin \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}_{\text{BK}}(\mathcal{Q}), x \in I_j \wedge \exists q \in \mathcal{Q}, S_q \cap I_j = \emptyset$.

For a given query \mathcal{Q} , several valid partition certificates could be given. However, we are interested in the smallest partition certificate of \mathcal{Q} , as it takes the least time to be computed.

► **Definition 6.** *For a given query instance $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, let δ denote the size of the smallest partition certificate of \mathcal{Q} .*

Measure δ is known as the *alternation* of the query instance [12], measuring its difficulty. Notice $|\mathcal{I}(\mathcal{Q})| \leq \delta$ holds. Figure 2 shows the smallest partition certificate (of size $\delta = 8$) for sets S_1 and S_2 of our running example. Barbay and Kenyon [11, 12] proved a lower bound of $\Omega(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ comparisons for the set intersection problem. They also gave an optimal intersection algorithm running in $O(\delta \sum_{i \in \mathcal{Q}} \lg(n_i/\delta))$ time.



■ **Figure 2** Vertical lines show the smallest partition certificate $\mathcal{P} = \{[0..1], [2..2], [3..4], [5..6], [7..7], [8..11], [12..12], [13..15]\}$ of size $\delta = 8$ of the universe $[0..16)$ for the intersection of sets $S_1 = \{1, 3, 7, 8, 9, 10, 11, 12\}$ and $S_2 = \{2, 5, 7, 12, 15\}$.

3 Trie Intersection Certificates: A Revisit to Trabb-Pardo Algorithm

In this section we revisit an old divide-and-conquer intersection algorithm by Trabb-Pardo [50], not only to review it but also to prove an adaptive bound on its running time. Algorithm 1 shows the pseudocode. Given a query instance $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, the algorithm must

■ **Algorithm 1** TP-Intersection(sets S_1, \dots, S_k ; universe $[L..R)$).

```

Result: The set intersection  $S_1 \cap \dots \cap S_k$ 
1 begin
2   // Base cases
3   for  $i \leftarrow 1$  to  $k$  do
4     if  $S_i = \emptyset$  then
5       return  $\emptyset$ 
6   if  $L = R$  then
7     return  $\{L\}$            // Universe of size 1, all sets are the same singleton
8   else
9     // Divide
10     $M \leftarrow \lfloor (R + L)/2 \rfloor$ 
11    for  $i \leftarrow 1$  to  $k$  do
12       $S_{i,l} \leftarrow \{x \in S_i \mid x \in [L..M)\}$ 
13       $S_{i,r} \leftarrow \{x \in S_i \mid x \in [M..R)\}$ 
14    // Conquer
15     $R_1 \leftarrow \text{TP-Intersection}(S_{1,l}, \dots, S_{k,l}, [L..M))$ 
16     $R_2 \leftarrow \text{TP-Intersection}(S_{1,r}, \dots, S_{k,r}, [M..R))$ 
17    // Combine
18    return  $R_1 \cup R_2$            // Disjoint set union

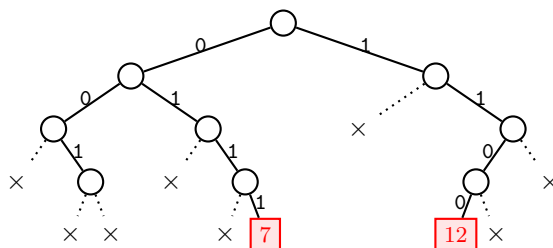
```

be invoked as $\text{TP-Intersection}(S_{i_1}, \dots, S_{i_k}, [0..u))$. The main idea is to divide the universe into two halves, to then split each set according to this universe division. This differs from, e.g., Baeza-Yates's algorithm [9, 10], which splits according to the median of one of the sets. The *Divide* steps (lines 10 and 11) can be implemented using binary search. At the first level of recursion, the most-significant bit of every element in sets $S_{i,l}$ is 0, as they belong to the left half of the universe. Similarly, for $S_{i,r}$ the most-significant bit is 1 as all elements belong to the right half. At each node of the recursion tree, the current universe is divided into two halves, to then recurse on the sets split accordingly.

As sets are known in advance to queries and set splits carried out by Algorithm 1 depend just on the universe, the Divide step of Algorithm 1 can be implemented efficiently by using a suitable set representation that not only stores the set values, but also precomputes the set splits carried out recursively by the algorithm. Trabb-Pardo proposes to represent each $S_i \in \mathcal{S}$ using $\text{bintrie}(S_i)$, mimicking the way set S_i is recursively split by Algorithm 1. The left child of the root represents all elements whose most-significant bit is 0, i.e., elements in set $S_{i,l}$ of Algorithm 1 (line 10) in the first level of recursion; similarly for $S_{i,r}$, containing

all elements in S_i whose most-significant bit is 1. To simulate the recursive execution of Algorithm 1 on the binary tries, one must carry out a DFS traversal in synchronization on all tries involved in the query, following the same path in all of them and stopping (and backtracking if needed) as soon as we reach a dead end in one of the tries (which correspond to dotted lines in Figure 1), or we reach a leaf node in all the tries (in which case we have found an element belonging to the intersection). In this way, (1) we stop as soon as we detect a universe interval that does not have any element in the intersection, and (2) we find the relevant elements when we arrive at the leaves.

To analyze Algorithm 1, we introduce the concept of *trie intersection certificates*, denoted $\text{cert}(\mathcal{Q})$, as an alternative to existing certificates [20, 12]. Figure 3 shows a possible $\text{cert}(\mathcal{Q})$ for the intersection of S_1 and S_2 from Figure 1. Let $\text{path}(v)$ denote the binary string labelling the root-to- v path, and $\text{depth}(v) = |\text{path}(v)|$. For a query \mathcal{Q} , a binary trie $\text{cert}(\mathcal{Q})$ is a



■ **Figure 3** Trie certificate for the intersection $\{1, 3, 7, 8, 9, 10, 11, 12\} \cap \{2, 5, 7, 12, 15\}$. This trie shows the nodes that must be checked to determine that the result is, in this case, $\{7, 12\}$. This corresponds to the intersection of the binary tries representing these sets.

trie partition certificate if: (1) for any internal node v of $\text{cert}(\mathcal{Q})$ such that $\text{path}(v) = b$, there exists an internal node v_i with $\text{path}(v_i) = b$ in every $\text{bintrie}(S_i)$, $i \in \mathcal{Q}$; (2) for any void external node v with $\text{depth}(v) = d \leq \lceil \lg u \rceil$ and $\text{path}(v) = b$, there exists at least a set S_i ($i \in \mathcal{Q}$) such that there is no node v_i with $\text{path}(v_i) = b$ in $\text{bintrie}(S_i)$. So, the universe interval $[\text{dec}(b \cdot 0^{\ell-d}).. \text{dec}(b \cdot 1^{\ell-d})]$ has no element in the intersection, where $\text{dec}(x)$ denotes the decimal representation of a binary string x . We call them *fail nodes*, shown as “ \times ” in Figure 3; and (3) for any valid external node v with $\text{depth}(v) = \ell$ corresponding to $\text{path}(v) = b$, there exists a valid external node v_i with $\text{path}(v_i) = b$ in every $\text{bintrie}(S_i)$, $i \in \mathcal{Q}$. We call them *success nodes* as they correspond to elements in $\mathcal{I}(\mathcal{Q})$.

Notice that $\text{cert}(\mathcal{Q})$ is the trie obtained by intersecting $\text{bintrie}(S_i)$, for all $i \in \mathcal{Q}$, and that it is the smallest trie that allows us to prove the correctness of the intersection. For instance, Figure 3 shows $\text{cert}(\mathcal{Q})$ for a given query. Interestingly, the recursion tree of Algorithm 1 is exactly $\text{cert}(\mathcal{Q})$, as the algorithm stops as soon as one arrives at a fail node. The external nodes of $\text{cert}(\mathcal{Q})$ cover the universe $[0..u)$ with intervals, similar to Barbay and Kenyon partition certificates, as stated by the following definition.

► **Definition 7.** Given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, its trie partition certificate is a partition of the universe $[0..u)$ into a set of intervals, we say that $\text{cert}(\mathcal{Q})$ induces the following partition of $[0..u)$ into a set of intervals that we call trie partition certificate:

$$\mathcal{P}_{\text{AC}}(\mathcal{Q}) = \bigcup_{l \in \mathcal{E}(\text{cert}(\mathcal{Q}))} \left\{ [\text{dec}(\text{path}(l) \cdot 0^{\ell - \text{depth}(l)}).. \text{dec}(\text{path}(l) \cdot 1^{\ell - \text{depth}(l)})] \right\},$$

where $\mathcal{E}(\text{cert}(\mathcal{Q}))$ denotes the set of external nodes of $\text{cert}(\mathcal{Q})$.

For instance, the trie certificate of Figure 3 induces the following trie partition certificate of the universe $[0..16)$: $\{[0..1], [2..2], [3..3], [4..5], [6..6], [7..7], [8..11], [12..12], [13..13], [14..15]\}$.

Since $\text{cert}(\mathcal{Q})$ is the recursion tree of Algorithm 1, its running time is $O(k|\text{cert}(\mathcal{Q})|)$. As in the worst-case one must traverse completely all tries $\text{bintrie}(S_i)$, $i \in \mathcal{Q}$, we have:

$$k|\text{cert}(\mathcal{Q})| \leq \sum_{i \in \mathcal{Q}} \text{trie}(S_i) \leq \sum_{i \in \mathcal{Q}} n_i \lg \frac{u}{n_i} + 2n_i - 2,$$

where the last bound is from Lemma 4 (1). Next we prove an adaptive bound for $k|\text{cert}(\mathcal{Q})|$.

► **Theorem 8.** *Given a query instance $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$ with alternation measure δ and over sets with universe $[0..u]$, algorithm *TP-Intersection* computes $\mathcal{I}(\mathcal{Q}) = \cap_{i \in \mathcal{Q}} S_i$ in time $O(k\delta \lg(u/\delta))$.*

Proof. Consider a smallest partition certificate $\mathcal{P}_{\text{BK}}(\mathcal{Q}) = \{I_1, \dots, I_\delta\}$ of universe $[0..u]$, such that $|I_i| = L_i$ for $i = 1, \dots, \delta$. Let us think now of the worst-case smallest $\text{cert}(\mathcal{Q})$ we could have, by covering the δ intervals in $\mathcal{P}_{\text{BK}}(\mathcal{Q})$ with as many external nodes of $\text{cert}(\mathcal{Q})$ as possible. For any $I_j \in \mathcal{P}_{\text{BK}}(\mathcal{Q})$ formed by elements not in $\mathcal{I}(\mathcal{Q})$, there exists a set of external fail nodes in $\text{cert}(\mathcal{Q})$ that cover I_j . This is because when traversing the tries $\text{bintrie}(S_i)$ in coordination, $i \in \mathcal{Q}$, the algorithm stops as long as one gets into one of the cover nodes of I_j , since it does not belong to at least one of the tries. According to Lemma 2 (1), a contiguous range of L leaves (corresponding to the values in I_j) can be covered with up to $O(\lg L)$ nodes. Thus, in the worst-case, $\text{cert}(\mathcal{Q})$ has $O(\sum_{i=1}^{\delta} \lg L_i)$ external nodes that overall cover $[0..u]$. Now, recall that the external nodes of $\text{cert}(\mathcal{Q})$ cover the contiguous range of leaves corresponding to $[0..u]$. Hence, according to Lemma 2 (3), these external nodes have $O(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ ancestors, so overall $\text{cert}(\mathcal{Q})$ has $O(\sum_{i=1}^{\delta} \lg L_i + \lg u)$ nodes. The sum is maximized when $L_i = u/\delta$, for all $1 \leq i \leq \delta$, hence $\text{cert}(\mathcal{Q})$ has $O(\delta \lg(u/\delta))$ nodes. The result follows from the fact that for each node in $\text{cert}(\mathcal{Q})$ the algorithm runs in time $O(k)$. ◀

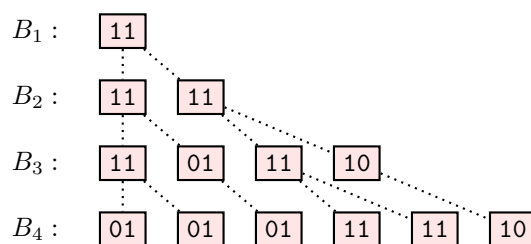
4 Compressed Intersectable Sets

We devise next a space-efficient representation of $\text{bintrie}(S)$, for a set $S = \{x_1, \dots, x_n\} \subseteq [0..u]$ of n elements such that $0 \leq x_1 < \dots < x_n < u$. This representation will also allow for efficient intersections, supporting Trabb-Pardo's [50] algorithm.

We represent $\text{bintrie}(S)$ level-wise [30]. Let $B_1[1..2l_1], \dots, B_\ell[1..2l_\ell]$ be bit vectors such that B_i represents the l_i nodes at level i of $\text{bintrie}(S)$ ($1 \leq i \leq \ell$), from left to right. Each node is encoded using 2 bits, indicating the presence (using bit **1**) or absence (bit **0**) of the left and right children, respectively. In this way, the feasible codewords for trie nodes are **01**, **10**, and **11**, whereas **00** is not a valid codeword. The codewords of all nodes at level $i \geq 1$ in the trie are concatenated from left to right to form B_i . The j -th node at level i (from left to right) is stored at positions $2j - 1$ and $2j$. We say that $2j - 1$ is the position of such node in B_i .

Let p be the position in B_i corresponding to a node v at level i of $\text{bintrie}(S)$. As the nodes are stored level-wise and from left to right, the number of **1**s before position p in B_i equals the number of nodes in level $i + 1$ that are before the child(ren) of node v . So, $2B_i.\text{rank}_1(p - 1) + 1$ yields the position of B_{i+1} where the first child of node v is. Figure 4 illustrates our representation.

The total number of **1**s in the bit vectors of our representation equals the number of edges in the trie. That is, there are $\text{trie}(S)$ **1**s. Besides, the trie has $\text{trie}(S) + 1$ internal nodes and leaves: n of them are leaves, so $\text{trie}(S) - n + 1$ are internal. In our representation we only need to represent the internal trie nodes. As we encode each node using 2 bits, the total space usage for B_1, \dots, B_ℓ is $2(\text{trie}(S) - n + 1)$ bits. On top of them we use Clark's data structure [16] to support rank in $O(1)$ time, adding $o(\text{trie}(S))$ extra bits overall.



■ **Figure 4** Level-wise bit vector representation of $\text{bintrie}(S)$ for $S = \{1, 3, 7, 8, 9, 10, 11, 12\}$. Dotted lines are implicit, as they are computed using operation rank_1 on the bit vectors.

Given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, we traverse $\text{bintrie}(S_{i_1}), \dots, \text{bintrie}(S_{i_k})$ using a recursive DFS traversal as in Algorithm 1. Besides the query itself, our algorithm receives: (1) an integer value, $level$, indicating the current recursion level, and (2) integer values r_1, \dots, r_k , indicating the current nodes in each trie, represented as the positions of these nodes within B_{level} . Algorithm 2 shows the pseudo-code of our adaptive and compressed algorithm to compute the compact representation for $\text{bintrie}(\mathcal{I}(\mathcal{Q}))$ (denoted T_I in the pseudocode). The algorithm uses a binary variable s , initialized with $\mathbf{11}$, which stores the bitwise-and of all current node codewords (line 4). So, $s = \mathbf{00}$ means that recursion must stop, $s = \mathbf{10}$ indicates to go down only to the left, $s = \mathbf{01}$ just to the right, and $s = \mathbf{11}$ to both children.

Lines 9–13 carry out the needed computation to go down to the left child. In particular, we compute the positions of the left-subtrie roots using rank_1 operation. Then, in line 13 we recursively go down to the left. The result of that recursion is stored in variable $lChild$, indicating with a $\mathbf{1}$ that the left recursion yielded a non-empty intersection, $\mathbf{0}$ otherwise. A similar procedure is carried out for the right child in lines 14–21. Line 17 determines whether we have already computed the rank_1 s corresponding to the left child. If that is not the case, we compute them in line 18. In this way, we compute only one rank_1 operation per traversed node in the tries, which is important in practice. Just as for the left child, we store the result of the right-child recursion in variable $rChild$ in line 21. Finally, in line 22 we determine whether the left and right recursions yielded an empty intersection or not. If both $lChild = rChild = \mathbf{0}$, the intersection was empty on both children, so we return $\mathbf{0}$. Otherwise, we append $lChild$ and $rChild$ to $T_I.B_{level}$, as that is the codeword of the corresponding node in T_I . Note how we actually generate the output trie T_I in postorder, after we visited both children of the current nodes, despite the input is traversed in preorder. Thus, we write the output in time proportional to its size. Although the total running time is still proportional to $|\text{cert}(\mathcal{Q})|$, this can save important time in practice.

Besides computing $\mathcal{I}(\mathcal{Q})$, a distinctive feature of our algorithm is that it also allows one to obtain the sequence $\langle \text{rank}(S_{i_1}, x), \dots, \text{rank}(S_{i_k}, x) \rangle$, for all $x \in \mathcal{I}(\mathcal{Q})$, for free (in asymptotic terms). The idea is to compute $\langle \text{bintrie}(S_{i_1}).B_\ell.\text{rank}_1(r_1), \dots, \text{bintrie}(S_{i_k}).B_\ell.\text{rank}_1(r_k) \rangle$ every time the recursion reaches level ℓ (i.e., just before the **return** of line 7 in Algorithm 2). Outputting this information is important for several applications, such as cases where set elements have satellite data associated to them. For an element $x_j \in S_i$, the associated data d_j is stored in an auxiliary array $D_i[1..n_i]$ such that $D[\text{rank}(S_i, x_j)] = d_j$. Typical applications are inverted indexes in IR (where ranking information, such as frequencies, is associated to inverted list elements), and the Leapfrog Triejoin algorithm [51] (where at each step we must compute the intersection of sets, and for each element in the intersection we must go down following a pointer associated to it).

■ **Algorithm 2** AC-Intersection(query \mathcal{Q} ; roots r_1, \dots, r_k ; level).

```

Result: The binary trie  $T_I$  representing  $\mathcal{I}(\mathcal{Q}) = \bigcap_{i \in \mathcal{Q}} S_i$ 
1 begin
2    $s \leftarrow 11$  // binary encoding
3   for  $i \in \mathcal{Q}$  do
4      $s \leftarrow s \& (\text{bintrie}(S_i).B_{\text{level}}[r_i] \cdot \text{bintrie}(S_i).B_{\text{level}}[r_i + 1])$ 
5   if  $\text{level} = \ell$  then
6     append  $s$  to  $T_I.B_\ell$ 
7     return 1
8    $lChild \leftarrow 0$ ;  $rChild \leftarrow 0$ 
   // Go down to the left in the tries
9   if  $s$  is 10 or 11 then
10     $lRoots \leftarrow \emptyset$ 
11    for  $i \in \mathcal{Q}$  do
12       $lRoots \leftarrow lRoots \cup \{2 \times \text{bintrie}(S_i).B_{\text{level}}.\text{rank}_1(r_i - 1) + 1\}$ 
13     $lChild \leftarrow \text{AC-Intersection}(\mathcal{Q}, lRoots, \text{level} + 1)$ 
   // Go down to the right in the tries
14  if  $s$  is 01 or 11 then
15     $rRoots \leftarrow \emptyset$ 
16    for  $i \in \mathcal{Q}$  do
17      if  $s = 01$  then
18         $rRoots \leftarrow rRoots \cup \{2 \times \text{bintrie}(S_i).B_{\text{level}}.\text{rank}_1(r_i - 1) + 2\}$ 
19      else
20         $rRoots \leftarrow rRoots \cup \{lRoots_i + 2\}$ 
21     $rChild \leftarrow \text{AC-Intersection}(\mathcal{Q}, rRoots, \text{level} + 1)$ 
   // Output written in postorder
22  if  $lChild \neq 0$  or  $rChild \neq 0$  then
23    append  $lChild \cdot rChild$  to  $T_I.B_{\text{level}}$ 
24    return 1
25  else
26    return 0

```

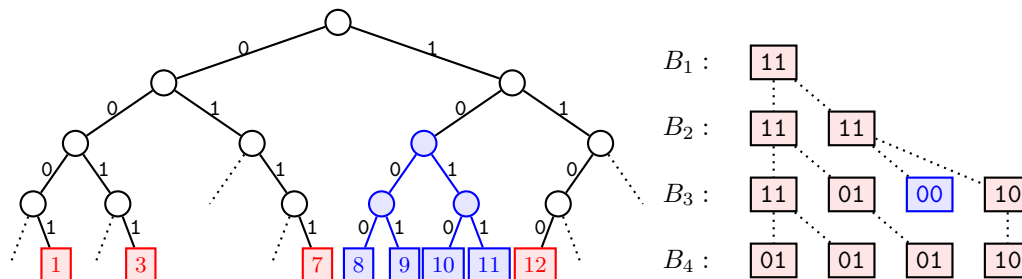
We have proved the following theorem:

► **Theorem 9.** *Let $\mathcal{S} = \{S_1, \dots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u]$. There exists a data structure able to represent each set S_i using $2(\text{trie}(S_i) - n_i + 1) + o(\text{trie}(S_i))$ bits, such that given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(\mathcal{Q}) = \bigcap_{i \in \mathcal{Q}} S_i$ can be computed in $O(k\delta \lg(u/\delta))$ time, where δ is the alternation measure of \mathcal{Q} . Besides, for every $x \in \mathcal{I}(\mathcal{Q})$, the data structure also allows one to obtain the sequence $\langle \text{rank}(S_{i_1}, x), \dots, \text{rank}(S_{i_k}, x) \rangle$ asymptotically for free.*

5 Compressing Runs of Elements

Next, we exploit the presence of runs of successive elements in the input sets to reduce both the space usage of the binary trie representation, as well as intersection time. Runs tend to be captured by full subtrees in the corresponding binary tries. See, e.g., the full subtree whose leaves correspond to elements 8, 9, 10, 11 in the binary trie of Figure 5. Let v be a $\text{bintrie}(S)$ node whose subtree is full. Let $\text{depth}(v) = d$. If $b = \text{path}(v)$, the $2^{\ell-d}$ leaves

covered by v correspond to the integer interval $[\text{dec}(b \cdot \mathbf{0}^{\ell-d}).. \text{dec}(b \cdot \mathbf{1}^{\ell-d})]$. So, the subtree of v can be removed, keeping just v , saving space and still being able to recover the removed elements.



■ **Figure 5** Left side, the binary trie representing set $\{1, 3, 7, 8, 9, 10, 11, 12\}$. Notice that the subtree whose leaves correspond to elements 8, 9, 10, 11 is a full subtree. Right side, our compact representation removing full subtrees and encoding their roots with **00**.

► **Definition 10.** Let $S \subseteq [0..u)$ be a set of n elements. We define $\text{rTrie}(S)$ as the number of edges in $\text{bintrie}(S)$ after removing the maximal full subtrees.

This immediately implies $\text{rTrie}(S) \leq \text{trie}(S) \leq 2\text{gap}(S)$, yet we can prove tighter bounds. Assume a set S with r runs of ℓ_1, \dots, ℓ_r successive elements each, respectively. The ℓ_i elements of a given run correspond to ℓ_i contiguous leaves in $\text{bintrie}(S)$ which, according to Lemma 2 (item 1), are covered by at most $2\lceil \lg(\ell_i/2) \rceil$ nodes. This is a pessimistic case that removes the least edges, so we analyze it. Among the cover nodes, there are 2 whose subtrees have 0 edges, 2 whose subtrees have 2 edges, 2 whose subtrees have 6 edges, and so on. In general, for each $i = 1, \dots, \lceil \lg(\ell_i/2) \rceil$, there are 2 cover nodes whose subtrees have $2^i - 2$ edges. If we remove them all, the total number of edges removed is $2 \sum_{i=i}^{\lceil \lg(\ell_i/2) \rceil} (2^i - 2) \leq 2\ell_i - 4\lg \ell_i$. This removes the least edges belonging to full subtrees, so we can bound

$$\text{rTrie}(S) \leq \text{trie}(S) - \sum_{i=1}^r (2\ell_i - 4\lg \ell_i). \tag{1}$$

We can also prove the following bounds.

► **Lemma 11.** Given a set $S \subseteq [0..u)$ of n elements, it holds that

1. $\text{rTrie}(S) \leq 2 \cdot \min \{ \text{rle}(S) + \sum_{i=1}^r \lg \ell_i, \text{gap}(S) \}$.
2. $\exists a \in [0..u)$, such that $\text{rTrie}(S + a) \leq \min \{ \text{rle}(S) - \sum_{i=1}^r \ell_i + 3 \sum_{i=1}^r \lg \ell_i, \text{gap}(S) \} + 2n - 2$.
3. $\text{rTrie}(S + a) \leq \min \{ \text{rle}(S) - \sum_{i=1}^r \ell_i + 3 \sum_{i=1}^r \lg \ell_i, \text{gap}(S) \} + 2n - 2$ on average, assuming $a \in [0..u)$ is chosen uniformly at random.

Proof. Since S has r runs of ℓ_1, \dots, ℓ_r elements, we can rewrite $\text{gap}(S) = \sum_{i=1}^r (\lceil \lg(z_i - 1) \rceil + 1) + \sum_{i=1}^r (\ell_i - 1)$. As $\text{rTrie}(S) \leq \text{trie}(S) \leq 2\text{gap}(S)$, and $\text{rTrie}(S) \leq \text{trie}(S) - \sum_{i=1}^r (2\ell_i - 4\lg \ell_i)$ (Equation 1), it holds that

$$\begin{aligned}
\text{rTrie}(S) &\leq \text{trie}(S) - \sum_{i=1}^r (2\ell_i - 4\lg \ell_i) \\
&\leq 2\left(\sum_{i=1}^r (\lceil \lg(z_i - 1) \rceil + 1) + \sum_{i=1}^r (\ell_i - 1)\right) - \sum_{i=1}^r (2\ell_i - 4\lg \ell_i) \\
&= 2\sum_{i=1}^r (\lceil \lg(z_i - 1) \rceil + 1) + 4\sum_{i=1}^r \lg \ell_i = 2(\text{rle}(S) + \sum_{i=1}^r \lg \ell_i),
\end{aligned}$$

proving item 1. Items 2 and 3 can be proved similarly from items 2 and 3 of Lemma 4. ◀

In our compact representation, we encode a full-subtree cover node using **00**. Recall that **00** is an invalid codeword, so we use it as a special mark. See Figure 5 for an illustration.

Given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, the procedure to compute $\mathcal{I}(\mathcal{Q})$ is similar to that of Algorithm 2. The only difference is that if in a given trie $\text{bintrie}(S_i)$ we arrive at a node encoded **00**, every possible set element in the subtree of the node belongs to S_i . In other words, the intersection within the current subtrees is independent of S_i , so we can safely temporarily exclude $\text{bintrie}(S_i)$ from the intersection and continue intersecting the remaining tries. To implement this idea, we keep boolean flags f_1, \dots, f_k such that f_j corresponds to $\text{bintrie}(S_{i_j})$. The idea is that at each point during the synchronized DFS traversal, only tries whose flag is true participate in the intersection. Initially, we set $f_i \leftarrow \text{true}$, for $1 \leq i \leq k$. If, during the intersection process, we arrive at a node encoded **00** in $\text{bintrie}(S_i)$, we set $f_i \leftarrow \text{false}$. When the recursion at a node encoded **00** in $\text{bintrie}(S_i)$ finishes, we set $f_i \leftarrow \text{true}$ again. If, at a given point, all tries have been temporarily excluded but one, let us say $\text{bintrie}(S_j)$, we only need to traverse the current subtree in S_j , copying it verbatim to the output. If this subtree contains nodes encoded **00**, they will appear in the output. This way, the maximal runs of successive elements in the output will be covered by nodes encoded **00**. This fact is key for the adaptive running time of our algorithm, as we shall see below.

We analyze our algorithm introducing the following variant of partition certificates.

► **Definition 12.** *Given a query instance $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, a run-partition certificate for it is a partition of the universe $[0..u)$ into a set of intervals $\mathcal{P}_{\text{AC}}^r(\mathcal{Q}) = \{I_1, I_2, \dots, I_p\}$, such that the following conditions hold:*

1. $\forall x \in \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}_{\text{AC}}^r(\mathcal{Q}), \text{ such that } x \in I_j \wedge \mathcal{I}(\mathcal{Q}) \cap I_j = I_j;$
2. $\forall x \notin \mathcal{I}(\mathcal{Q}), \exists I_j \in \mathcal{P}_{\text{AC}}^r(\mathcal{Q}), \text{ such that } x \in I_j \wedge \exists q \in \mathcal{Q}, S_q \cap I_j = \emptyset.$

Let ξ denote the size of the smallest run-partition certificate $\mathcal{P}_{\text{AC}}^r(\mathcal{Q})$ of \mathcal{Q} . We call ξ the run alternation measure.

Item 2 is the same as for Barbay and Kenyon's partition certificates, corresponding to intervals of elements not in $\mathcal{I}(\mathcal{Q})$. Item 1, on the other hand, corresponds to elements in $\mathcal{I}(\mathcal{Q})$ which, unlike Barbay and Kenyon certificates, are not necessarily covered by singletons: our definition allows one to cover a run of successive elements in $\mathcal{I}(\mathcal{Q})$ using a single interval. Clearly, $\xi \leq \delta$ holds. Besides, although $|\mathcal{I}(\mathcal{Q})| \leq \delta$ holds, in our case there can be query instances such that $\xi < |\mathcal{I}(\mathcal{Q})|$. Figure 6 illustrates our definition for an intersection of 4 sets on the universe $[0..15)$. Notice that $\xi = 5$, whereas $|\mathcal{I}(\mathcal{Q})| = 6$ and $\delta = 9$.

We must also introduce a fourth type of node to our trie certificate definition of Section 3. If for an internal node v of $\text{cert}(\mathcal{Q})$ with $\text{path}(v) = b$, it holds that there is a node v_i with $\text{path}(v_i) = b$ in every $\text{bintrie}(S_i)$, $i \in \mathcal{Q}$, and the subtrees of all v_i s is full, then v is called an *internal success node*. It is important to note that every interval I_j from item 1 of Definition 12 is covered only by internal success nodes. Also, internal success nodes only cover intervals from item 1 of Definition 12.

$S_{i_1} :$	7	8	9	10	11	12	13	14	15			
$S_{i_2} :$	5	6	7	8	9	10	11	12	13	14		
$S_{i_3} :$	4	5	6	7	8	9	11	12	13	14		
$S_{i_4} :$					8	9	10	11	12	13	14	15

■ **Figure 6** A query instance $\mathcal{Q} = \{S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4}\}$ and its smallest run-partition certificate $\mathcal{P}_{\text{AC}}^r(\mathcal{Q}) = \{[0..7], [8..9], [10..10], [11..14], [15..15]\}$ of size $\xi = 5$.

Our main result is stated in the following theorem:

► **Theorem 13.** *Let $\mathcal{S} = \{S_1, \dots, S_N\}$ be a family of N integer sets, each of size $|S_i| = n_i$ and universe $[0..u)$. There exists a data structure able to represent each set S_i using $2\text{rTrie}(S_i)(1 + o(\text{rTrie}(S_i)))$ bits, such that given a query $\mathcal{Q} = \{i_1, \dots, i_k\} \subseteq [1..N]$, the intersection $\mathcal{I}(\mathcal{Q}) = \bigcap_{i \in \mathcal{Q}} S_i$ can be computed in $O(k\xi \lg(u/\xi))$ time, where ξ is the run alternation measure of \mathcal{Q} .*

Proof. Consider the smallest run-partition certificate $\mathcal{P}_{\text{AC}}^r(\mathcal{Q}) = \{I_1, \dots, I_\xi\}$ of universe $[0..u)$, such that $|I_i| = L_i$ for $i = 1, \dots, \xi$. Let us cover these ξ intervals with as many nodes of the smallest $\text{cert}(\mathcal{Q})$ as possible. As we already saw in the proof of Theorem 8, all intervals I_i such that $I_i \cap \mathcal{I}(\mathcal{Q}) = \emptyset$ are covered by at most $O(\lg L_i)$ nodes in $\text{cert}(\mathcal{Q})$. We now prove the same for intervals $I_j \subseteq \mathcal{I}(\mathcal{Q})$, which are covered by internal success nodes of $\text{cert}(\mathcal{Q})$. The only thing to note is that our algorithm stops as soon as it arrives to an internal success node. As there can be $O(\lg L_j)$ such cover nodes, universe $[0..u)$ can be covered by $O(\lg u + \sum_{i=1}^{\xi} \lg L_i) = O(\lg u + \sum_{i=1}^{\xi} \lg(u/\xi))$ nodes, hence $\text{cert}(\mathcal{Q})$ has $O(\xi \lg(u/\xi))$ nodes overall. The result follows from the fact that at each node the algorithm runs in time $O(k)$. ◀

6 Implementation

We implemented bit vectors B_1, \dots, B_ℓ in plain form using class `bit_vector<>` from the `sds1` library [27]. We support `rank1` on them using different data structures to obtain the following schemes. (`trie v`, `rTrie v`): the variants defined in Section 4 and 5, respectively, using `rank_support_v` for `rank1`. It uses $\sim 25\%$ extra space on top of the bit vector, supporting `rank1` in $O(1)$ time. (`trie v5`, `rTrie v5`): use `rank_support_v5`, requiring $\sim 6.25\%$ extra space on top of the bit vectors, supporting `rank1` in $O(1)$ time. This alternative is smaller, yet slower in practice. (`trie IL`, `rTrie IL`): use `rank_support_il`, aiming at reducing the number of cache misses to compute `rank1`. We use block size 512, requiring $\sim 12.5\%$ extra space on top of the bit vectors, while supporting `rank1` in $O(1)$ time.

Most state-of-the-art alternatives we compare with do not support operation `rank(S, x)`. So, to be fair, we do not store any `rank1` data structure for the last-level bit vector B_ℓ . Recall that `rank(S, x)` is equivalent to a `rank1` on the corresponding position of B_ℓ . We implemented Algorithm 2 on our compact trie data structures, following the descriptions from Sections 4 and 5 very closely. We implemented, however, two alternatives for representing the output: (1) the binary trie representation, and (2) the plain array representation. In our experiments we will use the latter, to be fair: all testes alternatives produce their outputs in plain form.

We also implemented a simple multithreaded version of our algorithm. Let t denote the number of available threads. Then, we define $c = \lceil \lg t \rceil$. Our algorithm proceeds as in Algorithm 2, generating a binary trie of height c (that we will call *top trie*), with at most t leaves. Then, we execute Algorithm 2 again, this time in parallel, with each thread

starting from a different leaf of the top trie. Each thread generates its own output in parallel, using our compact trie representation. Once all threads finish, we concatenate these tries to generate the final output. We just need to count, in parallel, how many nodes there are in each level of the trie. Then, we allocate a bit vector of the appropriate size for each level, where each thread will write its own part of the output in parallel. This simple approach does not guarantee load balancing among threads, however it works relatively well in practice.

Our source code and instructions to replicate our experiments are available at <https://github.com/jpcastillo/compressed-binary-tries>.

7 Experimental Results

We experimentally evaluate our approaches on a server with an i7 10700k CPU, 8 cores and 16 threads at 4.70 GHz, 32 GB of RAM (DDR4-3.6GHz) running in dual channel, and Ubuntu 20.04 LTS OS. Our implementation is developed in C++, compiled with g++ 9.3.0 and optimization flags `-O3` and `-march=native`.

In our tests, we used families of sets corresponding to inverted indexes of three standard document collections: Gov2 [17], ClueWeb09 [1], and CC-News [38]. For Gov2 and ClueWeb09 collections, we used the freely-available inverted indexes and query logs by Daniel Lemire (see [34] for details), corresponding to the URL-sorted document enumeration [48] (which tends to yield runs of successive elements in the sets). The query log contains 20,000 random queries from the TREC million-query track (1MQ). Each query has at least 2 query terms. Also, each term is in the top-1M most frequently queried terms. For CC-News we use the freely-available inverted index by Mackenzie et al. [38] in *Common Index File Format* (CIFF) [37], as well as their query log of 9,666 queries. Table 1 shows a summary of statistics of the collections. In all cases, we only keep sets with at least 4,096 elements.

■ **Table 1** Dataset summary and average space usage (in bits per integer, bpi) for different compression measures and baseline representations.

	Gov2	ClueWeb09	CC-News
# Lists	57,225	131,567	79,831
# Integers	5,509,206,378	14,895,136,282	18,415,151,585
u	25,205,179	50,220,423	43,495,426
$\lceil \lg u \rceil$	25	26	26
$\text{gap}(S)$	2.25	3.25	3.70
$\text{rle}(S)$	1.99	3.33	4.23
$\text{trie}(S)$	3.48	4.56	5.18
$\text{rTrie}(S)$	2.51	4.00	5.12
Elias γ	3.71	5.74	6.81
Elias δ	3.64	5.40	6.69
Fibonacci	3.90	5.35	6.09
Elias γ 128	4.07	6.10	7.05
Elias δ 128	4.00	5.77	7.17
Fibonacci 128	4.26	5.71	6.45
$\text{rrr_vector}\langle\rangle$	11.82	19.94	11.29
$\text{sd_vector}\langle\rangle$	8.45	8.52	7.17

As baseline, Table 1 also shows the average bit per integer (bpi) for different compression measures on our tested set collections. We also show the average bpi for different integer compression approaches, namely Elias γ and δ [22], Fibonacci [25], `rrr_vector<>` [47], and `sd_vector<>` [42], all of them from the `sds1` library [27]. In particular, Elias γ , δ , and Fibonacci codes are known for yielding highly space-efficient set representations in IR indexing [15], hence they are a strong baselines for comparison. We show a plain version of them, as well as variants with blocks of 128 integers. The latter are needed to speed up decoding. However, these approaches are relatively slow to be decoded [15, See Table 6.9], and hence yield higher intersection times. On the other hand, `sd_vector<>` uses $n \lg(u/n) + 2n + o(n)$ bits to encode a set of n elements and universe $[0..u)$. Finally, `rrr_vector<>` uses $\mathcal{B}(n, u) + o(u)$ bits of space. As it can be seen, the $o(u)$ -bit term yields a higher space usage.

Next, we compare our approaches with state-of-the-art set compression alternatives available at the project *Performant Indexes and Search for Academia*¹ (PISA) [39]:

- **IPC**: the Binary Interpolative Coding approach by Moffat et al. [40]. This is a highly space-efficient approach, with a relatively slow processing performance [15, 40].
- **PEF Opt**: the highly competitive approach by Ottaviano and Venturini [43].
- **OptPFD**: The Optimized PForDelta approach by Yan et al. [53].
- **SIMD-BP128**: The highly efficient approach by Lemire and Boytsov [35], aimed at decoding billions of integers per second using vectorization capabilities of modern processors.
- **Simple16**: The approach by Zhang et al. [54], a variant of the **Simple9** approach [5] that combines a relatively good space usage and an efficient intersection time.
- **VarintGB**: The approach used in Google and presented by Dean [19].
- **Varint-G8IU**: by Stepanov et al. [49], using SIMD instructions to speed-up set processing.

We also compared with the following approaches, available from their authors:

- **Roaring**: the compressed bitmap approach by Lemire et al. [36], widely used as indexing tool on several systems and platforms [3]. Roaring bitmaps are highly competitive, leveraging modern CPU hardware architectures. We use the code from the authors [2].
- **RUP**: The recent recursive universe partitioning approach by Pibiri [45], using also SIMD instructions to speed up processing. We use the code from the author [44].

Table 2 shows the average experimental intersection time (in milliseconds per query) and space usage (in bits per integer) for all the alternatives tested. Figure 7 (in the Appendix) shows the same results, using space vs. time plots. Our approaches introduce competitive trade-offs, as follows:

Results for Gov2: `rTrie` uses 1.166–1.329 times the space of **PEF**, the former being 1.549–2.442 times faster. `rTrie` uses 0.481–0.548 times the space of **Roaring**, the former being up to 1.415 times faster. Finally, `rTrie` uses 0.837–0.954 times the space of **RUP**, the former being up to 1.428 times faster.

Results for ClueWeb09: `rTrie` uses 1.188–1.361 times the space of **PEF**, the former being 2.117–3.316 times faster. Also, `rTrie` uses 0.551–0.631 times the space of **Roaring**, the former being 1.221–1.913 times faster. Finally, `rTrie` uses 0.823–0.943 times the space of **RUP**, the former being 1.391–2.178 times faster.

Results for CC-News: for this dataset, the resulting inverted lists have considerably less runs, hence the space usage of `trie` and `rTrie` are about the same. However, `trie` is faster than `rTrie`, as the code to handle runs introduces an overhead that does not pay

¹ <https://github.com/pisa-engine/pisa>

■ **Table 2** Average intersection time (milliseconds per query) and space usage (in bits per integer) for all alternatives tested.

Data Structure	Gov2		ClueWeb09		CC-News	
	Space	Time	Space	Time	Space	Time
IPC	3.34	8.66	5.15	30.18	5.87	68.98
Simple16	4.65	2.44	6.72	8.66	6.88	19.74
OptPFD	4.07	2.15	6.28	7.79	6.50	11.80
PEF Opt	3.62	1.88	5.85	6.50	5.80	17.33
VarintGB	10.80	1.43	11.40	7.34	11.04	12.38
Varint-G8IU	9.97	1.38	10.55	5.25	10.24	12.09
SIMD-BP128	6.07	1.29	8.98	4.47	7.36	15.90
Roaring	8.77	1.09	12.62	3.75	9.86	5.56
RUP	5.04	1.10	8.44	4.27	8.41	5.44
trie (v5)	5.18	1.21	7.46	2.81	8.77	8.72
trie (IL)	5.41	1.06	7.83	2.42	9.30	7.46
trie (v)	5.85	0.77	8.50	1.64	9.99	5.21
rTrie (v5)	4.22	1.22	6.95	3.07	8.73	9.74
rTrie (IL)	4.42	1.10	7.31	2.62	9.16	8.13
rTrie (v)	4.81	0.77	7.96	1.96	9.95	6.09

off in this case. So, we will use `trie` to compare here. It uses 1.512–1.722 times the space of `PEF`, the former being 1.987–3.326 times faster. `trie` uses 0.889–1.013 times the space of `Roaring`, the former being up to 1.067 times faster. Finally, `trie` uses 1.043–1.188 times the space of `RUP`, the former being up to 1.044 times faster.

We can conclude that in all tested datasets, at least one of our trade-offs is the fastest and competitive in space usage, outperforming the highly-engineered ultra-efficient set compression techniques we tested.

8 Conclusions

Trie partition certificates, the main concept we introduced as an alternative to existing certificates by Demaine et al. [20] and Barbay and Kenyon [12], allowed us to introduce our main contributions. In particular, we were able to prove that Trabb-Pardo’s intersection algorithm [50] works in $O(k\delta \lg(u/\delta))$ time, where δ is the alternation measure of the query instance [12]. Thus, Trabb-Pardo’s intersection algorithm was likely the first adaptive intersection algorithm that ever existed, appearing about 22 years before Demaine et al.’s adaptive approach. The lack of analysis on this algorithm (the original author only analyzed his algorithm in the average case) might explain the lack of consideration regarding this algorithm, in particular in practice. Motivated by this result, we introduced compressed representations of integer sets preserving the running time of Trabb-Pardo’s algorithm, and even improving it. Summarizing, our proposals: (1) use compressed space usage, (2) have adaptive intersection computation time, and (3) have highly competitive practical performance.

Multiple avenues for future research are open now. For instance, novel data structures supporting operation `rank1` have emerged recently [32]. These offer interesting trade-offs, using less space than then ones we used, with competitive operation times. Another interesting

line is that of alternative binary trie compact representations. E.g., a DFS representation [13] (rather than BFS, as the one used in this paper), which would potentially reduce the number of cache misses when traversing the tries. Finally, our representation would support dynamic sets (where insertion and deletion of elements are allowed) if we use dynamic binary tries [6].

References

- 1 The Lemur Project. <https://lemurproject.org/>. Accessed March 14, 2023.
- 2 Roaring bitmaps. <https://github.com/RoaringBitmap/CRoaring>. Accessed March 14, 2023.
- 3 Roaring bitmaps: A better compressed bitset. <https://roaringbitmap.org/>. Accessed March 14, 2023.
- 4 A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- 5 V. Ngoc Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.
- 6 D. Arroyuelo, P. Davoodi, and S. Rao Satti. Succinct dynamic cardinal trees. *Algorithmica*, 74(2):742–777, 2016.
- 7 D. Arroyuelo, J. Fuentes-Sepúlveda, and D. Seco. Three success stories about compact data structures. *Communications of the ACM*, 63(11):64–65, 2020.
- 8 D. Arroyuelo and R. Raman. Adaptive succinctness. *Algorithmica*, 84(3):694–718, 2022.
- 9 R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 3109, pages 400–408. Springer, 2004.
- 10 R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th International Conference on String Processing and Information Retrieval (SPIRE)*, LNCS 3772, pages 13–24. Springer, 2005.
- 11 J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–399. ACM/SIAM, 2002.
- 12 J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Transactions on Algorithms*, 4(1):4:1–4:18, 2008. doi:10.1145/1328911.1328915.
- 13 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005. doi:10.1007/s00453-004-1146-6.
- 14 P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *Proc. 18th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 4835, pages 739–750. Springer, 2007.
- 15 S. Büttcher, C. Clarke, and G. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.
- 16 D. Clark. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997.
- 17 C. Clarke, F. Scholer, and I. Soboroff. TREC terabyte track. <https://www-nlpir.nist.gov/projects/terabyte/>. Accessed March 14, 2023.
- 18 H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010. doi:10.1016/j.tcs.2010.06.002.
- 19 J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. 2nd ACM International Conference on Web Search and Data Mining (WSDM’09)*, pages 1–1, 2009.
- 20 E. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752. ACM/SIAM, 2000.
- 21 B. Ding and A. König. Fast set intersection in memory. *Proc. VLDB Endowment*, 4(4):255–266, 2011. doi:10.14778/1938545.1938550.

- 22 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
- 23 R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Pearson, 2011.
- 24 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.
- 25 A. S. Fraenkel and S. T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55, 1996. doi:10.1016/0166-218X(93)00116-H.
- 26 T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012.
- 27 S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.
- 28 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM/SIAM, 2003.
- 29 A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.
- 30 G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.
- 31 S. T. Klein and D. Shapira. Searching in compressed dictionaries. In *Proc. Data Compression Conference (DCC)*, page 142. IEEE Computer Society, 2002.
- 32 F. Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 13617, pages 257–272. Springer, 2022.
- 33 R. M. Layer and A. R. Quinlan. A parallel algorithm for n-way interval set intersection. *Proc. IEEE*, 105(3):542–551, 2017. doi:10.1109/JPROC.2015.2461494.
- 34 D. Lemire. Document identifier data set. <https://lemire.me/data/integercompression2014.html>. Accessed March 14, 2023.
- 35 D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.
- 36 D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O’Hara, F. Saint-Jacques, and G. Ssi Yan Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice & Experience*, 48(4):867–895, 2018. doi:10.1002/spe.2560.
- 37 J. Lin, J. Mackenzie, C. Kamphuis, C. Macdonald, A. Mallia, M. Siedlaczek, A. Trotman, and A. de Vries. Supporting interoperability between open-source search engines with the common index file format, 2020. doi:10.48550/ARXIV.2003.08276.
- 38 J. M. Mackenzie, R. Benham, M. Petri, J. R. Trippas, J. S. Culpepper, and A. Moffat. CC-News-En: A large english news corpus. In *Proc. 29th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 3077–3084. ACM, 2020.
- 39 A. Mallia, M. Siedlaczek, J. Mackenzie, and T. Suel. PISA: performant indexes and search for academia. In *Proc. of the Open-Source IR Replicability Challenge co-located with 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–56, 2019.
- 40 A. Moffat and L. Stuiiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.
- 41 G. Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.
- 42 D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

- 43 G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proc. of 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 273–282, 2014.
- 44 G. E. Pibiri. Sliced indices. https://github.com/jermp/s_indexes. Accessed March 14, 2023.
- 45 G. E. Pibiri. Fast and compact set intersection through recursive universe partitioning. In *Proc. Data Compression Conference (DCC)*, pages 293–302. IEEE, 2021.
- 46 G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):125:1–125:36, 2021. doi:10.1145/3415148.
- 47 R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007.
- 48 F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of 29th European Conference on IR Research (ECIR)*, LNCS 4425, pages 101–112. Springer, 2007.
- 49 A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM'11)*, pages 317–326, 2011.
- 50 L. Trabb-Pardo. *Set Representation and Set Intersection*. PhD thesis, STAN-CS-78-681, Department of Computer Science, Stanford University, 1978. D. E. Knuth, advisor.
- 51 T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106. OpenProceedings.org, 2014.
- 52 I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd Edition*. Morgan Kaufmann, 1999.
- 53 H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web (WWW)*, pages 401–410, 2009.
- 54 J. Zhang, X. Long, , and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. 17th International Conference on World Wide Web (WWW)*, pages 387–396, 2008.
- 55 J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006. doi:10.1145/1132956.1132959.

A Plots of Experimental Results

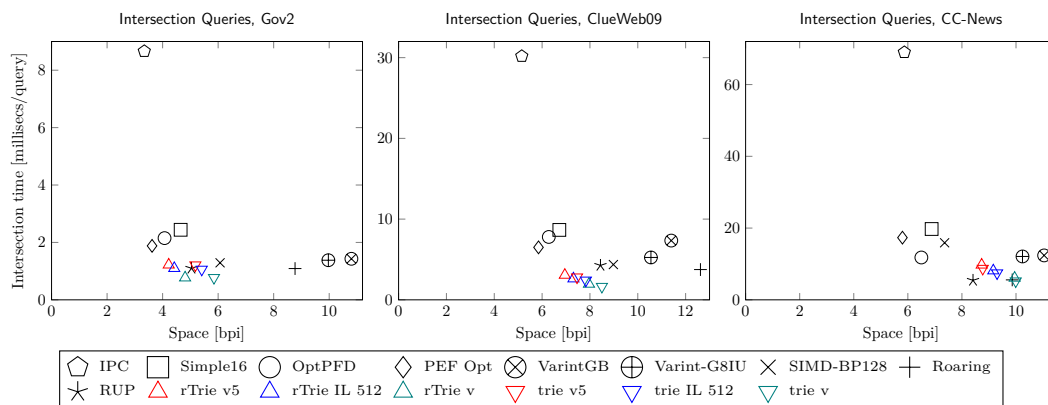


Figure 7 Space vs. time trade-off for all alternative tested on the 3 datasets.

Approximation Algorithms for the Longest Run Subsequence Problem

Yuichi Asahiro ✉ 


Kyushu Sangyo University, Fukuoka, Japan

Mingyang Gong ✉

University of Alberta, Edmonton, Canada

Guohui Lin ✉ 

University of Alberta, Edmonton, Canada

Hiroataka Ono ✉ 


Nagoya University, Nagoya, Japan

Hiroshi Eto ✉ 

Kyushu Institute of Technology, Iizuka, Japan

Jesper Jansson ✉ 

Kyoto University, Kyoto, Japan

Eiji Miyano ✉ 

Kyushu Institute of Technology, Iizuka, Japan

Shunichi Tanaka ✉

Kyushu Institute of Technology, Iizuka, Japan

Abstract

We study the approximability of the LONGEST RUN SUBSEQUENCE PROBLEM (LRS for short). For a string $S = s_1 \cdots s_n$ over an alphabet Σ , a *run of a symbol* $\sigma \in \Sigma$ in S is a maximal substring of consecutive occurrences of σ . A *run subsequence* S' of S is a sequence in which every symbol $\sigma \in \Sigma$ occurs in at most one run. Given a string S , the goal of LRS is to find a longest run subsequence S^* of S such that the length $|S^*|$ is maximized over all the run subsequences of S . It is known that LRS is APX-hard even if each symbol has at most two occurrences in the input string, and that LRS admits a polynomial-time k -approximation algorithm if the number of occurrences of every symbol in the input string is bounded by k . In this paper, we design a polynomial-time $\frac{k+1}{2}$ -approximation algorithm for LRS under the k -occurrence constraint on input strings. For the case $k = 2$, we further improve the approximation ratio from $\frac{3}{2}$ to $\frac{4}{3}$.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms

Keywords and phrases Longest run subsequence problem, bounded occurrence, approximation algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.2

Funding This work is partially supported by NSERC Canada, and JSPS KAKENHI Grant Numbers JP17K00024, JP20H05967, JP21K11755, JP21K19765, JP22H00513, JP22H03550, and JP22K11915.

Acknowledgements The authors would like to thank the anonymous reviewers for their suggestions and detailed comments that helped to improve the presentation of the paper.

1 Introduction

The main goal of genome analysis is to study and compare genetic content among organisms, and thus genome sequencing to determine the complete sequence of a genome is one of its most important stages. Since the first whole genome was obtained [10], genome sequencing technologies have significantly improved. Almost all the current DNA sequencing technologies are based on the following process: First, tens or hundreds of millions of fragments from random positions on the DNA sequence are read via shotgun sequencing. Second, these randomly extracted fragments, called reads, are merged to form a set of contiguous sequences, called contigs, by using an assembly algorithm. Then, the contigs are ordered correctly in a phase called *scaffolding*. One commonly used approach for scaffolding is to rearrange contigs by comparing two or more incomplete assemblies of related samples (see, for example, [8]).



© Yuichi Asahiro, Hiroshi Eto, Mingyang Gong, Jesper Jansson, Guohui Lin, Eiji Miyano, Hiroataka Ono, and Shunichi Tanaka;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 2; pp. 2:1–2:12

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the context of the scaffolding phase of genome assembly, the ONE-SIDED SCAFFOLD FILLING PROBLEM [9], TWO-SIDED SCAFFOLD FILLING PROBLEM [7], ONE-SIDE-FILLED LONGEST COMMON SUBSEQUENCE PROBLEM [3], and TWO-SIDE-FILLED LONGEST COMMON SUBSEQUENCE PROBLEM [4] were formulated as combinatorial optimization problems on two strings. For those problems, their computational complexities were proved, and then fixed-parameter tractable algorithms, approximation algorithms, and exponential-time exact algorithms were proposed in [2, 3, 4, 7]. Very recently, as a different formulation of the scaffolding phase, Schrinner et al. [11, 12] introduced the LONGEST RUN SUBSEQUENCE PROBLEM (LRS for short), defined as follows: For a string $S = s_1 \cdots s_n$ over an alphabet Σ , a *run of a symbol* $\sigma \in \Sigma$ in S is a maximal substring of consecutive occurrences of σ . A *run subsequence* S' of S is a sequence in which every symbol $\sigma \in \Sigma$ occurs in at most one run. Given a string S , the goal of LRS is to find a longest run subsequence S^* of S such that the length $|S^*|$ is maximized over all the run subsequences of S .

► **Example 1.** Consider the string $S = abacacbbab$ over the alphabet $\Sigma = \{a, b, c\}$. It contains (i) four runs of symbol a , i.e., a in the first position, a in the third position, a in the fifth position, and a in the ninth position, (ii) three runs of symbol b , i.e., b in the second position, bb in the seventh and eighth positions, and b in the tenth position, and (iii) two runs of c , i.e., c in the fourth position, and c in the sixth position in S . The numbers of occurrences of a , b , and c are four, four, and two, respectively.

An optimal solution to LRS on input S is $S^* = aacbbb$. For example, the leftmost run aa of length two in S^* is obtained from the leftmost substring aba in S by deleting the second character b . One sees that S^* is a run subsequence, i.e., S^* contains (at most) one run for every symbol. The length of S^* is seven. Note that $S' = aaacbbb$ is another optimal solution since $|S'|$ is also seven. ┘

Schrinner et al. [12] showed that LRS is NP-hard. Subsequently, Dondi and Sikora [5] showed that LRS is APX-hard even if each symbol has at most two occurrences in the input string, and that LRS admits a polynomial-time $\min\{|\Sigma|, k\}$ -approximation algorithm if the number of occurrences of every symbol in the input string is bounded by k .

In this paper, we propose the following improved approximation algorithms for LRS:

- We first design a polynomial-time $\frac{k+1}{2}$ -approximation algorithm for LRS, when the number of occurrences of every symbol is at most k .
- For the case $k = 2$, we further improve the approximation ratio from $\frac{3}{2}$ to $\frac{4}{3}$.

Related work. The fixed-parameter tractability and the parameterized complexity of LRS have been previously investigated [5, 12]: Schrinner et al. [12] showed that there is an $O(|\Sigma| \cdot |S| \cdot 2^{|\Sigma|})$ -time algorithm, given a string S over an alphabet Σ as input of LRS, i.e., LRS is fixed-parameter tractable when parameterized by the size $|\Sigma|$ of the alphabet on which the input string is defined. Dondi and Sikora [5] showed that LRS can be solved by a randomized algorithm in $O(2^r \cdot r \cdot |S|^3)$ time and polynomial space, where r is the number of different runs in a solution, and thus $r \leq |S|$. They also proved that LRS admits a polynomial kernel when parameterized by the length of the solution, but that it does not admit a polynomial kernel when parameterized by the size $|\Sigma|$ of the alphabet or by the number r of runs.

2 Preliminaries

Let Σ be a finite alphabet of symbols. A *string* $S = s_1 \cdots s_n$ is a sequence of n characters, each of which is a symbol in Σ . Two or more characters in S can be the same symbol in Σ . For a string $S = s_1 \cdots s_n$, $|S|$ denotes the length of S , i.e., $|S| = n$. A *subsequence* of S is a

sequence $s_{i_1} \cdots s_{i_m}$, such that $1 \leq i_1 < i_2 < \cdots < i_m \leq |S|$. Let $S[i]$ denote the character of S in the i th position for $1 \leq i \leq |S|$, and $S[i, j]$ denote the substring of S that starts from the i th position and ends at the j th position. For a symbol σ , we denote by σ^k a string that is the concatenation of k occurrences of symbol σ for some integer $k \geq 1$. A *run* in S is a substring $S[i, j]$ such that: (1) $S[i] = S[i+1] = \cdots = S[j]$; (2) $S[i-1] \neq S[i]$ if $i > 1$; and (3) $S[j+1] \neq S[j]$ if $j < |S|$. For any $\sigma \in \Sigma$, a run in S of the form σ^k is called a *length- k σ -run* in S . Observe that if $S[i, j]$ is a σ -run, then it has length $j - i + 1$. Given a string S on alphabet Σ , a *run subsequence* S' of S is a subsequence in which every symbol $\sigma \in \Sigma$ occurs in at most one run.

Let $occ(\sigma)$ be the number of occurrences of σ in the input string S . Let $occ_{max}(S) = \max_{\sigma \in \Sigma} occ(\sigma)$. For example, consider a string $S = abacaabbab$. Then, S includes four a -runs, a , a , a^2 , and a , three b -runs, b , b^2 , and b , and one length-1 c -run. The number $occ(a)$ of occurrences of a is five. Also, $occ(b) = 4$ and $occ(c) = 1$. Therefore, $occ_{max}(S) = 5$.

Our problem LRS can be formulated as follows:

► **Problem 2** (LONGEST RUN SUBSEQUENCE PROBLEM, LRS). *Given an alphabet Σ and a string $S = s_1 \cdots s_n$ with $s_i \in \Sigma$, the goal of LRS is to find a longest run subsequence S^* of S , i.e., every $\sigma \in \Sigma$ occurs in at most one run in S^* and the length $|S^*|$ is maximized over all the run subsequences of S .*

Schrinner et al. [12] show that LRS is NP-hard by giving a polynomial-time reduction from the LINEAR ORDERING PROBLEM, which is shown to be NP-hard in [6]. In this paper we consider the following restricted LRS:

► **Problem 3** (k -LONGEST RUN SUBSEQUENCE PROBLEM, k -LRS). *If the maximum number $occ_{max}(S)$ of occurrences of symbols in the input S is bounded by k , then the problem is called the k -Longest Run Subsequence problem, k -LRS.*

One sees that 1-LRS is trivial since the length of all the runs in the input string S is one, and thus the input S itself is the optimal run subsequence. Dondi and Sikora [5] show that 2-LRS remains hard even from the approximation point of view; they give an L-reduction from the MINIMUM INDEPENDENT SET ON CUBIC GRAPH PROBLEM, which is shown to be APX-hard in [1]:

► **Proposition 4** ([5]). *2-LRS is APX-hard.*

Suppose that an input string of k -LRS is S over an alphabet Σ . Also, without loss of generality, we assume here that every symbol in Σ appears at least once, and the maximum number $occ_{max}(S)$ of occurrences of symbols in S is k . Note that the length of an optimal run subsequence is bounded by $k|\Sigma|$. Consider the following two simple algorithms, (i) and (ii):

(i) Arbitrarily select one run of every symbol $\sigma \in \Sigma$ in S , and construct a run subsequence S' by concatenating all the selected runs.

One sees that $|S'|$ is at least $|\Sigma|$. Therefore, we can conclude that k -LRS is k -approximable.

(ii) Find a symbol, say, σ of the maximum occurrences k , and construct another run subsequence $S'' = \sigma^k$.

Then, we can conclude that k -LRS is $|\Sigma|$ -approximable. By using those two algorithms, we obtain the following proposition:

► **Proposition 5** ([5]). *There is a $\min(|\Sigma|, k)$ -approximation algorithm for k -LRS.*

Since $\min(|\Sigma|, k) \leq \sqrt{|S|}$, the above proposition implies the following:

► **Corollary 6** ([5]). *The general LRS problem admits a $\sqrt{|S|}$ -approximation algorithm.*

3 A polynomial-time $\frac{k+1}{2}$ -approximation algorithm for k -LRS

In this section, we improve the approximation ratio for k -LRS from k to $\frac{k+1}{2}$. Our approximation algorithm **ALG** uses a very natural idea:

Algorithm ALG. Given an input string S over an alphabet Σ , **ALG** selects a longest σ -run in S for each $\sigma \in \Sigma$, and outputs the concatenation of all the selected longest runs.

► **Example 7.** Consider the input string $S = abacaabbab$ (for 5-LRS). The longest a -run, b -run, c -run are aa in the fifth and sixth positions, bb in the seventh and eighth positions, and c in the fourth position. Therefore, the output of **ALG** is $ALG = caabb$. ◻

We now prove that the above simple algorithm achieves the claimed approximability bound:

► **Theorem 8.** *ALG is a polynomial-time $\frac{k+1}{2}$ -approximation algorithm for k -LRS.*

Proof. Clearly, **ALG** returns a valid solution since one run is selected for every symbol in S , and runs in polynomial time. We bound its approximation ratio in the following. Let S be an input string of k -LRS. We assume that S consists of m symbols, i.e., $|\Sigma| = m$, and $occ_{max}(S) = k$. Then, suppose that OPT and ALG are solutions obtained by an optimal algorithm and our algorithm **ALG**, respectively, for the input S . We consider the following two cases: (**Case 1**) The length of every run in S is one, and (**Case 2**) the length of some run in S is at least two.

(**Case 1**). Suppose that the length of every run in S is one. Let m_ℓ be the number of symbols in OPT such that the length of the run of those symbols is exactly ℓ ($\leq k$).

First, the following two equalities hold:

$$|OPT| = \sum_{i=1}^k i \cdot m_i; \text{ and} \quad (1)$$

$$|ALG| = m. \quad (2)$$

Let D be the number of characters deleted from S by the optimal algorithm. Since $|\Sigma| = \sum_{i=0}^k m_i = m$ and $occ_{max}(S) = k$, the following is satisfied:

$$|OPT| = |S| - D \leq km - D. \quad (3)$$

We now derive a lower bound on D . Suppose that a symbol σ_2 in S appears exactly twice in the optimal solution OPT , i.e., OPT contains the length-2 σ_2 -run $\sigma_2\sigma_2$. Recall that the length of all the runs in the input string S is one. Namely, there is at least one different character, say, σ' between two σ_2 's in S . That is, σ' must be deleted from S in order to obtain the length-2 σ_2 -run. Since OPT contains m_2 symbols such that the length of the runs of those symbols is exactly two, the total number of deleted characters from S to obtain the length-2 runs is at least m_2 . It is important to note that the character-deletion to obtain each run is independently carried out, and therefore the number of deleted characters is not doubly counted. Similarly, the total number of deleted characters from S to obtain the length- ℓ runs is at least $(\ell - 1)m_\ell$ for each $3 \leq \ell \leq k$. As a result, we obtain the following lower bound on D :

$$D \geq m_2 + 2m_3 + \cdots + (k - 1)m_k = \sum_{i=2}^k (i - 1)m_i = \sum_{i=1}^k (i - 1)m_i. \quad (4)$$

From Eq.(3) and Eq.(4), the following inequality holds:

$$|OPT| \leq km - \sum_{i=1}^k (i-1)m_i.$$

From Eq.(1), this can be rewritten as:

$$|OPT| \leq (k+1)m - |OPT|,$$

and then rearranged to give:

$$|OPT| \leq \frac{(k+1)m}{2}.$$

From Eq.(2), we obtain the following approximation ratio:

$$\frac{|OPT|}{|ALG|} \leq \frac{k+1}{2}.$$

(Case 2). Suppose that the length of a σ -run in S is at least two and S consists of symbols in Σ . For every such symbol $\sigma \in \Sigma$, we consider a different symbol $\bar{\sigma}$, called a *dummy* symbol. Then, we insert $\bar{\sigma}$ between every consecutive two symbols $\sigma\sigma$ in S so that the two σ 's are not consecutive. Hence we obtain a longer sequence S_d such that the length of all the runs in S_d is one. For example, consider a string

$$S = abacaabbbab.$$

Then, we insert a dummy \bar{a} between the fifth and the sixth positions, a dummy \bar{b} between the seventh and the eighth positions, and the other dummy \bar{b} between the eighth and the ninth positions as follows:

$$S_d = abaca\bar{a}ab\bar{b}\bar{b}bbab.$$

Note that the number of occurrences of each dummy $\bar{\sigma}$ is at most $k-1$ since the maximum number $occ_{max}(S)$ of occurrences of (original) symbols in S is bounded by k . Suppose that OPT_d and ALG_d are solutions obtained by an optimal algorithm and our algorithm ALG , respectively, for the input S_d . One sees that the maximum number $occ_{max}(S_d)$ of occurrences of symbols in S_d is also bounded by k . Therefore, from the arguments in **(Case 1)**, the following inequality is satisfied:

$$\frac{|OPT_d|}{|ALG_d|} \leq \frac{k+1}{2}. \quad (5)$$

The original input S is a subsequence of S_d . Hence, the following clearly holds:

$$|OPT| \leq |OPT_d|. \quad (6)$$

Now consider ALG and ALG_d . (i) For each symbol σ such that the length of all the σ -runs is one, its dummy $\bar{\sigma}$ is not inserted into S_d . Hence, ALG and ALG_d contain one σ , but, of course, neither contains any $\bar{\sigma}$. (ii) If the maximum length of a σ -run in S is (at least) two for some symbol σ , then ALG contains (at least) two σ 's. On the other hand, ALG_d contains one σ and one dummy $\bar{\sigma}$ instead. From (i) and (ii), we have:

$$|ALG| \geq |ALG_d|. \quad (7)$$

From the three inequalities (5), (6), and (7), the following approximation ratio is obtained again:

$$\frac{|OPT|}{|ALG|} \leq \frac{|OPT_d|}{|ALG_d|} \leq \frac{k+1}{2}.$$

For both cases (**Case 1**) and (**Case 2**), the approximation ratio of ALG is bounded above by $\frac{k+1}{2}$. ◀

► **Remark 9.** To see that the approximation analysis above is tight, consider the following string S , where $|S| = n = 2k\ell$, and $\sigma_i \neq \sigma_j$ for $i \neq j$.

$$S = \overbrace{\sigma_1\sigma_2\sigma_1\sigma_2 \cdots \sigma_1\sigma_2}^{2k} \overbrace{\sigma_3\sigma_4\sigma_3\sigma_4 \cdots \sigma_3\sigma_4}^{2k} \cdots \overbrace{\sigma_{2\ell-1}\sigma_{2\ell}\sigma_{2\ell-1}\sigma_{2\ell} \cdots \sigma_{2\ell-1}\sigma_{2\ell}}^{2k}.$$

Namely, the length- $2k$ prefix string contains k σ_1 's and k σ_2 's alternatively. The next string of length $2k$ contains k σ_3 's and k σ_4 's alternatively, and so on. Then, we can obtain the following run subsequence S' by deleting $k-1$ σ_2 's from the first length- $2k$ prefix string, $k-1$ σ_4 's from the next string of length $2k$, and so on:

$$S' = \sigma_1^k \sigma_2 \sigma_3^k \sigma_4 \cdots \sigma_{2\ell-1}^k \sigma_{2\ell}.$$

Hence, the length of OPT is at least $|S'| = (k+1)\ell$. On the other hand, the solution ALG of our algorithm ALG for S contains one of the k σ_i 's for each $1 \leq i \leq 2\ell$:

$$ALG = \sigma_1 \sigma_2 \cdots \sigma_{2\ell}.$$

The length of ALG is 2ℓ . As a result,

$$\frac{|OPT|}{|ALG|} \geq \frac{k+1}{2}.$$

This shows that the analysis of the approximation ratio in the proof of Theorem 8 is tight. ◻

Recall that we can always return a run subsequence of length k as shown in the previous section, and k -LRS is $|\Sigma|$ -approximable. Therefore, we obtain the following corollary:

► **Corollary 10.** *There is a polynomial-time $\min\{|\Sigma|, \frac{k+1}{2}\}$ -approximation algorithm for k -LRS.*

4 A polynomial-time $\frac{4}{3}$ -approximation algorithm for 2-LRS

For 2-LRS, ALG achieves the approximation ratio of $\frac{3}{2}$. In this section we improve the approximation ratio to $\frac{4}{3}$.

As shown in Remark 9, the following string S is a bad example for ALG.

$$S = ababcdcdefef.$$

One sees that from the leftmost substring $S[1, 4] = abab$ of length four (resp. $S[5, 8] = cdcd$ and $S[9, 12] = efef$), we can only obtain a run subsequence of length at most three, i.e., the length of any optimal solution is at most nine. Therefore, one of the possible optimal solution OPT for S is:

$$OPT = aabccdeef.$$

The solution ALG of ALG for S is:

$$ALG = abcdef.$$

Namely, OPT has two a 's (resp. two c 's and two e 's), but ALG has only one a (resp. one c and one e). This observation suggests to us that if there is only one character, say, σ' between two occurrences of a symbol σ , then we should delete σ' and obtain a run $\sigma\sigma$ of length two. This is a basic strategy of our new algorithm ALG_2 .

Before describing details of ALG_2 , we give some definitions which are used in the following. Let S be an input string. Assume that all the symbols in Σ appear in S . We define several subsets of Σ in the following.

- Let $\Sigma_1 = \{\sigma \mid occ(\sigma) = 1, \sigma \in \Sigma\}$ be a set of symbols that appear exactly once in the input string S .
- Let $\Sigma_2 = \{\sigma \mid occ(\sigma) = 2, \sigma \in \Sigma\}$ be a set of symbols that appear exactly twice in the input string S .

Note that $\Sigma = \Sigma_1 \cup \Sigma_2$ in 2-LRS. Now, we consider a symbol $\sigma \in \Sigma_2$ and define several disjoint subsets of Σ_2 . In the following, by *distance* we mean the number of characters between the two occurrences of a symbol.

- If two σ 's consecutively appear in S , then we call σ a distance-0 symbol. Let $\Sigma_{2,0}$ be a subset of all the distance-0 symbols in Σ_2 .
- If there is one character between two σ 's, then we call σ a distance-1 symbol. Let $\Sigma_{2,1}$ be a subset of all the distance-1 symbols in Σ_2 .
- We define $\Sigma_{2,\geq 2} = \Sigma_2 \setminus (\Sigma_{2,0} \cup \Sigma_{2,1})$, i.e., for each $\sigma \in \Sigma_{2,\geq 2}$, σ appears twice in S and there are at least two characters between the two σ 's.

Next, consider a symbol $\gamma \in \Sigma_1$. As a special case, the left and the right symbols of γ can be the same symbol $\gamma' \in \Sigma_{2,1}$, i.e., the input string S possibly contains a substring $\gamma'\gamma\gamma'$ of length 3, called a *special triple*.

- Let Γ_1 be a set of center symbols of special triples. Note that $\Gamma_1 \subseteq \Sigma_1$.
- Let $\Gamma_{2,1}$ be a set of left and right symbols of special triples. Note that $\Gamma_{2,1} \subseteq \Sigma_{2,1}$.

One sees that $|\Gamma_1| = |\Gamma_{2,1}|$.

Finally, consider two symbols σ and σ' in $\Sigma_{2,1} \setminus \Gamma_{2,1}$ in the input string S such that the substring(s) containing σ and σ' can be represented by (i) $S = \cdots\sigma\sigma'\sigma\sigma'\cdots$, or (ii) $S = \cdots\sigma\lambda\sigma\cdots\sigma'\lambda'\sigma'\cdots$, where both λ and λ' are in $\Sigma_{2,\geq 2}$. (i) If S contains $\sigma\sigma'\sigma\sigma'$ as a substring, then we say that a pair of σ and σ' is called a Ψ -pair. Then, σ and σ' belong to a set $\Psi_{2,1}$. (ii) If $\lambda = \lambda'$, then we say that a pair of σ and σ' is a Λ -pair related to λ . Then, σ and σ' belong to a set $\Lambda_{2,1}$ and λ belongs to $\Lambda_{2,\geq 2}$. Note that $|\Lambda_{2,1}| = 2|\Lambda_{2,\geq 2}|$.

Algorithm. The following is a description of our algorithm ALG_2 . During execution of ALG_2 , we determine which characters are included into the run subsequence ALG_2 or not, step by step. Finally, ALG_2 outputs the concatenation of the characters (or the subsequences) included into ALG_2 in each step.

■ Algorithm ALG_2 .

Input An input string S over an alphabet Σ such that every symbol in Σ appears at most twice.

Output A run subsequence.

- Step 1.** Count the number of occurrences of every symbol in Σ , and divide Σ to two subsets Σ_1 and Σ_2 . Then, examine the distance of every symbol in Σ_2 , and obtain $\Sigma_{2,0}$, $\Sigma_{2,1}$, and $\Sigma_{2,\geq 2}$.
- Step 2.** Find all the special triples, all the Ψ -pairs, and all the Λ -pairs.
- Step 3.** For every $\sigma \in \Sigma_{2,0}$, the length-2 σ -run σ^2 is included into ALG_2 .
- Step 4.** For every $\sigma \in \Sigma_{2,1}$, execute the following:
- (i) For every special triple $\gamma'\gamma\gamma'$, the first two characters $\gamma' \in \Gamma_{2,1}$ and $\gamma \in \Gamma_1$ are included into ALG_2 . That is, the third character γ' of that special triple is not included into ALG_2 .
 - (ii) For every Ψ -pair of σ and σ' , i.e., for each string $\sigma\sigma'\sigma'$, its subsequence $\sigma\sigma'\sigma'$ is included into ALG_2 . That is, the third character σ of that string is not included into ALG_2 .
 - (iii) For every Λ -pair related to λ of σ and σ' , i.e., for two strings $\sigma\lambda\sigma$ and $\sigma'\lambda\sigma'$, two subsequences $\sigma\lambda$, and σ'^2 are included into ALG_2 . That is, the third character σ of the former string and the second character λ of the latter string are not included into ALG_2 .
 - (iv) For every $\sigma \in \Sigma_{2,1} \setminus (\Gamma_{2,1} \cup \Psi_{2,1} \cup \Lambda_{2,1})$, σ^2 is included into ALG_2 . That is, the character between the two σ 's is not included into ALG_2 .
- Step 5.** For every $\sigma \in \Sigma_{2,\geq 2} \setminus \Lambda_{2,\geq 2}$, only the first occurrence of σ is included into ALG_2 . That is, if neither of the two occurrences of σ is determined whether or not to be included into ALG_2 , then the first occurrence is included into ALG_2 and the other not into ALG_2 ¹. If only one occurrence remains undetermined, then it is included into ALG_2 .
- Step 6.** Every $\sigma \in \Sigma_1 \setminus \Gamma_1$ is included into ALG_2 .
- Step 7.** Output the concatenation of the characters and the subsequences included into ALG_2 in Step 3 through Step 6 as a run subsequence, and then halt.

► **Remark 11.** Importantly, the output run subsequence of ALG_2 includes at least one occurrence of every symbol in Σ . ┘

► **Example 12.** To clarify the behavior of ALG_2 , we take a look at the following input string of length 20:

$$S = abacdbdecefgfhhiijkjk.$$

One sees that $\Sigma_1 = \{g, i\}$, $\Sigma_{2,0} = \{h\}$, $\Sigma_{2,1} = \{a, d, e, f, j, k\}$, and $\Sigma_{2,\geq 2} = \{b, c\}$. (Step 3) $S[14, 15] = hh$ is included into ALG_2 . (Step 4-(i)) Since $f \in \Sigma_{2,1}$ and $g \in \Sigma_1$, $S[10, 12] = fgf$ is a special triple. Therefore, we select fg from fgf . (Step 4-(ii)) Since there is a substring $S[17, 20] = jkjk$, the pair of j and k is a Ψ -pair, $\Psi_{2,1} = \{j, k\}$. Then, jjk is included into ALG_2 . (Step 4-(iii)) S contains $S[1, 3] = aba$ and $S[5, 7] = dbd$ and thus the pair of a and d is a Λ -pair related to b ; $\Lambda_{2,1} = \{a, d\}$ and $\Lambda_{2,\geq 2} = \{b\}$. Hence, ab and dd are included into ALG_2 . (Step 4-(iv)) From $S[8, 10] = ece$, we obtain a run e^2 of length two, and $S[9] = c$ is not included into ALG_2 . (Step 5) The fourth character c is included into ALG_2 since $c \in \Sigma_{2,\geq 2} \setminus \Gamma_{2,\geq 2}$ and $S[9] = c$ is not included into ALG_2 in Step 4-(iv). (Step 6) The

¹ Alternatively, we can choose any one of the two occurrences of each symbol, to obtain the same approximation ratio.

16th character i is included into ALG_2 since $i \in \Sigma_1 \setminus \Gamma_1$. (Step 7) Finally, the following concatenation of the characters and the subsequences obtained in Step 3 through Step 6 is output as the run subsequence ALG_2 of length 15:

$$ALG_2 = abcddeefghhijkk.$$

□

► **Theorem 13.** ALG_2 is a polynomial-time $\frac{4}{3}$ -approximation algorithm for 2-LRS.

Proof. Clearly, ALG_2 returns a valid solution and runs in polynomial time. We bound its approximation ratio in the following. Suppose that OPT and ALG_2 are run subsequences obtained by an optimal algorithm and our algorithm ALG_2 , respectively, for the input string S .

We assume that the optimal run subsequence OPT consists of the following symbols (OPT1 through OPT4) or characters in special triples (OPT5):

- (OPT1) Consider symbols in $\Sigma_{2,\geq 2}$. Suppose that there are $m_{2,\geq 2,2}$ symbols such that two occurrences of each of them are included into OPT by deleting all the characters between two occurrences. Also, suppose that there are $m_{2,\geq 2,1}$ (resp. $m_{2,\geq 2,0}$) symbols such that one occurrence (resp. no occurrence) of each of them is included into OPT .
- (OPT2) Consider symbols in $\Sigma_{2,1} \setminus \Gamma_{2,1}$. Suppose that there are $m_{2,1,2}$ symbols such that two occurrences of each of them are included into OPT by deleting one character between two occurrences. Also, suppose that there are $m_{2,1,1}$ (resp. $m_{2,1,0}$) symbols such that one occurrence (resp. no occurrence) of each of them is included into OPT .
- (OPT3) Consider symbols in $\Sigma_{2,0}$. Suppose that there are $m_{2,0,2}$ (resp. $m_{2,0,0}$) symbols such that two occurrences (resp. no occurrence) of each of them are included into OPT . Remark that since the goal is to maximize the length of the run subsequence, we can assume that two occurrences (one run of length two) of the symbol in $\Sigma_{2,0}$ are completely included into OPT , or completely deleted.
- (OPT4) Consider symbols in $\Sigma_1 \setminus \Gamma_1$. Suppose that there are $m_{1,1}$ (resp. $m_{1,0}$) symbols such that one occurrence (resp. no occurrence) of each of them is included into OPT .
- (OPT5) Consider special triples. For example, take a look at $\gamma'\gamma\gamma'$ where $\gamma \in \Gamma_1$ and $\gamma' \in \Gamma_{2,1}$. One sees that we cannot select all the three characters into any solution subsequence since it can contain at most one run for every symbol. Therefore, OPT includes at most two characters of the special triple, γ'^2 , $\gamma'\gamma$, or $\gamma\gamma'$. Since the goal is to maximize the length of the run subsequence, we can assume that OPT includes one of the two characters of the special triple, or does not include any character from the special triple. Suppose that there are $m_{\gamma,2}$ (resp. $m_{\gamma,0}$) special triples such that two characters (resp. no character) of each of them are included into OPT .

Then, the length of OPT is calculated as follows:

$$|OPT| = \overbrace{2m_{2,\geq 2,2} + m_{2,\geq 2,1}}^{\text{OPT1}} + \overbrace{2m_{2,1,2} + m_{2,1,1}}^{\text{OPT2}} + \overbrace{2m_{2,0,2}}^{\text{OPT3}} + \overbrace{m_{1,1}}^{\text{OPT4}} + \overbrace{2m_{\gamma,2}}^{\text{OPT5}}. \quad (8)$$

Now, let D be the number of deleted symbols from S by the optimal algorithm. Then, D is counted by the above assumption:

$$D = \overbrace{m_{2,\geq 2,1} + 2m_{2,\geq 2,0}}^{\text{OPT1}} + \overbrace{m_{2,1,1} + 2m_{2,1,0}}^{\text{OPT2}} + \overbrace{2m_{2,0,0}}^{\text{OPT3}} + \overbrace{m_{1,0}}^{\text{OPT4}} + \overbrace{m_{\gamma,2} + 3m_{\gamma,0}}^{\text{OPT5}}. \quad (9)$$

Next, we consider a lower bound on D . As for symbols in $\Sigma_{2,\geq 2}$, we assumed in (OPT1) that there are $m_{2,\geq 2,2}$ symbols such that two occurrences of each of them are included into OPT , i.e., at least two characters between the two occurrences must be deleted. Also, as

for symbols in $\Sigma_{2,1} \setminus \Gamma_{2,1}$, we assumed in (OPT2) that there are $m_{2,1,2}$ symbols such that two occurrences of each of them are included into OPT , i.e., one character between the two occurrences must be deleted. As a result, the following inequality holds:

$$D \geq 2m_{2,\geq 2,2} + m_{2,1,2}. \quad (10)$$

Now, we estimate the length of the output run subsequence of ALG_2 .

- (ALG1) Consider symbols in $\Sigma_{2,0}$. In Step 3, two occurrences of every symbol in $\Sigma_{2,0}$ are included into ALG_2 , i.e., $2m_{2,0,2} + 2m_{2,0,0}$ characters are included into ALG_2 .
- (ALG2) Consider symbols in $\Gamma_{2,1}$. In Step 4-(i), one occurrence of every symbol in $\Gamma_{2,1}$ is included into ALG_2 , i.e., $m_{\gamma,2} + m_{\gamma,0}$ characters are totally included in ALG_2 .
- (ALG3) Consider symbols in Σ_1 . In Step 4-(i), every symbol in $\Gamma_1 (\subseteq \Sigma_1)$ is included into ALG_2 . In Step 6, every symbol in $\Sigma_1 \setminus \Gamma_1$ is included into ALG_2 . That is, all the symbols in Σ_1 are included into ALG_2 . In total, $m_{1,1} + m_{1,0} + m_{\gamma,2} + m_{\gamma,0}$ characters are included into ALG_2 .
- (ALG4) Consider symbols in $\Sigma_{2,\geq 2}$. In Step 4-(iii), one occurrence of every symbol in $\Lambda_{2,\geq 2} (\subseteq \Sigma_{2,\geq 2})$ is included into ALG_2 . Also, in Step 5, one occurrence of every symbol in $\Sigma_{2,\geq 2} \setminus \Lambda_{2,\geq 2}$ is included into ALG_2 . In total, $m_{2,\geq 2,2} + m_{2,\geq 2,1} + m_{2,\geq 2,0}$ characters are included into ALG_2 .
- (ALG5) Consider symbols in $\Sigma_{2,1} \setminus \Gamma_{2,1}$. Recall that $|\Sigma_{2,1} \setminus \Gamma_{2,1}| = m_{2,1,2} + m_{2,1,1} + m_{2,1,0}$. Consider a Ψ -pair of σ and σ' , i.e., a substring $\sigma\sigma'\sigma\sigma'$ of length four in S . In Step 4-(ii), three characters σ , σ' , and σ' are selected from the Ψ -pair of σ and σ' . Namely, we can see that three characters per two symbols are included into ALG_2 . Also, in Step 4-(iii), three characters σ , σ' , and σ' are selected from every Λ -pair of σ and σ' . Again, three characters per two symbols are included into ALG_2 . In Step 4-(iv), two occurrences of every symbol in $(\Sigma_{2,1} \setminus \Gamma_{2,1}) \setminus (\Psi_{2,1} \cup \Lambda_{2,1})$ are included into ALG_2 . As a result, at least $\frac{3}{2}(m_{2,1,2} + m_{2,1,1} + m_{2,1,0})$ characters are included into ALG_2 .

Then, the following inequality on the length of ALG_2 holds:

$$\begin{aligned} |ALG_2| \geq & \overbrace{m_{2,\geq 2,2} + m_{2,\geq 2,1} + m_{2,\geq 2,0}}^{\text{ALG4}} + \overbrace{\frac{3}{2}(m_{2,1,2} + m_{2,1,1} + m_{2,1,0})}^{\text{ALG5}} \\ & + \overbrace{2m_{2,0,2} + 2m_{2,0,0}}^{\text{ALG1}} + \overbrace{m_{1,1} + m_{1,0} + 2m_{\gamma,2} + 2m_{\gamma,0}}^{\text{ALG2 and ALG3}}. \end{aligned} \quad (11)$$

From Eq.(9) and Eq.(10), we obtain the following inequality:

$$\begin{aligned} & \frac{1}{3}(m_{2,\geq 2,1} + 2m_{2,\geq 2,0} - m_{2,1,2} + m_{2,1,1} + 2m_{2,1,0} + 2m_{2,0,0} + m_{1,0} + m_{\gamma,2} + 3m_{\gamma,0}) \\ & \geq \frac{2}{3}m_{2,\geq 2,2}. \end{aligned} \quad (12)$$

Therefore, from Eq.(8) and Eq.(12), $|OPT|$ is bounded as follows:

$$\begin{aligned} |OPT| &= \left(\frac{4}{3}m_{2,\geq 2,2} + \frac{2}{3}m_{2,\geq 2,2} \right) + m_{2,\geq 2,1} + 2m_{2,1,2} + m_{2,1,1} \\ & \quad + 2m_{2,0,2} + m_{1,1} + 2m_{\gamma,2} \\ & \leq \frac{4}{3}m_{2,\geq 2,2} + \frac{4}{3}m_{2,\geq 2,1} + \frac{2}{3}m_{2,\geq 2,0} + \frac{5}{3}m_{2,1,2} + \frac{4}{3}m_{2,1,1} + \frac{2}{3}m_{2,1,0} \\ & \quad + 2m_{2,0,2} + \frac{2}{3}m_{2,0,0} + m_{1,1} + \frac{1}{3}m_{1,0} + \frac{7}{3}m_{\gamma,2} + m_{\gamma,0} \end{aligned} \quad (13)$$

One can verify that the following is satisfied from Eq.(11) and Eq.(13):

$$\frac{|OPT|}{|ALG_2|} \leq \frac{4}{3}. \quad \blacktriangleleft$$

► Remark 14. Again, we can show the tightness for the approximation ratio $\frac{4}{3}$ of ALG_2 . Consider the following string S , where $|S| = n = 6\ell$.

$$S = \sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6\sigma_4\sigma_5\sigma_6 \cdots \sigma_{3\ell-2}\sigma_{3\ell-1}\sigma_{3\ell}\sigma_{3\ell-2}\sigma_{3\ell-1}\sigma_{3\ell}.$$

Then, we can find the following run subsequence S' :

$$S' = \sigma_1^2\sigma_2\sigma_3\sigma_4^2\sigma_5\sigma_6 \cdots \sigma_{3\ell-2}^2\sigma_{3\ell-1}\sigma_{3\ell}$$

Therefore, the length of OPT is at least $|S'| = 4\ell$. On the other hand, the solution of our algorithm ALG_2 for S contains only one of the two σ_i 's for each $1 \leq i \leq 3\ell$ since every symbol is in $\Sigma_{2, \geq 2}$:

$$ALG_2 = \sigma_1\sigma_2 \cdots \sigma_{3\ell}.$$

The length of ALG_2 is 3ℓ . As a result,

$$\frac{|OPT|}{|ALG_2|} \geq \frac{4}{3}.$$

This shows that the above approximation analysis is tight. \blacktriangleright

5 Conclusion

We have presented a polynomial-time $\frac{k+1}{2}$ -approximation algorithm for k -LRS, where the number of occurrences of every symbol in the input string is at most k . Then, for the case $k = 2$, we have reduced the approximation ratio to $\frac{4}{3}$. The current approximation algorithm for 2-LRS is a little bit complicated, and thus might be simplified to obtain the same approximation ratio. Future work is to further improve the approximation ratio of $\frac{4}{3}$ for 2-LRS, and to design an even better approximation algorithm for general k -LRS. It would also be useful to derive tight bounds on the polynomial-time approximation hardness of k -LRS in terms of k .

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 Yuichi Asahiro, Jesper Jansson, Guohui Lin, Eiji Miyano, Hirotaka Ono, and Tadatashi Utashima. Polynomial-time equivalences and refined algorithms for longest common subsequence variants. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPICs*, pages 15:1–15:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.15.
- 3 Mauro Castelli, Riccardo Dondi, Giancarlo Mauri, and Italo Zoppis. The longest filled common subsequence problem. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPICs*, pages 14:1–14:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.14.

- 4 Mauro Castelli, Riccardo Dondi, Giancarlo Mauri, and Italo Zoppis. Comparing incomplete sequences via longest common subsequence. *Theor. Comput. Sci.*, 796:272–285, 2019. doi:10.1016/j.tcs.2019.09.022.
- 5 Riccardo Dondi and Florian Sikora. The longest run subsequence problem: Further complexity results. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPICs*, pages 14:1–14:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.14.
- 6 Martin Grötschel, Michael Jünger, and Gerhard Reinelt. A cutting plane algorithm for the linear ordering problem. *Oper. Res.*, 32(6):1195–1220, 1984. doi:10.1287/opre.32.6.1195.
- 7 Haitao Jiang, Chunfang Zheng, David Sankoff, and Binhai Zhu. Scaffold filling under the breakpoint and related distances. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 9(4):1220–1229, 2012. doi:10.1109/TCBB.2012.57.
- 8 Junwei Luo, Yawei Wei, Mengna Lyu, Zhengjiang Wu, Xiaoyan Liu, Huimin Luo, and Chaokun Yan. A comprehensive review of scaffolding methods in genome assembly. *Briefings Bioinform.*, 22(5), 2021. doi:10.1093/bib/bbab033.
- 9 Adriana Muñoz, Chunfang Zheng, Qian Zhu, Victor A. Albert, Steve Rounsley, and David Sankoff. Scaffold filling, contig fusion and comparative gene order inference. *BMC Bioinform.*, 11:304, 2010. doi:10.1186/1471-2105-11-304.
- 10 F. Sanger, G.M. Air, B.G. Barrell, N.L. Brown, A.R. Coulson, J.C. Fiddes, C.A. Hutchison III, P.M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage ϕ x174 DNA. *Nature*, 265:687–695, 1977.
- 11 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.
- 12 Sven Schrinner, Manish Goel, Michael Wulfert, Philipp Spohr, Korbinian Schneeberger, and Gunnar W. Klau. Using the longest run subsequence problem within homology-based scaffolding. *Algorithms Mol. Biol.*, 16(1):11, 2021. doi:10.1186/s13015-021-00191-8.

Optimal LZ-End Parsing Is Hard

Hideo Bannai ✉ 

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

Mitsuru Funakoshi ✉ 

Department of Informatics, Kyushu University, Fukuoka, Japan
Japan Society for the Promotion of Science, Tokyo, Japan

Kazuhiro Kurita ✉ 

Nagoya University, Japan

Yuto Nakashima ✉ 

Department of Informatics, Kyushu University, Fukuoka, Japan

Kazuhisa Seto ✉ 

Faculty of Information Science and Technology, Hokkaido University, Sapporo, Japan

Takeaki Uno ✉

National Institute of Informatics, Tokyo, Japan

Abstract

LZ-End is a variant of the well-known Lempel-Ziv parsing family such that each phrase of the parsing has a previous occurrence, with the additional constraint that the previous occurrence must end at the end of a previous phrase. LZ-End was initially proposed as a greedy parsing, where each phrase is determined greedily from left to right, as the longest factor that satisfies the above constraint [Kreft & Navarro, 2010]. In this work, we consider an optimal LZ-End parsing that has the minimum number of phrases in such parsings. We show that a decision version of computing the optimal LZ-End parsing is NP-complete by showing a reduction from the vertex cover problem. Moreover, we give a MAX-SAT formulation for the optimal LZ-End parsing adapting an approach for computing various NP-hard repetitiveness measures recently presented by [Bannai et al., 2022]. We also consider the approximation ratio of the size of greedy LZ-End parsing to the size of the optimal LZ-End parsing, and give a lower bound of the ratio which asymptotically approaches 2.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Data Compression, LZ-End, Repetitiveness measures

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.3

Related Version *Previous Version*: <https://arxiv.org/abs/2302.02586>

Funding This work was partially supported by JSPS KAKENHI Grant Numbers JP20H05964 (for YN).

Hideo Bannai: JSPS KAKENHI Grant Number JP20H04141.

Mitsuru Funakoshi: JSPS KAKENHI Grant Number JP20J21147.

Kazuhiro Kurita: JSPS KAKENHI Grant Number JP21K17812, JP22H03549, JP21H05861, and JST ACT-X Grant Number JPMJAX2105.

Yuto Nakashima: JSPS KAKENHI Grant Number JP21K17705, JP23H04386, and JST ACT-X Grant Number JPMJAX200K.

Kazuhisa Seto: JSPS KAKENHI Grant Number JP21H05839.

Acknowledgements We would like to thank Dominik Köppl for discussion. We also gratefully acknowledge the comments of anonymous reviewers for improving the manuscript.



© Hideo Bannai, Mitsuru Funakoshi, Kazuhiro Kurita, Yuto Nakashima, Kazuhisa Seto, and Takeaki Uno;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 3; pp. 3:1–3:11

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the context of lossless data compression, various repetitiveness measures – especially those based on dictionary compression algorithms – and relations between them have recently received much attention (see the excellent survey by Navarro [12, 13]). One of the most fundamental and well-known measures is the LZ77 parsing [15], in which a string is parsed into z phrases such that each phrase is a single symbol, or the longest substring which has a previous occurrence. LZ-End [9, 10] is a variant of LZ77 parsing with the added constraint that a previous occurrence of the phrase must end at the end of a previous phrase. More formally, the LZ-End parsing is a factorization q_1, \dots, q_{z_e} of a given string that can be greedily obtained from left to right: each phrase q_i is either (1) a symbol that is the leftmost occurrence of the symbol or (2) the longest prefix of the remaining suffix $q_i \cdots q_{z_e}$ that is a suffix of $q_1 \cdots q_j$ for some $j < i$. It is known that LZ-End parsing can be computed in linear time [6], and there exists a space-efficient algorithm [5].

While there is no known data structure of $O(z)$ size that provides efficient random access to arbitrary positions in the string, it was recently shown that $\tilde{O}(1)$ time access could be achieved with $O(z_e)$ space [8]. Furthermore, concerning the difference between z and z_e , an upper bound of $z_e = O(z \log^2(n/z))$ was shown [8], where n is the length of the (uncompressed) string. On the other hand, there is an obvious bound of $z_e = \Omega(z \log n)$ for the unary string, since a previous occurrence of an LZ-End phrase cannot be self-referencing, i.e., overlap with itself, while an LZ77 phrase can. Notice that $z \leq z_{no} \leq z_e$ holds for any string, where z_{no} is the number of phrases in the LZ77 parsing that does not allow self-referencing. A family of strings such that the ratio z_e/z_{no} asymptotically approaches 2 (for large alphabet [10], for binary alphabet [4]) is known, and it is conjectured that $z_e \leq 2z_{no}$ holds for any strings [10].

While the phrases in the parsings described above are chosen greedily (i.e., longest), we can consider variants which do not impose such constraint, e.g., in an *LZ-End-like* parsing, each phrase q_i is either (1) a symbol that is the leftmost occurrence of the symbol or (2) a (not necessary longest) prefix of the remaining suffix which is a suffix of $q_1 \cdots q_j$ for some $j < i$. We refer to an LZ-End-like parsing with the smallest number z_{end} of phrases, an *optimal* LZ-End parsing [12], and call the original, the *greedy* LZ-End parsing.¹ Thus $z \leq z_{no} \leq z_{end} \leq z_e$ holds.

Interestingly, $z_{end} \leq g$ holds, where g is the size of the smallest context free grammar that derives (only) the string, while a similar relation between z_e and g does not seem to be known [12].

This brings us to two natural and important questions about the measure z_{end} :

- How efficiently can we compute z_{end} ?
- How much smaller can z_{end} be compared to z_e ?

In this work, we answer a part of the above questions. Namely:

1. We prove the NP-hardness of computing z_{end} .
2. We present an algorithm for exact computation by MAX-SAT.
3. We give a lower bound of the maximum value of the ratio z_e/z_{end} .

In Section 3, we give the hardness result. Our reduction is from the vertex cover problem: finding a minimum set U of vertices such that every edge in a graph is incident to some vertex in U . In Section 4, we show a MAX-SAT formulation for computing the optimal

¹ Notice that we do not need the distinction for LZ77, since the greedy LZ77 parsing is also an optimal LZ77-like parsing.

LZ-End parsing that follows an approach by Bannai et al. that allows computing NP-hard repetitiveness measures using MAX-SAT solvers [1]. In Section 5, we consider the ratio z_e/z_{end} . We give a family of binary strings such that the ratio asymptotically approaches 2. Note that we can easily modify this result to a larger alphabet. Since $(z_e/z_{end}) \leq (z_e/z_{no})$, the bound is tight, assuming that the conjecture by Kreft and Navarro [10] holds.

Related work

The LZ77 and LZ78 are original members of the LZ family [15, 16]. It is well-known that the (greedy) LZ77 parsing produces the optimal version of the parsing [11]. The LZ78 parsing satisfies that each phrase can be represented as a concatenation of a previous phrase and a symbol. The NP-hardness of computing the optimal version of the LZ78 variant was shown [2]. This hardness result is also given by a reduction from the vertex cover problem. However, our construction of the reduction for the LZ-End differs from that for the LZ78 since these parsings have very different structures. The smallest string attractor [7] is one of the most fundamental repetitiveness measures. It is also known that computing the smallest string attractor of a given string is NP-hard [7]. The hardness result was proven by a reduction from the set-cover problem.

2 Preliminaries

2.1 Strings

Let Σ be an *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is the string of length 0. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any strings x and y , $x \cdot y$ denotes the concatenation of two strings. We will sometimes abbreviate “ \cdot ” (i.e., $x \cdot y = xy$). For a string $w = xyz$, x , y and z are called a *prefix*, *substring*, and *suffix* of w , respectively. They are called a *proper prefix*, a *proper substring*, and a *proper suffix* of w if $x \neq w$, $y \neq w$, and $z \neq w$, respectively. The i -th symbol of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j . For convenience, let $w[i..j] = \varepsilon$ when $i > j$. We will sometimes use $w[i..j]$ to denote $w[i..j - 1]$. For any string w , let $w^1 = w$ and let $w^k = ww^{k-1}$ for any integer $k \geq 2$, i.e., w^k is the k -times repetition of w .

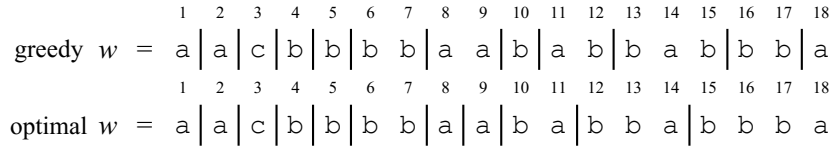
2.2 LZ-End parsing

We give a definition of the LZ-End parsing, which is a variant of the Lempel-Ziv family.

► **Definition 1** ((Greedy) LZ-End parsing). *The LZ-End parsing of a string w is the parsing $LZEnd(w) = q_1, \dots, q_{z_e}$ of w such that q_i is either a symbol that is the leftmost occurrence of the symbol or the longest prefix of $q_i \cdots q_{z_e}$ that occurs as a suffix of $q_1 \cdots q_j$ for some $j < i$, which we call a source of the phrase.*

We refer to each q_i as a *phrase*. This definition, used in [8], is slightly different from the original version [9, 10] where a symbol is added to each phrase. The results in this paper hold for the original version as well (which we will show in the full version of the paper), but here we use this definition for simplicity. In this paper, we consider a more general version of the LZ-End parsing: a parsing $q_1, \dots, q_{z_{end}}$ of a string w such that q_i is a (not necessary longest) suffix of $q_1 \cdots q_j$ for some $j < i$. We call such a parsing with a minimum number z_{end} of phrases an *optimal LZ-End parsing* of w . We give an example of the greedy LZ-End parsing and the optimal LZ-End parsing in Figure 1.

3:4 Optimal LZ-End Parsing Is Hard



■ **Figure 1** Let $w = \text{aacbbbbbbaababbabbba}$. The greedy LZ-End parsing $LZEnd(w)$ of w is illustrated in the upper part of the figure. For the phrase at position 10, a longer substring $w[10..11] = \text{ba}$ has another previous occurrence at position 7, but there is no phrase that ends at position 8, and any longer substring does not have a previous occurrence. Therefore, the phrase starting at position 10 is b . The lower part of the figure shows an optimal LZ-End parsing (which is smaller than the greedy one) on the same string. Each phrase has a previous occurrence that ends at the end of some LZ-End phrase. The size of the greedy parsing is 12 and the size of the optimal parsing is 11.

2.3 Graphs

Let $G = (V, E)$ be a graph with the set of vertices V and the set of edges E . An edge $e = \{u, v\}$ is called an *incident edge of u* . We denote the set of incident edges of v as $\Gamma_G(v)$ and drop the subscript whenever it is clear from context. For an edge $e = \{u, v\}$, vertices u and v are the *end points* of e . For a subset of vertices $U \subseteq V$, U is a *vertex cover* if for any $e \in E$, at least one end point of e is contained in U . Let τ_G be the size of the minimum vertex cover of G (i.e., τ_G denotes the *vertex cover number* of G). Notice that computing τ_G is NP-complete [3].

2.4 Maximum Satisfiability (MAX-SAT) problem

Let $\{x_1, \dots, x_n\}$ be a set of literals and C be a conjunctive normal form (CNF) formula. Each variable in C is assigned a Boolean value (i.e., true or false). The goal of the *Satisfiability (SAT)* problem is to compute an assignment of variables that satisfies all clauses of C . The *Maximum Satisfiability (MAX-SAT)* problem is a variant of SAT, in which there are two types of clauses: hard clauses and soft clauses. A solution for MAX-SAT is a truth assignment of the variables such that all hard clauses are satisfied, and the number of soft clauses that are satisfied is maximized.

3 NP-hardness of computing the optimal LZ-End parsing

In this section, we consider the problem of computing the optimal LZ-End parsing of a given string. A decision version of the problem is given as follows.

► **Problem 2** (Decision version of computing the optimal LZ-End parsing (OptLE)). *Given a string w and an integer k , decide whether there exists an LZ-End parsing of size k or less.*

We show the NP-completeness of OptLE in the following and present an algorithm for exact computation in the next section.

► **Theorem 3.** *OptLE is NP-complete.*

Proof. We give a reduction from the vertex cover problem to OptLE. Let $G = (V, E)$ be a graph with a set of vertices $V = \{v_1, \dots, v_n\}$ and a set of edges $E = \{e_1, \dots, e_m\}$. Suppose that an input graph G of the vertex cover problem is connected and $|\Gamma(v)| \geq 2$ for any

\mathcal{P}	$v_1 v_1 v_1 \# v_1 v_1 \$ \# v_1 \$ \# \# v_1 \$ e_1 \# v_1 \$ e_3 \# e_1 e_1 v_1 \# e_3 e_3 v_1 \# $ $v_2 v_2 v_2 \# v_2 v_2 \$ \# v_2 \$ \# \# v_2 \$ e_1 \# v_2 \$ e_2 \# e_1 e_1 v_2 \# e_2 e_2 v_2 \# $ $v_3 v_3 v_3 \# v_3 v_3 \$ \# v_3 \$ \# \# v_3 \$ e_2 \# v_3 \$ e_3 \# e_2 e_2 v_3 \# e_3 e_3 v_3 \# $	
\mathcal{Q}	$e_1 e_1 e_1 \# e_2 e_2 e_2 \# e_3 e_3 e_3 \# $	
\mathcal{R}	$v_1 v_1 v_1 v_1 \$ e_1 e_1 e_1 v_1 v_1 \$ e_3 e_3 e_3 v_1 v_1 \$ \# \# $ $v_2 v_2 v_2 v_2 \$ e_1 e_1 e_1 v_2 v_2 \$ e_2 e_2 e_2 v_2 v_2 \$ \# \# $ $v_3 v_3 v_3 v_3 \$ e_2 e_2 e_2 v_3 v_3 \$ e_3 e_3 e_3 v_3 v_3 \$ \# \# $	greedy
\mathcal{S}	$\$ e_1 e_1 e_1 \# \$ e_2 e_2 e_2 \# \$ e_3 e_3 e_3 \# $	
\mathcal{R}	$v_1 v_1 v_1 v_1 \$ e_1 e_1 e_1 v_1 v_1 \$ e_3 e_3 e_3 v_1 v_1 \$ \# \# $ $v_2 v_2 v_2 v_2 \$ e_1 e_1 e_1 v_2 v_2 \$ e_2 e_2 e_2 v_2 v_2 \$ \# \# $ $v_3 v_3 v_3 v_3 \$ e_2 e_2 e_2 v_3 v_3 \$ e_3 e_3 e_3 v_3 v_3 \$ \# \# $	optimal
\mathcal{S}	$\$ e_1 e_1 e_1 \# \$ e_2 e_2 e_2 \# \$ e_3 e_3 e_3 \# $	

■ **Figure 2** Let $G = (V, E)$ be the complete graph of three vertices v_1, v_2, v_3 and $e_1 = \{v_1, v_2\}$, $e_2 = \{v_2, v_3\}$, $e_3 = \{v_1, v_3\}$. \mathcal{W}_G and the greedy parsing and an optimal parsing are illustrated in the figure. The first two parts (\mathcal{P} and \mathcal{Q}) share the same parsing. The last two parts (\mathcal{R} and \mathcal{S}) are different. The upper part in the figure shows the greedy parsing and the lower part shows an optimal parsing. For instance, in the optimal parsing, we can choose $\$e_1^3$ and $\$e_3^3$ as phrases by using non-greedy parsing in \mathcal{R}_1 . In other words, we can reduce two phrases in \mathcal{S} -part by adding one phrase in \mathcal{R}_1 . In this example, the optimal parsing represents a vertex cover $\{v_1, v_3\} \subset V$ of G (since \mathcal{R}_2 selects the greedy parsing and the others are not).

$v \in V$. We identify each vertex v_i as a symbol v_i and each edge e_i as a symbol e_i . We also introduce the symbol $\$$, and a set of symbols that occur uniquely in the string. The latter is represented, for simplicity, by the special symbol $\#$, i.e., $\#$ represents a different symbol each time it occurs in our description. We consider the string \mathcal{W}_G defined by graph G as follows.

- $\mathcal{W}_G = \prod_{i=1}^n \mathcal{P}_i \cdot \prod_{j=1}^m \mathcal{Q}_j \cdot \prod_{i=1}^n \mathcal{R}_i \cdot \prod_{j=1}^m \mathcal{S}_j$
- $\mathcal{P}_i = v_i^3 \# v_i^2 \$ \# v_i \$^2 \# \cdot \mathcal{X}_i \cdot \mathcal{Y}_i$
- $\mathcal{Q}_j = e_j^3 \#$
- $\mathcal{R}_i = v_i^4 \$ \prod_{e_j \in \Gamma(v_i)} (e_j^3 v_i^2 \$) \$ \#$
- $\mathcal{S}_j = \$ e_j^3 \#$
- $\mathcal{X}_i = \prod_{e_j \in \Gamma(v_i)} (v_i \$ e_j \#)$
- $\mathcal{Y}_i = \prod_{e_j \in \Gamma(v_i)} (e_j^2 v_i \#)$

An example of this string is illustrated in Figure 2. Note that we use i for representing indices of vertices and j for indices of edges.

Before we show the detail of the reduction, we start with an intuitive description of our reduction. The string \mathcal{W}_G consists of four parts (which are represented by \mathcal{P} , \mathcal{Q} , \mathcal{R} , and \mathcal{S}). (1) The first two parts are non-functional parts. They can only be parsed in a single sensible way. These parts play a role as sources of the third part (\mathcal{R} -part). (2) In the third part, \mathcal{R}_i corresponds to the vertex v_i . Roughly speaking, \mathcal{R}_i can be parsed in two sensible ways such that the parsing represents whether the vertex v_i is in the vertex cover or not. If a vertex v_i is in the vertex cover, then the parsing of \mathcal{R}_i needs one more phrase. (3) In the last part, \mathcal{S}_j corresponds to the edge e_j . \mathcal{S}_j will be parsed into two phrases iff one of the incident vertex of the edge e_j is in the vertex cover. Otherwise, \mathcal{S}_j has three phrases. Overall, minimizing the number of vertices for the vertex cover of a graph G corresponds to minimizing the total penalties in the \mathcal{R} -part such that every \mathcal{S}_j will be parsed into two phrases.

3:6 Optimal LZ-End Parsing Is Hard

We show that the number of phrases of the optimal parsing of \mathcal{W}_G is less than $13n+22m+k$ if and only if the vertex cover number τ_G is less than k .

First, we observe an optimal LZ-End parsing of \mathcal{W}_G . Let us consider a parsing of $\prod_{i=1}^n \mathcal{P}_i$. In this part, the greedy parsing gives $10n + 13m$ phrases. In the greedy parsing of $\prod_{i=1}^n \mathcal{P}_i$, phrases v_i^2 in $v_i^2\$\#, v_i\$\$ in $v_i\$\^2\#, v_i\$\#e_j\#$, and the second occurrence of e_j^2 have length 2, and the other phrases have length 1. It is easy to see that this parsing is a smallest possible parsing of $\prod_{i=1}^n \mathcal{P}_i$. Moreover, other parsings of the same size do not affect the parsing of the rest of the string; candidates for a source cannot be increased by selecting any other parsings since the phrases of length 2 are preceded by unique symbols $\#$. Hence, we can choose this greedy parsing as a part of an optimal parsing.

In the second part $\prod_{j=1}^m \mathcal{Q}_j$, the greedy parsing also gives an optimal parsing which has $3m$ phrases (i.e., each \mathcal{Q}_j is parsed into three phrases since e_j^2 occurs in \mathcal{P}_i for some i and e_j^3 is unique in $\prod_{i=1}^n \mathcal{P}_i \cdot \prod_{j=1}^m \mathcal{Q}_j$). This parsing is also a smallest possible parsing and does not affect any parsings of the rest of the string.

The remaining suffix $\prod_{i=1}^n \mathcal{R}_i \cdot \prod_{j=1}^m \mathcal{S}_j$ is a key of the reduction. The key idea is that \mathcal{S}_j represents whether the edge e_j is an incident edge of some vertex in a subset of vertices or not. $\$e_j^3$ in \mathcal{S}_j has exactly two previous occurrences in the \mathcal{R} -part (since each edge is incident to exactly two vertices). Hence \mathcal{S}_j can be parsed into two phrases (i.e., $\$e_j^3, \#$) if and only if $\$e_j^3$ has an occurrence which ends with an LZ-End phrase in the \mathcal{R} -part. Now we consider the greedy parsing of the \mathcal{R}_i -part (let $\Gamma(v_i) = \{e_{(i,1)}, \dots, e_{(i,|\Gamma(v_i)|)}\}$), which is as follows:

$$v_i^3, v_i\$\#e_{(i,1)}, e_{(i,1)}^2 v_i, \dots, v_i\$\#e_{(i,|\Gamma(v_i)|)}, e_{(i,|\Gamma(v_i)|)}^2 v_i, v_i\$\^2, \#.$$

The parsing has $2|\Gamma(v_i)| + 3$ phrases. We claim that this parsing is the smallest possible parsing: If the length of every phrase is at most 3, then $2|\Gamma(v_i)| + 3$ is the minimum size since the length of \mathcal{R}_i is $6|\Gamma(v_i)| + 7$. On the other hand, we can see that substrings of length at least 4 which contain a symbol v_i are unique in the whole string \mathcal{W}_G by the definition. Namely, $\$e_j^3$ is the only substring of length at least 4 which is not unique. Let us consider a parsing of \mathcal{R}_i such that the parsing has α length-4 phrases. In other words, we choose α incident edges out of $|\Gamma(v_i)|$ edges. Let (i_1, \dots, i_α) be the sequence of indexes of selected edges. We observe that the length of substrings that are covered by length at most 3 phrases. The length of the prefix of \mathcal{R}_i that is succeeded by the first length-4 phrase $\$e_{(i,i_1)}^3$ is $6(i_1 - 1) + 4$. This implies that there are at least $2(i_1 - 1) + 2$ phrases. The length of substring between $\$e_{(i,i_{d-1})}^3$ and $\$e_{(i,i_d)}^3$ is $6(i_d - i_{d-1} - 1) + 2$. Thus there are at least $2(i_d - i_{d-1} - 1) + 1$ phrases in each middle part. The length of the suffix that is preceded by the last length-4 phrase $\$e_{(i,i_\alpha)}^3$ is $6(|\Gamma(v_i)| - i_\alpha) + 5$. Since the last symbol is a unique symbol $\#$, there are at least $2(|\Gamma(v_i)| - i_\alpha) + 3$ phrases in the last part. Hence, there are at least

$$\alpha + 2(i_1 - 1) + 2 + \sum_{d=2}^{\alpha} (2(i_d - i_{d-1} - 1) + 1) + 2(|\Gamma(v_i)| - i_\alpha) + 3 = 2|\Gamma(v_i)| + 4$$

phrases. Thus the minimum number of phrases of \mathcal{R}_i is $2|\Gamma(v_i)| + 3$ and the above greedy parsing is the only candidate which is the minimum size. Notice that phrases of this parsing do not end with $\$e_j^3$. Let us consider the other possible parsing of \mathcal{R}_i -part as follows:

$$v_i^2, v_i^2\$, e_{(i,1)}^3, \dots, v_i^2\$, e_{(i,|\Gamma(v_i)|)}^3, v_i^2\$, \$, \#.$$

This parsing has $2|\Gamma(v_i)| + 4$ phrases. Notice that this parsing has phrases which end with $\$e_j^3$. Thus \mathcal{S}_j can be parsed into two phrases if we choose a non-greedy parsing such that there exists a phrase that ends at one of these positions. In other words, if we choose such a parsing

in the \mathcal{R}_i -part, we can reduce at most $|\Gamma(v_i)|$ phrases in the \mathcal{S} -part. These observations imply that \mathcal{R}_i is parsed into $2|\Gamma(v_i)| + 3$ or $2|\Gamma(v_i)| + 4$ phrases in any optimal parsing of \mathcal{W}_G .

Let us consider an optimal LZ-End parsing. Let r be the number of substrings \mathcal{R}_i which contain $2|\Gamma(v_i)| + 4$ phrases, and s be the number of substrings \mathcal{S}_j which contain exactly two phrases. Then the size of the parsing is

$$(10n + 13m) + (3m) + (2 \sum_{i=1}^n |\Gamma(v_i)| + 3n + r) + (3m - s) = 13n + 23m + r - s.$$

We consider a subset V' of vertices such that $v_i \in V'$ if and only if \mathcal{R}_i is parsed into $2|\Gamma(v_i)| + 4$ phrases (i.e., $|V'| = r$), and a subset E' of edges such that $e_j \in E'$ if and only if \mathcal{S}_j is parsed into three phrases (i.e., $|E'| = m - s$). If $E' = \emptyset$ (i.e., $s = m$), V' is a vertex cover of G . Otherwise, V' is not a vertex cover of G . However we can obtain the vertex cover number by using the parsing. Since the parsing is an optimal parsing, we can observe that there is no vertex v_i in $V \setminus V'$ which has two or more incident edges in E' (we can reduce two or more phrases in \mathcal{S} -part by adding one phrase in \mathcal{R}_i , a contradiction). This implies that we can obtain a vertex cover by choosing one vertex in $V \setminus V'$ for each edge in $E \setminus E'$. Then there exists an optimal LZ-End parsing of the same size which can directly represent a vertex cover. In other words, the vertex cover number is $r + m - s$ if there exists an optimal LZ-End parsing of $13n + 22m + (r + m - s)$ phrases. It is clear from the above constructions that there exists an optimal LZ-End parsing of $13n + 22m + k$ phrases iff the vertex cover number is k .

Since we can check a parsing is an LZ-End parsing in linear time, OptLE is clearly in class NP. \blacktriangleleft

4 MAX-SAT Formulation

An approach for exact computation of various NP-hard repetitiveness measures was shown in [1], where they formulated them as MAX-SAT instances so that very efficient solvers could be taken advantage of. Here, we show that this approach can be adapted to computing the optimal LZ-End parsing as well.

Let the input string be $T[1..n]$, and for any $i \in [2, n]$, let $M_i = \{j \mid 1 \leq j < i, T[j] = T[i]\}$. Below, we use 1 to denote true, and 0 to denote false. We introduce the following Boolean variables:

- p_i for all $i \in [1, n]$: $p_i = 1$ if and only if position i is a starting position of an LZ-End phrase. Note that $p_1 = 1$.
- c_i for all $i \in [1, n]$: $c_i = 1$ if and only if position i is the left-most occurrence of symbol $T[i]$.
- $r_{i \rightarrow j}$ for all $i \in [2, n]$ and $j \in M_i$: $r_{i \rightarrow j} = 1$ if and only if position i references position j via an LZ-End factor.

Notice that the truth values of c_i are all fixed for a given string and are easy to determine. Furthermore, the left-most occurrence must be beginning of a phrase, so, some values of p_i can also be fixed. For all $i \in [1, n]$:

$$c_i = p_i = 1 \text{ if } i \text{ is left-most occurrence of } T[i], \tag{1}$$

$$c_i = 0 \text{ otherwise.} \tag{2}$$

3:8 Optimal LZ-End Parsing Is Hard

The truth values of p_i define the factors, so in order to minimize the number of factors, we define the soft clauses as $\neg p_i$ for all $i \in [1, n]$. Below, we give other constraints between the variables that must be satisfied, i.e., hard clauses.

The symbol at any position must either be a left-most occurrence, or it must reference some position to its left. That is, for any $i \in [1, n]$:

$$c_i + \sum_{j \in M_i} r_{i \rightarrow j} = 1. \quad (3)$$

In order to ensure that references in the same LZ-End phrase are consistent, we have the following two constraints. The first ensures that if i references j and the symbols at positions $i - 1$ and $j - 1$ are different or do not exist (i.e., $j = 1$), positions i and $i - 1$ cannot be in the same LZ-End phrase. For all $i \in [2, n]$ and $j \in M_i$ s.t. $j = 1$ or $T[j - 1] \neq T[i - 1]$:

$$r_{i \rightarrow j} \implies p_i, \quad (4)$$

The second ensures that if position i references position j and i is not a start of an LZ-End phrase, then, position $i - 1$ must reference position $j - 1$. For all $i \in [2, n]$ and $j \in M_i \setminus \{1\}$ s.t. $T[j - 1] = T[i - 1]$:

$$r_{i \rightarrow j} \wedge \neg p_i \implies r_{i-1 \rightarrow j-1}. \quad (5)$$

Finally, the following constraints ensure that the reference of each LZ-End phrase must end at an end of a previous LZ-End phrase. For all $i \in [1, n]$ and $j \in M_i$:

$$\begin{cases} r_{i \rightarrow j} \wedge p_{i+1} \implies p_{j+1} & \text{if } i \in [1, n) \\ r_{i \rightarrow j} \implies p_{j+1} & \text{if } i = n. \end{cases} \quad (6)$$

It is easy to see that the truth assignments that are derived from any LZ-End parsing will satisfy the above constraints.

We now show that any truth assignment that satisfies the above constraints will represent a valid LZ-End parsing. The truth values for p_i implies a parsing where each phrase starts at a position i if and only if $p_i = 1$. Constraint (1),(2),(3) ensure that each position is either a left-most occurrence or references a unique previous position. Thus, it remains to show that the referencing of each position of a given factor is consistent (adjacent positions reference adjacent positions) and ends at a previous phrase end.

For any position i such that $c_i = 0$, let $j \in M_i$ be the unique value such that $r_{i \rightarrow j} = 1$. We can see that any such position i that is not at the beginning of a phrase (i.e., $p_i = 0$) will reference a position consistent with the reference of position $i - 1$: If $j = 1$ or $T[j - 1] \neq T[i - 1]$, then Constraint (4) would imply $r_{i \rightarrow j} = 0$. Thus, we have $j > 1$ and $T[j - 1] = T[i - 1]$, and from Constraint (5), we have that $r_{i-1 \rightarrow j-1}$, and the referencing inside a factor is consistent. Finally, from Constraint (6), the last reference in a phrase always points to an end of a previous LZ-End phrase.

The MAX-SAT instance contains $O(n^2)$ variables, and the total size of the CNF is $O(n^2)$: $O(n)$ clauses of $O(n)$ size (Constraint (3) using linear size encodings of cardinality constraints, e.g. [14]), and $O(n^2)$ clauses of size $O(1)$ (the soft clauses, and Constraints (4), (5), (6)).

We note that it is not difficult to obtain a MAX-SAT formulation for the original definition of LZ-End by minor modifications.

5 Approximation ratio of greedy parsing to optimal parsing

In this section, we consider an approximation ratio of the size z_e of the greedy LZ-End parsing to the size z_{end} of the optimal LZ-End parsing. Here, we give a lower bound of the ratio.

► **Theorem 4.** *There exists a family of binary strings such that the ratio z_e/z_{end} asymptotically approaches 2.*

Proof. Let $K = \sum_{i=1}^k 2^i (= 2^{k+1} - 2)$ for any positive integer $k \geq 1$. The following binary string w_k over an alphabet $\{a, b\}$ gives the lower bound:

$$w_k = aa \cdot \prod_{i=1}^k (a^{2^i}) \cdot b^4 \cdot \prod_{i=1}^K (a^i b^3).$$

It is easy to see that K is the length of the substring $\prod_{i=1}^k (a^{2^i})$. First, we show the greedy parsing of w_k . Let $W_0 = aa \cdot \prod_{i=1}^k (a^{2^i}) \cdot b^4$ (i.e., a prefix of w_k) and $W_j = W_{j-1} \cdot a^j b^3$ for any $1 \leq j \leq K$. Notice that $W_K = w_k$. We show that

$$LZEnd(W_j) = LZEnd(W_{j-1}), a^j b^2, b \quad (7)$$

by induction on j . Initially, we consider the greedy parsing of W_0 . The greedy parsing of the first run (i.e., maximal substring with a unique symbol) is $a, a, a^2, \dots, a^{2^k}$ of size $k+2$. The second run is parsed into three phrases b, b, b^2 . Thus

$$LZEnd(W_0) = a, a, a^2, \dots, a^{2^k}, b, b, b^2.$$

Moreover, we can see that the greedy parsing of W_1 is

$$LZEnd(W_1) = a, a, a^2, \dots, a^{2^k}, b, b, b^2, ab^2, b.$$

Hence Equation 7 holds for $j = 1$. Suppose that Equation 7 holds for any $j \leq p$ for some integer $p \geq 1$. We show that Equation 7 holds for $j = p+1$. Assume that there exists a phrase x of $LZEnd(W_{p+1})$ which begins in W_p and ends in a new suffix $a^{p+1}b^3$ of W_{p+1} . By the induction hypothesis, phrases of $LZEnd(W_p)$ which end with a are only in the first a 's run. This implies that x cannot end with a and x can be written as $x = x'ba^{p+1}b^\ell$ for some prefix x' of x and some positive integer ℓ . However, a^{p+1} only occurs in the first a 's run. Thus $LZEnd(W_{p+1})$ cannot have such a phrase x , namely $LZEnd(W_{p+1}) = LZEnd(W_p), S$ for some factorization S of the remaining suffix $a^{p+1}b^3$. It is easy to see that the remaining suffix $a^{p+1}b^3$ of W_{p+1} is parsed into $a^{p+1}b^2, b$. Hence Equation 7 holds for $j = p+1$, and it also holds for any j . Notice that $|LZEnd(w_k)| = 2K + k + 5$ holds.

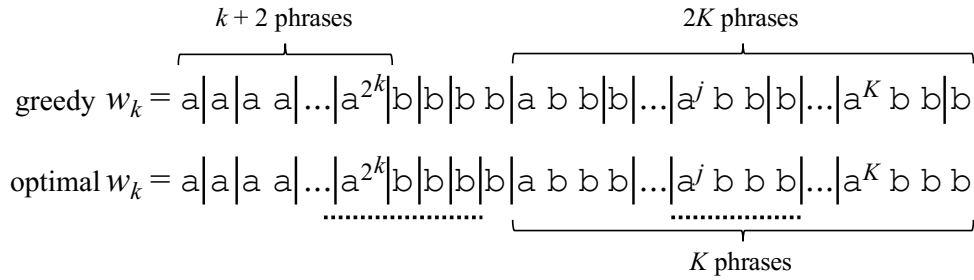
Finally, we give a smaller parsing of w_k . We consider the same parsing for the first run and a different parsing for the second run as b, b, b, b . In the greedy parsing, $a^j b^3$ cannot be a phrase since the only previous occurrence does not have an LZ-End phrase. We can use a substring $a^j b^3$ as a new phrase of W_j (see also Figure 3). Thus there exists an LZ-End parsing

$$a, a, a^2, \dots, a^{2^k}, b, b, b, b, a^1 b^3, \dots, a^K b^3.$$

The size of the parsing is $K + k + 6$.

Therefore the ratio z_e/z_{end} asymptotically approaches 2 for this family of strings. ◀

Note that this family of strings also gives a lower bound of the ratio z_e/z_{no} since $(z_e/z_{end}) \leq (z_e/z_{no})$ holds.



■ **Figure 3** Illustration for two variants of LZ-End parsings of a string w_k (Theorem 4). In the optimal parsing, we can choose $a^j b^3$ (dotted lines) as a phrase for each j ($1 \leq j \leq K$) by adding a single letter phrase b .

6 Conclusions

In this paper, we first studied the optimal version of the LZ-End variant. We showed the NP-completeness of the decision version of computing the optimal LZ-End parsing and presented an approach for exact computation of the optimal LZ-End by formulating as MAX-SAT instances. We also gave a lower bound of the possible gap (as the ratio) between the greedy LZ-End and the optimal LZ-End. Finally, we note possible future work in the following.

- Our reduction from the vertex cover problem uses a polynomially large alphabet. How can we construct a reduction with a small alphabet?
- The most interesting remaining problem is an upper bound of the ratio discussed in Section 5. We conjecture that there exists a constant upper bound (i.e., $z_e/z_{end} \leq c$ for any strings where c is a constant). This implies that the greedy parsing gives a constant-approximation of the optimal parsing. On the other hand, if there exists a family of strings which gives $c > 2$ or non-constant ratio, then the conjecture $z_e \leq 2z_{no}$ does not stand.

References

- 1 Hideo Bannai, Keisuke Goto, Masakazu Ishihata, Shunsuke Kanda, Dominik Köppl, and Takaaki Nishimoto. Computing np-hard repetitiveness measures via MAX-SAT. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPICs*, pages 12:1–12:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ESA.2022.12.
- 2 Sergio De Agostino and James A. Storer. On-line versus off-line computation in dynamic text compression. *Information Processing Letters*, 59(3):169–174, 1996. doi:10.1016/0020-0190(96)00068-3.
- 3 Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.
- 4 Takumi Ideue, Takuya Mieno, Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, and Masayuki Takeda. On the approximation ratio of LZ-End to LZ77. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 114–126. Springer, 2021. doi:10.1007/978-3-030-86692-1_10.

- 5 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in compressed space. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4-7, 2017*, pages 350–359. IEEE, 2017. doi:10.1109/DCC.2017.73.
- 6 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in linear time. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPICs*, pages 53:1–53:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ESA.2017.53.
- 7 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.
- 8 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-End parsing. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 2847–2866. SIAM, 2022. doi:10.1137/1.9781611977073.111.
- 9 Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In James A. Storer and Michael W. Marcellin, editors, *2010 Data Compression Conference (DCC 2010), 24-26 March 2010, Snowbird, UT, USA*, pages 239–248. IEEE Computer Society, 2010. doi:10.1109/DCC.2010.29.
- 10 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.
- 11 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.
- 12 Gonzalo Navarro. Indexing highly repetitive string collections. *CoRR*, abs/2004.02781, 2020. arXiv:2004.02781.
- 13 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021. doi:10.1145/3434399.
- 14 Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005. doi:10.1007/11564751_73.
- 15 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.
- 16 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

Sliding Window String Indexing in Streams

Philip Bille  

DTU Compute, Technical University of Denmark, Lyngby, Denmark

Johannes Fischer  

Department of Computer Science, Technische Universität Dortmund, Germany

Inge Li Gørtz  

DTU Compute, Technical University of Denmark, Lyngby, Denmark

Max Rishøj Pedersen  

DTU Compute, Technical University of Denmark, Lyngby, Denmark

Tord Joakim Stordalen  

DTU Compute, Technical University of Denmark, Lyngby, Denmark

Abstract

Given a string S over an alphabet Σ , the *string indexing problem* is to preprocess S to subsequently support efficient pattern matching queries, that is, given a pattern string P report all the occurrences of P in S . In this paper we study the *streaming sliding window string indexing problem*. Here the string S arrives as a stream, one character at a time, and the goal is to maintain an index of the last w characters, called the *window*, for a specified parameter w . At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and all occurrences of P within the current window must be returned. The streaming sliding window string indexing problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching.

Our main result is a simple $O(w)$ space data structure that uses $O(\log w)$ time with high probability to process each character from both the input string S and any pattern string P . Reporting each occurrence of P uses additional constant time per reported occurrence. Compared to previous work in similar scenarios this result is the first to achieve an efficient worst-case time per character from the input stream with high probability. We also consider a delayed variant of the problem, where a query may be answered at any point within the next δ characters that arrive from either stream. We present an $O(w + \delta)$ space data structure for this problem that improves the above time bounds to $O(\log(w/\delta))$. In particular, for a delay of $\delta = \epsilon w$ we obtain an $O(w)$ space data structure with constant time processing per character. The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees of independent interest, inspired by the classic log-structured merge trees.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching; Theory of computation \rightarrow Data structures design and analysis

Keywords and phrases String indexing, pattern matching, sliding window, streaming

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.4

Related Version *Full Version*: <https://arxiv.org/abs/2301.09477>

Funding *Philip Bille*: Supported by Danish Research Council grant DFF-8021-002498.

Inge Li Gørtz: Supported by Danish Research Council grant DFF-8021-002498.

Max Rishøj Pedersen: Supported by Danish Research Council grant DFF-8021-002498.

Acknowledgements We thank the anonymous reviewers for their helpful comments.



© Philip Bille, Johannes Fischer, Inge Li Gørtz, Max Rishøj Pedersen, and Tord Joakim Stordalen; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 4; pp. 4:1–4:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P , report all occurrences of P within S . In this paper, we introduce a basic variant of string indexing called the *streaming sliding window string indexing (SSWSI) problem*. Here, the string S arrives as a stream one character at a time, and the goal is to maintain an index of a *window* of the last w characters, for a specified parameter w . At any point in time a pattern matching query for a pattern P may arrive, also streamed one character at a time, and we need to report the occurrences of P within the current window. The goal is to compactly maintain the index while processing the characters arriving in either stream efficiently. We consider two variants of the problem: a *timely* variant where each query must be answered immediately, and a *delayed* variant where it may be answered at any point within the next δ characters arriving from either stream, for a specified parameter δ . See Section 1.1 for precise definitions.

The SSWSI problem naturally captures scenarios where we want to index the most recent data (i.e. the window) of a stream while supporting efficient pattern matching. For instance, monitoring a high-rate data stream system where we cannot feasibly index the entire stream but still want to support efficient queries. Depending on the specific system we may require immediate answers to queries, or we may be able to afford a delay that allows for more efficient queries and updates.

The SSWSI problem has not been explicitly studied before in our precise formulation, but for the timely variant several closely related problems are well-studied. In particular, the *sliding window suffix tree problem* [8, 13, 24, 25, 28] is to maintain the *suffix tree* of the current window (i.e., the compact trie of the suffixes of the window) as each character arrives. With appropriate augmentation the suffix tree can be used to process pattern matching queries efficiently, leading to a solution to the timely SSWSI problem. For constant-sized alphabets, the best of these solutions [8] maintains the sliding window suffix tree in constant *amortized* time per character while supporting efficient pattern matching queries. The worst-case time for updates is $\Omega(w)$. The other solutions achieve similar amortized time bounds. This amortization cannot be avoided since explicitly maintaining the suffix tree after the arrival of a new character may incur $\Omega(w)$ changes.

Another closely related problem is the *online string indexing problem* [3, 4, 5, 7, 14, 21, 22, 23]. Here the goal is to process S one character at a time (in either left-to-right or right-to-left order), while incrementally building an index of the string read so far. The best of these solutions update the index in either constant time per character for constant-sized alphabets [23] or $O(\log \log n + \log \log |\Sigma|)$ time for any alphabet where each character fits in a constant number of machine words [21]. These solutions all heavily rely on processing the string in right-to-left order to avoid the inherent linear time suffix tree updates due to appending, as mentioned above. Therefore they cannot be applied in our left-to-right streaming setting. Alternatively, we can instead apply these solutions on the reverse of the string S , but then each pattern must be processed in reverse order, which also cannot be done in our setting. Also, note that these solutions index the entire string read so far. It is not clear if they can be adapted to efficiently index a sliding window.

Another line of work shows how to maintain a fully dynamic suffix array under insertions and deletions [1, 2, 19, 27]. These can also be used to solve SSWSI but are more general and lead to polylogarithmically slower bounds than our results while being more complicated.

Our main result is an efficient and simple solution to the SSWSI problem in both the timely and delayed variant. Let w denote the size of the window. For the timely variant, we present a string index that uses $O(w)$ space and processes a character from the stream S in $O(\log w)$ time. Each pattern matching query P is also supported in $O(\log w)$ time per character with additional $O(\text{occ})$ time incurred after receiving the last character of P , where occ is the number of occurrences of P in the current window. The index is randomized and both time bounds hold with high probability. Compared to previous suffix tree based approaches for indexing a sliding window, we improve the worst-case time bounds per character in the stream from $\Omega(w)$ to $O(\log w)$ with high probability. This is particularly important in the above mentioned applications, such as high-rate data stream systems. Our solution generalizes to the delayed variant of the problem. If we allow a delay of δ before answering each query we achieve $O(w + \delta)$ space while improving the above time bounds to $O(\log(w/\delta))$. In particular, if we allow a delay of $\delta = \epsilon w$ for any constant $\epsilon > 0$, we achieve linear space and optimal constant time (reporting the occurrences still takes $O(\text{occ})$ time, and we do not count the reporting time towards the delay). Note that $\delta \leq w$ is sufficient delay for optimal time bounds and we can assume $O(w + \delta) = O(w)$. The results hold on a word RAM and for any alphabet size, assuming that each character fits into a constant number of machine words.

The key idea to achieve our result is a novel and simple hierarchical structure of suffix trees inspired by log-structured merge trees [26]. Instead of maintaining a single suffix tree on the window we maintain a collection of suffix trees of exponentially increasing sizes that cover the current window. We show how to efficiently maintain the structure as new characters from the stream arrive by incrementally “merging” suffix trees, while supporting efficient pattern matching queries within the window.

Our solution uses randomization to construct suffix trees in linear time with high probability. Plugging in a deterministic construction algorithm such as the one by Ukkonen [30], we obtain a solution using $O(\log w \log |\Sigma|)$ time for both queries and updates. With more recent deterministic suffix tree solutions [6, 10, 14] we can improve this to obtain $O(\log w \log \log n)$ time per character for both queries and updates. Note that the $O(\log \log |\Sigma|)$ in the time bounds of [14] has been replaced by $O(\log \log n)$ here due to an additional sorting step using [17].

1.1 Setup and Results

We formally define the problem as follows. Let S be a stream over any alphabet Σ where each character fits in a constant number of machine words. For given integer parameters $w \geq 1$ and $\delta \geq 0$, the δ -delayed streaming sliding window string indexing $((w, \delta)$ -SSWSI) problem is to maintain a data structure that, after receiving the first i characters of S , supports

- **Report(P):** report all the occurrences of P in $S[i - w + 1, i]$ before an additional δ characters have arrived from either stream.
- **Update():** process the next character in the stream S .

In the **Report(P)** query the pattern string P is also streamed. When P is streamed it interrupts the stream S , arrives one character at a time, and all characters of P arrive before the streaming of S resumes. Furthermore, we do not assume that we know the length of P before the arrival of its last character. Although P is streamed we assume random access to its characters after they arrive, as any pattern that fits in the window is at most w characters long and we can afford to store it. The delay is counted from after the last character of P arrives. Characters from S and from new patterns count towards the delay, while reported occurrences do not (otherwise it would be impossible to answer the query in time if there are more than δ occurrences).

4:4 Sliding Window String Indexing in Streams

We define the *timely streaming sliding window string indexing* (w -SSWSI) problem to be $(w, 0)$ -SSWSI, that is, queries must be answered immediately as the last character of the pattern arrives.

We show the following general main result.

► **Theorem 1.** *Let S be a stream and let $w \geq 1$ and $\delta \geq 0$ be integers. We can solve the (w, δ) -SSWSI problem on S with an $O(w + \delta)$ space data structure that supports **Update** and **Report** in $O(\log \frac{w}{\delta+1})$ time per character with high probability. Furthermore, **Report** uses additional worst-case constant time per reported occurrence.*

Here, with high probability means with probability at least $1 - \frac{1}{w^d}$ for any constant d . Theorem 1 provides a trade-off in the delay parameter δ . In particular, plugging in $\delta = 0$ in Theorem 1 we obtain a solution to the timely SSWSI problem that uses $O(w)$ space and $O(\log w)$ time per character for both **Update** and **Report**. Compared to the previous work on sliding window stream indexing [8, 13, 18, 24, 25, 28, 29] this improves the worst-case bounds on the **Update** operation from $\Omega(w)$ to $O(\log w)$ with high probability and also removes the restriction on the alphabet. At the other extreme, plugging in $\delta = \epsilon w$ for constant $\epsilon > 0$ in Theorem 1 we obtain a solution to the delayed SSWSI problem that uses $O(w)$ space and optimal constant time per character with high probability. All our results hold on a word RAM where each machine word has at least $\log w$ bits, and where each character of the alphabet fits into a constant number of machine words.

1.2 Techniques

We obtain our result for the timely variant, but without high probability guarantees, as follows. At all times we maintain at most $\log w$ suffix trees that do not overlap and together cover the window. The trees are organized by the *log-structured merge technique* [26], where the rightmost tree is the smallest and their sizes increase exponentially towards the left. For each new character that arrives we append its suffix tree to the right side of our data structure. Whenever there are two trees of the same size next to each other we “merge” them by constructing a new suffix tree covering them both. Each character from S is involved in at most $\log w$ merges and each merge takes expected linear time, so we spend expected amortized $O(\log w)$ time per character in S . We deamortize the updates by temporarily keeping both trees while merging them in the background. Note that for each adjacent pair of suffix trees we also store a suffix tree approximately covering them both, referred to as *boundary trees* (see details below).

We find the occurrences of a pattern P in the window by querying each of these trees, which takes $O(\log w)$ time per character in P . For adjacent pairs of trees larger than $|P|$ we find the occurrences of P crossing from one into the other using the boundary trees. The remaining trees cover a suffix of the window of length $O(|P|)$, and we grow a suffix tree to answer queries in this suffix *at query time*. Our data structure has some “overhang” on the left side of the window, and we use range maximum queries to report only the occurrences that start inside the window.

This solution is generalized to incorporate a delay of δ as follows. We store the $O(\log(w/\delta))$ largest trees from the timely solution and leave a suffix of size $\Theta(\delta)$ of the window uncovered by suffix trees. We answer queries as follows. If $|P| > \delta/4$ we say that P is *long*, and otherwise it is *short*. For long patterns we do as in the timely case; the suffix tree we grow at query time now must also contain the uncovered suffix, but it still has size $O(|P|)$ since the uncovered part of the window has length $O(\delta) = O(|P|)$. We show how to do this in $O(\log(w/\delta))$ time per character in P . For short patterns we utilize that they are smaller

than the delay to temporarily buffer the queries and later batch process them. We buffer up to $O(\delta \log(w/\delta))$ work and deamortize it over $\Theta(\delta)$ characters, obtaining the same bound as for long patterns. Updates run in the same bound since each character from S is involved in at most $O(\log(w/\delta))$ merges before it leaves the window.

Finally, we improve the time bounds by proving that for any substring S' of our window, we can construct the suffix tree over S' in $O(|S'|)$ time with probability $1 - w^{-d}$ for any constant $d > 1$. We do so by reducing the alphabet $\Sigma' = \{c \in S'\}$ of S' to rank-space $\{1, 2, \dots, |\Sigma'|\}$ from which the algorithm by Farach-Colton et al. [12] can construct the suffix tree in worst-case linear time. For large strings ($|S'| > w^{1/5}$) we pick a hash function from $\Sigma \rightarrow [0, w^c]$ that with high probability is injective on S' , and then we use radix sort to reduce to rank-space in linear time. For small strings ($|S'| \leq w^{1/5}$) we pick a hash function from $\Sigma \rightarrow [0, w/\log w]$ that is injective with (almost) high probability, and use this to manually construct a mapping into rank space in $O(S')$ time. This mapping algorithm uses additional $O(w/\log w)$ space, but we construct at most $O(\log w)$ suffix trees at any time so the total space is linear.

1.3 Outline

In Section 2 we cover the preliminaries, including some useful facts about suffix trees. In Section 3 we give a solution to the timely SSWSI problem that supports each operation in expected logarithmic time per character. In Section 4 we show how to generalize this to incorporate delay, and in Section 5 we show how to get good probability guarantees, proving Theorem 1.

2 Preliminaries

Given a string X of length n over an alphabet Σ , the i th character is denoted $X[i]$ and the substring starting at $X[i]$ and ending at $X[j]$ is denoted $X[i, j]$. The substrings of the form $X[i, n]$ are the *suffixes* of X .

A *segment* of X is an interval $[i, j] = \{i, i+1, \dots, j\}$ for $1 \leq i \leq j \leq n$. We will sometimes refer to segments as strings, i.e., the segment $[i, j]$ refers to the string $X[i, j]$. The definition differs from “substring” by being specific about position; even if $X[1, 2] = X[3, 4]$ we have $[1, 2] \neq [3, 4]$. A *segmentation* of X is a decomposition of X into disjoint segments that cover it. For instance, $x_1 = [1, i]$ and $x_2 = [i+1, n]$ is a segmentation of X into two parts. The two segments x_1 and x_2 are *adjacent* since x_2 starts immediately after x_1 ends, and for a pair of adjacent segments we define the *boundary* (x_1, x_2) to be the implicit position between i and $i+1$.

The *suffix tree* [31] T over X is the compact trie of all suffixes of $X\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of X , and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $O(n)$ space by implicitly representing the string associated with each edge using two indices into X . Farach-Colton et al. [12] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ . For alphabets of the form $\Sigma = \{0, \dots, n^c\}$ for constant $c \geq 1$ this implies that T can be built in worst-case $O(n)$ time using radix sort. For larger alphabets we can reduce to the polynomial case in expected linear time using hashing, building T in expected linear time (see Section 5 for details).

The *suffix array* L of X is the array where $L[i]$ is the starting position of the i th lexicographically smallest suffix of X . Note that $L[i]$ corresponds to the i th leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string

spelled out by the root-to- v path. The descendant leaves of v exactly correspond to the suffixes of X that start with s_v , and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L .

We augment the suffix tree to support efficient pattern matching queries as follows. First, we use the well-known FKS perfect hashing scheme [15] to store the edges of the suffix tree, so we can for any node determine if there is an outgoing edge matching a character $a \in \Sigma$ in worst-case constant time. Note that this construction takes *expected* linear time. Furthermore, we also build a *range maximum query* data structure over L . This data structure supports range maximum queries, i.e., given a range $[\alpha, \beta]$ return the $j \in [\alpha, \beta]$ maximizing $L[j]$. It also supports range minimum queries, defined analogously. The data structure can be built in linear time and supports queries in constant time [16]. Finally, we preprocess the suffix tree in linear time such that each internal node v stores the range $[\alpha, \beta]_v$ into L corresponding to the occurrences of s_v .

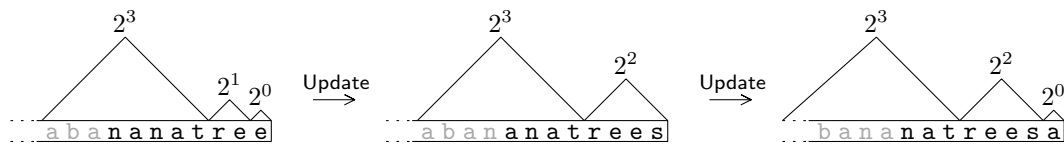
We can use this structure to efficiently find all the occurrences of P in $O(|P| + \text{occ})$ time, where occ is the number of occurrences, or the leftmost and rightmost occurrence of P in $O(|P|)$ time. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v . First we find the locus by walking downwards in the suffix tree, matching each character in P in worst-case constant time using the dictionary. Once we have found v we can report all the occurrences in $[\alpha, \beta]_v$ in $O(\text{occ})$ time. Alternatively, we can find the rightmost occurrence of P in constant time by doing a range maximum query on the range $[\alpha, \beta]_v$ in L , which returns the $j \in [\alpha, \beta]_v$ maximizing the *string position* $L[j]$. We can also find the leftmost occurrence by doing a range minimum query.

Finally, note that it is possible to deamortize algorithms with *expected* running time using the standard technique of distributing the work evenly. Specifically, if an algorithm runs in expected λn time we can do λ work for $n - 1$ steps; by linearity of expectation only expected λ work remains for the last step.

3 The Timely SSWSI Problem

Here we present a solution for the timely variant that matches the bounds in Theorem 1 in expectation. Section 5 shows how to get the bounds with high probability. Throughout this section we assume without loss of generality that w is a power of two. Section 3.3 briefly mentions how to generalize to arbitrary w .

The main idea is as follows. We maintain a suffix of S of length at least w . This suffix is segmented into at most $\log w$ segments whose sizes are distinct powers of two, in increasing order from right to left. The length of the suffix we store is at most $2^0 + \dots + 2^{\log w} = 2w - 1$. When a new character arrives, we append a new size-one segment to our data structure and merge equally-sized segments until they all have distinct sizes again. We also discard the largest segment when it no longer intersects the window. For each segment we store a suffix tree, and for every pair of adjacent segments we store a *boundary tree* approximately covering them both (see below). To support queries we query the suffix tree for each individual segment, and also each boundary tree. For the segments larger than the pattern, the boundary trees are sufficient to find the occurrences crossing the respective boundary. The remaining trees cover a suffix of S that is $O(|P|)$ long, and we grow a suffix tree at query time to find the remaining occurrences in this suffix.



■ **Figure 1** Example of updating the data structure with a window size of $w = 8$. Here we illustrate the segments by the suffix trees built over them. Characters outside of the window are gray. As the character **s** arrives we construct a new suffix tree of size one, which is then immediately merged with the existing size-one suffix tree over **e** into a size-two suffix tree over **es**, which is then merged into the final size-four suffix tree over **rees**. After receiving **a** we again have a size-one suffix tree. Note that after three more updates the suffix tree of size eight will no longer overlap the window and will be discarded.

3.1 Data Structure

At any point, the data structure contains a suffix s of S of length $w \leq |s| \leq 2w - 1$ and a segmentation of s into at most $\log w$ segments. Specifically, if $|s| = 2^{b_1} + \dots + 2^{b_k}$ for integers $b_1 < \dots < b_k$ then we have the segmentation s_1, \dots, s_k where $|s_i| = 2^{b_i}$, and s is the concatenation of the strings s_k, s_{k-1}, \dots, s_1 , in that order. The set $\{b_1, \dots, b_k\}$ is unique and corresponds to the 1-bits in the binary encoding of $|s|$. Three different configurations can be seen in Figure 1.

For each segment s_i we store the suffix tree T_i over s_i , along with a range maximum query data structure over the suffix array of s_i . For each boundary (s_{i+1}, s_i) we store the *boundary tree* B_i , which is the suffix tree over the substring centered at the boundary and extending $|s_i|$ characters in both directions. We augment B_i with an additional data structure that we will use for reporting occurrences across the boundary. Let BL_i be the suffix array corresponding to B_i . We define the *modified suffix array* BL'_i as

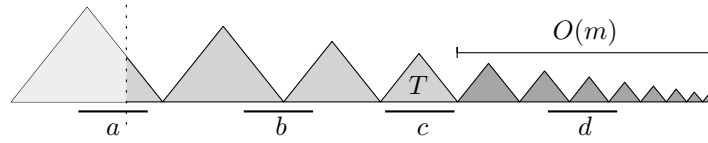
$$BL'_i[j] = \begin{cases} BL_i[j] & \text{if } BL_i[j] \text{ corresponds to a suffix starting in } s_{i+1} \\ -\infty & \text{if } BL_i[j] \text{ corresponds to a suffix starting in } s_i \end{cases}$$

We store a range maximum query data structure over BL'_i . Each of the data structures use $O(s_i)$ space, so the whole data structures uses $O(s) = O(w)$ space.

We note a few properties of the data structure. Let $S[n]$ be the most recent character to arrive and let $W_n = S[n - w + 1, n]$ be the current window. Then W_n is a suffix of s since $|s| \geq w$. The largest, and leftmost, segment s_k always has size $2^{\log w} = w$; it is not larger since $\log w$ bits are sufficient to represent $|s| \leq 2w - 1$, and it is always there since $|s| \geq w$ cannot be represented with $\log w - 1$ bits. For the same reason, s_k always intersects at least partially with W_n , and each of s_1, \dots, s_{k-1} are fully contained in W_n .

3.2 Queries

The idea is as follows, as exemplified in Figure 2. Any occurrence of a pattern P that is fully contained in a segment is found using the suffix tree over that segment. Similarly, any occurrence that only crosses a single boundary far enough away from the end of the window is found in the respective boundary tree. Note that in the leftmost segment we must be careful to not report any occurrences that start before the left window boundary. The remaining occurrences are not contained in any of the trees in the data structure (either because they cross multiple boundaries or because they cross a single boundary (s_{i+1}, s_i)



■ **Figure 2** Illustration of how we answer queries for a pattern P of length m . The lines denoted a , b , c , and d indicate occurrences of P . The segmentation is illustrated by the trees over the segments. The leftmost window boundary is marked with a vertical dashed line. Note that the leftmost segment intersects only partially with the window. The tree T marks the smallest segment of size at least m . The segments to the right of T are all smaller than m , so they cover at most $m + m/2 + \dots + 1 = O(m)$ characters. To answer the query we match P in the tree over each segment and in each boundary tree, and we also build a suffix tree over the segments smaller than m at query time. We find b because the respective boundary tree is sufficiently large. We find c because it is fully contained in a segment. We find d in the suffix tree that we build at query time. Note that a is not contained in the window; we avoid reporting it by recursively using range maximum queries to find the *rightmost* occurrence of P in the leftmost segment.

but start more than $|s_i|$ characters to the left of the boundary). However, these occurrences are all located within a substring of size $O(m)$ ending at position $S[n]$, so we build, at query time, a suffix tree to find these occurrences.

Let P be the length- m pattern being queried, $S[n]$ be the most recent character to arrive, and let W_n , the suffix s , the segmentation s_1, \dots, s_k , and the indices $b_1 < \dots < b_k$ be defined as above. As mentioned, any occurrence of P in W_n must either be fully contained within one of the segments, or it must cross the boundary between two adjacent segments. We will show how to handle each of these cases separately.

Fully Contained in a Segment

Fix a specific segment s_i . As each character of P arrives we match it in T_i . When the last character arrives we have a (possibly empty) range $[\alpha, \beta]$ into the suffix array of s_i corresponding to the occurrences of P . If s_i is not the leftmost segment then it is fully contained in W_n and we report all the occurrences. Otherwise, $s_i = s_k$ is the leftmost segment, which might overlap only partially with W_n , and it may contain occurrences of P that are not contained in the window. However, note that the intersection between W_n and s_k is a suffix of s_k . Therefore, if an occurrence of P in s_k starts inside W_n it also ends inside W_n . We find all such occurrences as follows. Let L_k be the suffix array of s_k . As described in Section 2 we find the index j of the rightmost occurrence of P by doing a range maximum query on the range $[\alpha, \beta]$ in L_k . If $L_k[j]$ is not inside W_n then none of the occurrences are, and we are done. Otherwise we recurse on $[\alpha, j - 1]$ and $[\beta, j - 1]$. Matching P in the trees of all the segments takes $O(\log w)$ overall time per character of P . Reporting each occurrence takes constant time since range maximum queries run in constant time.

Crossing a Boundary

We now show how to report the occurrences of P that span a boundary. The main idea is as follows, as illustrated in Figure 3. Let s_i be the smallest segment where $|s_i| \geq m$. Consider any boundary (s_{j+1}, s_j) to the left of s_i , i.e., where $j \geq i$. Since both of these segments have size at least $|s_i| \geq m$, the boundary tree B_j extends at least m characters in both directions from the boundary. Therefore, all the occurrences of P crossing the boundary are contained in B_j , and none of them can cross another boundary as well. Now

consider the suffix \mathcal{R} of s containing the $m - 1$ last characters of s_i and extending to the end of s . This substring contains all the other boundary-crossing occurrences. Furthermore, all the occurrences in \mathcal{R} cross at least one boundary since the longest consecutive part of a single segment in \mathcal{R} is the $m - 1$ characters in s_i . Note that the length of \mathcal{R} is at most $m - 1 + |s_{i-1}| + |s_{i-1}|/2 + \dots + 1 < m - 1 + 2|s_{i-1}| < 3m$ since $|s_{i-1}| < m$. Thus, the number of boundary-crossing occurrences of P equals the number of occurrences in \mathcal{R} plus the number of occurrences crossing the boundaries $(s_k, s_{k-1}), (s_{k-1}, s_{k-2}), \dots, (s_{i+1}, s_i)$.

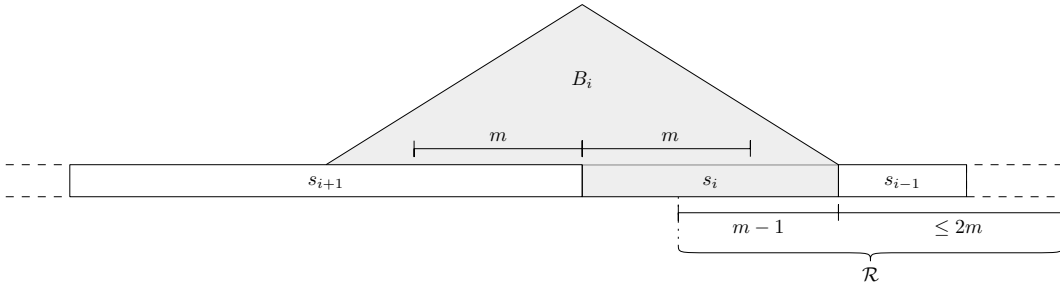
The algorithm for finding the occurrences in the sufficiently large boundary trees is as follows. Fix a boundary (s_{x+1}, s_x) . We match each character of P in B_x as it arrives. When the last character arrives we know if $|s_x| \geq m$, and also the range $[\alpha, \beta]$ corresponding to the occurrences of P in the boundary tree. If $|s_x| \geq m$ (hence $x \geq i$) we report the occurrences as follows. As above we do a range maximum query to find the j maximizing $BL'_x[j]$. If $BL'_x[j] = -\infty$ then all occurrences of P start in s_x , and there are no occurrences crossing the boundary. Otherwise, $BL'_x[j]$ corresponds to the starting position of the rightmost occurrence of P in s_{x+1} . Since all of P has arrived and we now know m , we know that this occurrence crosses the boundary if and only if $BL'_x[j] \geq |s_x| - m + 2$ (recall that B_x extends $|s_x|$ characters in both directions from the boundary). If it does not cross the boundary, then none of the other occurrences do either. Otherwise we report $BL'_x[j]$ and recurse on $[\alpha, j - 1]$ and $[j + 1, \beta]$ to find the remaining occurrences. Matching P in all boundary trees takes $O(\log w)$ overall time per character, and reporting each occurrence with range maximum queries takes constant time.

We now show how to find the occurrences of P in \mathcal{R} with the same bounds. Assume that we know that $2^\ell \leq m < 2^{\ell+1}$ for some integer ℓ . We build the suffix tree over the last $3 \cdot 2^{\ell+1}$ characters of s , deamortized over receiving the first $2^{\ell-1}$ characters of P . Over the next $2^{\ell-1}$ characters we match P in the tree, at a rate of two characters per new character from P . Then, when the 2^ℓ th character arrives, we have caught up to the stream P , and we match the remaining $m - 2^\ell$ characters as they arrive. When the last character arrives we have matched P in a tree of size at least $3m$, and we can start reporting occurrences. Note that we are overestimating the size of the tree, and it potentially includes some occurrences of P that are contained in s_i . To avoid reporting these, we also build a range maximum query data structure over the suffix array such that we can use recursive range maximum queries. When deamortized, we construct the tree in expected constant time per character of P . Matching P also takes constant time per character. We know that $m \leq w$, so we run this algorithm simultaneously for each of the $\log w$ different choices for ℓ , using expected $O(\log w)$ time per character in P . Note that the trees use $O(w)$ space in total since the sum of the space is a geometric sum where the largest term is $O(w)$.

3.3 Amortized Updates

We show how to support updates in amortized $O(\log w)$ time. Let $S[n]$ be the last character to arrive and as in the description of the data structure let $b_1 < b_2 < \dots < b_k$ be the positions of the 1-indices in the binary encoding of $|s|$. When the new character $c = S[n + 1]$ arrives, we update s and the segmentation $s_1 \dots s_k$ to create the new suffix s' with the new segmentation s'_1, \dots, s'_k . See Figure 1 for an example.

If $|s| < 2w - 1$ then we set $s' = sc$. The segmentation of s' corresponds to the unique binary encoding of $|s'| = |s| + 1$, so we update the segmentation analogously to a “binary increment”. One way to do so is as follows. We create a new segment of size one over c . If there was not already a segment of size one, then we add the new segment and we are done. Otherwise we *merge* (see below) the two size-one segments to create a segment of size



■ **Figure 3** The segment s_i is the smallest segment where $|s_i| \geq m$. For each boundary (s_{j+1}, s_j) where $j \geq i$, the tree B_j is large enough to find all occurrences of P across the boundary. All other occurrences of P that cross a boundary must be in \mathcal{R} , the string covering the $m - 1$ rightmost characters of s_i and extending to the end of the window. The length of \mathcal{R} is no more than $m - 1 + |s_{i-1}| + |s_{i-1}|/2 + \dots + 1 < 3m$.

two. The process cascades until we reach a size 2^b that does not exist in the segmentation of s (i.e., the smallest index $b \notin \{b_1, \dots, b_k\}$). At this point we replace all of the segments s_{b-1}, \dots, s_1 with s'_1 covering the last 2^b characters of s' . The remaining segments for s' are the same as the segments s_{b+1}, \dots, s_k . If $|s| = 2w - 1$ then there is a segment of each size $2^0, 2^1, \dots, 2^{\log w}$. Since the segments have decreasing size from left to right, the $\log w - 1$ rightmost segments cover the last $2^0 + \dots + 2^{\log w - 1} = w - 1$ characters of s . Thus, after c arrives, the leftmost segment of size $2^{\log w} = w$ no longer intersects the window. We remove it by setting $s' = s[w + 1, |s|]c$, and update the segmentation as above.

Let s_a, s_b and s_c be three adjacent segments, in that order. To *merge* s_b and s_c we combine them into a new segment s_d that spans them both, construct the suffix tree over s_d , and construct a range maximum query data structure on the suffix array of s_d . Furthermore, since s_a and s_d are now adjacent we also construct the boundary-spanning suffix tree for the boundary (s_a, s_d) that extends $|s_d|$ characters in each direction. The construction of all of these data structures takes expected $O(|s_d|)$ time (see Section 2). Thus, it takes expected constant time per character every time it moves into a new, larger segment. Each character is contained in at most $\log w$ segments before it leaves the window, so the amortized update time is expected $O(\log w)$ per character.

Note that all but the last merge are unnecessary to actually compute s'_1 ; in the amortized setting we can simply determine where the cascade will end and immediately construct the suffix tree over the corresponding segment. However, the cascading merges will come into play in the deamortized variant.

Also note that if w is not a power of two we can use a similar scheme where we allow either two simultaneous trees of size $2^{\lceil \log w \rceil}$, or one tree of size $2^{\lceil \log w \rceil}$. In both cases, there are some straightforward edge cases for when to remove the leftmost segment.

3.4 Deamortized Updates

We now show how to deamortize the updates. Unfortunately the previous construction cannot be directly deamortized since the suffix tree construction algorithm by Farach-Colton et al. [12] requires access to the whole string. Therefore, if a new character c causes a cascade of merges resulting in a new segment of size 2^i we have to build the suffix tree over that segment when c arrives.

Instead, we modify the structure slightly. When two segments of size 2^i become adjacent we temporarily keep both while deamortizing the cost of merging them over the *next* 2^i characters of S , doing expected constant work per character. Note that queries are unaffected,

with one exception for reporting occurrences across the boundaries; there might now be two adjacent segments s_{i+1} and s_i of the same size that are both the smallest segment at least as large as $|P|$. In this case the suffix \mathcal{R} extends only $m - 1$ characters into the rightmost segment s_i . The boundary tree for (s_{i+1}, s_i) is large enough to report all occurrence crossing that boundary since both segments have size at least $|P|$. Furthermore, \mathcal{R} potentially becomes twice as long, so we adjust the constants of the trees that we grow at query time.

To bound the time for updates we show that we are constructing at most $\log w$ suffix trees at any point, from which it follows that the update time is expected $O(\log w)$. To do so we show the following lemma.

► **Lemma 2.** *When the construction of a segment of size 2^i finishes there is exactly one segment of each size $2^{i-1}, \dots, 2^0$.*

Proof. The proof is by induction on i . For $i = 1$, when two size-one segments become adjacent we merge them when the next character c from S arrives. This results in a segment of size two, as well as a size-one segment containing c , proving the base case.

Inductively, consider the first time two segments of size 2^i become adjacent. By the induction hypothesis, there is one segment of each size $2^0, 2^1, \dots, 2^{i-1}$ to the right of these two segments. For another segment of size 2^i to be constructed, we must first receive one more character, which triggers a merge that eventually cascades through all $i - 1$ of these segments. For this to happen, $1 + (2^0 + 2^1 + \dots + 2^{i-1}) = 2^i$ more characters from S must arrive, where the 1 is for the next character to arrive, and 2^j is the amount of characters the j th merge is deamortized over. However, at this point the merge of the two segments of size 2^i is complete, so we constructed two new segments, one of size 2^{i+1} and one of size 2^i . By the induction hypothesis, there is also one segment of each size $2^0, \dots, 2^{i-1}$, concluding the proof. ◀

Lemma 2 implies that there are never more than two segments of the same size adjacent to each other, and therefore at most one merging process for each segment size $2^0, 2^1, \dots, 2^{\log w}$. To see this, consider the first time two segments a and b of size 2^i are adjacent. At this point, there are $2^0 + 2^1 + \dots + 2^{i-1} = 2^i - 1$ characters to the right of b . When the next segment c of size 2^i arrives there are $2^i - 1$ characters to the right of that, too. But then there are $|c| + 2^i - 1 = 2^i + 2^i - 1$ characters to the right of b . Thus 2^i new characters must have arrived in the meanwhile, and the merging of a and b is done.

We obtain the following theorem.

► **Theorem 3.** *Let S be a stream and let $w \geq 1$ be an integer. We can solve the w -SSWSI problem on S with an $O(w)$ space data structure that supports *Update* and *Report* in expected $O(\log w)$ time per character. Furthermore, *Report* uses additional worst-case constant time per reported occurrence.*

4 The Delayed SSWSI Problem

In this section we show how to improve the result from Section 3 if we are allowed a delay of δ . The main idea is as follows. As before, we maintain suffix trees of exponentially increasing sizes, although only the $O(\log(w/\delta))$ largest of them. As a result there are fewer trees to query, but also an *uncovered* suffix of size $\Theta(\delta)$ of the window for which we do not have any suffix trees. As in Section 3 we denote the part of S covered by suffix trees by s and we denote the uncovered suffix by t . As above, s is segmented into s_1, \dots, s_k .

We will first explain how to solve the problem when all patterns are *long*, that is, $|P| > \delta/4$, and then when all patterns are *short*, that is, $|P| \leq \delta/4$. Finally we show how to combine these solutions. When all the patterns are long we can afford to construct, at query time,

4:12 Sliding Window String Indexing in Streams

a suffix tree covering t . On the other hand, when all the patterns are short we can do both updates and queries in an offline fashion; we buffer queries and updates until we have approximately $\delta/2$ operations to do, at which point we can afford to construct a suffix tree over t in a deamortized manner. See Figure 4 for an example.

Throughout this section we assume without loss of generality that δ is a power of two. Otherwise we instead use a more restrictive delay of $\delta' = 2^{\lceil \log \delta \rceil}$ and achieve the same asymptotic bounds.

4.1 Long Patterns

We first show how to support queries if all patterns have a length $m > \delta/4$. We modify the data structure from Section 3 slightly. The smallest tree now has size $\delta/2$ as opposed to 1, so there are $\Theta(\log w - \log(\delta/2)) = O(\log(w/\delta))$ segments and boundary trees. The uncovered suffix t has length at most δ .

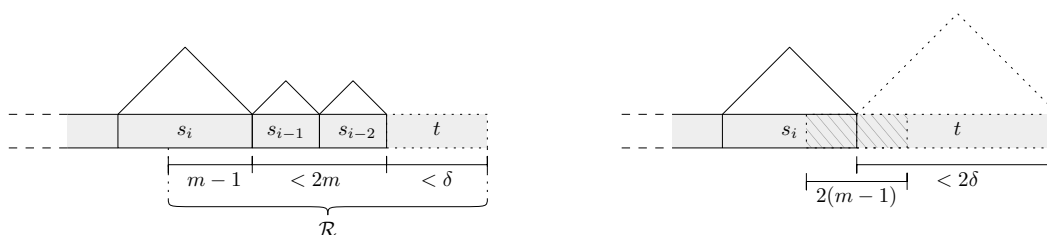
We answer queries the same way as in Section 3.2, with only small modifications. Let P be a pattern of length $m > \delta/4$. As before, let s_i be the smallest and rightmost segment with $|s_i| \geq m$. We find any occurrence within a segment or crossing a single boundary by using the suffix trees over each segment and the boundary trees to the left of s_i , as before. The remaining occurrences we again find by growing suffix trees of exponentially increasing sizes from the right window boundary. The only change is that we now grow the trees faster, as we must also cover t , and we can afford to let the smallest tree have size δ since we have $m > \delta/4$ characters in the pattern to deamortize the work over. As above, let \mathcal{R} be the string covering the $m - 1$ last characters of s_i and extending to the right window boundary, which now also includes t . As $|t| < \delta$ the length of \mathcal{R} is $|\mathcal{R}| < 3m + \delta < 7m$. Assuming $2^\ell \leq m < 2^{\ell+1}$, we build the suffix tree of size $7 \cdot 2^{\ell+1}$ and match P in it, amortized over the characters of P . As we have $m > \delta/4$ characters to deamortize the work over, we only do this for each choice of ℓ where $2^{\ell+1} \geq \delta$, which results in $O(\log w - \log \delta) = O(\log(w/\delta))$ work per character in P . As in Section 3.2 we use recursive range maximum queries to avoid double reporting any occurrences of P that are also in s . As there are also only $O(\log(w/\delta))$ segments and boundary trees we spend $O(\log(w/\delta))$ time per character in P . Note that we answer these queries without delay.

Updates are performed as follows. For each segment of $\delta/2$ characters that arrives we construct the suffix tree over it, deamortized over the next $\delta/2$ characters of S . We merge suffix trees as before, also deamortized over new characters of S . The induction proof from Section 3.4 still works by modifying the base case; the merging of two trees of size $\delta/2$ takes $\delta/2$ characters, at which point another tree of size $\delta/2$ is constructed. The inductive step follows from the fact that δ is a power of two. Thus, we spend expected $O(\log(w/\delta))$ time per update.

4.2 Short Patterns

We now show how to support queries if all patterns have a length $m \leq \delta/4$. We extend the data structure with a buffer of size δ . This buffer will contain queries that we have not yet answered and characters for S that we have not yet processed. The total space is still $O(w + \delta) = O(w)$.

Whenever a character from S arrives we append it to both t and to the buffer. When a pattern arrives we append the full pattern to the buffer, and along with it we store the current position of the right window boundary. Once the buffer has more than $\delta/2$ characters



■ **Figure 4** *Left:* Example of a query with a long pattern. Here s_i is the smallest and rightmost segment with $|s_i| \geq m$. Note that the non-indexed suffix t is less than $\delta < 4m$ characters long. *Right:* Example of a query with a short pattern. Note that for short patterns, s_i is always the rightmost segment. Any occurrence in s cross at most a single boundary and is found using the constructed trees. Any occurrence in t is found by the suffix tree over t that we construct when we flush the buffer. Any occurrence that cross the boundary (s, t) is found by the KMP automaton we build over the substring the extends $m - 1$ characters in both directions from the boundary, which is hatched in the figure.

(patterns and text combined) we immediately allocate a new buffer of size δ and *flush* the old buffer as follows. Note that at this point there are strictly less $\frac{3}{4}\delta$ characters in the buffer since each pattern is short.

When we flush the buffer, we first answer all the buffered queries, and then we process all the buffered updates. We deamortize this work over the next $\delta/4$ characters that arrive from either stream. To answer the buffered queries we do as follows. Let P_1, \dots, P_ℓ be the patterns in the buffer, let $m_i = |P_i|$, and let $M = \sum_{1 \leq i \leq \ell} m_i$. We have $M < \delta$. We start by building a suffix tree over t , along with a range maximum query data structure over the suffix array of t . This takes expected $O(\delta)$ time. An occurrence of P_i is either contained in s , or it crosses the boundary (s, t) , or it is contained in t . Since P_i is smaller than each segment s_j we can find all the occurrences within s using the suffix trees over the segments and the boundary trees in $O(m_i \log(w/\delta))$ time. To find the occurrences crossing the boundary we build the KMP matching automaton [20] for P_i . In it we match the string that is centered at the boundary (s, t) and extends $m_i - 1$ characters in each direction. This takes $O(m_i)$ time. To find the occurrences in t we match P_i in the suffix tree over t in $O(m_i)$ time. In total, this takes $O(M \log(w/\delta)) = O(\delta \log(w/\delta))$ time for all the patterns, or expected $O(\log(w/\delta))$ time per character when deamortized. Note however, that after P_i arrived more characters from S could have arrived and been appended to t . We must therefore take care not to report any occurrences of P_i that extend past what *was* the right window boundary when P_i arrived. The KMP automaton finds the occurrences in left-to-right order, and in t we avoid reporting too far right using recursive range minimum queries.

Finally, we process each update in the buffer in the order they arrived, using the same procedure as for long patterns. This takes $O(\log(w/\delta))$ time per update and $O(\delta \log(w/\delta))$ time in total. Thus flushing the buffer takes expected $O(\log(w/\delta))$ time per character since we deamortize the expected $O(\delta \log(w/\delta))$ work over $\delta/4$ characters. Since we allocate a new buffer immediately when we begin flushing, we will complete the flush before the next flush begins.

4.3 Both Long and Short Patterns

We now show how to combine the solutions for short and long patterns, to obtain a solution that handles patterns of any length. The data structure is the same as for small patterns above. As above, we append each new character to the buffer. However, whenever we start

streaming a pattern we also proceed as if P were long. If P turns out to fit in the buffer without triggering a flush (which might also happen if P is long), we simply discard the work we did for the long-pattern case. However, if adding P to the buffer results in more than $\frac{3}{4}\delta$ characters being in the buffer, then P must be long. We immediately start flushing the buffer (ignoring the characters related to P) and also continue processing P as a long pattern. Note that since we are potentially streaming a long pattern while batch processing the updates in the buffer, the data structure might change while we are matching in it. However, it only changes when a merge finishes, replacing a pair of suffix trees by a larger tree. If this happens we keep the old trees in memory until we are done processing the pattern, at which point we discard them.

We obtain the following theorem.

► **Theorem 4.** *Let S be a stream and let $w \geq 1$ and $\delta \geq 1$ be integers. We can solve the (w, δ) -SSWSI problem on S with an $O(w)$ space data structure that supports **Update** and **Report** in expected $O(\log(w/\delta))$ time per character. Furthermore, **Report** uses additional worst-case constant time per reported occurrence.*

5 Obtaining High Probability

In this section we show how to improve the time bounds to $O(\log(w/\delta))$ with probability $1 - w^{-d}$ for any constant $d \geq 1$.

The expectation in the time bounds in Section 4 comes from the construction of suffix trees (recall that we also build suffix trees at query time). Below, in Lemma 5, we prove that given a string \mathcal{K} of length $k = O(w)$ we can construct the suffix tree over \mathcal{K} in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$, using additional $O(w/\log w)$ space. We use this algorithm to construct suffix trees during updates and queries, deamortizing them as before and doing $O(\log(w/\delta))$ work per character that arrives. When a new character arrives from S or P , at most $O(\log(w/\delta)) = O(\log w)$ suffix tree constructions will finish. At this point, we finish constructing those trees that did not finish in time, that is, used more more time than what was allotted to them. By the union bound, the probability that any of them fail to finish in time (and thus incurring extra construction cost) is no more than $c \log w/w^{1+\epsilon}$ for some constant c which is no more than $1/w$ for large w . Thus, for each character from S or P we spend $O(\log(w/\delta))$ time with high probability in w . We obtain the $1 - 1/w^d$ probability bound by probability boosting, running $d = O(1)$ independent copies of the construction algorithm simultaneously. The algorithm from Lemma 5 uses additional $O(w/\log w)$ space, but we are never constructing more than $O(\log w)$ suffix trees, so the space usage is $O(w)$ in total.

Furthermore, as mentioned in Section 2, we previously used an FKS dictionary [15] to store the edges to support reporting queries in worst-case constant time per character in the pattern. The construction time of this dictionary is expected linear, so it can no longer be used. Instead we use a dictionary by Dietzfelbinger and Meyer auf der Heide [11]. If there are n elements in the dictionary it supports searches in worst-case constant time and any sequence of $\frac{1}{2}n$ updates takes constant time per update with probability $1 - 1/n^{d'}$ for any constant $d' \geq 1$. We store all the edges of all the suffix trees in one such dictionary. At all times, we keep $\Theta(w)$ dummy-elements in the dictionary to ensure that we get good probability bounds in terms of w , and we choose d' large enough that any sequence of $O(w)$ operations (e.g., the construction of any one of our suffix trees) runs in $O(w)$ time with probability $1 - 1/w^{d'+\epsilon}$.

Universal Hashing

Before we prove Lemma 5 we restate some basic facts about universal hashing, introduced by Carter and Wegman [9]. Let $M, m > 0$ be integers, \mathcal{H} be a set of functions $[0, M] \rightarrow [0, m]$, and $h \in \mathcal{H}$ be selected uniformly at random. Then \mathcal{H} is *universal* if $P[h(x) = h(y) \mid x \neq y] \leq 1/m$. Let $R \subseteq [0, M]$ and $|R| = r$. It follows from the union bound that h has a *collision* on R with probability at most

$$P[h(x) = h(y) \text{ for some } x \neq y] \leq \sum_{x \neq y \in R} P[h(x) = h(y)] = \frac{r(r-1)}{2} \cdot \frac{1}{m} < \frac{r^2}{m}. \quad (1)$$

In particular, if $m = r^c$ for constant $c \geq 1$ then h is *injective* (i.e., has no collisions) on R with probability at least $1 - 1/r^{c-2}$. Carter and Wegman gave several classes of universal hash functions from which we can sample a function uniformly at random in constant time.

Fast Suffix Tree Construction

We now prove Lemma 5, showing how to construct our suffix trees in linear time with high probability.

► **Lemma 5.** *Given a string \mathcal{K} of length $k \leq 2w$ there is an algorithm that uses $O(k + w/\log w)$ space and constructs the suffix tree over \mathcal{K} in $O(k)$ time with probability $1 - 1/w^{1+\epsilon}$ for some $\epsilon > 0$.*

Proof. Let $\sigma = \{\mathcal{K}[i] \mid i \in [1, k]\} \subseteq \Sigma$ be the alphabet of \mathcal{K} . We show how to, in $O(k)$ time, find a function $h : \Sigma \rightarrow [1, k^{O(1)}]$ such that h is injective on σ with probability at least $1 - 1/w^{1+\epsilon}$. If h is injective on σ , we can construct the suffix tree over \mathcal{K}' where $\mathcal{K}'[i] = h(\mathcal{K}[i])$ in time $O(\text{sort}(k, k^{O(1)})) = O(k)$ using radix sort. After the tree is constructed we can substitute for the original alphabet in linear time. Therefore, the construction algorithm finishes in $O(k)$ time with probability at least $1 - 1/w^{1+\epsilon}$ (otherwise we make no guarantee on the construction time and we can build the suffix tree in any way).

For some m to be determined later, let $f : \Sigma \rightarrow [1, m]$ be chosen uniformly at random from a class of universal hash functions. By Equation 1, the probability that f has a collision on σ is

$$P[f \text{ has collisions on } \sigma] < \frac{|\sigma|^2}{m} \leq \frac{k^2}{m}.$$

We divide into the cases of large trees ($k \geq w^{1/5}$) and small trees ($k < w^{1/5}$). If k is large then $w^{1/5} \leq k \leq 2w$, and we set $m = w^4$ so the probability that f has a collision is at most

$$\frac{k^2}{m} \leq \frac{(2w)^2}{w^4} = \frac{4}{w^2} \leq \frac{1}{w^{1+\epsilon}}$$

for some $\epsilon > 0$. We check whether f is injective by sorting the set $\{(x, f(x)) \mid x \in \sigma\}$ with respect to the $f(\cdot)$ -values and checking if two consecutive elements $(x, f(x))$ and $(y, f(y))$ have $x \neq y$ and $f(x) = f(y)$. This takes time $O(\text{sort}(k, w^4)) = O(k)$ using radix sort since $k \geq w^{1/5}$. If f is injective we set $h = f$, concluding the proof of the large case.

If k is small then we allocate an array A of length $w/\log w$ in constant time. For simplicity we assume that A is initialized such that $A[i] = 0$ for all i . This can be avoided using standard constant-time initialization schemes; assume each entry in A contains an arbitrary value initially. We maintain two other arrays B and C such that if we have written a value to $A[i]$

at least once then $A[i]$ is a pointer to some $B[j]$, $B[j]$ is a pointer to $A[i]$, and $C[j]$ stores the value most recently written to $A[i]$. From this we can determine if $A[i]$ has been initialized (check if the pointers match), and if it has not we can initialize it in constant time.

Then we set $m = w/\log w$ such that the probability that f has a collision is no more than

$$\frac{k^2}{m} < \frac{w^{2/5}}{w/\log w} = \frac{\log w}{w^{3/5}} = \frac{\log w}{w^{1/2}} \cdot \frac{1}{w^{1/10}} \leq \frac{1}{w^{1/10}}$$

for $w \geq 16$. We check if f is injective on σ by for each character x in \mathcal{K} setting $A[f(x)] = x$ and seeing if two distinct characters hash to the same index. If f is injective we then arbitrarily assign the values $1, \dots, |\sigma|$ to the now non-zero indices of A and let $h(x) = A[f(x)]$ (at this point we know σ since it is equal to the number of entries in A that we modified). To boost the probability of success we run this algorithm up to eleven times with independent choices for f . The probability that all of them fail is at most $1/w^{11/10} \leq 1/w^{1+\epsilon}$ concluding the proof for the small case. ◀

In conjunction with Theorems 3 and 4, this proves Theorem 1.

6 Conclusion and Future Work

We have studied two variants of the streaming sliding window string indexing problem; the timely variant, where queries must be answered immediately, and the delayed variant where a query may be answered at any point within the next δ characters received, for a specified parameter δ . For a sliding window of size w we have given an $O(w)$ space data structure that, in the timely variant, supports updates in $O(\log w)$ time with high probability and queries in $O(\log w)$ time with high probability per character in the pattern; each occurrence is reported in additional constant time. For the delayed variant we improved these bounds to $O(\log(w/\delta))$, where each occurrence is still reported in constant time.

One open problem is whether these bounds can be improved. Another is to find efficient solutions when queries may be interleaved with new updates to the stream. That is, while you are streaming a pattern, new characters of S might arrive that move the current window.

References

- 1 Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *Proc. 31st ISAAC*, volume 181, pages 63:1–63:16, 2020. doi:10.4230/LIPIcs.ISAAC.2020.63.
- 2 Amihood Amir and Itai Boneh. Dynamic suffix array with sub-linear update time and poly-logarithmic lookup time. *CoRR*, abs/2112.12678, 2021. arXiv:2112.12678.
- 3 Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014. doi:10.1137/110836377.
- 4 Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005. doi:10.1007/11575832_9.
- 5 Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008. URL: <http://dl.acm.org/citation.cfm?id=1347082.1347201>.
- 6 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic Indexing for Packed Strings. In *Proc. 28th CPM*, volume 78, pages 6:1–6:11, 2017. doi:10.4230/LIPIcs.CPM.2017.6.

- 7 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. doi:10.1016/j.jda.2012.07.003.
- 8 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. doi:10.3390/a11080118.
- 9 Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.
- 10 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. *Algorithmica*, 72(2):450–466, 2015. doi:10.1007/s00453-013-9860-6.
- 11 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In *Proc. 17th ICALP*, pages 6–19, 1990. doi:10.1007/BFb0032018.
- 12 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
- 13 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.
- 14 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. 26th CPM*, pages 160–171, 2005. doi:10.1007/978-3-319-19929-0_14.
- 15 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.
- 16 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th STOC*, pages 135–143. ACM, 1984. doi:10.1145/800057.808675.
- 17 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proc. 34th STOC*, pages 602–608, 2002. doi:10.1145/509907.509993.
- 18 Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *J. Discrete Algorithms*, 2(1):33–51, 2004. doi:10.1016/S1570-8667(03)00064-9.
- 19 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. 54th STOC*, pages 1657–1670, 2022. doi:10.1145/3519935.3520061.
- 20 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 21 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *53rd FOCS*, pages 283–292, 2012. doi:10.1109/FOCS.2012.79.
- 22 S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994. doi:10.1145/195058.195170.
- 23 Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017. doi:10.1007/s00453-016-0199-7.
- 24 N. Jesper Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Lund University, Sweden, 1999. URL: <http://lup.lub.lu.se/record/19255>.
- 25 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.
- 26 Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996. doi:10.1007/s002360050048.
- 27 Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *J. Discrete Algorithms*, 8(2):241–257, 2010. doi:10.1016/j.jda.2009.02.007.
- 28 M Senft. Suffix tree for a sliding window: An overview. In *Proc. WDS*, volume 5, pages 41–46, 2005.

4:18 Sliding Window String Indexing in Streams

- 29 Martin Senft and Tomáš Dvoraák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008. doi:10.1007/978-3-540-89097-3_12.
- 30 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 31 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

Faster Algorithms for Computing the Hairpin Completion Distance and Minimum Ancestor

Itai Boneh ✉

Reichman University, Herzliya, Israel

Dvir Fried ✉

Bar-Ilan University, Ramat-Gan, Israel

Adrian Miclăuș ✉

Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Alexandru Popa ✉

Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Abstract

Hairpin completion is an operation on formal languages that has been inspired by hairpin formation in DNA biochemistry and has many applications especially in DNA computing. Consider s to be a string over the alphabet $\{A, C, G, T\}$ such that a prefix/suffix of it matches the reversed complement of a substring of s . Then, in a hairpin completion operation the reversed complement of this prefix/suffix is added to the start/end of s forming a new string.

In this paper we study two problems related to the hairpin completion. The first problem asks the minimum number of hairpin operations necessary to transform one string into another, number that is called *the hairpin completion distance*. For this problem we show an algorithm of running time $O(n^2)$, where n is the maximum length of the two strings. Our algorithm improves on the algorithm of Manea (TCS 2010), that has running time $O(n^2 \log n)$.

In *the minimum distance common hairpin completion ancestor* problem we want to find, for two input strings x and y , a string w that minimizes the sum of the hairpin completion distances to x and y . Similarly, we present an algorithm with running time $O(n^2)$ that improves by a $O(\log n)$ factor the algorithm of Manea (TCS 2010).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases dynamic programming, incremental trees, exact algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.5

Funding This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0253, within PNCDI III.

Itai Boneh: Israel Science Foundation grant 810/21.

Dvir Fried: Supported in part by ISF grant no. 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

1 Introduction

1.1 Motivation and informal problem definition

Hairpin completion is an operation on formal languages that has been inspired by hairpin formation in DNA biochemistry and has many applications especially in DNA computing [11, 12, 14, 15]. This operation has been inspired by three biological principles: Watson-Crick complementarity, DNA annealing and DNA lengthening through polymerases. The DNA chain is a molecule consisting of two intertwined strands, each strand being composed by nucleotides: A(Adenine), C(cytosine), G(guanine) and T(thymine). The two strands which



© Itai Boneh, Dvir Fried, Adrian Miclăuș, and Alexandru Popa; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 5; pp. 5:1–5:18

Leibniz International Proceedings in Informatics

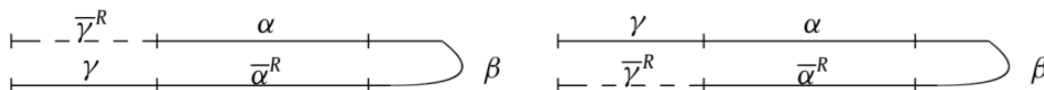


LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

form the DNA molecule are kept together by the hydrogen bond between the bases: A bonds with T and C with G. This paradigm is usually referred to as the Watson-Crick complementarity [25].

Another important bio-chemical principle is annealing, the process of fusing two single stranded molecules by complementary base. DNA lengthening through polymerases is a phenomenon that produces a complete double stranded DNA molecule as follows: one starts with two single strands such that one (called primer) is bonded to a part of the other (called template) through Watson-Crick complementarity and a polymerization buffer with many copies of the four nucleotides. The polymerases will then concatenate to the primer by complementing the template [22].

We now begin to informally explain the hairpin completion operation and how it can be related to the biological concepts presented above. Consider s to be a string over the alphabet $\{A, C, G, T\}$ such that a prefix/suffix of it matches to the reversed complement of a substring of s . Then, the reversed complement of this prefix/suffix is added to the beginning/ending of s forming a new string as can be visualized in Figure 1. The mathematical expression of this hypothetical situation defines the hairpin completion operation. Starting with a single string, one can generate a set of strings using this formal operation: via hairpin completion, a new string can be created for each possible pairing between a prefix or suffix and a complementary substring. In addition, one could be interested in knowing how many iterations of hairpin completion are required to transform one string into another. In this way, the hairpin completion distance between two strings was defined as the minimum number of times we must iterate the hairpin completion operation, starting from one of the two string, in order to obtain the other. Further, one can also be interested in finding for two strings, a common ancestor that minimizes the sum of the hairpin completion distances to those strings. This ancestor is called minimum distance common hairpin completion ancestor.



■ **Figure 1** An illustration of the left and right hairpin completion operations.

1.2 Previous and related work

The hairpin completion operation has been introduced by Cheptea, Martin-Vide and Mitrana [4]. In several papers, the hairpin completion and other familiar operations have been studied [3, 5, 6, 7, 8, 9, 13, 17, 19, 20, 21, 22, 23, 24].

Hairpin reduction [3, 22, 23] was introduced as an inverse operation for hairpin completion. The hairpin reduction of a string x consists of all strings y such that x can be obtained from y by hairpin completion. Further, two variants of hairpin completion were considered, as they seem more appropriate for practical implementation: hairpin lengthening and bounded hairpin completion [9, 19, 21]. The first variant consist of adding a prefix or a suffix of γ . The second variant assumes that the length of the added prefix or suffix is bounded by a constant. Besides the algorithmic aspects, hairpin completion operation has been studied from the language theory point of view in several papers [5, 6, 8, 13, 17].

Manea and Mitrana introduced the minimum distance common k -hairpin completion ancestor of two strings in [22] where they presented a cubic time algorithm to compute the ancestor. Afterwards, Manea, Martin-Vide, and Mitrana [20] suggested a cubic time

algorithm to tackle the k -hairpin completion distance problem. In addition, in [18] improved the time complexity to $O(n^2 \log n)$ to both problems, where n is the length of the longest string.

1.3 Our results

The focus of this paper is on two algorithmic problems related to iterated hairpin completion: k -hairpin completion distance and minimum distance common k -hairpin completion ancestor. Our main results are improving the upper bound on both problems with a $\log n$ factor, from $O(n^2 \log n)$ to $O(n^2)$. For the k -hairpin completion distance, our speedup is based on using incremental tree, a data structure proposed by Kaplan and Shafrir [10] which can support in constant time the following operations in a weighted tree: return the edge with minimum weight on a path and add a leaf to the tree. Our algorithm for finding a minimum distance k -hairpin completion ancestor of two strings (x, y) is based on dynamic programming technique presented in [18]. As in [18], we are interested in constructing the table DP_x , where $DP_x[i][j]$ represents the minimum number of k -hairpin completion operations to transform $x[i \dots j]$ into x . Similarly, we would like to compute a table DP_y . Our speedup relies in an $O(n^2)$ time algorithm for computing these tables by rephrasing the problem of computing DP_x in terms of shortest distances in a graph and replacing the segment tree used in [18] with doubly linked list and changing the order we process the cells in the matrix.

2 Preliminaries

We start with basic notations related to strings. An alphabet Σ is a finite, non-empty set of symbols. Throughout this paper, we mostly discuss strings over the alphabet $\Sigma = \{A, C, G, T\}$. For a letter $x \in \Sigma$, we denote as \bar{x} the letter in Σ that is complementary to x . For the previously mentioned alphabet, we have $\bar{A} = T$ and $\bar{C} = G$. The set of all strings over an alphabet Σ is denoted by Σ^* . The empty string is denoted as λ , and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. Given a string $w \in \Sigma^*$, we denote by $|w|$ its length. If $w = xy$, $x, y \in \Sigma^*$ then x is called prefix and y a suffix. For a string w , $w[i \dots j]$ denotes the substring of w starting at position i and ending at position j , $1 \leq i \leq j \leq |w|$. Given a string $s \in \Sigma^+$, we denote by $\bar{s} = \bar{s}_1 \bar{s}_2 \dots \bar{s}_{|s|}$ the complement of the string s and s^R the reversed string of s , i.e. $s^R = s_{|s|} s_{|s|-1} \dots s_1$.

Incremental tree is a data structure introduced by Kaplan and Shafrir [10] based on a similar structure of Alstrup and Holm [1] for the level ancestor problem, to maintain a rooted tree T , with an integer weight on each edge, such that the following operations are supported in $O(1)$ amortized time:

- $\text{add-leaf}_T(v, w, c)$: Add a new leaf v with parent w to T . The weight of the edge (v, w) is c .
- $\text{add-root}_T(v, c)$: Add a new root v to T . The old root (r) becomes a child of v and the weight of edge (r, v) is c .
- $\text{min-edge}_T(v, w)$: Returns the edge with minimum weight on the path from v to w .
- $\text{change-weight}_T(v, c)$: v is a leaf or v 's parent is the root of T . Changes the weight of the edge between v and its parent to c .

From this data structure we will use just add-leaf_T and min-edge_T operations.

2.1 Hairpin Operations

For a string $x \in \Sigma^+$ and a positive integer $k \in \mathbb{N}$, k -hairpin completion is a family of transformations that can be applied to x . When applying a *left k -hairpin completion*, we select a non-empty suffix γ of x such that x can be partitioned into $x = \alpha\beta\alpha^R\gamma$ with

5:4 Faster Algorithms for Computing HCD and MDCHCA

$\alpha, \beta, \gamma \in \Sigma^+$ and $|\alpha| = k$. We execute the left hairpin operation by appending $\overline{\gamma^R}$ to the beginning of s . Formally, the set of strings that can be obtained from x by applying a single left k -hairpin operation is denoted as

$$HCL_k(x) = \{\overline{\gamma^R}x \mid x = \alpha\beta\overline{\alpha^R}\gamma, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

A *right k -hairpin completion* is defined in a symmetrical manner and the set of strings that can be obtained from s by applying a single right k -hairpin completion operation is denoted as

$$HCR_k(x) = \{x\overline{\gamma^R} \mid x = \gamma\alpha\beta\overline{\alpha^R}, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

► **Example 1.** The string $s = GAATCT$ can be partitioned into $\alpha = GA$, $\beta = A$, $\overline{\alpha^R} = TC$ and $\gamma = T$. Applying the left hairpin completion operation on s with this partitioning yields the string $AGAATCT$. Also, s can be partitioned into $\gamma = GA$, $\alpha = A$, $\beta = TC$, $\overline{\alpha^R} = T$ and by applying right hairpin completion operation we obtain $GAATCTTC$.

Collectively, the set of strings that can be obtained from x either by applying a right or a left k -hairpin completion operation is denoted as

$$HC_k(x) = HCL_k(x) \cup HCR_k(x)$$

The hairpin completion is the variant of the k -hairpin completion where we do not place a bound on the length of prefix. The hairpin completion of x is defined by:

$$HC(x) = \bigcup_{k \geq 1} HC_k(x)$$

We extend the notation of hairpin completion to sets of strings in the following way, for a set $L \subseteq \Sigma^*$ and a positive integer k ,

$$HC_k(L) = \bigcup_{x \in L} HC_k(x) \quad HC(L) = \bigcup_{x \in L} HC(x)$$

For every non negative integers k, i and string $x \in \Sigma^+$, we denote as $HC_k^i(x)$ the set of strings that can be obtained from x using exactly i k -hairpin completion operations and $HC_k^*(x)$ as the set of strings that are obtainable from x using any number of k -hairpin completion operations. Similarly, we denote as $HC^i(x)$ and $HC^*(x)$ the sets of strings obtainable from x by applying i (resp. any number) of hairpin operations, respectively. Formally,

$$HC_k^0(x) = \{x\} \quad HC_k^{i+1}(x) = HC_k(HC_k^i(x)) \quad HC_k^*(x) = \bigcup_{i \geq 0} HC_k^i(x)$$

$$HC^0(x) = \{x\} \quad HC^{i+1}(x) = HC(HC^i(x)) \quad HC^*(x) = \bigcup_{i \geq 0} HC^i(x)$$

$$HC_k^*(L) = \bigcup_{x \in L} HC_k^*(x) \quad HC^*(L) = \bigcup_{x \in L} HC^*(x)$$

► **Definition 2 (k -Hairpin Completion Common Ancestor).** A string w is a common k -hairpin completion ancestor of two strings x and y if $\{x, y\} \subseteq HC_k^*(w)$. We denote the set of common k -hairpin ancestors of x and y as $HCA_k(x, y)$.

► **Definition 3** (*k*-Hairpin Completion Distance). *Given two strings x and y such that $|x| \leq |y|$, the k -hairpin completion distance between x and y is the minimal number of k -hairpin operations required to obtain y from x . Formally*

$$HCD_k(x, y) = \begin{cases} \min\{p \mid x \in HC_k^p(y)\} \\ \infty, x \notin HC_k^*(y) \end{cases}$$

► **Definition 4** (Minimum Distance k -hairpin Completion Ancestor). *For two strings $x, y \in \Sigma^*$, a k -hairpin completion ancestor $w \in HCA_k(x, y)$ is a minimum distance k -hairpin completion ancestor of x and y if $\forall w' \in HCA_k(x, y)$ it holds that $HCD_k(w, x) + HCD_k(w, y) \leq HCD_k(w', x) + HCD_k(w', y)$, i.e. w minimizes the sum of the k -hairpin completion distances from x and from y .*

► **Definition 5** (Border). *Given a string $s[1 \dots n] \in \Sigma^+$, $Border(s)$ is the length of the longest prefix of the string s which is also a complemented reversed suffix of this string. Formally, $Border(s) = \max(\{t \mid s[1 \dots 1+t-1] = \overline{s[n-t+1 \dots n]^R}\} \cup \{0\})$. This definition can be easily extended for any substring $s[i \dots j]$ in the following way: $Border(s[i \dots j]) = \max(\{t \mid s[i \dots i+t-1] = \overline{s[j-t+1 \dots j]^R}\} \cup \{0\})$*

► **Remark 6.** Note that the above definition for border is different than the common definition, which is usually the largest prefix of x which is also a suffix of x .

Since in the k -hairpin completion operation we have to make sure that $|\alpha| = k$, we introduce the definition of k -Border.

► **Definition 7** (k -Border). *Given a string $s \in \Sigma^+$, k -Border(s) = $\max(Border(s) - k, 0$).*

Hairpin reduction is the inverse operation of hairpin completion. The hairpin reduction of a string x consists of all strings y such that x can be obtained from y by hairpin completion. For a string $x \in \Sigma^+$ and a positive integer $k \in \mathbb{N}$, k -hairpin reduction is a family of transformations that can be applied to x . When applying a *left hairpin reduction*, we select a non-empty prefix γ of x such that x can be partitioned into $\gamma\alpha\beta\alpha^R\gamma^R$ with $\alpha, \beta, \gamma \in \Sigma^+$ and $|\alpha| = k$. We execute the left hairpin reduction operation by deleting γ . Formally, the set of strings that can be obtained from x by applying a single left k -hairpin reduction operation is denoted as

$$HRL_k(x) = \{\alpha\beta\alpha^R\gamma^R \mid x = \gamma\alpha\beta\alpha^R\gamma^R, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

A *right k -hairpin reduction* operation is defined in a symmetrical manner and the set of strings that can be obtained from x by applying a single right k -hairpin reduction operation is denoted as

$$HRR_k(x) = \{\gamma\alpha\beta\alpha^R \mid x = \gamma\alpha\beta\alpha^R\gamma^R, |\alpha| = k, \alpha, \beta, \gamma \in \Sigma^+\}$$

The set of strings that can be obtained from x either by applying a left or a right k -hairpin reduction operation is denoted as

$$HR_k(x) = HRL_k(x) \cup HRR_k(x)$$

The hairpin reduction is the variant of the k -hairpin reduction where we do not place a bound on the length of prefix. The hairpin reduction of x is defined by:

$$HR(x) = \bigcup_{k \geq 1} HR_k(x)$$

We make the following observation.

► **Observation 8.** Let $x[1 \dots n]$ be a string with k -Border l .

$$HRL_k(x) = \bigcup_{j \in [1 \dots l]} \{x[j + 1 \dots n]\} \quad HRR_k(x) = \bigcup_{j \in [1 \dots l]} \{x[1 \dots n - j]\}$$

$$HR_k(x) = \bigcup_{j \in [1 \dots l]} \{x[j + 1 \dots n], x[1 \dots n - j]\}$$

Now we are ready to introduce the problems that we study in this paper.

► **Problem 1** (Hairpin completion distance). Let Σ be the alphabet and $x, y \in \Sigma^+$. Compute $HCD_k(x, y)$.

► **Problem 2** (Minimum distance common hairpin completion ancestor). Let Σ be the alphabet and $x, y \in \Sigma^+$. Compute a minimum-distance common k -hairpin completion ancestor of x, y .

2.2 Suffix Tree and Extension queries

The *suffix tree* [26] is a useful string data structure.

► **Definition 9.** Let S_1, \dots, S_k be strings over alphabet Σ and let $\$ \notin \Sigma$.

A trie of strings S_1, \dots, 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 a 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 $S = S[1], \dots, S[n]$ be a string over alphabet Σ . Let $\{S_1, \dots, S_n\}$ be the set of suffixes of S , where $S_i = S[i \dots n]$, $i = 1, \dots, n$. A suffix tree of S is the compacted trie of the suffixes S_1, \dots, 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 10** (Weiner [26]). 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 $\sigma = \min(|\Sigma|, n)$.

For two strings $S[1 \dots n]$ and $T[1 \dots m]$, a string $P[1 \dots p]$ is a *common prefix* of S and T if $S[1 \dots p] = T[1 \dots p] = P$. We say that P is the *longest common prefix* (LCP) of S and T if P is a common prefix and $m = p$ or $n = p$ or $S[p + 1] \neq T[p + 1]$. Similarly, a string $A[1 \dots a]$ is a *common suffix* of S and T if $S[n - a + 1 \dots n] = T[m - a + 1 \dots m] = A$. A is the *longest common suffix* (LCS) if $n = a$ or $m = a$ or $S[n - a] \neq T[m - a]$. Collectively, we refer to LCP and LCS as longest common extensions (LCE).

By preprocessing the suffix tree of a string S for level ancestor queries [2], we can obtain the following.

► **Lemma 11** (Longest Common Extension Data Structure). A string S can be preprocessed in $O(n)$ time to support the following queries in $O(1)$ time.

1. $LCP(i, j)$ - return the length of the longest common prefix of $S[i \dots n]$ and $S[j \dots n]$
2. $LCS(i, j)$ - return the length of the longest common suffix of $S[1 \dots i]$ and $S[1 \dots j]$

By constructing the above data structure for the string $x\$x^R$ with $\$ \notin \Sigma$, we obtain the following.

► **Corollary 12.** *We can process a string $S[1 \dots n]$ in linear time to construct a data structure for answering the following query in $O(1)$ time.*

k -Border($S[i \dots j]$) – Return the length of the k -Border of $S[i \dots j]$.

3 Hairpin completion distance

In this section we study Problem 1.

Our algorithm is based on the dynamic programming technique presented in [18]. For the sake of clarity, we briefly describe this technique. Without loss of generality, we assume that $|x| \leq |y|$ and $n = |y|$, $m = |x|$. We are interested in computing a dynamic programming table $DP[n][n]$ with dimensions $n \times n$. For every two indices $1 \leq i \leq j \leq n$, we define $DP[i][j]$ to be the minimum number of k -hairpin completion operations to transform x into $y[i \dots j]$. Formally, $DP[i][j] = HCD_k(x, y[i \dots j])$.

► **Definition 13.** *Given two non-negative integers i, j ($1 \leq i \leq j \leq n$), L_j represents the DP values of all strings that can generate the substring $y[i \dots j]$ through a single left k -hairpin completion operation (elements of the set $HRL_k(y[i \dots j])$).*

▷ Claim 14. $L_j = \{DP[i+1][j], \dots, DP[i+l][j]\}$ where l is the k -Border of $y[i \dots j]$.

► **Definition 15.** *Given two non-negative integers i, j ($1 \leq i \leq j \leq n$), R_i represents the DP values of all strings that can generate the substring $y[i \dots j]$ through a single right k -hairpin completion operation (elements of the set $HRR_k(y[i \dots j])$).*

▷ Claim 16. $R_i = \{DP[i][j-l], \dots, DP[i][j-1]\}$ where l is the k -Border of $y[i \dots j]$.

Correctness of Claim 14 and Claim 16 is based on Observation 8.

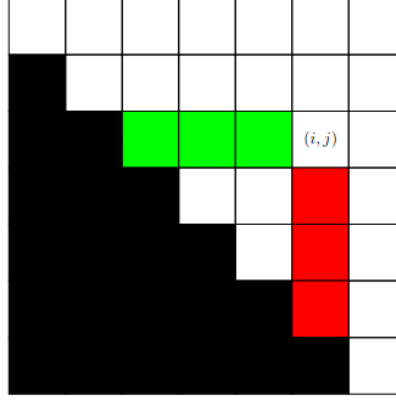
► **Lemma 17.** *Given two non-negative integers i, j ($1 \leq i \leq j \leq n$), we have that $DP[i][j] = \min(\min L_j, \min R_i) + 1$.*

For the proof of Lemma 17 we refer to [18].

All positions in DP are initialized with ∞ . We start by considering the base cases. These are represented by all subsequences $y[i \dots j] = x$. To determine them, we use any pattern matching algorithm which runs in linear time, for example KMP [16] and set $DP[i][j] = 0$. Analyzing the elements of the sets L_j and R_i , it can be seen that they actually represent continuous blocks from line i or column j . Thus, determining the minimum values from each of those sets is a range minimum query.

► **Definition 18.** *Given two non-negative integers i, j ($1 \leq i \leq j \leq n$), DSL_j represents the data structure that keeps the DP values of column j and DSR_i represents the data structure that keeps the DP values of row i . (Note that we don't have to keep the values below the main diagonal)*

Naively, DSL_j and DSR_i could be arrays, which leads to constant update time, but linear query time. The overall complexity of the algorithm with this naive implementation is $O(n^3)$. In [18], the algorithm is implemented using segment trees, which leads to a logarithmic time for queries and for updates.



■ **Figure 2** We compute the DP matrix in increasing order of difference $j - i$ (parallel with the main diagonal). Red line represents DSL_j and the green line DSR_i .

Considering that the update operations are append-like, i.e. they are only done after the first/last index of DSL_j and DSR_i , we propose using an incremental tree. The advantages of this approach consist in the fact that this structure can perform query and update operations in constant time. Practically, we keep an incremental tree for each row and column. A row or a column in the matrix represents a particular case of a tree, more precisely a chain. For the range minimum query needed in the computation of $DP[i][j]$ we use incremental tree's min-edge_T operation. After we compute the $DP[i][j]$ value, we have to add to DSR_i and DSL_j . This can be done by using the add-leaf_T operation.

■ **Algorithm 1** An $O(n^2)$ algorithm for Problem 1.

Input: $x, y \in \Sigma^+$
Output: $HCD_k(x, y)$

- 1: $DP[i][j] = \infty, \forall 1 \leq i \leq j \leq n$
- 2: Find all pairs (i, j) such that $x = y[i \dots j]$ and set $DP[i][j] = 0$.
- 3: **for** $len \leftarrow m + 1$ to n **do**
- 4: **for** $i \leftarrow 1$ to $n - len + 1$ **do**
- 5: $j \leftarrow i + len - 1$
- 6: **if** $DP[i][j] = \infty$ **then**
- 7: $x \leftarrow \text{min-edge}_{DSR_i}(j - k\text{-Border}(s[i \dots j]), j - 1)$
- 8: $y \leftarrow \text{min-edge}_{DSL_j}(i + 1, i + k\text{-Border}(s[i \dots j]))$
- 9: $DP[i][j] = \min(x, y) + 1$
- 10: **end if**
- 11: $\text{add-leaf}_{DSR_i}(j, j - 1, DP[i][j])$
- 12: $\text{add-leaf}_{DSL_j}(i, i + 1, DP[i][j])$
- 13: **end for**
- 14: **end for**
- 15: **return** $DP[1][n]$

► **Theorem 19.** *Algorithm 1 solves Problem 1 in $O(n^2)$.*

Proof.

Correctness. We prove the correctness of the algorithm by induction over the algorithm execution. The base cases correspond to the substrings $y[i \dots j] = x$. In these cases, $DP[i][j] = 0$ because no operation is needed to convert x to $y[i \dots j]$. Suppose we want to calculate the value of $DP[i][j]$. We remind that $DP[i][j] = \min(\min L_j, \min R_i) + 1$. We can rewrite the elements of the set R_i in the following form $DP[i][p]$ with $j - \text{Border}(i, j) + k \leq p < j$ and the elements of the set L_j in the form $DP[s][j]$ with $i < s \leq i + \text{Border}(i, j) - k$. Taking into account the iteration order (increasing according to the difference $j - i$) and $j > i$, we obtain the following inequalities: $j - i > j - s$ and $j - i > p - i$. Thus, it is guaranteed that when we want to calculate $DP[i][j]$ all the necessary values are already calculated.

Complexity. Line 1 runs in $O(n^2)$ and Line 2 in $O(n + m)$. For each cell above the main diagonal we have two queries and two updates both done in $O(1)$ amortized time. The overall time complexity is therefore $O(n^2)$. ◀

4 Minimum distance common hairpin completion ancestor

In this section we study Problem 2.

Our algorithm is based on the dynamic programming technique described in [18], but we replace the segment tree with a linked list and change the order of processing the cells in the matrix. Without loss of generality, we assume that $|x| \geq |y|$ and $n = |x|, m = |y|$. As in [18], we are interested in constructing the table $DP_x[1 \dots n][1 \dots n]$ with $DP_x[i][j] = HCD_k(x[i \dots j], x)$. Similarly, we would like to compute a table $DP_y[1 \dots m][1 \dots m]$ with $DP_y[i][j] = HCD_k(y[i \dots j], y)$. Our speedup relies on an $O(n^2)$ time algorithm for computing DP_x and DP_y .

We are interested in rephrasing the problem of computing DP_x in terms of shortest distances in a graph. We present the following definition.

► **Definition 20 (Hairpin Deletion Graph).** For a string $x[1 \dots n]$, we define the Hairpin Deletion Graph $G_h(x) = (V, E)$ of x as follows.

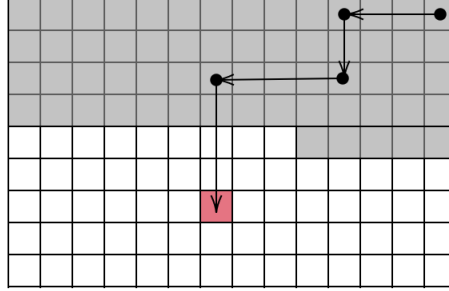
- $V = \{x[i \dots j] \mid 1 \leq i \leq j \leq n\}$ is the set of substrings of x .
- $E = \{(x[i \dots j], x[a \dots b]) \mid x[i \dots j] \in HR_k(x[a \dots b])\}$ I.e. there is a directed edge from substring A to the substring B if A can be obtained from B by applying a single hairpin completion operation.

It is easy to see that $HCD_k(x[i \dots j], x)$ is exactly the length of the shortest path from $x[1 \dots n]$ to $x[i \dots j]$ in $G_h(x)$. Following Observation 8, we present the following characterization of the edges in $G_h(x)$.

► **Corollary 21.** Let $x[1 \dots n]$ be a string and let $A = x[i \dots j]$ be a substring of x with k -Border length l . The set of edges emerging from A in $G_h(x)$ is

$$E_A = \bigcup_{p \in [1 \dots l]} \{(A, x[i + p \dots j]), (A, x[i \dots j - p])\}$$

We call edges from $A = x[i \dots j]$ to a suffix $[i + p \dots j]$ a downward edge and an edge from A to a prefix $[i \dots j - p]$ a leftward edge. When a path in $G_h(x)$ uses a downward (resp. left) edge, we say that it takes a step down (resp. leftward). For example, if $P = (v_1, v_2 \dots v_z)$ is a path in $G_h(x)$, and the edge (v_{z-1}, v_z) is a downward (resp. leftward) edge, we say that P ends with a step left (resp. down).



■ **Figure 3** A demonstration of a restricted path to the red square. The grey squares resemble cells that precede the cell (i, j) .

Then, the algorithm computes the cells of DP_x row by row from top to bottom, iterating a row in decreasing order of the columns. Formally, when iterating the cell $DP_x[i][j]$, the algorithm have already computed the cells $DP_x[a][b]$ with $a < i$ and the cells $DP_x[i][b]$ with $b > j$. The order of the iteration implies a total order on the pairs $i, j \in [n] \times [n]$.

► **Definition 22** (Iteration Order). For two pairs of integers $(i_1, j_1), (i_2, j_2) \in [n] \times [n]$, we say that (i_1, j_1) precedes (i_2, j_2) (denoted as $(i_1, j_1) < (i_2, j_2)$) if the cell $DP_x[i_1][j_1]$ is iterated before $DP_x[i_2][j_2]$ by our algorithm. Similarly, we say (i_2, j_2) proceeds (i_1, j_1)

We proceed to introduce a useful concept used by the algorithm.

► **Definition 23** (Restricted Path). For a string $x[1 \dots n]$ and integers $i, j \in [n]$, a path $P = (x, v_1, v_2 \dots, v_z, A)$ from x to A in $G_h(x)$ is (i, j) -restricted if for every $r \in [z]$ we have $v_r = x[a_r \dots b_r]$ such that (a_r, b_r) precedes (i, j) . For $(i, j) = (0, 0)$, we say that there is no $(0, 0)$ -restricted path.

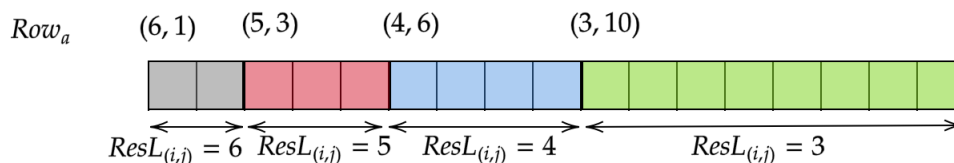
For integer pairs $(i, j), (a, b) \in [n][n]$ such that $(i, j) < (a, b)$, we denote as $ResL_{(i,j)}[(a, b)]$ the length of the shortest (i, j) -restricted path from x to $x[a \dots b]$ in $G_h(x)$ that ends with a step left. Similarly, we denote as $ResD_{(i,j)}[(a, b)]$ the length of the shortest (i, j) -restricted path from x to $x[a \dots b]$ in $G_h(x)$ that ends with a step down.

We make the following observations regarding the structure of $ResL_{(i,j)}[(a, b)]$ and $ResD_{(i,j)}[(a, b)]$.

► **Lemma 24.** For every $i, j \in [n]$ and $a, b \in [n] \times [n - 1]$ such that $(i, j) < (a, b)$ it is satisfied that $ResL_{(i,j)}[(a, b + 1)] \leq ResL_{(i,j)}[(a, b)]$.

Proof. If there is no (i, j) -restricted path that ends with a step leftwards from $(0, 0)$ to (a, b) , the claim is vacuously true. Otherwise, let $P = (1, n), (x_1, y_1), (x_2, y_2) \dots (x_d, y_d), (a, b)$ be the shortest (i, j) -restricted path from $(1, n)$ to (a, b) that ends with a step to the left. Since P ends with a step leftwards, we have $x_d = a$ and there is an edge from (a, y_d) to (a, b) . According to Corollary 21, there is also an edge from (a, y_d) to $(a, b + 1)$. Therefore, we can replace (a, b) with $(a, b + 1)$ in P to obtain an (i, j) -restricted path P' with length $|P|$ from $(1, n)$ to $(a, b + 1)$. ◀

The following symmetric statement can be proven in a similar manner.



■ **Figure 4** The list Row_a above the a 'th row of DP_x . Every pair (δ, β) appears above the cell (a, β) . The content of Row_a implies $ResL(i, j)$ for the cells in the a 'th row. Every cell in the green region has the boundary predecessor $(3, 10)$ in Row_k . So for every b in the green region, $ResL(i, j)[(a, b)] = 3$.

► **Lemma 25.** For every $i, j \in [n]$ and $a, b \in [n-1] \times [n]$ such that $(i, j) < (a, b)$ it is satisfied that $ResD_{(i, j)}[(a, b)] \leq ResD_{(i, j)}[(a+1, b)]$.

Lemma 24 and Lemma 25 suggest that the values of $ResL(i, j)$ (resp. $ResD(i, j)$) in every row (resp. column) are monotonic.

For every row $k \in [n]$ of DP_x , the algorithm maintains a corresponding double-sided linked list Row_k . Similarly, for every column $k \in [n]$ the algorithm maintains a list Col_k . Conceptually, Row_k (resp. Col_k) compactly represents the values $ResL(i, j)[(a, b)]$ (resp. $ResD(i, j)(a, b)$) for all the cells (a, b) in row k (resp. in column k). Every list stores a sequence of pairs of integers (δ, β) . The first value δ is called the *distance* and the second value β is called the *boundary*. We call such pairs *boundary pairs*. The pairs are stored in increasing order of distances. For an integer x , we call the pair (δ, β) in a list the *boundary predecessor* (resp. *boundary successor*) of x in Row_k if β is the minimal (resp. maximal) boundary in the list that is at least (resp. at most) x .

When processing $DP_x[i][j]$, we are interested in maintaining the following invariant regarding the pairs stored in Row_a (for every $a \in [n]$):

Let $b \in [n]$ be an integer such that $(i, j) < (a, b)$ and let (δ, β) be the boundary predecessor of b in Row_a . It holds that $ResL(i, j)[(a, b)] = \delta$. Equivalently: Let $(\delta_1, \beta_1), (\delta_2, \beta_2) \dots (\delta_z, \beta_z)$ be the pairs in Row_a . Note that due to Lemma 24, the pairs in Row_k are naturally stored in decreasing order of their boundaries. If an integer b satisfies $b \in [\beta_r \dots \beta_{r-1} - 1]$ for some $r \in [z]$, then $ResL(i, j)[(a, b)] = \delta_r$. For a visualization, see Figure 4.

Essentially, the list Row_a stores an implicit representation of the shortest (i, j) -restricted paths that end with a left step to the cells in row a .

Similarly, the list Col_b stores an implicit representation of $ResD(i, j)[(a, b)]$ for vertices in column b as follows. For every $a \in [n]$ such that $(i, j) < (a, b)$ with boundary successor (δ, β) in Col_b , it holds that $ResD(i, j)[(a, b)] = \delta$.

Throughout the iterations, we maintain the pair $r = (\delta_r, \beta_r)$ such that when we iterate $DP_x[i][j]$, the pair r is the boundary predecessor j in Row_i . We also store n pairs $c_1, c_2 \dots c_n$ such that when iterating $DP_x[i][j]$, the pair $c_j = (\delta_c^j, \beta_c^j)$ is the boundary successor of i in Col_j . We initialize every list Row_k with a single pair $(\infty, 1)$ and every list Col_k with a single pair (∞, n) . For the sake of consistency, we treat the initialization of the algorithm as a phase in the iteration in which a dummy cell $(0, 0)$ is the currently iterated cell. Note that the initialization for the lists suggests that for every vertex, there is no $(0, 0)$ -restricted path that ends with a step to downwards or leftwards.

Processing a cell. When processing $DP_x[i][j]$, we first obtain the distance to $DP_x[i][j]$ using r and c_j . The shortest path to (i, j) must end with a step to the left from a vertex in row i or with a step downwards from a vertex in column j . Note that all the cells to the right of (i, j) and above it have already been processed. It follows that the shortest path to (i, j) is an (i', j') -restricted path with (i', j') being the cell processed in the previous iteration. Let $r = (\delta_r, \beta_r)$ and $c_j = (\delta_c^j, \beta_c^j)$. r is the boundary predecessor of i in Row_i , so according to the invariant we have $ResL_{(i', j')} = \delta_r$. Similarly, $ResD_{(i', j')} = \delta_c^j$. We can therefore set $DP_x[i][j] = \min(\delta_r, \delta_c^j)$.

The remaining task is to update the lists and the pointers in a manner that preserves our invariants. We make the following claims.

▷ **Claim 26.** For $(i, j) \in [n] \times [n]$ and $(a, b) \in [n] \times [n]$ such that $(i, j) < (a, b)$, if an (i, j) -restricted path to (a, b) visits the vertex (i, j) - (i, j) must be the second to last vertex in the path (i.e. the next vertex is (a, b)).

Proof. According to Corollary 21, every edge emerging from (i, j) enter a vertex (i', j') such that $(i, j) < (i', j')$. The only vertex in an (i, j) -restricted path that is allowed to proceed (i, j) is the destination vertex. ◁

Claim 26 suggests the following.

► **Corollary 27.** Let $(i', j') \in [n] \times [n]$ and let $(i, j) \in [n] \times [n]$ be the vertex immediately following (i', j') in the iteration order. Let $(a, b) \in [n] \times [n]$ such that $(i, j) < (a, b)$. If there is no edge from (i, j) to (a, b) we have $ResL_{(i, j)}[(a, b)] = ResL_{(i', j')}[(a, b)]$ and $ResD_{(i, j)}[(a, b)] = ResD_{(i', j')}[(a, b)]$

Furthermore, by Corollary 27 and Corollary 21 together, we obtain the following.

► **Corollary 28.** For $k \neq i$ (resp. $k \neq j$), the list Row_k (resp. Col_k) does not need to be updated after the cell (i, j) is processed in order to satisfy the invariant.

It follows from Corollary 28 that we only need to update the lists Row_i and Col_j to represent shortest (i, j) -restricted paths instead of representing shortest (i', j') -restricted paths. In other words, we need to update Row_i and Col_j to consider paths that use the vertex (i, j) . Specifically, paths that use (i, j) as a second to last vertex (Claim 26)

We update the lists as follows. Let l be the k -Border of $x[i \dots j]$ and let d be the recently calculated $d = DP_x[i][j] = \min(\delta_r, \delta_c^j)$. According to Corollary 21, there is an edge from (i, j) to $(i, j - z)$ with $z \in [1 \dots l]$, and only to those vertices in the i 'th row. We call these vertices the *contested* vertices. For every contested vertex, there is an (i, j) -restricted path that ends with a step to the left via the vertex (i, j) . This path has length $d + 1$. Our task is to update Row_i such that every contested vertex (a, b) in the list with $ResL_{i', j'}[(a, b)] > d + 1$ is updated to have $ResL_{(i, j)}[(a, b)] = d + 1$. Every $(a, b) \in Row_i$ with $ResL_{(i', j')}[(a, b)] \leq d + 1$ needs to keep its current distance. The distances to uncontested vertices in the i 'th row do not require an update (Corollary 27).

Assume w.l.o.g that $d = \delta_r$ (the case in which $d = \delta_c^j$ is treated symmetrically). We may need to add the boundary pair $(d + 1, j - l)$ to Row_i to represent the newly available (i, j) -restricted paths. First, observe that $r = (d, \beta_r)$ should not be removed from Row_i . This is due to the cost of the newly available paths via (i, j) being $d + 1$ - longer than the paths already represented by Row_i for the vertices (i, b) with $b \in [\beta_r \dots j]$. We follow the list pointer from r to obtain its boundary predecessor $r' = (\delta_1, \beta_1)$ in Row_i with $\beta_1 < \beta_r$ and $\delta_1 > d$. We consider the following cases.

Case 1.a: $\delta_1 = d + 1$ and $j - l \geq \beta_1$. In this case, Row_i already represents the shortest restricted paths with cost $d + 1$ to the vertices (i, k) with $k \in [j - l \dots \beta_r - 1]$. Therefore, no update is required for Row_i .

Case 1.b: $\delta_1 > d + 1$ and $j - l \geq \beta_1$. In this case, we need to add the boundary pair $(d + 1, j - l)$ after the boundary border r in Row_i . The following pairs in Row_i have boundaries smaller than $j - l$ and therefore represent the shortest paths uncontested vertices and do not need to be changed. If $j - l = \beta_1$, we also remove r' from Row_i , as it is redundant.

Case 2 : $j - l < \beta_1$. In this case, adding the pair $(d + 1, j - l)$ after the pair r to Row_i may be insufficient. We also need to remove every pair (δ, β) in Row_i with $\beta \in [j - l \dots \beta_1]$. All of those pairs are now redundant in Row_i - as they represent paths with a length at least $d + 1$ to contested vertices. We execute the deletion of these pairs in a straightforward manner by following the links from r' until we reach a pair $r^* = (\delta, \beta)$ with $\beta < j - l$. When r^* is finally met, we insert $(d + 1, j - l)$ to Row_i between the r and r^* .

We proceed to treat Col_j . If $\delta_c^j = d$, the treatment of Col_j is completely symmetric to the treatment of Row_i . Otherwise, δ_c^j . As in the treatment of Row_i , our task is to add a representation of the paths with length $d + 1$ to vertices (a, j) with $a \in [i \dots i + l]$.

Namely, every pair (δ, β) in Col_j with $\delta < i + l$ should be removed (including c^j), as it represents a path with length at least $d + 1$ to one of the vertices (a, j) with $a \in [i, i + l]$. We execute the required deletion in a straightforward manner. Starting from c^j , we proceed to the next pair in the list until a pair (δ, β) with $\beta > j + l$ is found. We then remove the pairs iterated in this process from Col_j and append $(d + 1, i + l)$ to the beginning of the list. This concludes the updates to Row_i and to Col_j . We note that if r of c_j is removed from Row_i or from Col_j , respectively, the new pair $(d + 1, j - l)$ (or respectively, $(d + 1, i + l)$) is becoming the new boundary predecessor (resp. boundary successor) of j in Row_i (resp. of i in Col_j).

Finally, we need to update r and c_j to be the boundary predecessor and successors required for the next iterated cell. If $i < n$, the i value will remain the same on the next iteration. In this case, if $\beta_r = j$, we update r to be the next element in Row_i . If $\beta_r < j$, we do not need to update r as it is also the boundary predecessor of $j - 1$ in Row_i .

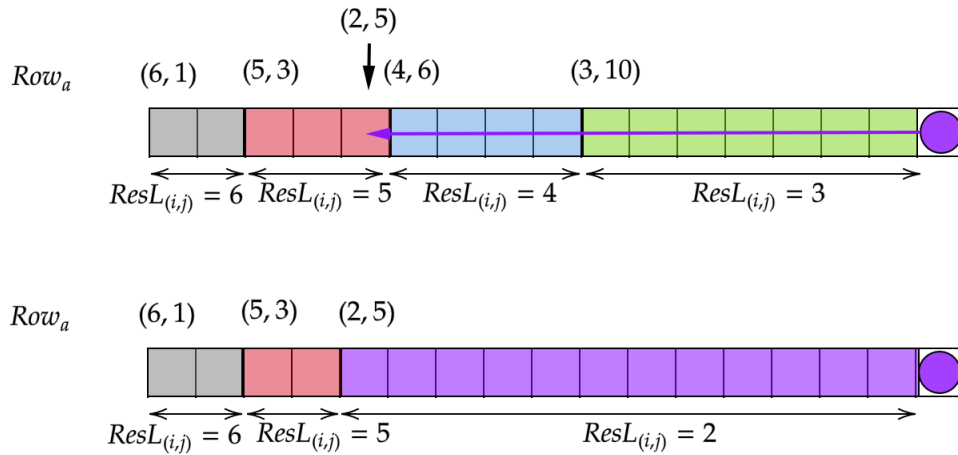
If $i = n$, the next iteration is the first step in row $i + 1$. It follows that Row_{i+1} is still in its initialized state, and we set the only pair in $(\infty, 1) \in Row_{i+1}$ to be r .

As for the c_j pointers, we need to update all of them every time a new row is met. When moving from row i to row $i + 1$, every c_j needs to be updated from the predecessor of i in Col_j to the predecessor of $i + 1$ in Col_j . This is done in a symmetric manner to the update of r .

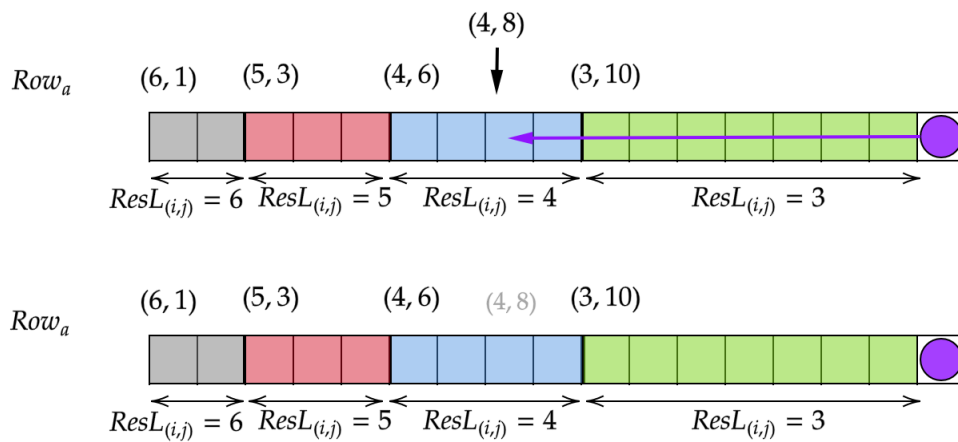
► **Lemma 29.** *The time complexity of the algorithm is $O(n^2)$.*

Proof. When a cell (i, j) is processed, the value in $DP_x[i][j]$ is decided in constant time. In the process of maintaining the lists invariant, at most one pair is added to the list Row_i and to the list Col_j . Several pairs may be removed from these lists, but since every element can be removed at most once throughout the algorithm - the overall time complexity for treating the lists is $O(n^2)$. The pointer r is updated in constant time during the processing of $DP_x[i][j]$. The pointers $c^1, c^2 \dots c^n$ are all updated in $O(n)$ when a table row is visited for the first time, which happens n times throughout the algorithm. ◀

After we compute the tables DP_x and DP_y we have to find a common substring z of x, y such that $HCD_k(z, x) + HCD_k(z, y)$ is minimum among all common substrings of x, y . We use the algorithm *Compute_MDCA* presented in [18] which can return the answer in

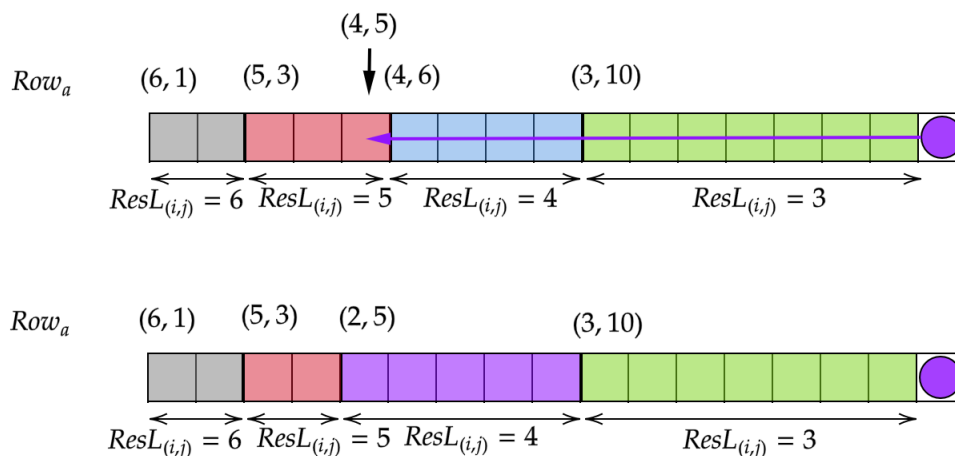


■ **Figure 5** Case 2: The shortest path to the purple vertex is discovered to be 2. The k -Border of the substring corresponding to this vertex is 11 (denoted as the purple left arrow), thus creating new restricted paths to the vertices covered by the purple arrow (recall that in $G_h(x)$, there is a directed edge from the purple vertex to every one of the vertices covered by the purple arrow). The appropriate update to the list is removing the pairs (3, 10) and (4, 6) as the vertices in the green area and in the blue area are now accessible via a shorter path with length 2 via the purple vertex. This new paths are represented by the newly added pair (2, 5).



■ **Figure 6** Case 1.a: The distance to the purple vertex is discovered to be 3, enabling new paths with length 4 to the vertices touched by the purple arrow (representing the length of the k -Border of the substring corresponding to the purple vertex). These new paths does not improve upon the restricted paths already represented in the list, so the pair (4, 8) representing these new paths is simply not added to the list.

$O(n^2)$. To be clear, we provide a brief explanation of the algorithm. In the first stage, the algorithm builds a trie with all the suffixes of the string x . Then, it will traverse the trie for every suffix of y and at every match it will compute the sum of DP values. In short, this algorithm determines in quadratic time all the common substrings of x and y and keeps the one with the minimum sum of distances. We add the pseudocode for the algorithm described in this section in the appendix.



■ **Figure 7** Case 2: The distance to the purple vertex is discovered to be 3. This creates new restricted paths with length 4 to the vertices touched by the purple arrow (representing the k -Border of the string corresponding to the purple vertex). For the vertices in the green area, this is not an improvement, as we already have a representation to a path with length 3 to those vertices. The distances to the vertices in the blue area and to the vertex in the red area touched by the purple arrow are longer or equal to 4. To represent this, we add the boundary pair $(4, 5)$ and remove the boundary pair $(4, 6)$ (as the k -Border $(4, 6)$ represented the distances to the vertices in the blue interval, which are now represented by $(4, 5)$).

5 Conclusions and future work

In this paper we study two problems related to the hairpin completion operation. We propose a quadratic time algorithm for solving these two problems, thus improving the runtime over previous work by Manea [18]. Notice that both our algorithms compute the dynamic programming table of the respective problem explicitly.

A question that arises from our work is can one find an algorithm that solves one of these problems by computing a small subset of cells in the dynamic programming table, which implies a runtime of $o(n^2)$. An interesting and challenging open problem is to provide an $o(n^2)$ algorithm for any of the two problems studied in this paper (not necessary with uses of the dynamic programming's formula), or present a lower bound matching with known problems.

For other variants of hairpin problems (see, e.g., [9, 20, 21]), we believe our techniques can help understand them better and help with designing efficient algorithms for these problems.

References

- 1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, pages 73–84, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
- 2 Michael A Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- 3 Henning Bordihn, Victor Mitrană, Andrei Păun, and Mihaela Păun. Hairpin completions and reductions: Semilinearity properties. *Natural Computing: An International Journal*, 20(2):193–203, June 2021.

- 4 Daniela Cheptea, Carlos Martin-Vide, and Victor Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. *Transgressive Computing*, January 2006.
- 5 Volker Diekert and Steffen Kopecki. Complexity results and the growths of hairpin completions of regular languages (extended abstract). In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, pages 105–114, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- 6 Volker Diekert and Steffen Kopecki. It is NL-complete to decide whether a hairpin completion of regular languages is regular. *Computing Research Repository – CORR*, 22, January 2011. [arXiv:22](#).
- 7 Volker Diekert, Steffen Kopecki, and Victor Mitrana. On the hairpin completion of regular languages. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing – ICTAC 2009*, pages 170–184, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 8 Volker Diekert, Steffen Kopecki, and Victor Mitrana. Deciding regularity of hairpin completions of regular languages in polynomial time. *Information and Computation*, 217:12–30, 2012.
- 9 Masami Ito, Peter Leupold, Florin Manea, and Victor Mitrana. Bounded hairpin completion. *Information and Computation*, 209(3):471–485, 2011. Special Issue: 3rd International Conference on Language and Automata Theory and Applications (LATA 2009).
- 10 Haim Kaplan and Nira Shafir. Path minima in incremental unrooted trees. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms – ESA 2008*, pages 565–576, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 11 Lila Kari, Stavros Konstantinidis, Elena Losseva, Petr Sosík, and Gabriel Thierrin. Hairpin structures in DNA words. In Alessandra Carbone and Niles A. Pierce, editors, *DNA Computing*, pages 158–170, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 12 Lila Kari, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. On hairpin-free words and languages. In Clelia De Felice and Antonio Restivo, editors, *Developments in Language Theory*, pages 296–307, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 13 Lila Kari, Steffen Kopecki, and Shinnosuke Seki. Iterated hairpin completions of non-crossing words. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science*, pages 337–348, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 14 Lila Kari, Elena Losseva, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. A formal language analysis of DNA hairpin structures. *Fundamenta Informaticae*, 71(4):453–475, 2006.
- 15 Lila Kari, Kalpana Mahalingam, and Gabriel Thierrin. The syntactic monoid of hairpin-free languages. *Acta Informatica*, 44(3):153–166, June 2007.
- 16 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.
- 17 Steffen Kopecki. On iterated hairpin completion. *Theoretical Computer Science*, 412(29):3629–3638, 2011.
- 18 Florin Manea. A series of algorithmic results related to the iterated hairpin completion. *Theoretical Computer Science*, 411(48):4162–4178, 2010.
- 19 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening. In Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes, editors, *Programs, Proofs, Processes*, pages 296–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 20 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. On some algorithmic problems regarding the hairpin completion. *Discrete Applied Mathematics*, 157(9):2143–2152, 2009. Optimal Discrete Structures and Algorithms.
- 21 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening: language theoretic and algorithmic results. *Journal of Logic and Computation*, 25(4):987–1009, January 2013.
- 22 Florin Manea and Victor Mitrana. Hairpin completion versus hairpin reduction. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *Computation and Logic in the Real World*, pages 532–541, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

- 23 Florin Manea, Victor Mitrana, and Takashi Yokomori. Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. *Theoretical Computer Science*, 410(4):417–425, 2009. Computational Paradigms from Nature.
- 24 Florin Manea, Victor Mitrana, and Takashi Yokomori. Some remarks on the hairpin completion. In *International Journal of Foundations of Computer Science*, 2010.
- 25 James D. Watson and Francis H. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.
- 26 Peter Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

A

 Appendix

■ **Algorithm 2** An $O(n^2)$ algorithm for Problem 2.

Input: $x, y \in \Sigma^+$

Output: a string z such that $HCD_k(z, x) + HCD_k(z, y)$ is minimum

- 1: $DP_x = \text{ComputeDP}(x)$
 - 2: $DP_y = \text{ComputeDP}(y)$
 - 3: **return** $\text{Compute_MDCA}(x, y, DP_x, DP_y)$
-

■ **Algorithm 3** Updates the list Row_i .

-
- 1: **procedure** UPDATEROW(i, j, r)
 - 2: **while** r is not NULL and $\beta_r > j - k\text{-Border}(s[i \dots j])$ and $\delta_r > DP_x[i][j]$ **do**
 - 3: delete r from Row_i
 - 4: $r = r \rightarrow next$
 - 5: **end while**
 - 6: add $(j - k\text{-Border}(s[i \dots j]), DP_x[i][j] + 1)$ to Row_i
 - 7: **end procedure**
-

■ **Algorithm 4** Updates the list Col_j .

-
- 1: **procedure** UPDATECOL(i, j, c_j)
 - 2: **while** c_j is not NULL and $\beta_{c_j}^j < i + k\text{-Border}(s[i \dots j])$ and $\delta_{c_j}^j > DP_x[i][j]$ **do**
 - 3: delete c_j from Col_j
 - 4: $c_j = c_j \rightarrow next$
 - 5: **end while**
 - 6: add $(i + k\text{-Border}(s[i \dots j]), DP_x[i][j] + 1)$ to Col_j
 - 7: **end procedure**
-

Algorithm 5 *ComputeDP*.

Input: $x \in \Sigma^+$
Output: DP_x

- 1: $DP[i][j] = \infty, \forall 1 \leq i \leq j \leq n$ $\triangleright n$ is the length of the input string
- 2: $DP_x[1][n] = 0$ \triangleright Base case
- 3: add $(n - k\text{-Border}(s[1 \dots n]), 1)$ to Row_1
- 4: **for** $i \leftarrow n - 1$ to 1 **do** \triangleright Compute the first line of DP_x
- 5: **if** $i \leq \beta_r$ **then**
- 6: $DP_x[1][i] = \delta_r$
- 7: **if** $j - k\text{-Border}(s[1 \dots i]) < \beta_r$ **then**
- 8: UPDATEROW(1, i, r)
- 9: **end if**
- 10: **end if**
- 11: **end for**
- 12: **for** $i \leftarrow 2$ to n **do**
- 13: **for** $j \leftarrow n$ to 1 **do**
- 14: **if** $\delta_r < \delta_c^j$ **then**
- 15: **if** $j \geq \beta_r$ **then**
- 16: $DP_x[i][j] = \delta_r$
- 17: **if** $j - k\text{-Border}(s[i \dots j]) < \beta_r$ **then**
- 18: UPDATEROW(i, j, r)
- 19: UPDATECOL(i, j, c_j)
- 20: **end if**
- 21: **end if**
- 22: **else**
- 23: **if** $i \leq \beta_c^j$ **then**
- 24: $DP_x[i][j] = \delta_c^j$
- 25: **if** $i + k\text{-Border}(s[i \dots j]) > \beta_c^j$ **then**
- 26: UPDATECOL(i, j, c_j)
- 27: UPDATEROW(i, j, r)
- 28: **end if**
- 29: **end if**
- 30: **end if**
- 31: **end for**
- 32: **end for**
- 33: **return** DP_x

On Distances Between Words with Parameters

Pierre Bourhis ✉ 

Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France

Aaron Boussidan ✉

LIGM, Université Gustave Eiffel, CNRS, Marne-la-Vallée, France

Philippe Gambette ✉ 

LIGM, Université Gustave Eiffel, CNRS, Marne-la-Vallée, France

Abstract

The edit distance between parameterized words is a generalization of the classical edit distance where it is allowed to map particular letters of the first word, called parameters, to parameters of the second word before computing the distance. This problem has been introduced in particular for detection of code duplication, and the notion of words with parameters has also been used with different semantics in other fields. The complexity of several variants of edit distances between parameterized words has been studied, however, the complexity of the most natural one, the Levenshtein distance, remained open.

In this paper, we solve this open question and close the exhaustive analysis of all cases of parameterized word matching and function matching, showing that these problems are NP-complete. To this aim, we also provide a comparison of the different problems, exhibiting several equivalences between them. We also provide and implement a MaxSAT encoding of the problem, as well as a simple FPT algorithm in the alphabet size, and study their efficiency on real data in the context of theater play structure comparison.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String matching, edit distance, Levenshtein, parameterized matching, parameterized words, parameter words, instantiable words, NP-completeness, MAX-SAT

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.6

Supplementary Material *Software (Source Code)*: <https://github.com/AaronFive/paramatch>
archived at `swh:1:dir:3a72b0d85a4a2be9126900473b8f3e6d03c12a52`

Funding ANR-18-CE23-0003 CQFD

1 Introduction

Measuring the similarity between text strings is a fundamental problem in computer science, and has applications in bioinformatics [23], databases [1, 16] and natural language processing [27]. Among the measures of similarities between strings, the Levenshtein distance [28] is the most commonly used, both for its practicality and its ease of computation. This distance quantifies the minimum number of operations of insertion, deletion, and substitution needed to transform a string into another one. It has a wide range of applications, ranging from biological sequence alignment [33] to dialect pronunciation differences [25] or signature authentication [34]. Computing the edit distance between two strings of length n and m can be achieved in time $O(nm)$, by computing the distance between all their prefixes, and storing the results in a dynamic programming fashion [37]. The success of the Levenshtein distance generated many extensions and generalization on more complex models, such as trees [38] or automata [32].

However, a limitation of the Levenshtein distance is that it only captures proximity between strings (or objects) written on the same alphabet. Evaluating the proximity of strings written on different alphabets is a problem that arises in various applications, such as bioinformatics [35], image processing [17] and code duplication [6, 7]. In all those contexts,



© Pierre Bourhis, Aaron Boussidan, and Philippe Gambette;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 6; pp. 6:1–6:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the technique used is the one of parameterized matching [6, 7]. Instead of using classical strings, parameterized matching uses “parameterized words” written using both constant parts, which are expensive to rename, and parameters, which are meant to be renamed freely. Formally, two equal-length strings u and v over an alphabet Π are said to be parameterized matching if there exists a 1-to-1 function $f : \Pi \rightarrow \Pi$ such that $f(u) = v$, where $f(u)$ is defined as $f(u_1) \dots f(u_{|u|})$.

Words with parameters also occur in other frameworks, and are often used in slightly different ways. The first of those frameworks was initially introduced in the context of Ramsey theory in the 80s [36], and is called “parameter words”. In this context, parameters are labelled according to their order of first occurrence. Parameter words are also equipped with a composition operation, where parameters of the first word can be instantiated by characters or parameters of the second word. Parameter words can also be seen as equivalence classes of parameterized words, which are the main focus of this article.

A second framework using parameters is the one of parameterized regular expressions introduced in [10], where parameters can only be instantiated by constants, and not by other parameters. The focus in this context is therefore made on the set of all possible valuations of the parameters. Then, when defining algorithmic problems on such objects, two distinct semantics can be studied: either the “certainty semantics”, where all valuations need to have some property, or the “possibility semantics”, where at least one valuation needs to have this property. To make a difference with the parameterized word framework mentioned below, we choose to call these words “instantiable words”. Finally, this notion of words with parameters can also be seen as a refined version of partial words (words containing a wildcard character) [15]. The notion of partial words is also important in the context of databases where paths of incomplete graphs can be interpreted as instantiable words [9].

This article aims at studying similarity by using edit distances in the framework of words with parameters. In this framework, the pattern matching problem, which consists in looking for the first string as a subword of the second string, has been extensively studied, either looking for exact occurrences, with efficient algorithms [4, 19, 30] or approximate ones, which is often NP-hard [21, 22]. In the case where we compare the two input strings in their entirety, various exact parameterized matching problems have been studied for parameterized pattern matching, namely string parameterized matching [7], single pattern parameterized matching [7, 3], multiple pattern parameterized matching, or 2-dimensional parameterized matching, many of those works being compiled in [29] and [31]. Different approximate variants of parameterized matching using edit distance have already been studied, but the problem has not been completely solved: the first work on the topic is [8], in which Baker introduces a form of approximate parameterized pattern matching in which the replacement of any substring by another one that is in parameterized matching with it is considered as a base edit operation. Parameterized matching under the Hamming distance, i.e., with a distance allowing only substitutions, has been covered in [24], where the authors prove that the string matching problem with at most k mismatches can be solved in time $O(m + k^{1.5})$. The LCPS (Longest Common Parameterized Subsequence) problem, equivalent to the parameterized pattern matching problem with insertions and deletions, is shown to be NP-hard in [26], which also provides an approximation algorithm. Those two different complexity classes for these problems raise the question of the complexity of the problem under the Levenshtein distance. This problem was left as an open question in the conclusion of [24].

Our paper establishes that this problem is NP-complete. Moreover, the result also extends to any possible edit distances obtained from deletion, insertion, and substitution as soon as substitution is not the only operation allowed, as summarized in Figure 1. Our main

d	\emptyset	D	I	DI
\emptyset	P [8]	NP-complete (Th. 12)	NP-complete (Cor. 14)	NP-complete [26]
S	P [24]	NP-complete (Cor. 14)	NP-complete (Cor. 14)	NP-complete (Th. 13)

■ **Figure 1** Complexity of the variants of parameterized matching PM^d , depending on the kind of operations (D: deletion, I: insertion, S: substitution) allowed in the edit distance d .

proof also implies the main theorem of [26] with a different NP-completeness reduction. This contrasts with the problems of exact parameterized pattern matching which are all polynomial-time solvable, as well as all variants of the string matching problem with deletions, insertions or substitutions.

We also extend these results to function matching, which is the problem obtained by relaxing the 1-to-1 restriction in parameterized matching, as defined in [2]. This generalization, by breaking the symmetry of parameterized matching, actually gives rise to two close but different problems, depending of the order of operations that are considered. We study the links between all these problems and their computational complexity, and study two practical ways to solve them, parameterized complexity and the use of maxSAT solvers.

We also make a direct connection with the framework of instantiable words, more precisely with a natural problem of distance between languages. We show how instantiable word problems can be reduced to parameterized matching ones, under the right assumptions. This allows us to open new perspectives on the complexity of several language repair problems.

In Section 2, we give basic definitions and notations, and recall the existing formalism of parameterized matching and instantiable words. In Section 3 we discuss approximate parameterized matching and its various generalizations. We also link it to instantiable words. In Section 4, we first prove a collection of technical results that build up to the NP-completeness proofs for parameterized matching and function matching problems defined above. In Section 5, we study two approaches to solve one of the variants of parameterized matching in practice, a simple FPT algorithm parameterized by the alphabet size and a MaxSAT encoding. We show in Section 6 that these implementations can solve real instances of the problem, motivated by structure comparison of theater plays.

Finally, in Section 7, we conclude the paper and give a few perspectives on the notion of distance between parameterized languages.

2 Notations and Definitions

2.1 Basic Notations on Words and Editions

Words

An **alphabet** is a set of letters. A word on an alphabet A is a finite sequence of letters from A , indexed starting from 1. Let u be a word on A . Unless defined differently, we note u_i the i -th letter of u , and $|u|$ is the length of u . If $i \notin [1, |u|]$, u_i is defined as the empty word ε . If x is a letter from A , $|u|_x$ is the number of times x appears in u . Similarly, if X is a set of letters, $|u|_X = \sum_{x \in X} |u|_x$ is the number of occurrences of letters of X in u . If f is a function defined on an alphabet A , we extend it to A^* in the usual way, so that $f(u) = f(u_1) \dots f(u_{|u|})$. If f is a function, we denote by $\mathcal{D}(f)$ the domain of f . Two functions f and g are said to be **compatible** if $f|_{\mathcal{D}(g) \cap \mathcal{D}(f)} = g|_{\mathcal{D}(g) \cap \mathcal{D}(f)}$. The identity function on D is defined as $Id_D(x) = x$ for all x in D .

Edit Operations

In this paper, we consider the three classical **edit operations** which are *deletion*, *substitution* and *insertion*. Let $u = u_1 \dots u_n$ be a word of size n . Let i be an integer between 0 and n and x be a letter of the alphabet, the **insertion** at position i is the operation that transforms u to $u_1 \dots u_i x u_{i+1} \dots u_n$. Let j be an integer between 1 and n , the **deletion** at position j is the operation that transforms u into $u_1 \dots u_{j-1} u_{j+1} \dots u_n$. Let y be a letter of the alphabet and $y \neq u_j$, the **substitution** to y at position j is the operation that transforms u into $u_1 \dots u_{j-1} y u_{j+1} \dots u_n$. A **sequence of operations** or **rewriting sequence** ρ is a sequence of edit operations. We denote by $\rho(u)$ the word obtained by applying the edit operations of ρ one after another, in the order defined by ρ , to u .

Distances

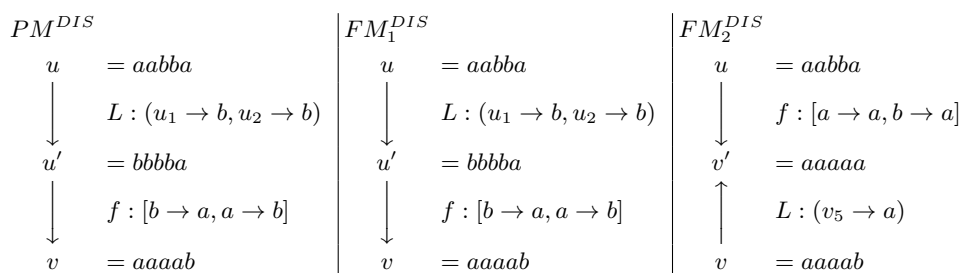
Given a set of edit operations E and two words u and v , the **edit distance** between u and v is defined as the length of a shortest sequence of operations of E changing u into v . We denote by D the distance obtained on words by allowing only deletion operations: that is to say $D(u, v) = k$ iff v can be obtained by deleting k letters from u . Similarly, we note I and S the distances obtained by allowing only insertions and substitutions respectively (note that S is the Hamming distance). We also combine these notations to define DI as the distance with insertions and deletions, and so on. We also denote the Levenshtein distance DIS by L . Note that some of these edit distances are not metrics, because they are not symmetrical. We emphasize this by calling **symmetric edit distances** the distances DI, S , and L .

2.2 Comparing Words with Parameters

Conceptually, a word with parameters is a word in which some letters are not yet determined. In order to distinguish the parameters from the constants, we split the alphabet into Σ , the alphabet of the constants and Π , the alphabet of the parameters. By definition, these alphabets are finite. A word with parameters can either be seen as representing a “word template” (i.e., a word with variable parts), or a set of words (determined by all possible affectations of its parameters). Depending on the definition chosen, comparing two words w_1 and w_2 is done in two different ways. In the first setting [6, 7, 8, 31, 2, 5, 24, 29, 26, 17], parameters of w_1 are renamed through a function f that maps the set of parameters to itself, and acts as identity on the set of constants. It is then possible to compare $f(w_1)$ and w_2 , which are written on the same alphabet. In the second setting, constants are seen as the concrete values parameters can take [11]. Parameters are instantiated through two functions f_1 and f_2 that map constants to themselves, but also map parameters to constants. The words $f_1(w_1)$ and $f_2(w_2)$ are then made only of constants, and can be compared. Formally, this gives rise to the two following different definitions:

On the one hand, a **parameterized word** is a word on an alphabet $\Sigma \cup \Pi$. In all that follows, Σ and Π are two disjoint alphabets, respectively called the **alphabet of constants** and the **alphabet of parameters**. Alphabets Σ and Π are considered to be finite, unless specified otherwise.

Two parameterized words u and v are said to be in **function matching** if there exist $f_\Pi : \Pi \rightarrow \Pi$ and $f : \Pi \cup \Sigma \rightarrow \Pi \cup \Sigma$ such that $f|_\Pi = f_\Pi$, $f|_\Sigma = Id_\Sigma$, and $f(u) = v$. In the classical setting [6], f is also **constrained to be 1-to-1**, and this relationship is called **parameterized matching**. Note that parameterized matching is an equivalence relation on parameterized words. Testing if two words are parameterized matching can be achieved in linear time [7].



■ **Figure 2** Side-by-side comparison of PM^{DIS} , FM_1^{DIS} and FM_2^{DIS} .

On the other hand, an **instantiable word** is a word on the alphabet $\Sigma \cup \Pi$. Given $f : \Pi \rightarrow \Sigma$, we extend it to constants by setting $f(x) = x$ for all $x \in \Sigma$, and we then define the **language of an instantiable word** u to be $L(u) = \{w \in \Sigma^* \mid \exists f : \Pi \rightarrow \Sigma, f(u) = w\}$. This definition is akin to the L_\diamond semantic of a parameterized regular expression defined in [11], but restricted here to a single instantiable word. Two instantiable words w_1 and w_2 describe the same elements if their languages are equal, i.e. $L(w_1) = L(w_2)$.

3 Different Definitions for Different Semantics and Problems

In this section, we introduce various new approximate variants of parameterized matching, and compare them, highlighting their differences on examples.

3.1 Variants of Parameterized Matching

In parameterized matching, the function f renaming parameters is generally considered to be 1-to-1. In this paper, we also consider the **function matching problem**, which is the case where f is not constrained to be injective anymore, as defined in [2]. We also introduce multiple approximate variants of the parameterized matching problems, depending on several edit distances obtained by combining insertion, deletion and substitution operations.

3.1.1 Edit distances for parameterized matching between two strings: PM^d

► **Definition 1.** *If d is an edit distance, we denote by PM^d the parameterized matching problem under d , which is the following:*

- **Input:** *two parameterized words u, v , a parameter alphabet Π , an alphabet Σ of constants, and a natural number k .*
- **Problem:** *Does there exist u' such that $d(u, u') \leq k$ and u' and v are parameterized matching, i.e. there exists a 1-to-1 function $f : \Pi \cup \Sigma \rightarrow \Pi \cup \Sigma$ such that $f|_\Sigma = Id_\Sigma$, $f(\Pi) = \Pi$, and $f(u') = v$?*

In that case, we say that u' and f **realize** the matching between u and v . We sometimes write that only f or u' realize the matching if the other one is not relevant to a proof.

In cases where Σ and Π are already defined, we omit them and simply call $PM^d(u, v, k)$ the result of the decision problem. Furthermore, $PM^d(u, v)$ denotes the minimum integer k (potentially infinite) such that $PM^d(u, v, k)$ is true.

We can note that this problem can be solved in polynomial time adapting the classical dynamic programming algorithm [33, 37] when the alphabet sizes are fixed.

3.1.2 Edit distances for function matching between 2 strings: FM_i^d

To denote **function matching problems**, we use FM instead of PM : FM^D denotes the function matching problems with deletions.

Furthermore, if \mathcal{P} is one of the problems defined above, we note \mathcal{P}_1 the problem where edit operations are only applied to the first argument, and \mathcal{P}_2 the one where they are only applied to the second argument.

► **Definition 2.** *The FM_1^d and FM_2^d problems are defined as follows. For both problems, the input is the following:*

- **Input:** two parameterized words u, v , a parameter alphabet Π , a constant alphabet Σ , and a natural number k .

The problems are then:

- **Problem FM_1^d :** $\exists u'$ such that $d(u, u') \leq k$ and u' and v are function matching?
- **Problem FM_2^d :** $\exists v'$ such that $d(v, v') \leq k$ and u and v' are in function matching?

Note that the renaming function f is always applied to one input only. These definitions are illustrated on an example in Figure 2.

3.2 Comparing Variants of PM

In this subsection, we compare the different variants of our problem.

Regarding the one-to-one parameterized matching PM , note that the definition we give above is designed to be easily extended to the different variants when we drop the one-to-one restriction. In [24], the authors consider that the “correct way for defining the edit distance problem” is “to allow the operations and then apply the edit distance”. By extending the definition of FM_1^d and FM_2^d to define PM_1^d and PM_2^d in the case of one-to-one matching, we see that it is actually possible to switch the order of operations, and to reverse them (deletions then become insertions and vice versa, and the renaming function f^{-1} is well-defined), in this case. This makes our definition consistent with the quote from [24] above. Formally, this gives the following equalities, for all parameterized words u and v : $PM_1^I(u, v) = PM_1^D(v, u) = PM_2^I(u, v) = PM_2^D(v, u)$.

More generally, it holds that for every edit distance d , $PM_1^d(u, v) = PM_1^{d^{-1}}(v, u) = PM_2^{d^{-1}}(u, v) = PM_2^d(v, u)$, where d^{-1} denotes the converse distance of d , i.e. d^{-1} contains deletions if d contains insertions, insertions if d contains deletions, and substitutions if d contains substitutions.

However, for function matching, we only have the following equalities: $FM_1^I(u, v) = FM_2^D(u, v)$ and $FM_1^D(u, v) = FM_2^I(u, v)$.

By taking $u = ab$ and $v = cc$, we can notice that $FM_1^I(u, v) = 0$ and $FM_1^D(u, v) = \infty$, so the equality $FM_1^I(u, v) = FM_1^D(v, u)$ does not hold.

Finally, note the following inequalities:

► **Proposition 3.** *Let u and v be parameterized words over $\Sigma \cup \Pi$. Then:*

1. $FM_1^d(u, v) \leq PM^d(u, v)$;
2. If d is a symmetric edit distance, $FM_2^d(u, v) \leq FM_1^d(u, v)$.

Proof. The first point comes from the fact that any solution to PM^d is also a solution to FM_1^d . For the second point, let $k = FM_1^d(u, v)$, and let u' and f realize $FM_1^d(u, v)$. We construct a word v' , obtained by applying to v the same operations applied to u to obtain u' , but “mirrored”. That is to say, for every operation used in u , we apply an operation in v , in the following way:

- If a letter a is inserted in u , there exists a position i in u' such that $u'_i = a$, and $f(u'_i) = v_i$. Hence, we delete v_i in v .
- Similarly, if a letter is substituted for another letter a' in u , there exists i such that $u'_i = a$, and we substitute v_i to $f(a)$.
- If a letter a is deleted in u at position i , we insert $f(a)$ in v at position i instead.

It then holds that $f(u) = v'$, and hence $PM_2^d(u, v) \leq k$. ◀

Note that the above proof does not work to prove the converse inequality between FM_1^d and FM_2^d , as it would require to consider an element of $f^{-1}(a)$, which might be empty. This is illustrated in the following example, on the alphabet $\Pi = \{a, b\}$:

► **Example 4.** Let $N \in \mathbb{N}$ and consider $u = a^N b^N b$ and $v = a^N a^N b$. u and v are not in parameterized matching, hence $FM_1^{DIS}(u, v) > 0$ and $FM_2^{DIS}(u, v) > 0$. By substituting the last b in v for a a , and picking a function f such that $f(a) = f(b) = a$, we get $FM_2^{DIS}(u, v) = 1$ (see Figure 2 for an example with $N = 2$). For FM_1^{DIS} , since b appears in v , it holds that for any function f realizing FM_1^{DIS} , $f(a) = b$ or $f(b) = b$. Hence, at least N occurrences of b appear in $f(u)$. Since there is only one occurrence of b in v , it is clear that $FM_1^{DIS}(u, v) \geq N - 1$.

The difference between FM_1^d and FM_2^d comes from the fact that Π is fixed in the input. In the case where Π could be extended, both problems can be shown equivalent (for example if we allow a new letter c in the example of Figure 2, we also get $FM_1^{DIS}(u, v) = 1$ by setting $u_5 \rightarrow c$ and $f : [a \rightarrow a, b \rightarrow a, c \rightarrow b]$), by using the same proof as Proposition 3.

3.3 Instantiable Words versus Parameterized Words

The parameterized word formalism and the instantiable word formalism give rise to two different definitions of distances between words. Given an edit distance d on words, there are two ways to extend it to words with parameters. Let w_1 and w_2 be two words over $\Sigma \cup \Pi$. The two possible extensions are the following:

- The distance between w_1 and w_2 is defined as $d(w_1, w_2) = PM^d(w_1, w_2)$. Alternatively, the function distance between w_1 and w_2 is defined as $FM_1^d(w_1, w_2)$.
- The distance between w_1 and w_2 is the distance between their respective languages seen as sets, that is to say $d(w_1, w_2) = d(L(w_1), L(w_2)) = \sup_{u \in L(w_1)} \inf_{v \in L(w_2)} d(u, v)$. Equivalently, $d(w_1, w_2) \leq k$ if and only if for all $f_1 : \Pi \rightarrow \Sigma$, there exists $f_2 : \Pi \rightarrow \Sigma$ such that $d(f_1(w_1), f_2(w_2)) \leq k$.

This second definition stems from the definition of distance between languages, as defined and studied in [12, 13, 14].

► **Example 5.** Consider the words $u = axyb$ and $v = xby$, on the alphabets $\Sigma = \{a, b\}$ and $\Pi = \{x, y\}$, and consider the distance S . On the one hand, $FM_1^S(u, v) = 4$, because regardless of the matching chosen, every letter of $f(u)$ has to be substituted. On the other hand, for any function $f_1 : \Pi \rightarrow \Sigma$, choosing f_2 such that $f_2(x) = a$ and $f_2(y) = b$ yields a distance $d(f_1(u), f_2(v))$ of at most 2, by substituting the 2 middle letters.

Given a big enough alphabet, those two definitions can in fact be shown equivalent:

► **Proposition 6.** *Let w_1 and w_2 be words over $\Sigma \cup \Pi$, and let d be a symmetric edit distance on $\Sigma \cup \Pi$. Suppose $|\Sigma| \geq |w_1| + |w_2|$, and let k be an integer. Then, the following are equivalent:*

1. $FM_1^d(w_2, w_1) \leq k$
2. $d(L(w_1), L(w_2)) \leq k$

Notice how w_1 and w_2 change position between the two distances. This is not benign, as FM_1^d is not symmetric.

Proof. Suppose $FM_1^d(w_2, w_1) \leq k$. There exists $f : \Pi \rightarrow \Pi$ such that $d(f(w_2), w_1) \leq k$. For this proof, we will use the characterization of the distance between languages with f_1 and f_2 . Let $f_1 : \Pi \rightarrow \Sigma$. Define $f_2 = f_1 \circ f$. Since $d(w_1, f(w_2)) \leq k$, we have $d(f_1(w_1), f_1 \circ f(w_2)) \leq k$, by following the same edit operations. Hence $d(f_1(w_1), f_2(w_2)) \leq k$.

Suppose now $d(L(w_1), L(w_2)) \leq k$. Let $f_1 : \Pi \rightarrow \Sigma$ be a 1-to-1 function such that for all parameters x in w_1 , $f_1(x)$ does not appear in w_1 or w_2 . This is possible since Σ is large enough. There exists $f_2 : \Pi \rightarrow \Sigma$ such that $d(f_1(w_1), f_2(w_2)) \leq k$. Let $h : \Sigma \rightarrow \Pi \cup \Sigma$ be such that if $x \in \Pi$, $h(f_1(x)) = x$, and if $x \notin f_1(\Pi)$, $h(x) = x$. We then have $h \circ f_1 = Id$. What is more, since h is injective, $d(f_1(w_1), f_2(w_2)) = d(h(f_1(w_1)), h(f_2(w_2))) = d(h(f_2(w_2)), w_1)$. Hence, $FM_1^d(w_2, w_1) \leq k$. ◀

4 Hardness Results for Approximate Parameterized Matching

In this section, we study the complexity of the various parameterized matching problems. We show the NP-completeness of the simplest problems using only deletions, which will be sufficient to show the NP-completeness of all the other problems. We start by proving some practical lemmas, and then proceed to the reductions.

4.1 “Block by block” Lemmas

In this section, we regroup a few useful technical lemmas. We start off by stating two simple results on distance and words, for which the proofs can be found in Appendix A. We then turn to block lemmas, which will later be useful in the proofs of Theorems 12, 17 and 15, to combine the various gadgets defined during the reduction.

This lemma simply states a commutativity result between the application of a matching f and the rewriting steps.

► **Lemma 7.** *Let d be a distance, k an integer and u, v two parameterized words such that $PM^d(u, v) \leq k$, and let f realize this parameterized match. Then: $d(f(u), v) \leq k$. The same result holds for $FM_1^d(u, v)$.*

Proof Idea. The proof is done by induction on k . We discuss whether the $(k+1)$ -th operation is an insertion, a deletion, or a substitution, and show that a corresponding operation can be used in $f(u)$. ◀

► **Lemma 8.** *Let z, u and v be (parameterized) words, and let d be a distance. Then $d(zu, zv) = d(u, v)$.*

Proof Idea. We show that we can consider every rewriting operation to be applied in u only: if z is modified during an optimal rewriting sequence, the words have some redundancy, and the same operations could have been carried in u instead. We proceed again by induction, and focus on the base case by studying the 3 possible cases, one for each type of operation. ◀

Next, we turn to prove “block by block” matching lemmas. Those results state that it is possible to encode multiple parameterized matching instances into a single one. They hold for every type of problems considered here, but their proofs vary slightly; we present them in order of increasing complexity. Note that all the constructions given can be achieved in polynomial time.

► **Lemma 9.** *Let u_1, \dots, u_n and v_1, \dots, v_n be parameterized words over $\Sigma \cup \Pi$ such that for $1 \leq i \leq n$, $k_i = |u_i| - |v_i| \geq 0$, and $k = \sum_{i=1}^n k_i$. There exist u and v two parameterized words over $\{\#\} \cup \Sigma \cup \Pi$, where $\#$ is a fresh variable, such that the following are equivalent:*

1. $PM^D(u, v) = k$
2. For all $1 \leq i \leq n$, $PM^D(u_i, v_i) = k_i$ and the applications f_i realizing those matchings are all compatible.

Proof. The idea behind this proof and all the following ones is that we can introduce a separator $\#$ to concatenate all the words, and that this separator will never be touched by any deletions or applications of f .

Let $\#$ be a fresh constant. We define $u = u_1\#u_2\#\dots\#u_n$, and $v = v_1\#v_2\#\dots\#v_n$.

2. \implies 1.: For every $1 \leq i \leq n$, take u'_i and f_i to realize the matchings. We can obtain $u' = u'_1\#u'_2\#\dots\#u'_n$ from u by applying the same deletions. Taking f to be the smallest function extending all the f_i , we get $PM^D(u, v) \leq k$.

1. \implies 2.: Assume $PM^D(u, v) \leq k$. Let u' and f realize this parameterized match. Since the $\#$ symbols are constants, we have $f(\#) = \#$. Since u' is obtained from u by deletions, we have $|u'|_{\#} \leq |u|_{\#}$. Since f is injective and $f(\#) = \#$, $|f(u')|_{\#} \leq |f(u)|_{\#}$. Hence, it holds that $|v|_{\#} = |f(u')|_{\#} \leq |f(u)|_{\#} = |u|_{\#}$. Since $|u|_{\#} = |v|_{\#}$, this is an equality, and $|f(u')|_{\#} = |f(u)|_{\#}$. Hence $|u'|_{\#} = |u|_{\#}$, and no $\#$ character is deleted. The word u' is then of the form $u'_1\#u'_2\#\dots\#u'_n$, where $|u'_i|_{\#} = 0$ and $D(u_i, u'_i) = k_i$ for all i . Thus, $f(u') = f(u'_1)\#f(u'_2)\#\dots\#f(u'_n) = v_1\#v_2\#\dots\#v_n$. Since no other $\#$ letter appear in any $f(u'_i)$ and v_i , we can deduce that $f(u'_i) = v_i$ for all i . Finally, this yields $PM^D(u_i, v_i) = k_i$, and taking all the $f_i = f$ gives all compatible functions, which concludes the proof. ◀

In this proof, we used a constant $\#$. However, it can also be conducted without using a constant alphabet; indeed, constants can be encoded with parameters, as shown in Appendix B.

Lemma 9 is still valid if PM^D is replaced by FM_2^D . This time, we conduct this proof without resorting to the use of constants. This result will be used twice: once for the proof of theorem 17, and again to show that we can once more encode constants into Π using Lemma 25 in Appendix B.

► **Lemma 10.** *Let u_1, \dots, u_n and v_1, \dots, v_n be parameterized words over Π such that $k_i = |v_i| - |u_i| \geq 0$, and $k = \sum_{i=1}^n k_i$. Then there exist u and v , two parameterized words over $\Pi \cup \{\#\}$, where $\#$ is a fresh variable, such that the following are equivalent:*

1. $FM_2^D(u, v) \leq k$
2. For all $1 \leq i \leq n$, $FM_2^D(u_i, v_i) \leq k_i$, and the applications f_i realizing those matchings are all compatible.

Proof Idea. The same technique as Lemma 9 is used but u and v are defined as $u = \#^{k+1}u_1\#u_2\#\dots\#u_n$ and $v = \#^{k+1}v_1\#v_2\#\dots\#v_n$ where $\#^{k+1}$ denotes $k+1$ repetitions of the character $\#$. The full proof can be found in Appendix A. ◀

Finally, the same block result holds for FM_1^D , and will be used in the proof of theorem 15.

► **Lemma 11.** *Let u_1, \dots, u_n and v_1, \dots, v_n be parameterized words over Π such that for every $1 \leq i \leq n$, $k_i = |u_i| - |v_i| \geq 0$, and $k = \sum_{i=1}^n k_i$. Then there exist u and v two parameterized words over $\Pi \cup \{\#\}$, where $\#$ is a fresh variable, such that the following are equivalent:*

1. $FM_1^D(u, v) \leq k$
2. For all $1 \leq i \leq n$, $FM_1^D(u_i, v_i) \leq k_i$, and the applications f_i realizing those matchings are all compatible.

Proof Idea. The difference with Lemma 10 is that some $\#$ symbols might be deleted, while some base letters could be mapped to $\#$. To ensure this does not happen, we define $u = \#^N u_1 \#^N u_2 \dots \#^N u_n \#^N$ and $v = \#^N v_1 \#^N v_2 \dots \#^N v_n \#^N$. The full proof can be found in Appendix A. \blacktriangleleft

The technique of block-by-block matching will be used in all the reductions, to encode multiple constraints in a single PM or FM instance.

4.2 1-to-1 Parameterized Matching PM

We now focus on the complexity of the PM^d problems. These problems, as well as function matching problems, are all clearly in NP: given the list of deletion, insertion or substitution operations to do and the matching to apply, it is easy to check that the solution is correct.

For the reductions in this paper, we always assume that words are written without constants, that is to say $\Sigma = \emptyset$, since this is sufficient for NP-completeness results. This choice is also motivated by the results of Appendix B, which show that Σ can in most cases be coded into Π .

► **Theorem 12.** *The 1-to-1 Parameterized Matching with deletions PM^D is NP-complete.*

The proof is a reduction from the NP-complete problem **3-coloring**[20]. Given an instance G of **3-coloring**, we will construct two words u and v . The word v will represent the list of vertices and edges of G , while the word u will list the color of each vertex, and the possible coloring of each pair of vertices joined by an edge. By deleting characters from u , we make a choice for the coloring of each vertex, and thus each edge. The function f answering the parameterized matching problem will assign a choice of color to each vertex. The instance that we define should be positive iff G is 3-colorable. More formally:

Proof. The 3-Coloring problem is defined as follows:

- **Input:** $G = (V, E)$ a graph with vertices V and edges E
 - **Output:** A coloring $c : V \rightarrow \{c_1, c_2, c_3\}$ such that for all $\{u, v\} \in E$, $c(u) \neq c(v)$
- Let $G = (V, E)$ be an instance of **3-Coloring**, and let $V = \{x_1, \dots, x_n\}$ be the set of its n vertices, and $E = \{e_1, \dots, e_m\}$ be the set of its edges. The parameter alphabet Π , of polynomial size in $O(|G|)$ will contain:
- x_1, \dots, x_n , corresponding to the vertices of G ;
 - n copies of the parameters corresponding to the colors c_1, c_2 and c_3 : c_1^i, c_2^i, c_3^i for $1 \leq i \leq n$;
 - for every edge e , the delimiters Y^e and $\square_1^e, \dots, \square_6^e$;
 - $2n$ bottom symbols, \perp_1^i, \perp_2^i for $1 \leq i \leq n$, which will be used to fix the image of some parameters.

First, we define words that will encode the constraint that each vertex is colored, and we make sure that the unused color variables cannot be assigned elsewhere. For $1 \leq i \leq n$, $u_1^i = u_{\perp}^i = c_1^i c_2^i c_3^i$, $v_1^i = x_i$ and $v_{\perp}^i = \perp_1^i \perp_2^i$. We then define words that include all possible colorings of each edge, and we make sure to use enough brackets. For every edge $e = \{x_i, x_j\}$, we define $u_2^e = \square_1^e c_1^i c_2^j \square_1^e \square_2^e c_1^i c_3^j \square_2^e \square_3^e c_2^i c_1^j \square_3^e \square_4^e c_2^i c_3^j \square_4^e \square_5^e c_3^i c_1^j \square_5^e \square_6^e c_3^i c_2^j \square_6^e$ and $v_2^e = Y^e x_i x_j Y^e$.

Applying Lemma 9 to $u_1^1, \dots, u_1^n, u_{\perp}^1, \dots, u_{\perp}^n, u_2^{e_1}, \dots, u_2^{e_m}$ and $v_1^1 \dots v_1^n, v_{\perp}^1, \dots, v_{\perp}^n, v_2^{e_1}, \dots, v_2^{e_m}$, we obtain u and v . Let $k = |u| - |v| = 3n + 20m$. We now show that G is 3-colorable $\Leftrightarrow PM^D(u, v) \leq k$.

\Rightarrow : Suppose G is 3-colorable. Let $c : V \rightarrow \{c_1, c_2, c_3\}$ be a 3-coloring of G . We define f in the following way, for $1 \leq y \leq 3$:

$$f(c_y^i) = \begin{cases} x_i & \text{if } c(x_i) = c_y, \\ \perp_1^i & \text{if } y \text{ is the smallest integer in } \{1, 2, 3\} \text{ such that } c(x_i) \neq c_y, \\ \perp_2^i & \text{otherwise.} \end{cases}$$

For every edge $e = \{x_i, x_j\} \in E$, since c is a valid coloring, and since every allowed arrangement of the colors is in u_2^e , there exists a unique factor of the form $\square_y^e f^{-1}(x_i) f^{-1}(x_j) \square_y^e$ in u_2^e , for some $1 \leq y \leq 3$. Hence, we define $f(\square_y^e) = Y^e$. The function f can then be extended in any way to be 1-to-1 (the remaining characters whose image under f are not yet defined will all be deleted in what follows, so their image doesn't matter).

By using f defined in this way:

- For $1 \leq i \leq n$, $PM^D(u_1^i, v_1^i) \leq 2$, by deleting the 2 colors not matching the color of x_i ;
- For $1 \leq i \leq n$, $PM^D(u_\perp^i, v_\perp^i) \leq 1$;
- For every edge $e \in E$, $PM^D(u_2^e, v_2^e) \leq 20$, by keeping only the factor delimited by the \square_y^e symbols defined above.

Thus Lemma 9 yields $PM^D(u, v) \leq k$.

\Leftarrow : We now suppose u and v are a parameterized match with k deletions. The following can then be derived about f :

1. Since the u_1^i and v_1^i are matching for $1 \leq i \leq n$, there exists an element $c \in \{c_1^i, c_2^i, c_3^i\}$ such that $f(c) = x_i$. Each of these matchings is done with exactly 2 deletions, for a total of $2n$.
2. Since the u_\perp^i and v_\perp^i are in matching, the two other colors that are not sent to x_i are sent to \perp_1^i and \perp_2^i . Each of these matchings is done with exactly one deletion, for a total of n .
3. For every edge $e \in E$, u_2^e and v_2^e are in matching. Let $u_2^{e'}$ realize this matching. For every $1 \leq i \leq n$ and $1 \leq i' \leq 3$ the colors $c_{i'}^e$ have images that are different from Y^e , so there necessarily exists $1 \leq y \leq 3$ such that $f(\square_y^e) = Y^e$. Furthermore, since f is injective, $|v_2^e|_{Y^e} = |u_2^{e'}|_{\square_y^e}$. Since $|v_2^e|_{Y^e} = |u_2^e|_{\square_y^e} = 2$, no \square_y^e is deleted from u . Since there are two characters between the Y^e in v_2^e and none outside, $u_2^{e'}$ has the same structure, and all other $\square_{y'}^e$ for $y' \neq y$ and all other colors are deleted from u_2^e .

Finally, $u_2^{e'}$ is of the form $\square_y^e c_t c_{t'} \square_y^e$, where $t \neq t'$ are elements of $\{1, 2, 3\}$. Each of these matchings is done with exactly 20 deletions, for a total of $20m$.

The function f then implies a coloring of G . Formally, we define $col(c_y^i) = c_y$ for $1 \leq i \leq n$ and $1 \leq y \leq 3$. We can then define $c : V \rightarrow \{c_1, c_2, c_3\}$ such that $c(x_i) = col(f^{-1}(x_i))$. Point 1 above ensures that this function definition is correct. Furthermore, for every edge $e = \{x_i, x_j\}$, point 3 ensures that $c(x_i) \neq c(x_j)$, and thus c is a valid coloring of G . \blacktriangleleft

This first NP-completeness results yields a few immediate corollary results, and in particular, the NP-completeness of the problem under the Levenshtein distance:

► **Theorem 13.** *The problem PM^{DIS} of parameterized matching under the Levenshtein distance is NP-complete.*

Proof. We do a simple reduction from PM^D . Let u, v, k be an instance of PM^D . If the instance is trivially false (that is to say, $k \neq |u| - |v|$), answer negatively. Else, consider u, v, k as an instance of PM^{DIS} . If this is a negative instance for PM^{DIS} , it is also negative

6:12 On Distances Between Words with Parameters

for PM^D . Furthermore, if it is a positive instance for PM^{DIS} , exactly k deletions should be applied, and so no substitution or insertion are used in that solution. Hence, that solution is also a solution to PM^D , and the reduction holds. ◀

The same result in fact holds for all other distances, and in particular the longest common sub-word distance ID . This proves once again the result shown in [26]:

► **Corollary 14.** *The problems PM^I , PM^{DI} , PM^{IS} , PM^{DS} are all NP-complete.*

Proof. From Section 3.2, PM^I and PM^D are equivalent in the 1-to-1 case. For the other problem, we do an immediate reduction from PM^I or PM^D analog to the proof of Theorem 13. ◀

We now turn to proofs of NP-completeness without the restriction asking f to be injective.

4.3 Function Matching FM_1^d

The problem considered in this section is the one where both deletions and f are applied to the first word. A reduction very similar to the one given for PM^D is used.

► **Theorem 15.** *FM_1^D is NP-complete.*

Proof Idea. The reduction follows the same idea as in Theorem 12. Since the function f realizing the matchings is not injective in this version, it will be used to send every vertex to its color. Moreover, we add more “sink” \perp letters to force the image of every unused letter. The full proof can be found in Appendix A. ◀

This again ensures the NP-completeness of the problem for all edit distances, using the same proof as for Theorem 13.

► **Corollary 16.** *The problem FM_1^{DIS} of function matching under the Levenshtein distance is NP-complete. The problems FM_1^I , FM_1^{ID} , FM_1^{IS} , FM_1^{DS} are all NP-complete too.*

We can notice that the problem FM_1^S , where substitution is the only operation allowed, is polynomial-time solvable. Intuitively, for each parameter, consider the possible parameters that it could be mapped to, and their respective number of occurrences. Then, choose the letter with the highest number of occurrences for the mapping. The remaining letters are then substituted.

4.4 Function Matching FM_2^d

The problem considered in this section is the one where deletions are applied to the second word, while f is applied to the first word.

► **Theorem 17.** *FM_2^D is NP-complete.*

Proof Idea. The proof is very similar to the previous case, but the bracketing has to be adapted. Separators Y^e are duplicated enough times to ensure that no vertex variable is mapped to them. The full proof can be found in Appendix A. ◀

► **Corollary 18.** *FM_1^I , FM_2^{DI} , and FM_2^L are all NP-complete.*

Proof. FM_1^I is equivalent to FM_2^D . For the two other problems, we use a reduction from FM_2^D exactly like in Corollary 14. ◀

This last result completes the picture of NP-completeness proofs, and indicates that computing the distances between parameterized words defined in Section 3.3 is in general an NP-complete problem.

Similarly to FM_1^S , FM_2^S is also polynomial-time solvable.

5 Approaches to Solve Parameterized Matching

In this section, we discuss two ways to get around the difficulty of the parameterized matching problems. The first one is to design an FPT algorithm in the alphabet size, and the second one is to translate the problem into a SAT formalism, with the intent of using a SAT-solver.

5.1 An FPT Algorithm in the Alphabet Size

The fact that Σ and Π are part of the input is what makes the various parameterized matching problems NP-hard. When the alphabet size is considered fixed, a simple polynomial algorithm can be used, which generalizes the “naïve” brute force algorithm of Theorem 1 of [26]:

■ **Algorithm 1** Simple FPT algorithm for FM^d .

```

 $m \leftarrow 0$ 
for all functions  $f : \Pi \rightarrow \Pi$  do
   $dist \leftarrow d(f(u), v)$ 
  if  $dist \leq m$  then
     $m \leftarrow dist$ 
  end if
end for

```

► **Theorem 19.** Let d be a distance. Algorithm 1 computes $FM^d(u, v)$ in time $O(|\Pi|^{|\Pi|}|u||v|)$

Proof. Algorithm 1 uses an exhaustive search and finds $\min_{f: \Pi \rightarrow \Pi} d(f(u), v)$, which is the definition of $FM^d(u, v)$. Furthermore, there are $|\Pi|^{|\Pi|}$ functions from Π to Π , and computing $d(f(u), v)$ is done in time $O(|f(u)||v|) = O(|u||v|)$, hence a total running time in $O(|\Pi|^{|\Pi|}|u||v|)$. ◀

Note that this also leads to a similar algorithm for PM^d by iterating over injective functions rather than all functions from Π to Π .

5.2 A MaxSat Formulation of Parameterized Matching

In this section, we encode PM^d problems into SAT problems, with the intent of solving them with a SAT solver. More precisely, we will use the weighted max-SAT variant of SAT, which is defined in the following way:

- **Input:** a set V of literals, a formula $\varphi = \bigwedge_{i=1}^n \varphi_i$ on V in conjunctive normal form (CNF), a weight function $w : [1, n] \rightarrow \mathbb{N}$.
- **Output:** a valuation $\nu : V \rightarrow \{0, 1\}$ such that $\sum_{\nu \models \varphi_i} w(i)$ is maximal.

Moreover, we will sometimes use a partially weighted variant of Max-SAT, which is defined in the following way:

6:14 On Distances Between Words with Parameters

- **Input:** a set V of literals, a satisfiable formula φ_c on V in CNF, a formula $\varphi_w = \bigwedge_{i=1}^n \varphi_i$ on V in CNF and a weight function $w : \llbracket 1, n \rrbracket \rightarrow \mathbb{N}$.
- **Output:** a valuation $\nu : V \rightarrow \{0, 1\}$ such that $\nu \models \varphi_c$ and $\sum_{\nu \models \varphi_i} w(i)$ is maximal.

In that case, clauses of φ_c are called “hard” clauses while clauses of φ_w are called “soft clauses”. We give a proof of the equivalence in Proposition 26 of Appendix C.

We will define an encoding of an instance (u, v) of PM^d such that an assignment of the variables of V will define an alignment between u and v . First, we make a link between the ID edit distance and the length of the optimal alignment between two strings.

► **Definition 20.** *Let u and v be two words on Π , such that $p = |u|$ and $p' = |v|$. A set $A \subset \llbracket 1, |u| \rrbracket \times \llbracket 1, |v| \rrbracket$ is an alignment between u and v iff the following are true:*

1. *Each position of u appears at most once: For all $1 \leq i \leq p$ and $1 \leq j, j' \leq p'$, if $(i, j) \in A$ and $(i, j') \in A$, then $j = j'$.*
2. *Each position of v appears at most once: For all $1 \leq j \leq p'$ and $1 \leq i, i' \leq p$, if $(i, j) \in A$ and $(i', j) \in A$, then $i = i'$.*
3. *There are no crossings: if $(i, j) \in A$, $(i', j') \in A$, and $i' > i$, then $j' > j$.*
4. *Aligned positions match in u and v : if $(i, j) \in A$, then $u_i = v_j$.*

An alignment relates to the insertion/deletion distance ID in the following way:

► **Theorem 21.** *Let u, v be words on Π and $k \leq |u| + |v|$ be an integer. The following are equivalent:*

1. *There exists an alignment A such that $2|A| = |u| + |v| - k$*
2. *$ID(u, v) \leq k$.*

Proof. The proof, which works by induction, can be found in Appendix C. ◀

We now turn to the max-SAT encoding of our problem.

► **Theorem 22.** *Let u and v be two words over Π . There exists a formula $\varphi_{u,v} = \varphi_c \wedge \varphi_w$ and a weight function w , instance of the partially weighted Max-SAT problem such that the following are equivalent:*

- *ν is a solution to this partially weighted Max-SAT instance and satisfies k clauses of φ_w*
- *There exists a function $f : \Pi \rightarrow \Pi$ and an alignment between $f(u)$ and v of size k .*

The formula φ uses $|m||p| + |\Pi|^2$ variables and is of size $O(m^2p^2)$, where $m = |u|$ and $p = |v|$. Moreover, there exists φ^{inj} of size $O(|\Pi|^3)$ such that the above result is true for f injective by replacing φ_c with $\varphi'_c = \varphi_c \wedge \varphi^{inj}$.

In particular, finding the valuation maximizing k gives a maximal alignment between u and v , and with Theorem 21, the distance $ID(u, v)$.

Proof. For this proof, we fix an ordering on the alphabet $\Pi = \{a_1, \dots, a_n\}$.

We define the set of literals V as $V = \{x_{i,j} \mid 1 \leq i \leq |u|, 1 \leq j \leq |v|\} \cup \{y_{a,b} \mid a \in \Pi, b \in \Pi\}$. Intuitively, $x_{i,j}$ represents a match between position i and j in the alignment, and $y_{a,b}$ will represent the fact that $f(a) = b$. We define the following sets of formulas, where all indices i are taken between 1 and m and all j between 1 and p , and a and b are taken in Π :

$$\begin{array}{lll}
\forall i \forall j' \neq j, & \varphi_{i,j,j'}^{A_1} \equiv x_{i,j} \implies \neg x_{i,j'} & \text{(NoDouble i)} \\
\forall j \forall i' \neq i, & \varphi_{i,i',j}^{A_2} \equiv x_{i,j} \implies \neg x_{i',j} & \text{(NoDouble j)} \\
\forall i' > i \forall j' < j, & \varphi_{i',i,j,j'}^C \equiv x_{i,j} \implies \neg x_{i',j'} & \text{(NoCrossing)} \\
\forall a \forall b \neq b', & \varphi_{a,b,b'}^f \equiv y_{a,b} \implies \neg y_{a,b'} & \text{(Function)} \\
\forall a \neq a' \forall \neq b, & \varphi_{a,a',b}^{inj} \equiv y_{a,b} \implies \neg y_{a',b} & \text{(Injectivity)} \\
\forall i \forall j, & \varphi_{i,j}^M \equiv x_{i,j} \implies y_{u_i,v_j} & \text{(Match)} \\
\forall i, & \varphi_i^{\exists} \equiv \bigvee_{1 \leq j \leq p} x_{i,j} & \text{(ExistsMatch)}
\end{array}$$

We then define φ_c as the conjunction of all the formulas (NoDouble i), (NoDouble j), (NoCrossing), (Function), and (Match). Furthermore, we define φ^{inj} as the conjunction of all the (Injectivity) formulas. Lastly, we define $\varphi_w = \bigwedge_{1 \leq i \leq m} \varphi_i^{\exists}$, and set $w(C) = 1$ for every clause C of φ_w .

There are $m \binom{p}{2}$ (NoDouble i) formulas, $p \binom{m}{2}$ (NoDouble j), $\binom{m}{2} \binom{p}{2}$ (NoCrossing), $n \binom{n}{2}$ (Function) and (Injectivity) formulas, pm (Match) formulas and n (ExistsMatch) formulas.

We now prove both implications of the theorem. Suppose ν is a valuation satisfying φ_c and k clauses of φ_w . We define, for all $a, b \in Pi$, $f(a) = b$ if and only if $\nu(y_{a,b}) = \top$. Since ν satisfies all the (Function) formulas, this is a correct definition of a (partial) function. We define $A = \{(i, j) \mid \nu(x_{i,j}) = \top\}$. A is an alignment between $f(u)$ and v . Indeed: (NoDouble i) and (NoDouble j) ensures point 1. and 2. of Definition 20, (NoCrossing) ensures point 3., and Match ensures point 4. The size of A is the number of $x_{i,j}$ instantiated to \top , which is exactly the number of clauses of φ_c satisfied, i.e., k .

Suppose now that there exists a function $\Pi \rightarrow \Pi$ and an alignment A between $f(u)$ and v . Similarly, we define $\nu(y_{a,b}) = \top$ if and only if $f(a) = b$, and $\nu(x_{i,j}) = \top$ if and only if $(i, j) \in A$. Since A is an alignment, ν satisfies (NoDouble i), (NoDouble j), and (NoCrossing). Since f is a function, (Function) is satisfied. Finally, if $\nu(x_{i,j}) = \top$, then $(i, j) \in A$, and since A is a matching, $f(u)_i = f(u_i) = v_j$ and $\nu(y_{u_i,v_j}) = \top$.

The proof for φ^b is the same, and (Injectivity) ensures the injectivity of f . \blacktriangleleft

What is more, this proof can be adapted to change the ID distance to the Levenshtein distance, simply by choosing to consider all the (Match) formulas as soft clauses.

6 Experiments

The two approaches presented in Section 5 were implemented in Python to solve PM^{ID} . They are available under the GPL license at <https://github.com/AaronFive/paramatch>. The FPT algorithm of Section 5.1 is implemented in the function `parameterizedAlignment` of file `fpt_alphabet_size.py`. The MaxSAT-reduction of Section 5.2 is implemented in the function `make_sat_instance` of file `sat_instance.py`. The MaxHS solver [18] available at <http://www.maxhs.org> is used by our script to solve the MaxSAT instances derived from the PM^{ID} instances.

Our initial motivation to introduce parameterized matching under various distances is theater play comparison. To represent the structure of a theater play, we represent each character by a letter of the alphabet, and create the parameterized word obtained by considering the succession of all consecutive speakers. To check their adequacy with real data,

we use a corpus of theater plays in which each character is represented by one letter of the alphabet, and each act of the play is represented by a string corresponding to the sequence of speaking characters. A letter may be duplicated in this string if the corresponding characters has lines in the end of a scene and in the beginning of the next one. Therefore, the edit distance between two parameter words representing acts will be small if both acts have a similar structure in terms of succession of speaking characters. We selected a corpus of 10 pairs of plays where one inspired the other, and performed 47 comparisons between pairs of acts. Among those comparisons, 26 were solved by the maxSAT algorithm and all by the FPT algorithm (detailed results are presented in the supplementary material available at <https://github.com/AaronFive/paramatch/tree/main/corpus10pairs>), with a 800 second timeout. The computation times are obtained on a XMG laptop running on Windows, with a 2.60 Ghz processor and 16 Gb RAM. Only the running time of MaxHS is provided, the encoding into a MaxSAT formula usually runs in approximately 1 second. Note that all instances are solved faster by the FPT algorithm than by the MaxSAT approach. The analysis of running times depending on the product of the lengths of the input strings (see supplementary material) shows that the MaxSAT approach may be relevant for strings with more than 10 distinct characters, but where the product of the length of input strings may not exceed 2000.

7 Conclusion

In this paper, we studied the complexity of several variants of the edit distance problem between parameterized words. We proved the NP-completeness of all previously unsolved cases, including the Levenshtein distance left open in [24], and provided practical approaches to solve real instances of those problems. We also studied similar problems for various definitions of words with parameters, namely parameter words and parameterized expressions, proving some relationships with parameterized word problems.

As future work, we will study the restrictions introduced in [21, 22] for a pattern matching problem with patterns in the parameter, in order to obtain polynomial time algorithms for the edit distance between parameterized words. Moreover, we will explore the question of distance between sets of words, in particular when they are defined through generalizations of automata. These problems are variants of the notion of distance between regular languages as defined in [12]. In this context, we can notice that different notions of automata can be considered: either automata generating parameterized words, or automata using parameters to define languages over classical words, with two different semantics as defined in [11].

References

- 1 Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *International conference on foundations of data organization and algorithms*, pages 69–84. Springer, 1993.
- 2 Amihood Amir, Yonatan Aumann, Richard Cole, Moshe Lewenstein, and Ely Porat. Function matching: Algorithms, applications, and a lower bound. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, pages 929–942, 2003. doi:10.1007/3-540-45061-0_72.
- 3 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994. doi:10.1016/0020-0190(94)90086-8.
- 4 Dana Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980. doi:10.1016/0022-0000(80)90041-0.

- 5 Alberto Apostolico, Péter L. Erdős, and Moshe Lewenstein. Parameterized matching with mismatches. *Journal of Discrete Algorithms*, 5(1):135–140, 2007. doi:10.1016/j.jda.2006.03.014.
- 6 Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 71–80, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/167088.167115.
- 7 Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26:1343–1362, 1997.
- 8 Brenda S. Baker. Parameterized diff. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 854–855, USA, 1999. Society for Industrial and Applied Mathematics.
- 9 Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying graph patterns. In Maurizio Lenzerini and Thomas Schwentick, editors, *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2011)*, pages 199–210. ACM, 2011. doi:10.1145/1989284.1989307.
- 10 Pablo Barceló, Juan Reutter, and Leonid Libkin. Parameterized regular expressions and their languages. *Theoretical Computer Science*, 474:21–45, 2013. doi:10.4230/LIPIcs.FSTTCS.2011.351.
- 11 Pablo Barceló, Leonid Libkin, and Juan Reutter. Parameterized regular expressions and their languages. *Theoretical Computer Science*, 474:21–45, 2011. doi:10.1016/j.tcs.2012.12.036.
- 12 Michael Benedikt, Gabriele Puppis, and Cristian Riveros. The cost of traveling between languages. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Automata, Languages and Programming*, pages 234–245, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- 13 Michael Benedikt, Gabriele Puppis, and Cristian Riveros. Regular repair of specifications. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 335–344, 2011. doi:10.1109/LICS.2011.43.
- 14 Michael Benedikt, Gabriele Puppis, and Cristian Riveros. Bounded repairability of word languages. *Journal of Computer and System Sciences*, 79(8):1302–1321, 2013. doi:10.1016/j.jcss.2013.06.001.
- 15 Francine Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words (Discrete Mathematics and Its Applications)*. Chapman, Hall/CRC, 2007.
- 16 William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. *SIGMOD Rec.*, 27(2):201–212, 1998. doi:10.1145/276305.276323.
- 17 Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Trans. Algorithms*, 11(2), October 2014. doi:10.1145/2650220.
- 18 Jessica Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction*. PhD thesis, University of Toronto, Canada, 2014. URL: <http://hdl.handle.net/1807/43539>.
- 19 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. PbwT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.
- 20 M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.
- 21 Pawel Gawrychowski, Florin Manea, and Stefan Siemer. Matching patterns with variables under hamming distance. In Filippo Bonchi and Simon J. Puglisi, editors, *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, volume 202 of *LIPIcs*, pages 48:1–48:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.48.

- 22 Pawel Gawrychowski, Florin Manea, and Stefan Siemer. Matching patterns with variables under edit distance. In Diego Arroyuelo and Barbara Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2022. doi:10.1007/978-3-031-20643-6_20.
- 23 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. URL: <https://www.wikidata.org/entity/Q55980413>.
- 24 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate parameterized matching. *ACM Trans. Algorithms*, 3(3):29–es, 2007. doi:10.1145/1273340.1273345.
- 25 Wilbert Jan Heeringa. *Measuring dialect pronunciation differences using Levenshtein distance*. PhD thesis, University of Groningen, 2004.
- 26 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. On the longest common parameterized subsequence. *Theoretical Computer Science*, 410(51):5347–5353, 2009. doi:10.1016/j.tcs.2009.09.011.
- 27 Lillian Jane Lee. *Similarity-based approaches to natural language processing*. PhD thesis, Harvard University, 1997.
- 28 Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10(8), pages 707–710. Soviet Union, 1966.
- 29 Moshe Lewenstein. *Parameterized Matching*, pages 635–638. Springer US, Boston, MA, 2008. doi:10.1007/978-0-387-30162-4_282.
- 30 Florin Manea and Markus L. Schmid. Matching patterns with variables. In Robert Mercas and Daniel Reidenbach, editors, *Combinatorics on Words - 12th International Conference, WORDS 2019, Loughborough, UK, September 9-13, 2019, Proceedings*, volume 11682 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2019. doi:10.1007/978-3-030-28796-2_1.
- 31 Juan Mendivelso, Sharma V. Thankachan, and Yoan Pinzón. A brief history of parameterized matching problems. *Discrete Applied Mathematics*, 274:103–115, 2020. Stringology Algorithms. doi:10.1016/j.dam.2018.07.017.
- 32 Mehryar Mohri. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(06):957–982, 2003.
- 33 Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
- 34 Sascha Schimke, Claus Vielhauer, and Jana Dittmann. Using adapted levenshtein distance for on-line signature authentication. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004)*, volume 2, pages 931–934. IEEE, 2004.
- 35 T. Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica (New York)*, 39(1):1–19, 2004. doi:10.1007/s00453-003-1067-9.
- 36 Bernd Voigt. The partition problem for finite abelian groups. *Journal of Combinatorial Theory, Series A*, 28(3):257–271, 1980.
- 37 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.
- 38 Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

A Details of the proofs

Proof of Lemma 7. We proceed by induction on k . If $k = 0$, then u and v are parameterized matching, and $f(u) = v$, thus $d(f(u), v) = 0$. Suppose the result holds until a fixed k . Suppose $PM^d(u, v) = k + 1$. There exist f , u'' and u' such that $d(u, u'') = 1$, $d(u'', u') = k$, and $f(u') = v$. Hence $PM^d(u'', v) \leq k$, and by induction hypothesis $d(f(u''), v) \leq k$. Moreover,

since $d(u, u'') = 1$, we get u'' from u by applying only one operation. We prove that regardless of this operation, $d(f(u), f(u'')) = 1$, and thus $d(f(u), v) \leq d(f(u), f(u'')) + d(f(u''), v) \leq k+1$ which will conclude the proof. There are 3 cases to consider:

- If the operation is a deletion, $u = v_1 x v_2$ and $u'' = v_1 v_2$ for some words v_1 and v_2 and some letter x . Then $f(u) = f(v_1) f(x) f(v_2)$ and we can obtain $f(v_1) f(v_2) = f(u'')$ by deleting $f(x)$.
- If it is an insertion, $u = v_1 v_2$ and $u'' = v_1 x v_2$, and we can similarly go from $f(u)$ to $f(u'')$ by inserting $f(x)$.
- If it is a substitution, $u = v_1 x v_2$ and $u'' = v_1 y v_2$, and we can go from $f(u)$ to $f(u'')$ by replacing $f(x)$ with $f(y)$.

Hence $d(f(u), f(u'')) = 1$, which concludes the proof for PM^d .

Since this proof does not use the fact that f is 1-to-1, it also stands for FM_1^d . ◀

Proof of Lemma 8. It is obvious that $d(zu, zv) \leq d(u, v)$, so we only prove $d(u, v) \leq d(zu, zv)$. We prove that any rewriting sequence from zu to zv can be modified such that no edit operation is applied in z . This will be enough to prove the result, as the edit sequence obtained can be seen as an edit sequence between u and v . We proceed by induction on the size of z . Suppose $|z| = 1$. Then $z = a \in \Sigma \cup \Pi$. We can consider that no character is modified twice in an edit sequence (i.e. no character is inserted and then deleted, or inserted and then substituted etc.), as that is always sub-optimal. Suppose z is modified. There are 3 possible cases:

1. There is an insertion in z , hence a word w ends up being inserted before a . Since $zv = av$ starts with a , w must start with an a , hence $w = aw'$. We insert $w'a$ to the right of z instead with the same operations. If z should be deleted or substituted, we apply the same operation to the new a instead. These operations yield the same result, and do not modify z .
2. There is a deletion in z , and hence a is deleted. Since this an optimal rewriting sequence, no a is created at that position through insertion or substitution afterwards. Since av starts with an a , u must be of the form $u = sau'$, where all the characters in s are deleted, and a isn't. Deleting sa instead of as yields the same result, and doesn't modify z .
3. There is a substitution in z , hence a is modified into a character $b \neq a$, that will not be further modified. Since av starts with a , an a has to be inserted in z , which is handled in case 1.

Hence, we can consider that every edit operations is done in u , and $d(au, av) = d(u, v)$. Suppose now that the result is proven for $|z| = k$, and let $z = az'$, with $|z'| = k$. Using the base case and the case for $|z| = k$, we have $d(zu, zv) = d(az'u, az'v) = d(zu, zv) = d(u, v)$, which concludes the proof. ◀

Proof of Lemma 10. Let $\#$ be a fresh parameterized letter. Let then $u = \#^{k+1} u_1 \# u_2 \# \dots \# u_n$ and $v = \#^{k+1} v_1 \# v_2 \# \dots \# v_n$, where $\#^{k+1}$ denotes $k+1$ repetitions of the character $\#$. The proof of the reverse direction is the same as in Lemma 9, so we only prove the other one.

Assume $FM_2^D(u, v) \leq k$. Let v' and f realize this parameterized match.

We prove that $f(\#) = \#$, and that no other character is sent to $\#$ by f . Indeed, v starts with $k+1$ symbols $\#$, which ensure that v' starts with the letter $\#$. Since u starts with $\#$ and $f(u) = v'$, $f(\#) = \#$. Furthermore, this implies that since $|u|_{\#} = k+n$, $|f(u)|_{\#} = |v'|_{\#} \geq k+n$. Since v' is obtained from v by deletions, we have $|v'|_{\#} \leq |v|_{\#} = k+n$. Hence $|v'|_{\#} = k+n$ and all those inequalities are equalities, which is only the case when no $\#$ symbols is deleted from v , and that for all $x \neq \#$, $f(x) \neq \#$.

6:20 On Distances Between Words with Parameters

Since all the $\#$ symbols are left untouched, the rest of the proof is the same as in Lemma 9, and all of the factors u_i and v_i are parameterized matching. \blacktriangleleft

Proof of Lemma 11. Let $\#$ be a fresh parameterized letter, and $N = k + 2$.

Let then $u = \#^N u_1 \#^N u_2 \dots \#^N u_n \#^N$ and $v = \#^N v_1 \#^N v_2 \dots \#^N v_n \#^N$. Once again, we only prove the non-trivial implication.

Suppose $FM_1^D(u, v) \leq k$, and let f and u' realize this matching. Since u starts with $k + 1$ copies of $\#$, u' starts with $\#$. Since v starts with $\#$ too, $f(\#) = \#$.

We now prove that we can consider that for all $x \neq \#$, $f(x) \neq \#$. This will also imply that no $\#$ symbol is deleted from u . Let $S = \{a \in \Pi \mid f(a) = \#\}$ be the set of symbols (different from $\#$) sent to $\#$. Since $|u|_{\#} = |v|_{\#}$, the number of deleted $\#$ symbols from u is exactly $|u|_S$, hence $|u|_S \leq k$. Let us now consider the leftmost occurrence of an element of S in u' , that we denote by a . The letter a appears in u in a factor of the form $\#^N w_1 a w_2 \#^N$. Since all $\#$ in v appear in blocks of size N , a must contribute to such a block, after deletions and application of f . We distinguish two cases:

1. The entirety of the word w_1 is deleted. In this case, at least one symbol $\#$ from the left $\#^N$ block is deleted; otherwise $f(\#^N)f(a) = \#^{N+1}$ would be a factor of v , which is impossible. Thus, choosing not to delete $\#$ and to delete a instead yields the same result.
2. w_1 is not deleted. Since no character from S appears to the left of a , $f(a)$ is the start of a $\#^N$ block. Furthermore, since $|u|_S \leq k$, it is not possible to form $\#^N$ with only a and w_2 , and characters from the right $\#^N$ contribute to it. Hence, at least one $\#$ symbol from this right block is deleted. Like before, the same result can be obtained by not deleting it, and deleting a instead.

Either way, we can repeat this process to eliminate all occurrences of characters of S and of deletions of $\#$, which proves that we can consider that for all $x \neq \#$, $f(x) \neq \#$. Once again, we are taken back to the conditions of Lemma 9, and the rest of the proof follows. \blacktriangleleft

Proof of Theorem 15. We define Π like in Theorem 12, and we add the letters $\perp_1, \perp_2, \perp_3, \perp_4$ and \perp_5 . Similarly, we define $u_1^i, v_1^i, u_{\perp}^i, v_{\perp}^i, u_2^e$, and v_2^e just like in Theorem 12. Additionally, we define for every edge e ,

$$u_{\perp}^e = \square_1^e \square_2^e \square_3^e \square_4^e \square_5^e \square_6^e \text{ and } v_{\perp}^e = \perp_1 \perp_2 \perp_3 \perp_4 \perp_5.$$

We then apply Lemma 11 with

$$u_1^1, \dots, u_1^n, u_{\perp}^1, \dots, u_{\perp}^n, u_2^{e_1}, \dots, u_2^{e_m}, u_{\perp}^{e_1}, \dots, u_{\perp}^{e_m}$$

and

$$v_1^1, \dots, v_1^n, u_{\perp}^1, \dots, v_{\perp}^n, v_2^{e_1}, \dots, v_2^{e_m}, v_{\perp}^{e_1}, \dots, v_{\perp}^{e_m}$$

to obtain u , v , and k . We show that G is 3-colorable $\Leftrightarrow FM_1^D(u, v) \leq k$.

\Rightarrow Suppose G is 3 colorable. Define f like in Theorem 12 on the c_y^i and \square_y^e . Let e be an edge and $k_e \in [1, 6]$ be the integer such that $f(\square_{k_e}^e)$ is defined. We map every remaining \square_y^e in the following way:

$$f(\square_i^e) = \begin{cases} \perp_i & \text{if } i < k_e, \\ Y^e & \text{if } i = k_e, \\ \perp_{i-1} & \text{if } i > k_e. \end{cases} \quad (1)$$

It is then easy to check that $d(f(u), v) = k$, and thus $FM_1^D(u, v) \leq k$.

\Leftarrow Suppose $FM_1^D(u, v) \leq k$, and let f and u' realize it. We define a coloring of G based on f . We note, for $1 \leq i \leq n$ and $1 \leq t \leq 3$, $col(c_t^i) = c_t$. If x_i is a vertex of G , define $c(x_i)$ to be $col(c_k^i)$, where c_k^i is the only element such that $f(c_k^i) = x_i$. We show in what follows that (1) this function definition is correct and (2) it is a valid coloring, i.e. if $e = \{x_i, x_j\}$ is an edge, $c(x_i) \neq c(x_j)$.

(1): The same points 1. and 2. from the proof of Theorem 12 apply, hence for every $1 \leq i \leq n$, exactly one element from $\{c_1^i, c_2^i, c_3^i\}$ is sent to x_i , while the two others are sent to \perp_1^i and \perp_2^i , hence the result.

(2): Let e be an edge. The words u_\perp^e and v_\perp^e are in matching, which is done with exactly one deletion. Hence, there exists k_e such that

$$f(\square_i^e) = \begin{cases} \perp_i & \text{if } i < k_e, \\ \perp_{i-1} & \text{if } i > k_e. \end{cases} \quad (2)$$

Moreover, u_2^e and v_2^e are in matching. Since Y^e appears in v_2^e and all the characters in u_2^e apart from $\square_{k_e}^e$ have an image different from Y^e , $f(\square_{k_e}^e) = Y^e$. Hence, the only characters that are not suppressed from u_2^e are the two characters between the $\square_{k_e}^e$. Denoting them by c and c' , the construction of the word ensures that $col(c) \neq col(c')$. Hence, if $e = \{x_i, x_j\}$, we have proven $c(x_i) \neq c(x_j)$, which is (2).

The coloring c is therefore valid, which concludes the proof. \blacktriangleleft

Proof of Theorem 17. Let $G = (V, E)$, with $V = \{x_1, \dots, x_n\}$ and $\{e_1, \dots, e_m\}$. Like in the 1-to-1 case, we construct factors u_i and v_i to encode vertex coloring. The parameter alphabet contains:

- x_1, \dots, x_n , corresponding to V ,
- the colors c_1, c_2, c_3 ,
- for every $e \in E$, the delimiters Y^e ,
- for every $e \in E$ and every $1 \leq i, j \leq 3$, $i \neq j$, the delimiters $Y_{i,j}^e$.

We define for $1 \leq i \leq n$, $u_1^i = x_i$ and $v_1^i = c_1 c_2 c_3$. If e is an edge and c_i and c_j are two colors, we denote $w^e(c_i, c_j) = Y_{i,j}^e Y_{i,j}^e Y_{i,j}^e c_i c_j Y_{i,j}^e Y_{i,j}^e Y_{i,j}^e$. For every edge $e = \{x_i, x_j\}$, we now define $u_2^e = Y^e Y^e Y^e x_i x_j Y^e Y^e Y^e$ and $v_2^e = w^e(c_1, c_2) w^e(c_1, c_3) w^e(c_2, c_1) w^e(c_2, c_3) w^e(c_3, c_1) w^e(c_3, c_2)$.

We now apply Lemma 10 with $u_1^1, \dots, u_1^n, u_2^{e_1}, \dots, u_2^{e_m}, v_1^1, \dots, v_1^n, v_2^{e_1}, \dots, v_2^{e_m}$, to obtain u and v .

\Rightarrow Suppose G is 3-colorable, and let $c : V \rightarrow \{c_1, c_2, c_3\}$ be a valid coloring. Define $f|_V = c$. For every edge $e = \{x_i, x_j\}$, let s and t be such that $c(x_i) = c_s$ and $c(x_j) = c_t$. We then define $f(Y^e) = Y_{s,t}^e$. It is easy to check now that $d(f(u), v) = k$, and hence $FM_2^D(u, v) \leq k$.

\Leftarrow Suppose now that $FM_2^D(u, v) \leq k$. We will show that $f|_V$ defines a 3-coloring of G , by showing that (1) for all $x \in V$, $f(x) \in \{c_1, c_2, c_3\}$ and (2) If $\{x, y\} \in E$, then $f(x) \neq f(y)$.

- Lemma 10 ensures that the words u_i and v_i are in matching, which proves (1).
- Lemma 10 also ensures that the words u^e and v^e are in matching. Let $e \in E$, with $e = x_s, x_t$. We have $|u_2^e|_{Y^e} = 6$, hence $|f(u_2^e)|_{f(Y^e)} \geq 6$. Since c_1, c_2 and c_3 each occur exactly 4 times in v_2^e , they cannot occur 6 times after deletions, and $f(Y^e) \notin \{c_1, c_2, c_3\}$. Hence, there exist $i \neq j$ with $1 \leq i, j \leq 3$ such that $f(Y^e) = Y_{i,j}^e$. This implies that all but one of the w^e factors from v_2^e are suppressed, and that the remaining one is $w^e(c_i, c_j)$. Hence $f(x_s) = c_i$ and $f(x_t) = c_j$, which proves (2). \blacktriangleleft

B

 Encoding Constant Alphabet Σ in Π

We show why it is always possible to consider that $\Sigma = \emptyset$ for certain problems. These results use the lemmas proved in Section 4.1.

► **Lemma 23.** *Let d be a distance, k an integer and u and v be two parameterized words over the alphabet of constants Σ and the alphabet of parameters Π . There exist words \tilde{u} and \tilde{v} over the alphabet of constants \emptyset and the alphabet of parameters $\Pi' = \Pi \uplus \Sigma$ such that the following are equivalent:*

- $PM^d(u, v, k)$ is realized by f ;
- $PM^d(\tilde{u}, \tilde{v}, k)$ is realized by f .

In particular, this implies that if $PM^d(\tilde{u}, \tilde{v}) \leq k$, all functions f realizing this matching verify that for all $x \in \Sigma$, $f(x) = x$, and for all $x \in \Pi$, $f(x) \in \Pi$.

Proof. Let $N = k + 1$. If $\Sigma = \{a_1, \dots, a_n\}$, we define z to be $a_1^N a_2^N \dots a_n^N u$ and $\tilde{u} = zu$, $\tilde{v} = zv$. It is clear that if $PM^d(u, v) \leq k$ then $PM^d(\tilde{u}, \tilde{v}) \leq k$, by following the same operations, and applying the same renaming function.

Suppose now that $PM^d(\tilde{u}, \tilde{v}) \leq k$, and let f and u' realize it. Let $i \in [1, n]$. All the letters of u between position Ni and $N(i + 1)$ are a_i . At most k of these positions can be modified with an edit operation. Since $N > k$, at least one of these positions is not modified, and thus there exists $j \in [Ni, N(i + 1)]$ such that $u'_j = a_i$. Since all letters in v between position Ni and $N(i + 1)$ are a_i , in particular $v_j = a_i$, and hence $f(a_i) = a_i$. This proves that for all $x \in \Sigma$, $f(x) = x$, and thus $f(z) = z$. Since f is 1-to-1, this entails $f(\Pi) \subseteq \Pi$. By Lemma 7, $d(f(\tilde{u}), \tilde{v}) \leq k$. Hence $d(f(zu), zv) = d(zf(u), zv) \leq k$ and by Lemma 8, $d(f(u), v) \leq k$. Hence $PM^d(u, v) \leq k$. ◀

► **Remark 24.** Note that the words \tilde{u} and \tilde{v} have a size increased by $N\Sigma$. If less operations are considered, it is possible to reduce this overhead. For example, in the case of PM^D , we can take z to be of the form $a_1 \dots a_n z^N$, to reduce the overhead to $N + \Sigma$.

Similarly, constants can be encoded in Π in some FM problems. We prove this result for FM_2^D , with the help of the block decomposition allowed by Lemma 10.

► **Lemma 25.** *Let u and v be two parameterized words over the alphabet of constants Σ and the alphabet of parameters Π . There exist words \tilde{u} and \tilde{v} over the alphabet of constants \emptyset and the alphabet of parameters $\Pi' = \Pi \uplus \Sigma$ such that the following are equivalent:*

- $FM_2^D(u, v, |v| - |u|)$ is realized by f ;
- $FM_2^D(\tilde{u}, \tilde{v}, |\tilde{v}| - |\tilde{u}|)$ is realized by f .

Proof. We write $\Sigma = \{a_1, \dots, a_n\}$ and $\Pi = \{b_1, \dots, b_m\}$. We define $z_\Sigma = a_1 \dots a_n$, and $z_\Pi = b_1 \dots b_m$. Let \tilde{u} and \tilde{v} be the words obtained by applying Lemma 10 to $z_\Sigma, b_1, b_2, \dots, b_m, u$ and $z_\Sigma, z_\Pi, z_\Pi, \dots, z_\Pi, v$. If $FM_2^D(u, v, k)$ is realized by a function f , it realizes $FM_2^D(\tilde{u}, \tilde{v}, |\tilde{v}| - |\tilde{u}|)$ too. Indeed, it is enough to apply the same operations in v , and to delete all the characters but $f(b_i)$ in the i -th copy of z_Π .

Suppose now that $FM_2^D(\tilde{u}, \tilde{v}) \leq k$, and let f realize it. Then, by Lemma 10, we have:

- $D(z, f(z)) = 0$, and hence $f(z) = z$, which implies that for all $x \in \Sigma$, $f(x) = x$.
- For every $1 \leq i \leq m$, $D(z_\Pi, f(b_i)) = |\Pi| - 1$. Hence $f(b_i)$ is a character of z_Π , which is some character $b_j \in \Pi$.
- $D(v, f(u)) \leq k$.

Hence f verifies $D(f(v), u) \leq k$ and respects the conditions on Π and Σ , which implies that it is also realizes $FM_2^D(u, v, k)$. ◀

The overhead to pay for this transformation is $O(|\Sigma| + |\Pi|^2 + k)$, where the term in k comes from the proof of Lemma 10.

Transposing the technique used for Lemma 25 is not sufficient to get a similar result for FM_1^D . The question thus remains open in this context.

C Proofs Regarding the Max-SAT Encoding

Proof of theorem 21. We proceed by induction on $|u| + |v|$. If $|u| + |v| = 0$, both u and v are the empty string, and the equivalence is trivial. Fix $n \in \mathcal{N}$ and suppose now that the result holds up for all words u, v such that $|u| + |v| \leq n - 1$. Let u and v be two words such that $|u| + |v| \leq n$. Without loss of generality, consider $|u| \geq |v|$.

Suppose $ID(u, v) \leq k$. Let ρ be a rewriting sequence between u and v of length k . If there is no deletion in u in ρ , there are only insertions in v , and v is a sub-word of u , and there exists another rewriting sequence ρ' only deleting letters from u . Hence, we can consider that there is at least a deletion in u in ρ . Let p be a position at which such a deletion occur, and let $a = u_p$. The word u can be written as $u = u'au''$ for some words u' and u'' . Define $w = u'u''$. It holds that $d(w, v) \leq k - 1$ and $|w| = |u| - 1$. By induction, there exists an alignment A between w and v such that $2|A| = |w| + |v| - (k - 1) = |u| + |v| - k$. We define

$$r(i) = \begin{cases} i & \text{if } i < p \\ i - 1 & \text{if } i > p \end{cases}, \text{ and } B = \{(r(i), j) \mid (i, j) \in A\}. \text{ Since } A \text{ is an alignment, so is } B: \text{ it}$$

satisfies conditions 1 to 3 of Definition 20, and since $w_r(i) = u_i$, it also satisfies condition 4. Finally, $|B| = |A|$, hence $2|B| = |u| + |v| - k$, hence the result.

Suppose now that there exists an alignment A such that $2|A| = |u| + |v| - k$. Similarly, consider p , a position in u such that there does not exist a j with $(p, j) \in A$. If no such position exist, since $|u| \geq |v|$, $u = v$ and the result is proven. Consider w the word obtained by deleting u_p from u . It then holds that $|w| = |u| - 1$ and that $2|A| = |u| + |v| - k = |w| + |v| - (k - 1)$. Defining B in the same way as above yields an alignment between w and v of the same size, and thus by induction, $d(w, v) \leq k - 1$, and since $d(u, w) = 1$, $d(u, v) \leq k$. ◀

► **Proposition 26.** *Weighted Max-SAT and partial weighted Max-SAT are equivalent.*

Proof. Encoding a weighted Max-SAT instance as a partially weighted Max-SAT instance is straightforward, as we just have to choose φ_c to be empty.


Conversely, given a satisfiable CNF formula φ_c , a CNF formula φ_w , and a weight function w on the clauses of φ_w , we can define a weighted Max-Sat instance in the following way:

- We define $\varphi = \varphi_c \wedge \varphi_w$
- We set $W = 1 + \sum_{C_i \text{ clause of } \varphi_c} w(C_i)$, and extend w to clauses of φ_c such that $w(C_j) = W$ for all clauses C_j of φ_c

If ν is a valuation, we denote by $w(\nu)$ the sum of the weights of all clauses it satisfies $\sum_{\nu \models C_i} w(C_i)$.

Since φ_c is satisfiable, there exists a valuation ν_c such that $\nu_c \models \varphi_c$, and $w(\nu_c) \geq |\varphi_c|W$. Let now ν be a valuation no satisfying a clause of φ_c . Then $w(\nu) \leq (|\varphi_c| - 1)W + (W - 1) < w(\nu_c)$, hence ν_c is not maximal and cannot be a solution to the weighted Max-SAT instance. ◀

Parameterized Algorithms for String Matching to DAGs: Funnels and Beyond

Manuel Cáceres  

Department of Computer Science, University of Helsinki, Finland

Abstract

The problem of String Matching to Labeled Graphs (SMLG) asks to find all the paths in a labeled graph $G = (V, E)$ whose spellings match that of an input string $S \in \Sigma^m$. SMLG can be solved in quadratic $O(m|E|)$ time [Amir et al., JALG 2000], which was proven to be optimal by a recent lower bound conditioned on SETH [Equi et al., ICALP 2019]. The lower bound states that no strongly subquadratic time algorithm exists, even if restricted to directed acyclic graphs (DAGs).

In this work we present the first parameterized algorithms for SMLG on DAGs. Our parameters capture the topological structure of G . All our results are derived from a generalization of the Knuth-Morris-Pratt algorithm [Park and Kim, CPM 1995] optimized to work in time proportional to the number of prefix-incomparable matches.

To obtain the parameterization in the topological structure of G , we first study a special class of DAGs called funnels [Millani et al., JCO 2020] and generalize them to k -funnels and the class ST_k . We present several novel characterizations and algorithmic contributions on both funnels and their generalizations.

2012 ACM Subject Classification Theory of computation \rightarrow Parameterized complexity and exact algorithms; Theory of computation \rightarrow Pattern matching; Mathematics of computing \rightarrow Graph algorithms

Keywords and phrases string matching, parameterized algorithms, FPT inside P, string algorithms, graph algorithms, directed acyclic graphs, labeled graphs, funnels

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.7

Funding This work was partially funded by the Academy of Finland (grants No. 352821, 328877).

Acknowledgements I am very grateful to Alexandru I. Tomescu for initial discussions on funnel algorithms, to Veli Mäkinen for discussions on applying KMP on DAGs, to Massimo Equi and Nicola Rizzo for the useful discussions, and to the anonymous reviewers for their useful comments.

1 Introduction

Given a labeled graph $G = (V, E)^1$ and a string S of length m over an alphabet Σ of size σ , the problem of *String Matching to Labeled Graph (SMLG)* asks to find all paths in G spelling S in their characters; such paths are known as *occurrences* or *matches* of S in G . This problem is a generalization of the classical *string matching (SM)* to a text T of length n , which can be encoded as an SMLG instance with a path labeled with T . Labeled graphs are present in many areas such as information retrieval [28, 73, 14], graph databases [11, 10, 75, 16] and bioinformatics [27], and SMLG is a primitive operation to locate information on them.

It is a textbook result [2, 29, 81] that the classical SM can be solved in linear $O(n + m)$ time. For example, the well-known *Knuth-Morris-Pratt* algorithm (KMP) [60] preprocesses S and then scans T while maintaining the longest matching prefix of S . However, for SMLG a recent result [12, 34] shows that there is no strongly subquadratic $O(m^{1-\epsilon}|E|)$, $O(m|E|^{1-\epsilon})$ time algorithm unless the *strong exponential time hypothesis (SETH)* fails, and the most

¹ We consider the case where each vertex is labeled with a single character from Σ .



© Manuel Cáceres;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 7; pp. 7:1–7:19

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

efficient current solutions [8, 72, 77, 52] match this bound, thus being optimal in this sense. Moreover, these algorithms solve the approximate version of SMLG (errors in S only) showing that both problems are equally hard under SETH, which is not the case for SM [13].

The history of (exact) SMLG. SMLG can be traced back to the publications of Manber and Wu [67] and Dubiner et al. [33] where the problem is defined for the first time, and solved in linear time on directed trees by using an extension of KMP. Later Akutsu [4] used a sampling on V and a suffix tree of S to solve the problem on (undirected) trees in linear time and Park and Kim [74] obtained an $O(N + m|E|)^2$ time algorithm for directed acyclic graphs (DAGs) by extending KMP on a topological ordering of G (we call this the *DAG algorithm*). Finally, Amir et al. [8] showed an algorithm with the same running time for general graphs with a simple and elegant idea that was later used to solve the approximate version [77, 52], and that has been recently generalized as the labeled product [78]. The lower bound of Equi et al. [34] shows that the problem remains quadratic (under SETH) even if the problem is restricted to deterministic DAGs with vertices of two kinds: indegree at most 1 and outdegree 2, and indegree 2 and outdegree at most 1 [34, Theorem 1], or if restricted to undirected graphs with degree at most 2 [34, Theorem 2]. Furthermore, they show how to solve the remaining cases (in/out-trees whose roots can be connected by a cycle) in linear time by an extension of KMP. Later they showed [35, 36] that the quadratic lower bound holds even when allowing polynomial indexing time.

An (important) special case. Gagie et al. [42] introduced *Wheeler graphs* as a generalization of prefix sortable techniques [38, 45, 37, 22, 83] applied to labeled graphs. On Wheeler graphs, SMLG can be solved in time $O(m \log |E|)$ [42] after indexing, however, it was shown that the languages recognized by Wheeler graphs (intuitively the set of strings they encode) are very restrictive [5, 6]. Later, Cotumaccio and Prezza [31] generalized Wheeler graphs to *p-sortable graphs*, capturing every labeled graph by using the parameter p : the minimum width of a colex relation over the vertices of the graph. On p -sortable graphs, SMLG can be solved in time $O(mp^2 \log(p\sigma))$ after indexing, however, the problems of deciding if a labeled graph is Wheeler or p -sortable are NP-hard [44]. In a recent work, Cotumaccio [30] defined *q-sortable graphs* as a relaxation of p -sortable ($q < p$), which can be indexed in $O(|E|^2 + |V|^{5/2})$ time but still solve SMLG in time $O(mq^2 \log(q\sigma))$.

1.1 Our results

We present parameterized algorithms for SMLG on DAGs. Our parameters capture the topological structure of G . These results are related to the line of research “FPT inside P” [43, 26, 40, 61, 1, 23, 25, 24, 66, 64] of finding parameterizations for polynomially-solvable problems.

All our results are derived from a new version of the DAG algorithm [74], which we present in Section 3. Our algorithm is optimized to only carry *prefix-incomparable* matches (Definition 3) and process them in time proportional to their size (Lemma 5 further optimized in Lemma 6 and Theorem 7). Prefix-incomparable sets suffice to capture all prefix matches

² N is the total length of the labels in G in a more general version of the problem where vertices are labeled with strings.

of S ending in a vertex v (Definition 4). By noting that the size of prefix-incomparable sets is upper-bounded by the structure of S (Lemma 27 in Appendix A), we obtain a parameterized algorithm (Theorem 28 in Appendix A) that beats the DAG algorithm on periodic strings.

To obtain the parameterization on the topological structure of the graph we first study and generalize a special class of DAGs called *funnels* in Section 4.

Funnels. Funnels are DAGs whose source-to-sink paths contain a *private* edge that is not used by any other source-to-sink-path. Although more complex than in/out-forests, their simplicity has allowed to efficiently solve problems that remain hard even when the input is restricted to DAGs, including: DAG partitioning [70], k -linkage [70], minimum flow decomposition [57, 56], a variation of network inhibition [62] and SMLG (this work). Millani et al. [70] showed that funnels can also be characterized by a partition into an in-forest plus an out-forest (the *vertex partition* characterization), or by the absence of certain forbidden paths (the *forbidden path* characterization), and propose how to find a minimal forbidden path in quadratic $O(|V|(|V| + |E|))$ time and a recognition algorithm running in $O(|V| + |E|)$ time. They used the latter to develop branching algorithms for the NP-hard problems of vertex and edge deletion distance to a funnel, obtaining a fix-parameter quadratic solution. Analogous to the minimum feedback set problem [54], the vertex (edge) deletion distance to a funnel problem asks to find the minimum number of vertices (edges) that need to be removed from a graph so that the resulting graph is a funnel.

We propose three (new) linear time recognition algorithms of funnels (Section 4.1), each based on a different characterization, improving the running time of the branching algorithm to parameterized linear time (see Appendix B). We generalize funnels to k -funnels by allowing private edges to be shared by at most k source-to-sink paths (Definitions 14 and 15). We show how to recognize them in linear time (Lemma 16) and find the minimum k for which a DAG is a k -funnel (Corollary 17 and Lemma 18). We then further generalize k -funnels to the class of DAGs \mathcal{ST}_k (Definition 20 and Lemma 21), which (unlike k -funnels for $k > 1$, see Figure 2) can be characterized (and efficiently recognized, see Lemma 22) by a partition into a graph of the class \mathcal{S}_k (generalization of out-forest, see Definition 19) and a graph of the class \mathcal{T}_k (generalization of in-forest, see Definition 19).

We obtain our parameterized results in Section 5 by noting that, analogous to the fact that in KMP we only need the longest prefix match, in the DAG algorithm we can bound the size of the prefix-incomparable sets by the number of paths from a source or the number of paths to a sink, $\mu_s(v)$ and $\mu_t(v)$, respectively (Lemma 23).

► **Theorem 1.** *Let $G = (V, E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k + \sigma m)$, where $k = \min(\max_{v \in V} \mu_s(v), \max_{v \in V} \mu_t(v))$.*

In particular, this implies linear time algorithms for out-forests and in-forests, and for every DAG in \mathcal{S}_k or \mathcal{T}_k for constant k . Finally, we solve the problem on DAGs in \mathcal{ST}_k (thus also in k -funnels), by using the vertex partition characterization of \mathcal{ST}_k (Lemma 22), solving the matches in each part separately with Theorem 1, and resolve the matches crossing from one part to the other with a precomputed data structure (Lemma 26).

► **Theorem 2.** *Let $G = (V, E)$ be a DAG, $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k^2 + m^2)$, where $k = \max_{v \in V}(\min(\mu_s(v), \mu_t(v)))$.*

2 Preliminaries

We work with a (directed) graph $G = (V, E)$, a function $\ell : V \rightarrow \Sigma$ labeling the vertices of G with characters from a finite alphabet Σ of size σ , and a sequence $S[1..m] \in \Sigma^m$.

Graphs. A graph $H = (V_H, E_H)$ is said to be a *subgraph* of G if $V_H \subseteq V$ and $E_H \subseteq E$. If $V' \subseteq V$, then $G[V']$ is the subgraph *induced by* V' , defined as $G[V'] = (V', \{(u, v) \in E : u, v \in V'\})$. We denote $G^r = (V, E^r)$ to be the *reverse* of G ($E^r = \{(v, u) \mid (u, v) \in E\}$). For a vertex $v \in V$ we denote by N_v^- (N_v^+) the set of *in(out)-neighbors* of v , and by $d_v^- = |N_v^-|$ ($d_v^+ = |N_v^+|$) its *in(out)degree*. A *source (sink)* is a vertex with zero in(out)degree. The *edge contraction* of $(u, v) \in E$ is the graph operation that removes (u, v) and merges u and v . A *path* P is a sequence $v_1, \dots, v_{|P|}$ of different vertices of V such that $(v_i, v_{i+1}) \in E$ for every $i \in \{1, \dots, |P| - 1\}$. We say that P is *proper* if $|P| \geq 2$, a *cycle* if $(v_{|P|}, v_1) \in E$, and *source-to-sink* if v_1 is a source and $v_{|P|}$ is a sink. We say that $u \in V$ *reaches* $v \in V$ if there is a path from u to v . If G does not have cycles it is called *directed acyclic graph* (DAG). A *topological ordering* of a DAG is a total order $v_1, \dots, v_{|V|}$ of V such that for every $(v_i, v_j) \in E$, $i < j$. It is known [53, 84] how to compute a topological ordering in $O(|V| + |E|)$ time, and we assume one $(v_1, \dots, v_{|V|})$ is already computed if G is a DAG³. An *out(in)-forest* is a DAG such that every vertex has in(out)degree at most one, if it has a unique source (sink) it is called an *out(in)-tree*. The *label* of a path $P = v_1, \dots, v_{|P|}$ is the sequence of the labels of its vertices, $\ell(P) = \ell(v_1) \dots \ell(v_{|P|})$.

Strings. We say that S has a *match* in (G, ℓ) if there is a path whose label is equal to S , every such path is an *occurrence* of S in (G, ℓ) . We denote $S[i..j]$ (also $S[i]$ if $i = j$, and the empty string if $j < i$) to be the substring of S between position i and j (both inclusive), we say that it is *proper* if $i > 1$ or $j < m$, a *prefix* if $i = 1$ and a *suffix* if $j = m$. We denote S^r to be the reverse of S ($S^r[i] = S[m - i + 1]$ for $i \in \{1, \dots, m\}$). A substring of S is called a *border* if it is a proper prefix and a proper suffix at the same time. The *failure function* of S , $f_S : \{1, \dots, m\} \rightarrow \{0, \dots, m\}$ (just f if S is clear from the context), is such that $f_S(i)$ is the length of the longest border of $S[1..i]$. We also use f_S to denote the in-tree $(\{0, \dots, m\}, \{(i, f_S(i)) \mid i \in \{1, \dots, m\}\})$, also known as the *failure tree* [46] of S . By definition, the lengths of all borders of $S[1..i]$ in decreasing order are $f_S(i), f_S^2(i), \dots, 0$. The matching automaton of S , $A_S : \{0, \dots, m\} \times \Sigma \rightarrow \{0, \dots, m\}$, is such that $A_S(i, a)$ is the length of the longest border of $S[1..i] \cdot a$. It is known how to compute f_S in time $O(m)$ [60] and A_S in time $O(\sigma m)$ [2, 29, 81], and we assume they are already computed⁴.

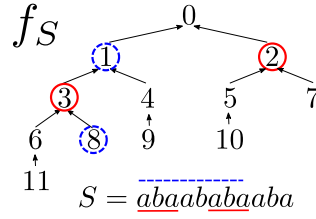
3 The DAG algorithm on prefix-incomparable matches

A key idea in our linear time parameterized algorithm is that of prefix-incomparable sets of the string S . We will show that one prefix-incomparable set per vertex suffices to capture all the matching information. See Figure 1 for an example of these concepts.

► **Definition 3** (Prefix-incomparable). *Let $S \in \Sigma^m$ be a string. We say that $i, j \in \{0, \dots, m\}, i < j$ are prefix-incomparable (for S) if $S[1..i]$ is not a border of $S[1..j]$. We say that $B \subseteq \{0, \dots, m\}$ is prefix-incomparable (for S) if for every $i < j \in B$, i and j are prefix-incomparable (for S).*

³ Our algorithms run in $\Omega(|V| + |E|)$ time.

⁴ Our algorithms run in $\Omega(\sigma m)$ time.



■ **Figure 1** A string $S = abaababaaba$ and its failure tree f_S , with $w = |\{i \in \{0, \dots, 11\} \mid \exists j, f_S(j) = i\}| = |\{7, 8, 9, 10, 11\}| = 5$. On the string it is shown in segmented (blue) and solid (red) lines that prefix $S[1..3] = aba$ is a border of prefix $S[1..8] = abaababa$, which can also be seen in the tree since 3 is an ancestor (parent in this case) of 8. In the tree two sets are shown $B_1 = \{1, 8\}$ in segmented (blue) circles, which is prefix-comparable, and $B_2 = \{2, 3\}$ in solid (red) circles, which is prefix-incomparable. If $B_1 \cup B_2 \cup \{0\}$ is B_v for some $v \in V$, then $PI_v = \{2, 8\}$.

In our algorithm we will compute for each vertex v a prefix-incomparable set representing all the prefixes of S that match with a path ending in v . More precisely, if B_v is the set of all the prefixes of S that match with a path ending in v , then the algorithm will compute $PI_v \subseteq B_v$ such that PI_v is prefix-incomparable and for every $i \in B_v$ there is a $j \in PI_v$ such that i is ancestor of j . Note that such a set always exists and it is unique, it corresponds to the leaves of $f_S[B_v]$. To obtain a linear time parameterized algorithm we show how to compute PI_v from the sets PI_u , $u \in N_v^-$, in time parameterized by the size of these sets.

► **Definition 4** (B_v, PI_v). Let $G = (V, E)$ be a DAG, $\ell : V \rightarrow \Sigma$ a labeling function, and $S \in \Sigma^m$ a string. For every $v \in V$ we define the sets:

- $B_v = \{i \in \{0, \dots, m\} \mid \exists P \text{ path of } G \text{ ending in } v \text{ and } \ell(P) = S[1..i]\}^5$
- $PI_v \subseteq B_v$ as the unique prefix-incomparable set such that for every $i \in B_v$ there is a $j \in PI_v$ such that $i = j$ or $S[1..i]$ is a border of $S[1..j]$

► **Lemma 5.** Let $G = (V, E)$ be a DAG, $v \in V$, $S \in \Sigma^m$ a string, f_S its failure tree and A_S its matching automaton. We can compute PI_v from PI_u for every $u \in N_v^-$ in time $O(w^2 \cdot d_v^-)$ or in time $O\left(\left(k_v := \sum_{u \in N_v^-} |PI_u|\right)^2\right)$, after $O(m)$ preprocessing time, where $w = |\{i \in \{0, \dots, m\} \mid \exists j, f_S(j) = i\}|$.

Proof. We precompute constant time lowest common ancestor (LCA)⁶ queries [3] of f_S in $O(m)$ time [41, 79, 20, 17, 18, 7, 39]. Note that with this structure we can check whether $i < j$ are prefix-incomparable in constant time ($LCA(i, j) < i$).

If v is a source we have that either $B_v = PI_v = \{0\}$ if $\ell(v) \neq S[1]$ or $PI_v = \{1\}$ if $\ell(v) = S[1]$, otherwise we proceed as follows. To obtain the $O(k_v^2)$ time, we first append all the elements of every PI_u for $u \in N_v^-$ into a list \mathcal{L} (of size k_v), then we replace every $i \in \mathcal{L}$ by $A_S(i, \ell(v))$, and finally we check (at most) every pair $i < j$ of elements of \mathcal{L} and test (in constant time) if they are prefix-incomparable, if they are not we remove i from the list. After these $O(|\mathcal{L}|^2) = O(k_v^2)$ tests $\mathcal{L} = PI_v$.

To obtain the $O(w^2 \cdot d_v^-)$ time, we process the in-neighbors of v one by one and maintain a prefix-incomparable set representing the prefix matches incoming from the already processed in-neighbors. That is, we maintain a prefix-incomparable set PI' , and when we process the next in-neighbor $u \in N_v^-$ we append all elements of PI' and $\{A_S(i, \ell(v)) \mid i \in PI_u\}$ into a

⁵ Here we consider that the empty path always exists and its label is the empty string, thus $0 \in B_v$.

⁶ $LCA(i, j)$ returns the lowest node in the tree f_S that is ancestor of both i and j .

list \mathcal{L}' of size $O(w)$ (by Lemma 27), then we use the same quadratic procedure applied on \mathcal{L} in time $O(w^2)$ to obtain the new PI' . After processing all in-neighbors in time $O(d_v^- \cdot w^2)$, we have $PI' = PI_v$. Next, we show the correctness of both procedures.

Let PI'_v be the result after applying one of the procedures explained before (that is the final state of PI' or \mathcal{L}), by construction PI'_v is prefix-incomparable. Now, consider $i \in B_v$ and a path P ending in v with $\ell(P) = S[1..i]$. If P is of length zero (only one vertex), then $i = 1$ and $\ell(P) = \ell(v) = S[1]$. Consider any $u \in N_v^-$, and any $j \in PI_u$ (there is at least one since $0 \in B_u$), this value is mapped to $A_S(j, \ell(v)) = A_S(j, S[1]) = j'$. By definition of the matching automaton, j' is the longest border of $S[1..j] \cdot S[1]$, thus $S[1]$ is a border of $S[1..j']$ or $j' = 1$, in both procedures j' can only be removed from PI'_v if a longer prefix contains $S[1..j']$ as a border and also $S[1]$. If P is a proper path and $i > 1$, consider the second to last vertex u of P . Note that $i - 1 \in B_u$, and thus there is $j \in PI_u$, such that $S[1..i - 1]$ is a border of $S[1..j]$, this value is mapped to $j' = A_S(j, \ell(v)) = A_S(j, S[i])$, which is the length of the longest border of $S[1..j] \cdot S[i]$, but since $S[1..i]$ is also a border of $S[1..j] \cdot S[i]$, then $S[1..i]$ is also a border of $S[1..j']$ or $j' = i$. Again, in both procedures j' can only be removed from PI'_v if a longer prefix contains $S[1..j']$ as a border and also $S[1..i]$. Finally, we note that $PI'_v \subseteq B_v$ since every $i \in PI'_v$ corresponds to a match of $S[1..i]$ by construction. \blacktriangleleft

We improve the dependency on w and k_v by replacing the quadratic comparison by sorting plus a linear time algorithm on the balanced parenthesis representation [51, 71] of f_S .

► **Lemma 6.** *We can obtain Lemma 5 in time $O(\text{sort}(w, m) \cdot d_v^-)$ or in time $O(\text{sort}(k_v, m))$, where $\text{sort}(n, p)$ is the time spent by an algorithm sorting n integers in the range $\{0, \dots, p\}$.*

Proof. We compute the balanced parenthesis (BP) [51, 71] representation of the topology of f_S , that is, we traverse f_S from the root in preorder, appending an open parenthesis when we first arrive at a vertex, and a closing one when we leave its subtree. As a result we obtain a balanced parenthesis sequence of length $2(m + 1)$, where every vertex $i \in f_S$ is mapped to its open parenthesis position $\text{open}[i]$ and to its close parenthesis position $\text{close}[i]$, which can be computed and stored at preprocessing time. Note that in this representation, i is ancestor of j (and thus prefix-comparable) if and only if $\text{open}[i] \leq \text{open}[j] \leq \text{close}[i]$. As such, if we have a list of $O(k_v)$ (or $O(w)$) (\mathcal{L} and \mathcal{L}' from Lemma 5) values, we can compute the corresponding prefix-incomparable sets as follows.

First, we sort the list by increasing open value, this can be done in $O(\text{sort}(k_v, m))$ (or $O(\text{sort}(w, m))$), since this sorting is equivalent to sort by increasing $\text{open}/2 \in \{0, \dots, m\}$ value. Then, we process the list in the sorted order, if two consecutive values i and j in the order are prefix-comparable (that is, if $\text{open}[j] \leq \text{close}[i]$) then we remove i and continue to the next value j . At the end of this $O(k_v)$ (or $O(w)$) time processing we obtain the desired prefix-incomparable set. \blacktriangleleft

If we use techniques for integer sorting [87, 90, 59] we can get $O(k_v \log \log m)$ (or $O(w \log \log m)$) time for sorting, however introducing m into the running time. We can solve this issue by using more advanced techniques [48, 9, 47] obtaining an $O(k_v \log \log k_v)$ (or $O(w \log \log w)$) time for sorting. However, we show that by using the suffix-tree [89, 68, 85] of S^r we can obtain a linear dependency on w and k_v .

► **Theorem 7.** *We can obtain the result of Lemma 5 in time $O(w \cdot d_v^-)$ or in time $O(k_v)$.*

Proof. We reuse the procedure of Lemma 6 but this time on the BP representation of the topology of the suffix-tree T_r of S^r , which has $O(m)$ vertices and can be built in $O(m)$ time [89, 68, 85]. Note that every suffix represented in T_r corresponds to a prefix of S (spelled

in the reverse direction). Moreover, $i \leq j$ are prefix-comparable if and only if the vertex representing i in T_r ($T_r[i]$) is an ancestor of $T_r[j]$, the same property as in f_S . Furthermore, if B is prefix-incomparable and $A(j, a) = j + 1$ for every $j \in B$, then the positions of the vertices in $A(B, a)$ in T_r follow the same order as the ones in B , since the suffix-tree is lexicographically sorted.

Now, we show how to obtain the prefix-incomparable set representing $A(PI_u, \ell(v))$ in $|PI_u|$ time assuming that PI_u is sorted by increasing (*open*) position in T_r .

We first separate PI_u into the elements $i \in M$ with $S[i + 1] = \ell(v)$ and $i \in E$ with $S[i + 1] \neq \ell(v)$ (in the same relative order as in PI_u , which is supposed to be in increasing order). Since M is prefix-incomparable the positions of the vertices in $A(M, \ell(v))$ in T_r follow the same order as the ones in M . We then obtain the list E_u by applying $T_r[A(i, \ell(v)) - 1]$ for every $i \in E$ (if $A(i, \ell(v)) = 0$ we do not process i), and then for any pair of consecutive elements x before y in E_u such that $y \leq x$ we remove y from E_u , and repeat this until no further such inversion remains, thus obtaining an increasing list in E_u representing vertices in T_r . Next, since E_u is sorted we can obtain the list PI_E of prefix-incomparable elements representing E_u , and finally apply $A(PI_E, \ell(v))$ (which also follows an increasing order in T_r), merge it with $A(M, \ell(v))$, and compute the prefix-comparable elements of this merge.

The correctness of the previous procedure follows by the fact that if there is an inversion $y < x$ in E_u , then the prefix $A(j, \ell(v)) - 1$ represented by y in T_r is a border of the prefix $A(i, \ell(v)) - 1$ represented by x (and thus is safe to remove y). For this, first note that i appears before j in E , then $S[i..1] <_{lex} S[j..1]$, and since i is prefix-incomparable with j there is a $k \geq 1$ such that $S[i..i + k - 1] = S[j..j + k - 1]$ and $S[i + k] <_{lex} S[j + k]$. Then, since y appears before x in E_u , then $S[A(j, \ell(v)) - 1..1] <_{lex} S[A(i, \ell(v)) - 1..1]$, but since $A(i, \ell(v)) - 1$ is a border of i and $A(j, \ell(v)) - 1$ is a border of j , $S[A(j, \ell(v)) - 1..1]$ must be a prefix of $S[A(i, \ell(v)) - 1..1]$, and thus $S[1..A(j, \ell(v)) - 1]$ is a border of $S[1..A(i, \ell(v)) - 1]$.

The corollary is obtained by maintaining the PI_v sets sorted by position in T_r , and noting that the previous procedure runs in linear $O(|P_u|)$ time. ◀

In Appendix A we show how to use Theorem 7 to derive a parameterized algorithm using parameter $w = |\{i \in \{0, \dots, m\} \mid \exists j, f_S(j) = i\}|$, improving on the classical DAG algorithm when S is a periodic string. Next, we will present our results on recognizing funnels and their generalization (Section 4), and how to use these classes of graphs and Theorem 7 to obtain parameterized algorithms using parameters related to the topology of the DAG (Section 5).

4 Funnels and beyond

Recall that funnels are DAGs whose source-to-sink paths have at least one *private* edge⁷, that is, an edge used by only one source-to-sink path. More formally,

► **Definition 8** (Private edge). *Let $G = (V, E)$ be a DAG and \mathcal{P} the set of source-to-sink paths of G . We say that $e \in E$ is private if $\mu(e) := |\{P \in \mathcal{P} \mid e \in P\}| = 1$. If $\mu(e) > 1$, we say that e is shared.*

► **Definition 9** (Funnel). *Let $G = (V, E)$ be a DAG and \mathcal{P} the set of source-to-sink paths of G . We say that G is a funnel if for every $P \in \mathcal{P}$ there exists $e \in P$ such that e is private.*

Millani et al. [70] showed two other characterizations of funnels.

⁷ For the sake of simplicity, we assume that there are no isolated vertices, thus any source-to-sink path has at least one edge.

► **Theorem 10** ([70]). *Let $G = (V, E)$ be a DAG. The following are equivalent:*

1. G is a funnel
2. There exists a partition $V = V_1 \dot{\cup} V_2$ such that $G[V_1]$ is an out-forest, $G[V_2]$ is an in-forest and there are no edges from V_2 to V_1
3. There is no path P such that its first vertex has more than one in-neighbor (a merging vertex) and its last vertex more than one out-neighbor (a forking vertex). Such a path is called forbidden

They also gave an $O(|V| + |E|)$ time algorithm to recognize whether a DAG G is a funnel, and an $O(|V|(|V| + |E|))$ time algorithm to find a minimal forbidden path in a general graph, that is, a forbidden path that is not contained in another forbidden path.

4.1 Three (new) linear time recognition algorithms

We first show how to find a minimal forbidden path in time $O(|V| + |E|)$ in general graphs, improving on the quadratic algorithm of Millani et al. [70].

► **Lemma 11.** *Let $G = (V, E)$ be a graph. In $O(|V| + |E|)$ time, we can decide if G contains a forbidden path, and if one exists we report a minimal forbidden path.*

Proof. In the bioinformatics community minimal forbidden paths are a subset of *unitigs* and it is well known how to compute them in $O(|V| + |E|)$ time (see e.g. [55, 50, 58, 69]), here we include a simple algorithm for completeness. We first compute the indegree and outdegree of each vertex and check whether there exists a forbidden path of length zero or one, all in $O(|V| + |E|)$ time, in the process we also mark all vertices except the ones with unit indegree and outdegree. If no path is found we iterate over the vertices one last time. If the current vertex is not marked we extend it back and forth to the closest marked vertices and mark the vertices in these extensions, finally we check whether the first vertex is merging and the last forking. This last step takes $O(|V|)$ time in total. ◀

Lemma 11 provides our first linear time recognition algorithm and, as opposed to the algorithm of Millani et al. [70], it also reports a minimal forbidden path given a general graph. Moreover, in Appendix B, we show that Lemma 11 provides a linear time parameterized algorithm for the NP-hard (and inapproximable) problem of deletion distance of a general graph to a funnel [63, 70]. Millani et al. [70] solved this problem in (parameterized) quadratic time and in (parameterized) linear time only if the input graph is a DAG.

Next, we show another linear time recognition algorithm, which additionally finds the partition $V = V_1 \dot{\cup} V_2$ from Theorem 10. Finding such a partition will be essential for our solution to SMLG. From now we will assume that the input graph is a DAG since this condition can be checked in linear time [53, 84].

► **Lemma 12.** *Let $G = (V, E)$ be a DAG. We can decide in $O(|V| + |E|)$ time whether G is a funnel. Additionally, if G is a funnel, the algorithm reports a partition $V = V_1 \dot{\cup} V_2$ such that $G[V_1]$ is an out-forest, $G[V_2]$ is an in-forest and there are no edges from V_2 to V_1 .*

Proof. We start a special BFS traversal from all the source vertices of G . The traversal only adds vertices to the BFS queue if they have not been previously visited (as a typical BFS traversal) and if its indegree is at most one. After the search we define the partition V_1 as the set of vertices visited during the traversal and $V_2 = V \setminus V_1$. Finally, we report the previous partition if there are no edges from V_2 to V_1 , and if every vertex of V_2 has outdegree at most one. All these steps run in time $O(|V| + |E|)$.

Note that if the algorithm reports a partition, then this satisfies the required conditions to be a funnel ($G[V_1]$ is an out-forest since every vertex visited in the traversal has indegree at most one). Moreover, if G is a funnel, we prove that V_2 is an in-forest and that there are no edges from V_2 to V_1 . For the first, suppose by contradiction that there is a vertex $v \in V_2$ with $d_v^+ > 1$, since every vertex is reached by some source in a DAG then there is a $u \in V_2$ with $d_u^- > 1$ (a vertex that was not added to the BFS queue) that reaches v , implying the existence of a forbidden path in G , a contradiction. Finally, there cannot be edges from V_2 to V_1 since the indegree (in G) of vertices of V_1 is at most one and its unique (if any) in-neighbor is also in V_1 by construction. ◀

Next, we present another characterization of funnels based on the structure of private/shared edges of the graph, which can be easily obtained by manipulating the original Definition 9.

► **Definition 13** (Funnel). *Let $G = (V, E)$ be a DAG. We say that G is a funnel if there is no source-to-sink path using only shared edges.*

As such, another approach to decide whether a DAG G is a funnel is to compute $\mu(e)$ for every $e \in E$ and then perform a traversal that only uses shared edges. Computing the number of source-to-sink paths containing e , that is $\mu(e)$, can be done by multiplying the number of source-to- e paths, $\mu_s(e)$, by the number of e -to-sink paths, $\mu_t(e)$, each of which can be computed in $O(|V| + |E|)$ time for all edges. The solution consists of a dynamic program on a topological order (and reverse topological order) of G with the following recurrences.

$$\begin{aligned} \mu_s(u) &= \mathbb{1}_{d_u^- = 0} + \sum_{u' \in N_u^-} \mu_s(u') \\ \mu_t(v) &= \mathbb{1}_{d_v^+ = 0} + \sum_{v' \in N_v^+} \mu_t(v') \\ \mu((u, v)) &= \mu_s(u) \cdot \mu_t(v) \end{aligned} \tag{1}$$

Where $\mathbb{1}_A$ is the characteristic function evaluating to 1 if A is true and to 0 otherwise. It is simple to observe that the previous dynamic programs can be computed in time $O(|V| + |E|)$ each and that for every $e = (u, v) \in E$, $\mu_s(e) = \mu_s(u)$ and $\mu_t(e) = \mu_t(v)$. By simplicity, in the following we will use $\mu_s(e)$ and $\mu_s(u)$ (also $\mu_t(e)$ and $\mu_t(v)$) interchangeably. The previous algorithm assumes constant time arithmetic operations on numbers up to $\max_{e \in E} \mu(e)$, which can be $O(2^{|V|})$. To avoid this issue, we note that it is not necessary to compute $\mu(e)$, but only to verify that $\mu(e) > 1$. As such, we can recognize shared edges as soon as we identify that $\mu(e) > 1$, that is whenever $\mu_s(e)$ or $\mu_t(e)$ is greater than one in their respective computation. A formal description of this algorithm can be found in Lemma 16.

4.2 Generalizations of funnels

To generalize funnels we will allow source-to-sink paths to use only shared edges, but require to have at least one edge shared by at most k different source-to-sink paths.

► **Definition 14** (k -private edge). *Let $G = (V, E)$ be a DAG. We say that $e \in E$ is k -private if $\mu(e) \leq k$. If $\mu(e) > k$ we say that e is k -shared.*

► **Definition 15** (k -funnel). *Let $G = (V, E)$ be a DAG. We say that G is a k -funnel if there is no source-to-sink path using only k -shared edges.*

7:10 Parameterized Algorithms for String Matching to DAGs

The next algorithm is a generalization of the last algorithm in Section 4.1 to decide if a DAG is a k -funnel. It assumes constant time arithmetic operations on numbers up to k .

► **Lemma 16.** *We can decide if a DAG $G = (V, E)$ is a k -funnel in $O(|V| + |E|)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(k)$.*

Proof. We process the vertices in a topological ordering and use Equation (1) to compute $\mu_s(e)$ in one pass, $\mu_t(e)$ in another pass and $\mu(e)$ in a final pass. To avoid arithmetic operations with numbers greater than k , we mark the edges having μ_s and μ_t greater than k as k -shared during the computations of μ_s, μ_t . Note that if $\mu_s(e) > k$ or $\mu_t(e) > k$ then $\mu(e) > k$. As such, before computing $\mu_s(e)$ ($\mu_t(e)$) we check if some of the edges from (to) the in(out)-neighbors is marked as k -shared. If that is the case we do not compute $\mu_s(e)$ ($\mu_t(e)$) and instead mark e as k -shared, otherwise we compute the respective sum of Equation (1), and if at some point the cumulative sum exceeds k we stop the computation and mark e as k -shared. Finally, we find all k -shared edges as the marked plus the unmarked with $\mu(e) = \mu_s(e) \cdot \mu_t(e) > k$, perform a traversal only using k -shared edges, and report that G is not a k -funnel if there is a source-to-sink path using only k -shared edges in time $O(|V| + |E|)$. ◀

We can use the previous result and exponential search [19, 15] to find the minimum k such that a DAG is a k -funnel.

► **Corollary 17.** *Let $G = (V, E)$ be a DAG. We can find the minimum k such that G is a k -funnel in $O((|V| + |E|) \log k)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(k)$.*

Assuming constant time arithmetic operations on numbers up to $\max_{e \in E} \mu(e)$ the problem is solvable in linear time by noting that the answer is equal to the weight of a widest path.

► **Lemma 18.** *Let $G = (V, E)$ be a DAG. We can find the minimum k such that G is a k -funnel in $O(|V| + |E|)$ time, assuming constant time arithmetic operations on numbers up to $\Theta(\max_{e \in E} \mu(e))$.*

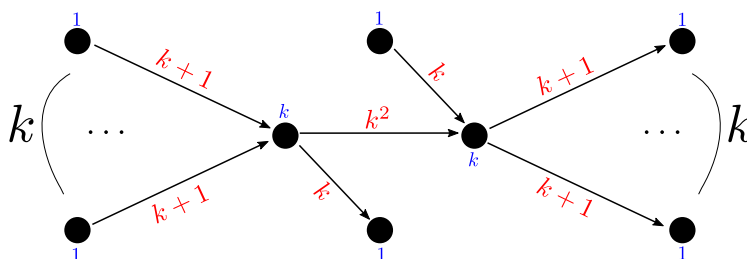
Proof. We compute $\mu(e)$ for every $e \in E$ by using the dynamic programming algorithm specified by Equation (1) on a topological ordering of G . Since constant time arithmetic operations are assumed for numbers up to $\max_{e \in E} \mu(e)$, the previous computation takes linear time. Then, we compute the weight of a source-to-sink path P maximizing $\min_{e \in P} \mu(e)$, and report this value. This problem is known as the widest path problem [76, 82, 65, 86, 80] and it can be solved in linear time on DAGs [88, 49] by a dynamic program on a topological order of the graph. By completeness, we show a dynamic programming recurrence to compute $W[e]$, the weight of a source-to- e path P maximizing $\min_{e' \in P} \mu(e')$.

$$W[e = (u, v)] = \mu(e) \cdot \mathbb{1}_{d_u^- = 0} + \sum_{u' \in N_u^-} \min(W[(u', u)], \mu((u', u)))$$

Finally, note that if we denote w to the weight of a widest path, then there is a source-to-sink path using only $w - 1$ -shared edges, G is not $w - 1$ -funnel. Moreover, there cannot be a source-to-sink path using only w -shared edges, since such a path would contradict w being the weight of a widest path. As such, w is the minimum k such that G is k -funnel. ◀

We now define three classes of DAGs closely related to k -funnels.

► **Definition 19.** *We say that a DAG $G = (V, E)$ belongs to the class \mathcal{S}_k (\mathcal{T}_k) if for every $v \in V$, $\mu_s(v)$ ($\mu_t(v)$) $\leq k$.*



■ **Figure 2** A DAG in \mathcal{ST}_k that is not a k -funnel, for every $k > 1$. The central edge is a forbidden path whose first vertex has indegree k and outdegree 2, and last vertex has indegree 2 and outdegree k , the rest of the edges have either a source tail or a sink head. The blue label next to each vertex v corresponds to $\min(\mu_s(v), \mu_t(v))$, since the maximum of these labels is k the graph belongs to \mathcal{ST}_k . The red label next to each edge e corresponds to $\mu(e)$, since there is a source-to-sink path with no k -private edge the graph is not a k -funnel.

► **Definition 20.** We say that a DAG $G = (V, E)$ belongs to the class \mathcal{ST}_k if for every $v \in V$, $\mu_s(v) \leq k$ or $\mu_t(v) \leq k$.

► **Lemma 21.** $\mathcal{S}_k, \mathcal{T}_k \subseteq k\text{-funnels} \subseteq \mathcal{ST}_k$.

Proof. We first prove that $\mathcal{S}_k, \mathcal{T}_k \subseteq k\text{-funnels}$. Consider $G \in \mathcal{S}_k$ ($G \in \mathcal{T}_k$), and take any source-to-sink path P of G . Let (u, v) be the last (first) edge of P , then by Equation (1) $\mu((u, v)) = \mu_s(u) \cdot \mu_t(v)$, but since $\mu_t(v) = 1$ (v is a sink) and $\mu_s(u) \leq k$ ($G \in \mathcal{S}_k$) (analogously, $\mu_s(u) = 1$ and $\mu_t(v) \leq k$), then $\mu((u, v)) \leq k$, and thus (u, v) is a k -private edge. To prove that $k\text{-funnels} \subseteq \mathcal{ST}_k$, suppose that G is a k -funnel, and by contradiction that there exists $v \in V$ with $\mu_s(v), \mu_t(v) > k$. Consider any source-to-sink path P using v . Now, let (u, w) be any edge in P before (after) v , then $\mu_t(w) \geq \mu_t(v) > k$ ($\mu_s(u) \geq \mu_s(v) > k$), and thus $\mu((u, w)) = \mu_s(u) \cdot \mu_t(w) > k$. As such, P does not have a k -private edge, a contradiction. ◀

For $k = 1$, \mathcal{S}_1 describes out-forests and \mathcal{T}_1 in-forests, thus being more restrictive than funnels. Moreover, we note that the in(out)-star of $k + 2$ vertices, that is $k + 1$ vertices pointing to a sink (pointed from a source), $\notin \mathcal{S}_k$ (\mathcal{T}_k), but this graph is a funnel. On the other hand, from the vertex partition characterization of funnels (Theorem 10 [70]) we have that $\mathcal{ST}_1 = (1\text{-})\text{funnels}$. However, for $k > 1$, the containment $k\text{-funnels} \subseteq \mathcal{ST}_k$ is strict (Figure 2).

By noting that the minimum k such that a DAG is in $\mathcal{S}_k, \mathcal{T}_k$ and \mathcal{ST}_k is $\max_{v \in V} \mu_s(v)$, $\max_{v \in V} \mu_t(v)$ and $\max_{v \in V} \min(\mu_s(v), \mu_t(v))$, respectively, we obtain the same results as in Lemmas 16 and 18 and Corollary 17 (with analogous assumptions on the cost of arithmetic operations) for recognition of $\mathcal{S}_k, \mathcal{T}_k$ and \mathcal{ST}_k .

Next, we prove that although the vertex partition characterization of funnels does not generalize to k -funnels, it does for the class \mathcal{ST}_k and it can be found efficiently.

► **Lemma 22.** Let $G = (V, E) \in \mathcal{ST}_k$ and k given as inputs. We can find, in $O(|V| + |E|)$ time, a partition $V = V_1 \dot{\cup} V_2$ such that $G[V_1] \in \mathcal{S}_k$, $G[V_2] \in \mathcal{T}_k$ and there are no edges from V_2 to V_1 . Moreover, if such a partition of a DAG G exists, then $G \in \mathcal{ST}_k$.

Proof. We set $V_1 = \{v \in V \mid \mu_s(v) \leq k\}$ and $V_2 = V \setminus V_1$. Note that finding V_1 takes linear time, since we can apply the algorithm described in Lemma 16 to compute the μ_s values (or decide that they are more than k)⁸. By construction we know that every $v \in V_1$ has

⁸ Recall that this procedure assumes constant time arithmetic operations of numbers up to $\Theta(k)$.

$\mu_s(v) \leq k$, and since $G \in \mathcal{ST}_k$ also every $v \in V_2$ has $\mu_t(v) \leq k$, thus $G[V_1] \in \mathcal{S}_k$, $G[V_2] \in \mathcal{T}_k$. Suppose by contradiction that there exists $e = (u, v) \in E \cap (V_2 \times V_1)$. As such, $\mu_s(u) > k$, but since $\mu_s(u) \leq \mu_s(v)$, then $\mu_s(v) > k$, a contradiction. Finally, if such a partition exists then $\mu_s(v) \leq k$ for every $v \in V_1$ and $\mu_t(v) \leq k$ for every $v \in V_2$, and thus $G \in \mathcal{ST}_k$. \blacktriangleleft

5 Parameterized algorithms: The DAG

The main idea to get the parameterized algorithms in this section is to bound the size of the PI_v sets by a topological graph parameter and use Lemma 5 and Theorem 7 to obtain a parameterized solution. As in the KMP algorithm [60] only one prefix-incomparable value suffices (the longest prefix match until that point), we show that $\mu_s(v)$ prefix-incomparable values suffice to capture the prefix matches up to v .

► **Lemma 23.** *Let $G = (V, E)$ be a DAG, $v \in V$, \mathcal{P}_{sv} the set of source-to- v paths, $\ell : V \rightarrow \Sigma$ a labeling function, $S \in \Sigma^m$ a string, and PI_v as in Definition 4. Then, $|PI_v| \leq \mu_s(v)$.*

Proof. Since any path ending in v is the suffix of a source-to- v path we can write B_v as:

$$B_v = \bigcup_{P_{sv} \in \mathcal{P}_{sv}} B_{P_{sv}} := \{i \in \{0, \dots, m\} \mid \exists P \text{ suffix of } P_{sv}, \ell(P) = S[1..i]\}$$

However, for every pair of values $i < j \in B_{P_{sv}}$, $S[1..i]$ is a border of $S[1..j]$ (it is a suffix since they are both suffixes of $\ell(P_{sv})$). As such, at most one value of $B_{P_{sv}}$ appears in PI_v , and then $|PI_v| \leq |\mathcal{P}_{sv}| = \mu_s(v)$. \blacktriangleleft

This result directly implies a parameterized string matching algorithm to DAGs in \mathcal{S}_k .

► **Lemma 24.** *Let $G = (V, E) \in \mathcal{S}_k$, $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O(|V|k + |E| + \sigma m)$.*

Proof. We compute the matching automaton A_S in $O(\sigma m)$ time. Then, we process the vertices in topological order, and for each vertex v we compute PI_v , the unique prefix-incomparable set representing B_v (all prefix matches of S with paths ending in v). We proceed according to Lemma 5 and Theorem 7 in $O(m)$ preprocessing time plus $O(k_v)$ time per vertex. There is a match of S in (G, ℓ) if and only if any PI_v contains m . The claimed running time follows since $k_v = \sum_{u \in N_v^-} |PI_u| \leq \sum_{u \in N_v^-} \mu_s(u) \leq \mu_s(v) \leq k$, by Lemma 23, Equation (1) and since $G \in \mathcal{S}_k$. \blacktriangleleft

A simple but interesting property about string matching to graphs is that we obtain the same problem by reversing the input (both the graph and the string), that is, S has a match in (G, ℓ) if and only if S^r has a match in G^r, ℓ . This fact, plus noting that $G \in \mathcal{S}_k$ if and only if $G^r \in \mathcal{T}_k$ gives the following corollary of Lemma 24.

► **Corollary 25.** *Let $G = (V, E) \in \mathcal{T}_k$, $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O(|V|k + |E| + \sigma m)$.*

With these two results and the fact that we can compute the minimum k such that a DAG is in $\mathcal{S}_k, \mathcal{T}_k$ in time $O((|V| + |E|) \log k)$ (see Corollary 17) we obtain our first algorithm parameterized by the topology of the DAG.

► **Theorem 1.** *Let $G = (V, E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k + \sigma m)$, where $k = \min(\max_{v \in V} \mu_s(v), \max_{v \in V} \mu_t(v))$.*

Our final result is a parameterized algorithm for DAGs in \mathcal{ST}_k (in particular for k -funnels). We note that the algorithm of Corollary 25 computes PI_v for S^r for every vertex in G^r . Recall that PI_v represents all the prefix matches of S^r with paths ending in v in G^r . In other words, it represents all suffix matches of S with paths starting in v in G . For clarity, let us call this set SI_v . The main idea of the algorithm for \mathcal{ST}_k is to use Lemma 22 to find a partitioning $V = V_1 \dot{\cup} V_2$ into \mathcal{S}_k and \mathcal{T}_k , use Lemma 24 and Corollary 25 to search for matches within each part and also to compute PI_v for every $v \in V_1$ and SI_v for every $v \in V_2$, and finally, to find matches using the edges from V_1 to V_2 . The last ingredient of our algorithm consists of preprocessing the answers to the last type of matches.

► **Lemma 26.** *Let $G = (V, E)$ a DAG, $(u, v) \in E$, $\ell : V \rightarrow \Sigma$ a labeling function, $S \in \Sigma^m$ a string and PI_u and SI_v as in Definition 4. We can decide if there is a match of S in (G, ℓ) using (u, v) in $O(|PI_u| \cdot |SI_v|)$ time, after $O(m^2)$ preprocessing time.*

Proof. We precompute a boolean table PS of $m \times m$ entries, such that $PS[i, j]$ is **true** if there is a length i' of a (non-empty) border of $S[1..i]$ (or $i' = i$) and a length j' of a (non-empty) border of $S[m - j + 1..m]$ (or $j' = j$) such that $i' + j' = m$, and **false** otherwise. This table can be computed by dynamic programming in $O(m^2)$ time as follows.

$$PS[i, j] = \begin{cases} \mathbf{false} & \text{if } i + j < m \vee i = 0 \vee j = 0 \\ i + j = m \vee PS[i, f_{S^r}(j)] \vee PS[f_S(i), j] & \text{otherwise} \end{cases}$$

We then use this table to test every $PS[i, j]$ with $i \in PI_u, j \in SI_v$ and report a match if any of these table entries is **true**, in total $O(|PI_u| \cdot |SI_v|)$ time.

Since every match of S using (u, v) must match a prefix $S[1..i]$ with a path ending in u and a suffix $S[i + 1..m]$ with a path starting in v , the previous procedure finds it (if any). ◀

► **Theorem 2.** *Let $G = (V, E)$ be a DAG, $\ell : V \rightarrow \Sigma$ a labeling function and $S \in \Sigma^m$ a string. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)k^2 + m^2)$, where $k = \max_{v \in V}(\min(\mu_s(v), \mu_t(v)))$.*

Proof. We first compute the minimum k such that the input DAG is in \mathcal{ST}_k in time $O((|V| + |E|) \log k)$ (see Corollary 17). Then, we obtain the partition of G into $G[V_1] \in \mathcal{S}_k, G[V_2] \in \mathcal{T}_k$ and no edges from V_2 to V_1 . We then search matches within $G[V_1]$ and $G[V_2]$ in time $O(|V|k + |E| + \sigma m)$ (Lemma 24 and Corollary 25) and we also keep PI_u for every $u \in V_1$ and SI_v for every $v \in V_2$. Finally, we process the matches using the edges (u, v) with $u \in V_1, v \in V_2$ in total $O(|E|k^2 + m^2)$ time (Lemma 26) since $O(|PI_u| \cdot |SI_v|) = O(k^2)$. ◀

6 Conclusions

In this paper we introduced the first parameterized algorithms for matching a string to a labeled DAG, a problem known (under SETH) to be quadratic even for a very special type of DAGs. Our parameters depend on the structure of the input DAG.

We derived our results from a generalization of KMP to DAGs using prefix-incomparable matches, which allowed us to bound the running time to parameterized linear. Further improvements on the running time of our algorithms remain open: is it possible to get rid of the automaton? or to combine prefix-incomparable and suffix-incomparable matches in better than quadratic (either in the size of the sets or the string)? (e.g. with a different tradeoff between query and construction time of the data structure answering these queries) and is there a (conditional) lower bound to combine these incomparable sets? (see e.g. [21]). Another

interesting question with practical importance is whether our parameterized approach can be extended to string labeled graphs with (unparameterized) linear time in the total length of the strings or extended to counting and reporting algorithms in linear time in the number of occurrences.

We also presented novel algorithmic results on funnels as well as generalizations of them. These include linear time recognition algorithms for their different characterizations, which we showed useful for the string matching problem but hope that can also help in other graph problems. We also showed how to find the minimum k for which a DAG is a k -funnel or $\in \mathcal{ST}_k$ (assuming constant time arithmetic operations on numbers up to $O(k)$) using an exponential search, but it remains open whether there exists a linear time solution.

References

- 1 Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2016)*, pages 377–391. SIAM, 2016.
- 2 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
- 3 Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. On finding lowest common ancestors in trees. *SIAM Journal on Computing*, 5(1):115–132, 1976.
- 4 Tatsuya Akutsu. A linear time pattern matching algorithm between a string and a tree. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM 1993)*, pages 1–10. Springer, 1993.
- 5 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, pages 911–930. SIAM, 2020.
- 6 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021.
- 7 Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.
- 8 Amihoud Amir, Moshe Lewenstein, and Noa Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.
- 9 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.
- 10 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):1–40, 2017.
- 11 Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.
- 12 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2016)*, pages 457–466. IEEE, 2016.
- 13 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi: 10.1137/15M1053128.
- 14 Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
- 15 Ricardo Baeza-Yates and Alejandro Salinger. Fast intersection algorithms for sorted sequences. In *Algorithms and Applications*, pages 45–61. Springer, 2010.
- 16 Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2013)*, pages 175–188, 2013.

- 17 Michael A Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN 2000)*, pages 88–94. Springer, 2000.
- 18 Michael A Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
- 19 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(SLAC-PUB-1679), 1976.
- 20 Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.
- 21 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132, pages 1–15. Schloss Dagstuhl-Leibniz Center for Informatics, 2019.
- 22 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics (WABI 2012)*, pages 225–235. Springer, 2012.
- 23 Manuel Cáceres, Massimo Cairo, Brendan Mumeey, Romeo Rizzi, and Alexandru I Tomescu. A linear-time parameterized algorithm for computing the width of a DAG. In *Proceedings of the 47th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2021)*, pages 257–269. Springer, 2021.
- 24 Manuel Cáceres, Massimo Cairo, Brendan Mumeey, Romeo Rizzi, and Alexandru I Tomescu. Minimum path cover in parameterized linear time. *arXiv preprint arXiv:2211.09659*, 2022.
- 25 Manuel Cáceres, Massimo Cairo, Brendan Mumeey, Romeo Rizzi, and Alexandru I Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 359–376. SIAM, 2022.
- 26 Manuel Cáceres, Brendan Mumeey, Edin Husic, Romeo Rizzi, Massimo Cairo, Kristoffer Sahlin, and Alexandru I Ioan Tomescu. Safety in multi-assembly via paths appearing in all path covers of a DAG. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 19(6):3673–3684, 2021.
- 27 Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in bioinformatics*, 19(1):118–135, 2018.
- 28 Jeff Conklin. Hypertext: An introduction and survey. *computer*, 20(09):17–41, 1987.
- 29 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- 30 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *Proceedings of the 32nd Data Compression Conference (DCC 2022)*, pages 272–281. IEEE, 2022.
- 31 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 2585–2599. SIAM, 2021.
- 32 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 5. Springer, 2015.
- 33 Moshe Dubiner, Zvi Galil, and Edith Magen. Faster tree pattern matching. *Journal of the ACM*, 41(2):205–213, 1994.
- 34 Massimo Equi, Roberto Grossi, Veli Mäkinen, Alexandru Tomescu, et al. On the complexity of string matching for graphs. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

- 35 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 *Proceedings of the 47th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2021)*, pages 608–622. Springer, 2021.
- 36 Massimo Equi, Veli Mäkinen, Alexandru I Tomescu, and Roberto Grossi. On the complexity of string matching for graphs. *ACM Transactions on Algorithms*, 2023.
- 37 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pages 184–193. IEEE, 2005.
- 38 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE, 2000.
- 39 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, pages 36–48. Springer, 2006.
- 40 Fedor V Fomin, Daniel Lokshtanov, Saket Saurabh, Michał Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3):1–45, 2018.
- 41 Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.
- 42 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.
- 43 Archontia C Giannopoulou, George B Mertzios, and Rolf Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. *Theoretical Computer Science*, 689:67–95, 2017.
- 44 Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing Wheeler graphs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144, pages 51:1–51:16, 2019.
- 45 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406, 2000.
- 46 Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1994)*, pages 697–704, 1994.
- 47 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, pages 602–608, 2002.
- 48 Yijie Han and Xiaojun Shen. Conservative algorithms for parallel and sequential integer sorting. In *Proceedings of the 1st International Computing and Combinatorics Conference (COCOON 1995)*, pages 324–333. Springer, 1995.
- 49 Tzvika Hartman, Avinatan Hassidim, Haim Kaplan, Danny Raz, and Michal Segalov. How to split a flow? In *2012 Proceedings IEEE INFOCOM*, pages 828–836. IEEE, 2012.
- 50 Benjamin Grant Jackson. *Parallel methods for short read assembly*. PhD thesis, Iowa State University, 2009.
- 51 Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*, pages 549–554. IEEE Computer Society, 1989.
- 52 Chirag Jain, Haowen Zhang, Yu Gao, and Srinivas Aluru. On the complexity of sequence-to-graph alignment. *Journal of Computational Biology*, 27(4):640–654, 2020.
- 53 Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

- 54 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.
- 55 John D Kececioglu and Eugene W Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1):7–51, 1995.
- 56 Shahbaz Khan, Milla Kortelainen, Manuel Cáceres, Lucia Williams, and Alexandru I Tomescu. Improving RNA assembly via safety and completeness in flow decompositions. *Journal of Computational Biology*, 2022.
- 57 Shahbaz Khan, Milla Kortelainen, Manuel Cáceres, Lucia Williams, and Alexandru I Tomescu. Safety and completeness in flow decompositions for RNA assembly. In *Proceedings of the 26th International Conference on Research in Computational Molecular Biology (RECOMB 2022)*, pages 177–192. Springer, 2022.
- 58 Carl Kingsford, Michael C Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):1–11, 2010.
- 59 David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983.
- 60 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.
- 61 Tomohiro Koana, Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. *Journal of Experimental Algorithmics*, 26:1–30, 2021.
- 62 Jonas Lehmann. The computational complexity of worst case flows in unreliable flow networks. B.S. thesis, Institute for Theoretical Computer Science, University of Lübeck, 2017.
- 63 Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *Proceedings of the 20th International Colloquium on Automata, Languages, and Programming (ICALP 1993)*, pages 40–51. Springer, 1993.
- 64 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I Tomescu. Chaining for accurate alignment of erroneous long reads to acyclic variation graphs. *bioRxiv*, 2022.
- 65 Thomas Magnanti, R Ahuja, and J Orlin. Network flows: theory, algorithms, and applications. *PrenticeHall, Upper Saddle River, NJ*, 1993.
- 66 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Transactions on Algorithms*, 15(2):1–21, 2019.
- 67 Udi Manber and Sun Wu. Approximate string matching with arbitrary costs for text and hypertext. In *Advances In Structural And Syntactic Pattern Recognition*, pages 22–33. World Scientific, 1992.
- 68 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- 69 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *Proceedings of the 7th International Workshop on Algorithms in Bioinformatics (WABI 2007)*, pages 289–301. Springer, 2007.
- 70 Marcelo Garlet Millani, Hendrik Molter, Rolf Niedermeier, and Manuel Sorge. Efficient algorithms for measuring the funnel-likeness of DAGs. *Journal of Combinatorial Optimization*, 39(1):216–245, 2020.
- 71 J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- 72 Gonzalo Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.
- 73 Jakob Nielsen. *Hypertext and hypermedia*. Academic Press Professional, Inc., 1990.
- 74 Kunsoo Park and Dong Kyue Kim. String matching in hypertext. In *Proceeding of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995)*, pages 318–329. Springer, 1995.

- 75 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.
- 76 Maurice Pollack. The maximum capacity through a network. *Operations Research*, 8(5):733–736, 1960.
- 77 Mikko Rautiainen and Tobias Marschall. Aligning sequences to general graphs in $O(V + mE)$ time. *bioRxiv*, page 216127, 2017.
- 78 Nicola Rizzo, Alexandru I Tomescu, and Alberto Policriti. Solving string problems on graphs using the labeled direct product. *Algorithmica*, pages 1–26, 2022.
- 79 Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
- 80 Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social choice and Welfare*, 36(2):267–303, 2011.
- 81 Robert Sedgewick and Kevin Wayne. *Algorithms (4th edn)*. Addison-Wesley, 2011.
- 82 Nachum Shacham. Multicast routing of hierarchical data. In *[Conference Record] SUPER-COMM/ICC'92 Discovering a New World of Communications*, pages 1217–1221. IEEE, 1992.
- 83 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.
- 84 Robert E Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.
- 85 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- 86 Ehsan Ullah, Kyongbum Lee, and Soha Hassoun. An algorithm for identifying dominant-edge metabolic pathways. In *2009 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Digest of Technical Papers*, pages 144–150. IEEE, 2009.
- 87 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS 1975)*, pages 75–84. IEEE, 1975.
- 88 Benedicte Vatinlen, Fabrice Chauvet, Philippe Chrétienne, and Philippe Mahey. Simple bounds and greedy algorithms for decomposing a flow into a minimal set of paths. *European Journal of Operational Research*, 185(3):1390–1401, 2008.
- 89 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973.
- 90 Dan E Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

A A parameterized algorithm: The String

A simple property about prefix-incomparable sets is that their sizes are bounded by the number of prefixes that are not a border of other prefixes of the string, equivalently, the number of leaves in the failure function of the string.

► **Lemma 27.** *Let $S \in \Sigma^m$ be a string, f_S its failure function/tree, and $B \subseteq \{0, \dots, m\}$ prefix-incomparable for S . Then, $|B| \leq w$ such that w is the number of leaves of f_S , equivalently $w := |\{i \in \{0, \dots, m\} \mid \nexists j, f_S(j) = i\}|$.*

Proof. First note that $i < j$ are prefix-incomparable if and only if i is not ancestor of j in f_S . Suppose by contradiction that $|B| > w$, and consider the w leaf-to-root paths of f_S . Note that these w leaf-to-root paths cover all the vertices of f_S . By pigeonhole principle, there must be $i < j \in B$ in the same leaf-to-root path, that is i is ancestor of j , a contradiction. ◀

► **Theorem 28.** *Let $G = (V, E)$ be a DAG, Σ a finite alphabet ($\sigma = |\Sigma|$), $\ell : V \rightarrow \Sigma$ a labeling function, $S \in \Sigma^m$ a string and f_S its failure function. We can decide whether S has a match in (G, ℓ) in time $O((|V| + |E|)w + \sigma m)$, where $w = |\{i \in \{0, \dots, m\} \mid \exists j, f_S(j) = i\}|$.*

Proof. We compute the matching automaton A_S in $O(\sigma m)$ time. Then, we process the vertices in topological order, and for each vertex v we compute PI_v , the unique prefix-incomparable set representing B_v (all prefix matches of S with paths ending in v). We proceed according to Lemma 5 and Theorem 7 in $O(m)$ preprocessing time plus $O(w \cdot d_v)$ time per vertex, adding up to $O(w(|V| + |E|))$ time in total. There is a match of S in (G, ℓ) if and only if any PI_v contains m . ◀

We note that $w \leq m$, thus our algorithm is asymptotically as fast as the DAG algorithm, which runs in time $\Omega((|V| + |E|)m)$. However, we note that for w to be $o(m)$, a (very) long prefix of S must be a (highly) periodic string. To see this, consider the longest prefix $S[1..i]$ of S , such that there exists $j > i$ with $i = f_S(j)$. By definition, $S[1..i]$ is a border of $S[1..j]$, thus $S[k] = S[k + j - i]$ for $k \in \{1, \dots, i\}$, that is $S[1..j]$ is a periodic string with period $j - i$. Finally, note that if $w = o(m)$, then $m - i \in o(m)$, and thus the period $j - i \in o(m)$.

B A linear time parameterized algorithm for the distance problems

Millani et al. [70] gave an $O(|V|(|V| + |E|))$ time algorithm to find a minimal forbidden path in a general graph. They used this algorithm to design branching algorithms (see e.g. [32]) for the problems of finding maximum sized sets $V' \subseteq V, d_v := |V'|$ and $E' \subseteq E, d_e := |E'|$, such that $G[V']$ and (V, E') are funnels, known as vertex and edge distance to a funnel. It is known that (unless $P = NP$) there is an $\epsilon > 0$ such that there is no polynomial time $|V|^\epsilon$ approximation [63] for the vertex version nor $(1 + \epsilon)$ approximation [70] for the edge version. The authors [70] noted that if we consider a minimal forbidden path P of G of length $|P| > 1$, then the edges of P can be contracted until $|P| = 1$ without affecting the size of the solution. Moreover, they noted that if we consider such a P , two in-neighbors of the first vertex and two out-neighbors of the last⁹, then V' must contain at least one of those 6 vertices and E' one of those 5 edges, deriving $O(6^{d_v}|V|(|V| + |E|))$ and $O(5^{d_e}|V|(|V| + |E|))$ time branching algorithms for each problem¹⁰ [70, Corollary 1]. The authors also developed a more involved branching algorithm, only for the edge distance problem on DAG inputs, running in time $O(3^{d_e}(|V| + |E|))$ [70, Theorem 4].

By noting that minimal forbidden paths can be further contracted to length zero (one vertex) in the vertex distance problem, and that a minimal forbidden path can be found in time $O(|V| + |E|)$ (Lemma 11) we obtain the following result.

► **Theorem 29.** *Let $G = (V, E)$ be a graph. We can compute the vertex (edge) deletion distance to a funnel in time $O(5^d(|V| + |E|))$, where d is the deletion distance.*

Proof. We follow the branching approach as in [70, Corollary 1], but in the case of vertex distance we further contract the forbidden paths to length 0, the correctness of this step follows by noting that any solution containing two different vertices in a forbidden path is not minimum, since we still get a funnel by removing one of them (from the solution). As such, the number of recursive calls is ≤ 5 for both problems. Moreover, by Lemma 11, we can find a minimal forbidden path in time $O(|V| + |E|)$. ◀

⁹ This structure is known as a *butterfly*.

¹⁰ After removing forbidden paths all cycles are vertex-disjoint thus the rest of the problem can be solved by removing one vertex (edge) per cycle in one $O(|V| + |E|)$ time traversal.

Optimal Near-Linear Space Heaviest Induced Ancestors

Panagiotis Charalampopoulos ✉ 

Birkbeck, University of London, UK

Bartłomiej Dudek ✉ 

Institute of Computer Science, University of Wrocław, Poland

Paweł Gawrychowski ✉ 

Institute of Computer Science, University of Wrocław, Poland

Karol Pokorski ✉ 

Institute of Computer Science, University of Wrocław, Poland

Abstract

We revisit the Heaviest Induced Ancestors (HIA) problem that was introduced by Gagie, Gawrychowski, and Nekrich [CCCG 2013] and has a number of applications in string algorithms. Let T_1 and T_2 be two rooted trees whose nodes have weights that are increasing in all root-to-leaf paths, and labels on the leaves, such that no two leaves of a tree have the same label. A pair of nodes $(u, v) \in T_1 \times T_2$ is *induced* if and only if there is a label shared by leaf-descendants of u and v . In an HIA query, given nodes $x \in T_1$ and $y \in T_2$, the goal is to find an induced pair of nodes (u, v) of the maximum total weight such that u is an ancestor of x and v is an ancestor of y .

Let n be the upper bound on the sizes of the two trees. It is known that no data structure of size $\tilde{O}(n)$ can answer HIA queries in $o(\log n / \log \log n)$ time [Charalampopoulos, Gawrychowski, Pokorski; ICALP 2020].¹ This (unconditional) lower bound is a polyloglog n factor away from the query time of the fastest $\tilde{O}(n)$ -size data structure known to date for the HIA problem [Abedin, Hooshmand, Ganguly, Thankachan; Algorithmica 2022]. In this work, we resolve the query-time complexity of the HIA problem for the near-linear space regime by presenting a data structure that can be built in $\tilde{O}(n)$ time and answers HIA queries in $\mathcal{O}(\log n / \log \log n)$ time. As a direct corollary, we obtain an $\tilde{O}(n)$ -size data structure that maintains the LCS of a static string and a dynamic string, both of length at most n , in time optimal for this space regime.

The main ingredients of our approach are fractional cascading and the utilization of an $\mathcal{O}(\log n / \log \log n)$ -depth tree decomposition. The latter allows us to break through the $\Omega(\log n)$ barrier faced by previous works, due to the depth of the considered heavy-path decompositions.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases data structures, string algorithms, fractional cascading

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.8

1 Introduction

The solutions to algorithmic problems on texts frequently involve the construction of text indexes that can be built efficiently and offer a broad functionality, without significantly increasing space usage. A prime example of such an index is the suffix tree, which is ubiquitous in stringology. The work of Weiner [22] that introduced it, showed that it can be used to efficiently solve a number of fundamental open problems such as the computation of occurrences of patterns (given in an online manner) in a text or the computation of the longest common substring of two strings. However, it is usually the case that a suffix tree needs to first be augmented with other data structures before it can efficiently answer

¹ The $\tilde{O}(\cdot)$ notation hides factors polylogarithmic in n .



more sophisticated queries, e.g., returning the longest common prefix of two substrings or the longest palindrome centered at some position; an augmentation with a lowest common ancestors data structure suffices for these examples [16, 17].

Crucially, a text index, such as the suffix tree, is built once and can then be queried an arbitrary number of times. This is increasingly relevant: in many real-world scenarios, large pieces of information are stored on servers and are constantly queried by a large number of remote clients. From this perspective, it makes sense to devote some time to preprocess the data stored on the server in order to be able to provide quick responses to remote users later.

The *Heaviest Induced Ancestors* problem, which was introduced by Gagie et al. [14] and is defined next, has been proved to be useful in solving several variants of the problem of computing a longest common substring of two strings [1, 4, 5, 8, 14].

We say that a tree is weighted if there is a weight associated with each node u of the tree, such that weights along root-to-leaf paths are increasing, i.e., for any node u other than the root the weight of u is larger than the weight of u 's parent. Further, we say that a tree is labelled if each of its leaves is given a distinct label from $[n]$, where n is the number of leaves. As an example of a rooted, weighted, and labelled tree, consider the suffix tree of a string $S\$$, where $\$$ does not occur in S , with the label of each leaf being the starting position of the corresponding suffix and the weight of each node being the length of the string it represents.

► **Definition 1.** For two rooted and weighted trees T_1 and T_2 on n leaves, we say that two nodes $u \in T_1$ and $v \in T_2$, are induced (by label ℓ) if and only if there are leaves x and y labelled with ℓ , such that x and y are weak descendants of u and v , respectively.

HEAVIEST INDUCED ANCESTORS (HIA)

Input: Two rooted, weighted, and labelled trees T_1 and T_2 on n leaves.

Query: Given a pair of nodes $u \in T_1$ and $v \in T_2$, return a pair of induced nodes (u', v') with the largest total weight, such that u' is an ancestor of u and v' is an ancestor of v .

Previous results and our contribution. Table 1 shows the state-of-the-art size vs. query-time tradeoffs for the HIA problem prior to our work and our result. Gagie et al. [14] presented several tradeoffs which have been since improved. We stress that the $\mathcal{O}(n \log^2 n)$ -size data structure with query-time $\mathcal{O}(\log n)$ included in Table 1 was only sketched in [14]. We briefly discuss this sketch in Appendix A, as some of the ideas involved are similar to the ones we use. The remaining $\tilde{\mathcal{O}}(n)$ -size known data structures found in Table 1 are due to Abedin et al. [1]. Charalampopoulos et al. [8] showed an unconditional lower bound for near-linear size data structures and a data structure with query-time $\mathcal{O}(1)$ and size $\mathcal{O}(n^{1+\epsilon})$ for any constant $\epsilon > 0$. We now formally state our main result, which matches the lower bound of [8].

► **Theorem 2.** For any $\epsilon > 0$, there is an $\mathcal{O}(n \log^{2+2\epsilon} n)$ -size data structure for the HIA problem that can be constructed in $\tilde{\mathcal{O}}(n)$ time and answers queries in $\mathcal{O}(\log n / \log \log n)$ time.

Applications of HIA. Before discussing some concrete applications of the HIA problem in string algorithms and the consequences of our results for them, we give a high-level description of how the HIA problem comes up in variants of computing an LCS.

Consider a string S and a chosen subset A of its positions, that we call *anchors*. Further, consider the following two tries: a trie \mathcal{T}^{\leftarrow} for the strings in $\{S[1..k-1]^R : k \in A\}$, where U^R denotes the reversal of U , and a trie $\mathcal{T}^{\rightarrow}$ for the strings in $\{S[k..|S|] : k \in A\}$. In other words, for every anchor $k \in A$, we have a path in the first trie for every prefix of $S[1..k-1]^R$ and a path in the second trie for every prefix of $S[k..|S|]$. We label each leaf of the two tries

■ **Table 1** Size vs. query-time tradeoffs for the HIA problem; the size is measured in machine words.

Size	Query time	Paper
$\tilde{O}(n)$	$\Omega(\log n / \log \log n)$	[8]
$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n / \log \log n)$	[1]
$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n \log \log n)$	[1]
$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\log n)$	sketched in [14], see Appendix A
$\mathcal{O}(n \log^{2+\epsilon} n)$	$\mathcal{O}(\log n / \log \log n)$	<i>this work</i>
$\mathcal{O}(n^{1+\epsilon})$	$\mathcal{O}(1)$	[8]

with the anchor it corresponds to. Now, observe that a substring $S[i..j]$ that crosses an anchor k , i.e., $i < k \leq j$, corresponds to an induced pair of nodes in the tries. Indeed, there is a path representing $S[i..k-1]^R$ in the first trie and a path representing $S[k..j]$ in the second trie. An illustration of this idea is provided in Figure 1. The set of anchors and the HIA queries performed in an application of this technique depends on the specific problem it is used for. For some of the usages, one may consider using a compressed form of tries [18].

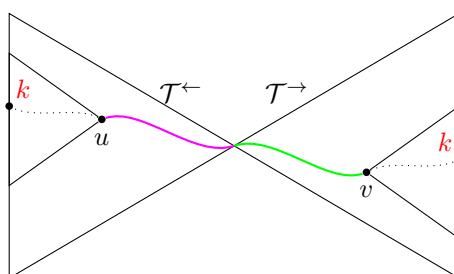
As a first application, consider the maintenance of an LCS of a static string T and a dynamic string S . By plugging our HIA data structure into the approach of [8], we obtain the following result, improving the state-of-the-art by polyloglog n factors, and matching the lower bound for the update-time when nearly-linear space is available [8, Theorem 1].

► **Corollary 3.** *We can maintain an LCS of a dynamic string S and a static string T , each of length at most n , in $\mathcal{O}(\log n / \log \log n)$ time per substitution operation using $\tilde{O}(n)$ space, after an $\tilde{O}(n)$ -time preprocessing.*

Further, the authors of [14] (implicitly) reduced to the HIA problem, the problem of preprocessing a text given in LZ77 compressed form so that one can compute its LCS with uncompressed patterns given online. Our HIA data structure yields the following result.

► **Corollary 4.** *Let S be a string of length N whose LZ77 parse consists of n phrases. We can store S in $\mathcal{O}(n \log N + n \text{ polylog } n)$ space such that, given a pattern P of length m , we can compute the LCS of S and P in $\mathcal{O}(m \log n / \log \log n)$ time. For each pattern P , the returned result may be (consistently) incorrect with probability inverse polynomial in n .²*

² Randomization is only used in the construction; all queries for the same pattern give identical results.



■ **Figure 1** An illustration of the anchoring technique for LCS computation, with the constructed tries \mathcal{T}^{\leftarrow} and $\mathcal{T}^{\rightarrow}$ drawn so that their roots are attached (in the middle). Any substring anchored at k , can be obtained by reading in a left-to-right manner the edge-labels from some node $u \in \mathcal{T}^{\leftarrow}$ to some node $v \in \mathcal{T}^{\rightarrow}$, that both have a leaf-descendant labelled with k .

Other applications of the HIA problem in string algorithms can be found in [1].

Tree Decompositions. One of the obvious divide-and-conquer techniques for efficiently solving algorithmic problems on trees is that of decomposing the tree(s) into smaller pieces and treating each of them separately. The most important attributes of a tree decomposition are usually its depth, i.e., the maximum number of pieces that one path can intersect, and the structure of each individual piece (e.g., pieces being paths may offer an advantage). We next describe some tree decompositions for a tree T with n nodes. For a node v , denote by $s(v)$ the number of nodes in v 's subtree.

Arguably, the most well-known tree decomposition is the *heavy-path decomposition* [17]. Abstractly, this decomposition is a partition of the edges into *light* and *heavy*, such that:

- all connected components after deleting the light edges are paths, called *heavy paths*;
- each root-to-leaf path consists of $\mathcal{O}(\log n)$ prefixes of heavy paths and $\mathcal{O}(\log n)$ light edges, i.e., the depth of the decomposition is $\mathcal{O}(\log n)$.

A heavy-path decomposition can be realized in several ways; two of which are as follows:

- HP1: Each non-leaf node u of the tree chooses a child v with maximum $s(v)$ and the edge from u to v is designated as heavy. The remaining edges outgoing from u are light.
- HP2: An edge (u, v) is designated as heavy if and only if $\lfloor \log s(u) \rfloor = \lfloor \log s(v) \rfloor$.³

Intuitively, using a heavy-path decomposition, one may often lift an algorithm that only works for paths and/or balanced trees to work for arbitrary trees – usually with some overhead.

All previous works on the HIA problem used heavy-path decompositions, which, as discussed, are of depth $\Omega(\log n)$. This adversely affects their query times as one may have to traverse the decomposition along a root-to-leaf path at query time. Thus, in order to achieve sublogarithmic query time, we considered tree decompositions of smaller depths. There are a couple of generalizations of heavy-path decompositions that have the sought depth, i.e., $\mathcal{O}(\log n / \log \log n)$. We next discuss two such decompositions that are also based on partitioning the edges into light and heavy. The caveat is that, for each of them, the connected components after the removal of the light edges are trees, which we call *heavy trees*, instead of paths and hence some extra work is required.⁴

The heavy α -tree decomposition, introduced by Bille et al. [6], is of depth $\mathcal{O}(\log_\alpha n)$ and is defined analogously to HP1: each non-leaf node u chooses its (at most) α heaviest (with respect to subtree-sizes) children; the edge from u to each of these children is designated as heavy, while all remaining edges outgoing from u are designated as light. By setting $\alpha = \lfloor \log n \rfloor$ one gets the sought depth.

An alternative is the so-called ART decomposition due to Alstrup et al. [2], which, for an input integer parameter b , has depth $\mathcal{O}(\log_b n)$. For ease of presentation, we consider b to be equal to $\lfloor \log n \rfloor$ so that the depth of the decomposition is $\mathcal{O}(\log n / \log \log n)$. A partition of the edges yields such an ART decomposition if and only if each heavy tree contains $\mathcal{O}(\log n)$ nodes that have more than one child (in the heavy tree). Alstrup et al. [2] showed how to compute an ART decomposition by computing a set of leafmost light edges (in the spirit of

³ In some works this has been called a centroid decomposition [10]. It should not be confused with the hierarchical decomposition of the tree obtained by recursively deleting a centroid node, that is, a node whose removal splits the tree into three roughly equal components [7, 15].

⁴ Heavy trees are sometimes called *micro trees*, while the tree obtained from T by contracting each micro tree is called a *macro tree*. We avoid this notation to not confuse with the so-called *micro-macro decomposition* [3], which, for a positive integer $k \leq n$, is a partition of the vertices of T into $\mathcal{O}(n/k)$ sets, such that each set S is of size $\mathcal{O}(k)$ it induces a subtree of T and has at most two vertices that have neighbours that are not in S .

HP2 with the base of the logarithm changed from 2 to $\lfloor \log n \rfloor$), removing them along with their descendants from the tree, and recursing. Here, for convenience, we compute an ART decomposition similar to the HP2-realization of a heavy-path decomposition: an edge (u, v) is heavy if and only if both $s(u)$ and $s(v)$ are in $(n/\lfloor \log n \rfloor^{k+1}, n/\lfloor \log n \rfloor^k]$. For each heavy tree, we call *branches* its maximal down-the-tree paths in which all nodes except the deepest one have exactly one child (in the heavy tree); each heavy tree has $\mathcal{O}(\log n)$ branches.

Our techniques. In order to answer an HIA query for nodes u and v , we consider $\mathcal{O}(\log n / \log \log n)$ pairs of heavy trees that consist of a heavy tree in the root-to- u path and a heavy tree in the root-to- v path. For each such pair, we compute an induced pair (x, y) of nodes in these heavy trees that are ancestors of u and v , respectively, and have maximum total weight. Similarly to previous work, we observe that not every pair of heavy trees needs to be considered. Instead, it suffices to consider a number of pairs of heavy trees linear to the depth of the tree decompositions by a procedure analogous to the natural algorithm for checking whether there are two elements of a sorted list that sum to a target t : start with two pointers, one at the beginning of the list and one at the end and move each of them in only one direction (either to the right or left). For each pair of heavy trees, we construct data structures that can efficiently handle each of the cases of how the locations of the lowest ancestors of u and v in the heavy trees relate to the locations of the lowest ancestors (in the heavy trees) of same-label leaves. Each data structure considers similar cases as previous work, however now we are working with two trees instead of two paths, and hence need to be more careful. This way, we reduce an HIA query to $\mathcal{O}(\log n / \log \log n)$ predecessor queries. By answering each of these predecessor queries independently, we obtain a data structure that answers HIA queries in $\mathcal{O}(\log n)$ time. Indeed, in our case, each predecessor query requires $\Omega(\log \log n)$ time to be answered independently.

However, crucially, we show how to design the data structures so that all predecessor queries need only two values: the preorder number of u or the preorder number of v . This is achieved by reordering the trees so that heavy edges come last. Then, larger preorder numbers correspond to a larger depth of the lowest ancestor on a branch. This means that the combination of our techniques with fractional cascading would yield a faster algorithm for answering all the predecessor queries; the first one for each queried value would take $\mathcal{O}(\log \log n)$ time, while all subsequent ones would take $\mathcal{O}(1)$ time each. The final technical hurdle is that fractional cascading requires the so-called underlying catalog graph to have polylogarithmic degree [21]. The construction of such a graph is straightforward if T_1 and T_2 are of polylogarithmic degree: roughly speaking, it suffices to consider the Cartesian product of two trees whose nodes represent branches and heavy trees of each of T_1 and T_2 . We overcome this difficulty in the general case by reducing the maximum degree of these trees prior to taking their Cartesian product while maintaining all of their desirable properties.

2 Preliminaries

We use $[n]$ to denote the set $\{1, 2, \dots, n\}$. Throughout the paper, we perform the same operations on T_1 and T_2 and define objects in these trees, so we are going to use T_\star to denote any of the trees. Similarly, we are going to use v_\star to denote a node $v_1 \in T_1$ or a node $v_2 \in T_2$ etc. as an abbreviation of writing that some property holds for v_i for both $i \in \{1, 2\}$.

Lowest Common Ancestor. LCA queries can be answered in constant time after an $\mathcal{O}(n)$ -time preprocessing [17].

LOWESTCOMMONANCESTOR (LCA)

Input: A rooted tree T .

Query: What is the node of largest depth that is an ancestor of both u and v ?

Predecessor query. For a static set S , a combination of x -fast tries [23] and deterministic dictionaries [20] yields an $\mathcal{O}(n)$ -size data structure that can be built in $\mathcal{O}(n)$ time and answers predecessor queries in $\mathcal{O}(\log \log U)$ time deterministically (cf., [12, Proposition 2]); this is optimal [19].

PREDECESSORQUERY

Input: A set S of n integers from $[U]$.

Query: For a given integer y , what is the largest $x \in S$ such that $x \leq y$?

Range Minimum Query. RMQs can be answered in constant time after an $\mathcal{O}(n)$ -time preprocessing [11, 13]. By setting, for each i , $S[i] := |U| - S[i]$, we get a structure for the symmetric RANGEMAXIMUMQUERY problem.

RANGEMINIMUMQUERY (RMQ)

Input: A sequence S of n integers from $[U]$.

Query: For given positions i and j , with $1 \leq i \leq j \leq n$, what is (the position of) the minimum among $S[i], S[i+1], \dots, S[j]$?

Deterministic Static Dictionary. A dictionary is a structure that stores a set of keys (often with associated values) and allows answering membership queries (or getting the value of a given key). There are multiple randomized solutions, but there is even a deterministic solution with $\mathcal{O}(n)$ space, $\tilde{\mathcal{O}}(n)$ -time preprocessing, and constant-time queries [20].

Fractional cascading. Consider a directed graph C , called the *catalog graph*, which has a sorted list (also called a catalog) in each of its nodes. Let the total size of the lists be n . Now, suppose that we want to answer queries of the following type: for a connected subgraph G of C and a query value v , find the predecessor of v in each of the lists stored in the nodes of G .

A naive way of solving this problem would be to ignore any preprocessing and run a separate binary search in the sorted list of each of the nodes of G , for a total of $\mathcal{O}(|G| \log n)$ time. Fractional cascading is a general optimization technique that allows the speed-up of multiple binary searches for the same value over multiple related sorted sequences of objects.

If the degree of each node of the catalog graph is bounded by a constant, the original solution of Chazelle and Guibas [9] answers a query in $\mathcal{O}(\log n + |G|)$ time after a linear-time preprocessing in the comparison model. To be precise, it is sufficient for the catalog graph to have *locally bounded degree* (as per Definition 1 of [9]). Unfortunately, in our case, this is not useful. A subsequent work of Shi and JáJá [21] achieved the same complexities for graphs of polylogarithmic maximum degree in the word RAM model of computation (in the original description, the graph is a tree, however, there is no difficulty in extending this to the general setting considered by Chazelle and Guibas [9]). Crucially, these data structures can also handle the case where the nodes of G are given one by one in an online manner; the only requirement is that each node (other than the first) must be a neighbour of some previous one. Note that the $\mathcal{O}(\log n)$ term in the complexities comes from performing a binary search in the first of the considered lists. Then, the predecessor of the query value v in each of the subsequently considered lists is obtained by following a constant number of

pointers, which can be retrieved in $\mathcal{O}(1)$ time. (During the preprocessing phase, the catalogs are augmented in an appropriate manner and said pointers are constructed.) In the word RAM model of computation, the first query can be solved faster using other data structures, e.g., in $\mathcal{O}(\log \log U)$ time with the structure discussed above for the PREDECESSORQUERY problem.

3 An $\tilde{\mathcal{O}}(n)$ -size Data Structure with Optimal Query Time

Let $b > 1$ be a parameter to be chosen later. Consider a rooted, weighted, and labelled tree T . The weight of a node u is denoted by $\text{weight}(u)$. For a node v , we denote by $s(v)$ the number of nodes in v 's subtree, including v . For an integer k , a node v is on layer k if and only if $n/b^{k+1} < s(v) \leq n/b^k$. An edge that connects nodes of the same layer is called *heavy* and the other edges are *light*. Each maximal subtree that does not contain any light edges is called a *heavy tree*. We stress that a heavy tree might be a singleton.

We decompose each heavy tree into *branches*, that is, maximal down-the-tree paths of nodes, where every node apart from the deepest one has one child (in the heavy tree). The last node is either a leaf of the heavy tree or has at least two children. Note that there can be branches consisting of a single node. We call a node *implicit* if it is an internal (non-leaf) node of the heavy tree with one child; otherwise, we call it *explicit*. For a heavy tree, we obtain a *compacted* version of it, called *compacted heavy tree*, by eliminating all the implicit nodes through the contraction of either of their incident edges. See an example in Figure 2.

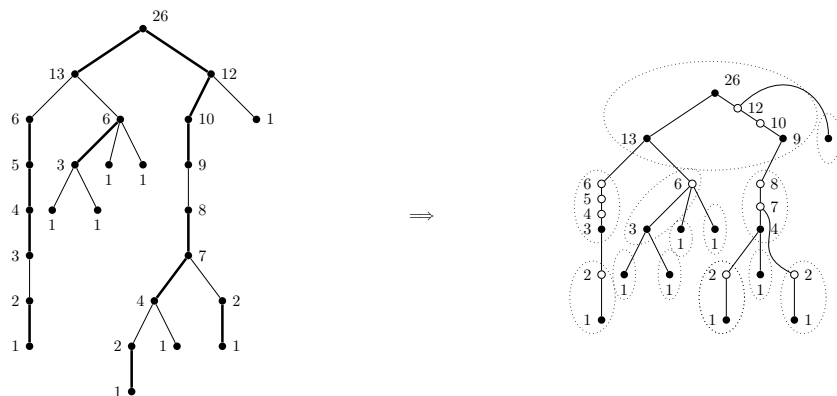


Figure 2 An example tree with $n = 26$ is shown. Each node is labelled with its subtree size, while $b = 3$. On the left, the heavy edges are thick, while the light edges are thin. On the right, the tree is decomposed into heavy trees (marked with ellipses). The compaction of heavy trees is illustrated as follows: empty circles denote implicit nodes, while full circles denote explicit nodes.

Observe that in a compacted tree, every non-leaf node has at least two children. Hence, there are fewer internal nodes than there are leaves of the tree. As every leaf in a heavy tree has a sufficiently big subtree underneath it, we obtain a bound on the total number of nodes inside a single compacted heavy tree; this shows that we obtain an ART decomposition [2].

► **Lemma 5.** *There are at most $\mathcal{O}(\log n / \log b)$ layers in a tree T on n leaves and each heavy tree has $\mathcal{O}(b)$ branches.*

Proof. Consider a heavy tree H of T and its compacted form H_C . Let r be the root of H . For every leaf ℓ of H , we have that $s(\ell) > s(r)/b$ as ℓ and r are nodes of the same heavy tree, and hence they are in the same layer of T . As the subtrees of T rooted at the leaves of H are

8:8 Optimal Near-Linear Space Heaviest Induced Ancestors

disjoint and their total size is at most $s(r)$, H contains at most b leaves. Further, as there are no internal nodes with one child in H_C , there are at most $b - 1$ non-leaf nodes in H_C , and hence H_C has $\mathcal{O}(b)$ nodes. All branches in H are disjoint, so there are $\mathcal{O}(b)$ of them.

Consider the k -th layer of the tree T and a node u in this layer. As the subtree of u has at least one node, we have $1 \leq s(u) \leq n/b^k$, so $k \leq \log_b n = \log n / \log b$. ◀

For an HIA query (v_1, v_2) , the paths from v_1 and v_2 to the roots of their respective trees are called *query paths*. The result of an HIA query is a pair of nodes (u_1, u_2) , that are on the query paths from v_1 and v_2 , respectively. To answer a query (v_1, v_2) , we identify the sequences of heavy trees B_1 and B_2 that contain nodes on the query paths from v_1 and v_2 , respectively, and perform *restricted HIA queries* for some pairs of those heavy trees. More precisely, in each step of the algorithm, having chosen two heavy trees $B_1[i]$ and $B_2[j]$, we try to find the pair of induced ancestors $(u_1, u_2) \in B_1[i] \times B_2[j]$ of (v_1, v_2) with the maximum combined weight or determine that there is no such pair. A pseudocode for this procedure is given as Algorithm 1.

■ **Algorithm 1** Algorithm for answering HIA queries.

```

1 function hia( $T_1, T_2, v_1, v_2$ )
2    $(r_1, r_2) \leftarrow \text{null}$ 
3    $(B_1, B_2) \leftarrow$  sequences of heavy trees on paths from  $v_1$  and  $v_2$  down-the-tree
4    $i \leftarrow 0$ 
5    $j \leftarrow |B_2| - 1$ 
6   repeat
7      $x_1 \leftarrow$  lowest ancestor of  $v_1$  in  $B_1[i]$ 
8      $x_2 \leftarrow$  lowest ancestor of  $v_2$  in  $B_2[j]$ 
9      $(u_1, u_2) \leftarrow$  restricted-hia( $v_1, v_2, B_1[i], B_2[j], x_1, x_2$ )
10    if  $(r_1, r_2) = \text{null}$  or  $\text{weight}(u_1) + \text{weight}(u_2) > \text{weight}(r_1) + \text{weight}(r_2)$ 
11      then
12         $(r_1, r_2) \leftarrow (u_1, u_2)$ 
13    if  $(i = |B_1| - 1)$  and  $(j = 0)$  then
14      return  $(r_1, r_2)$ 
15    if  $i + 1 < |B_1|$  and roots of  $B_1[i + 1]$  and  $B_2[j]$  are induced then
16       $i \leftarrow i + 1$ 
17    else
18       $j \leftarrow j - 1$ 
19  until false

```

To make the description more modular, we provide x_1 and x_2 to the restricted HIA query, where x_\star is the lowest ancestor of v_\star in the considered heavy tree. Note that x_1 is either the parent of the root of $B_1[i + 1]$ if $i + 1 < |B_1|$ or x_1 equals to v_1 otherwise, and similarly for x_2 . We note that in Algorithm 1, we ask restricted HIA queries about the same pair (v_1, v_2) for various pairs $(B_1[i], B_2[j])$ of heavy trees and it may be that $v_1 \notin B_1[i]$ and/or $v_2 \notin B_2[j]$, whereas we also provide nodes x_\star , in which v_\star connects to the respective heavy subtree in B_\star .

► **Lemma 6.** *Algorithm 1 performs $\mathcal{O}(\log n / \log b)$ restricted HIA queries to find the heaviest induced ancestors of nodes u and v .*

Proof. The paths from v_* to the roots pass heavy trees with monotonically decreasing indices of layers, so the sequences B_1 and B_2 contain at most $\mathcal{O}(\log n / \log b)$ elements. After every restricted HIA query, we either increase i or decrease j by 1, so the total number of restricted HIA queries that we perform is at most $|B_1| + |B_2| = \mathcal{O}(\log n / \log b)$.

The correctness follows from the monotonicity of being induced. For an induced pair of nodes, any pair of their (weak) ancestors is also induced. Conversely, if a pair is not induced, any pair of their (weak) descendants is also not induced.

This implies that if the roots of $B_1[i]$ and $B_2[j]$ are induced then there is no need to query for any pair $(B_1[i'], B_2[j'])$ with $(i' < i) \wedge (j' < j)$, as such a query would return a pair of (strict) ancestors of the pair returned by the restricted HIA query for $(B_1[i], B_2[j])$. We call a pair of trees $(B_1[i'], B_2[j'])$ *dominated*, if there exist $i > i'$ and $j > j'$ such that the roots of $B_1[i]$ and $B_2[j]$ are induced. We show that the algorithm performs restricted HIA queries at Line 9 exactly for those pairs of heavy trees that (i) are not dominated and (ii) for which the result of the restricted HIA query is not null.

The algorithm maintains the invariant that each of the restricted HIA queries is called for the pairs of trees for which their roots are induced and that the pair $(B_1[i], B_2[j])$ is not dominated. This is true for the first iteration, where the pair $(B_1[0], B_2[|B_2| - 1])$ is considered, because $B_2[|B_2| - 1]$ has a leaf and the root of $B_1[0]$ is the root of T_1 . Now we show that the invariant is maintained later. We start with the assumption that the result of `restricted-hia` $(v_1, v_2, B_1[i], B_2[j], x_1, x_2)$ is not null and this pair is non-dominated. Now, we have to distinguish between two cases.

Case 1: We next consider pair $(B_1[i + 1], B_2[j])$. It means that the check in Line 15 confirmed that the roots of the trees are induced, so the result of calling `restricted-hia` $(v_1, v_2, B_1[i + 1], B_2[j], x_1, x_2)$ is not null. Further, this pair of heavy paths is not dominated since $(B_1[i], B_2[j])$ is not dominated.

Case 2: We next consider pair $(B_1[i], B_2[j - 1])$. This can only happen if $i = |B_1| - 1$ or the roots of $B_1[i + 1]$ and $B_2[j]$ are not induced; in either of these cases, $(B_1[i], B_2[j - 1])$ is not dominated. Further, as the answer to the HIA query for pair $(B_1[i], B_2[j])$ is not null, the answer for $(B_1[i], B_2[j - 1])$ cannot be null either.

Thus, the invariant is maintained in both cases.

Clearly, the heaviest induced pair of ancestors of v_1, v_2 belongs to a pair of heavy trees that satisfy conditions (i) and (ii). We claim that we process all such pairs. Observe that for a fixed j there are two indices $0 \leq i_1 < i_2 \leq |B_1|$ such that the pair $B_1[i], B_2[j]$ is dominated for $0 \leq i < i_1$, non-dominated for $i_1 \leq i < i_2$, and corresponds to a null answer for $i_2 \leq i < |B_1|$. By the invariant, just after any decrease of j in Line 18 it holds that $i \geq i_1$, as $B_1[i], B_2[j]$ is non-dominated. Actually $i = i_1$, because the pair $B_1[i - 1], B_2[j]$ is dominated as in the previous step we considered the pair $B_1[i], B_2[j + 1]$ for which the answer was not null. As in the next steps we process all i up to (but excluding) i_2 , the claim follows. ◀

3.1 Restricted HIA Queries

In this subsection, we present a data structure that efficiently answers restricted HIA queries.

► **Theorem 7.** *For every two trees T_1, T_2 on n leaves and an integer parameter $b \in [n]$, there exists an $\mathcal{O}(nb^2 \log^2 n / \log^2 b)$ -size data structure that can be computed in $\tilde{\mathcal{O}}(nb^2 / \log^2 b)$ time and answers (i) queries about whether the roots of two given heavy trees are induced in constant time, (ii) any restricted HIA query `restricted-hia` $(v_1, v_2, H_1, H_2, x_1, x_2)$ in*

8:10 Optimal Near-Linear Space Heaviest Induced Ancestors

constant time plus the time required to answer a predecessor query about $\text{pre}(v_1)$ and one about $\text{pre}(v_2)$; these predecessor queries are performed on two out of $\mathcal{O}(b^2)$ (preprocessed) lists stored for the pair of heavy trees (H_1, H_2) .

We divide the proof into three parts: we first describe the preprocessing phase, then discuss the properties of the created data structure, and, finally, present the query procedure.

Preprocessing. First, we compute the partition of the edges of each of T_1, T_2 into heavy and light, and the implied heavy trees. For each node, we store its assignment to the heavy tree and to the branch to which it belongs. Further, for each T_* , we build a linear-size data structure for answering LCA queries in $\mathcal{O}(1)$ time [17]. For each node in T_* , we fix the order of its children such that the children that are in the same heavy tree are last. (The order of children that are connected to the parent with the same type of edge, heavy or light, is arbitrary.) Next, we compute preorder traversals of T_* , for each node u , we denote by $\text{pre}(u)$ the preorder number of u and by $T_*[p]$ the node of T_* whose preorder number is p ; we have $T_*[\text{pre}(u)] = u$. Additionally, for each label, we identify the leaves of T_1 and T_2 with that label (recall the labels in a single tree are pairwise distinct).

Next, for each pair (ℓ_1, ℓ_2) of leaves with the same label, we iterate over all pairs $(B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2])$ of heavy trees on their query paths and insert a point to the data structure for each pair of branches in $B_1^{\ell_1}[i_1] \times B_2^{\ell_2}[i_2]$. This procedure is formalized as Algorithm 2.

■ **Algorithm 2** Preprocessing for a pair of leaves (ℓ_1, ℓ_2) with the same label.

```

1 procedure add_label( $T_1, T_2, \ell_1, \ell_2$ )
2    $(B_1^{\ell_1}, B_2^{\ell_2}) \leftarrow$  sequence of heavy trees on query paths from  $\ell_1$  and  $\ell_2$ 
3   for  $i_1 \leftarrow 0, 1, \dots, |B_1^{\ell_1}| - 1$  do
4     for  $i_2 \leftarrow 0, 1, \dots, |B_2^{\ell_2}| - 1$  do
5       for  $e_1 \leftarrow$  branch of  $B_1^{\ell_1}[i_1]$  do
6         for  $e_2 \leftarrow$  branch of  $B_2^{\ell_2}[i_2]$  do
7            $w_1 \leftarrow$  LCA( $\ell_1$ , lowest node on  $e_1$ )
8            $w_2 \leftarrow$  LCA( $\ell_2$ , lowest node on  $e_2$ )
9           insert point  $(\text{pre}(w_1), \text{pre}(w_2))$  to  $D^{B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2]}[e_1, e_2]$ 

```

We call pairs (B_1, B_2) of heavy trees that are processed by Algorithm 2 *relevant*. We first run this algorithm once just to record all relevant pairs of heavy trees, without inserting any points to any structures. We then sort the relevant pairs, remove duplicates, and construct a deterministic dictionary over them [20]. This allows us to check in constant time if the roots of two trees are induced because this is equivalent to checking if the pair of trees is relevant. For each relevant pair of heavy trees, we initialize an array D^{B_1, B_2} indexed by pairs of branches (e_1, e_2) , where e_1 is a branch in B_1 and e_2 is a branch in B_2 . In each entry of the array, we create (store a pointer to) a data structure for the corresponding pair of branches. We then re-run Algorithm 2, inserting the points to the structures as needed, with the help of the deterministic dictionary built for relevant pairs of heavy trees. As each branch e_* belongs to a unique heavy tree, we often drop the superscript and write $D[e_1, e_2]$ instead of $D^{B_1, B_2}[e_1, e_2]$. We call a pair (e_1, e_2) of branches *relevant* if and only if pair of their assigned heavy trees is relevant. By Lemma 5, every query path is decomposed into $\mathcal{O}(\log n / \log b)$ parts on different layers and each heavy tree has $\mathcal{O}(b)$ branches. Hence, for every pair (ℓ_1, ℓ_2) of leaves with the same label, we insert a point to $\mathcal{O}(b^2 \log^2 n / \log^2 b)$ structures.

Finally, for each relevant pair of branches (e_1, e_2) , we perform the following postprocessing of structure $D[e_1, e_2]$:

- Remove all points (x, y) for which there exists another point (x', y') such that $x \leq x', y \leq y'$ and $(x, y) \neq (x', y')$. This can be done in $\tilde{O}(|D[e_1, e_2]|)$ time by sorting the points and processing them in the left-to-right order.
- Let $D_x[e_1, e_2]$ and $D_y[e_1, e_2]$ be the sets of x - and y -coordinates of the remaining points, respectively. We build a data structure for the PREDECESSORQUERY problem for each of $D_x[e_1, e_2]$ and $D_y[e_1, e_2]$ separately.
- We build a data structure for the RANGEMAXIMUMQUERY problem for the points remaining in $D[e_1, e_2]$ sorted by x -coordinate, where the weight of a point (x, y) is $\text{weight}(T_1[x]) + \text{weight}(T_2[y])$.

We call the above stage the postprocessing of $D[e_1, e_2]$.

To summarize the whole preprocessing stage for trees T_* , for each of the n labels we add $\mathcal{O}(b^2 \log^2 n / \log^2 b)$ points to structures $D[\cdot, \cdot]$, for a total number of $\mathcal{O}(nb^2 \log^2 n / \log^2 b)$ points. The postprocessing of all the structures $D[\cdot, \cdot]$ takes nearly linear time in their size and hence the total running time is $\tilde{O}(nb^2 / \log^2 b)$. The structures for the PREDECESSORQUERY and RANGEMAXIMUMQUERY problems have size linear in the number of elements they are built over and hence the total space is $\mathcal{O}(nb^2 \log^2 n / \log^2 b)$.

Properties of structures $D[e_1, e_2]$. In this paragraph, we show some properties of the structures $D[e_1, e_2]$ that are useful for answering restricted HIA queries efficiently.

► **Property 8.** *For every pair (w_1, w_2) added to $D^{B_1^{\ell_1}[i_1], B_2^{\ell_2}[i_2]}[e_1, e_2]$, w_* is either on e_* or on the path from the highest node of e_* to the root of $B_*^{\ell_*}[i_*]$.*

Proof. Recall that w_* is the lowest common ancestor of ℓ_* and the lowest node q on e_* . Observe that as $B_*^{\ell_*}[i_*]$ is on the query path from ℓ_* , the root r of $B_*^{\ell_*}[i_*]$ is an ancestor of both ℓ_* and q . Hence, w_* lies on the r -to- q path, which directly yields the statement. ◀

Note that after the first step of postprocessing, $D[e_1, e_2]$ satisfies the following property:

► **Property 9.** *After the postprocessing, for every pair (e_1, e_2) of branches, after sorting the points of $D[e_1, e_2]$ increasingly by x -coordinate, the sequence of points is also sorted decreasingly by y -coordinate.*

Informally, we can now consider a one-dimensional problem, with points forming a sequence that can be efficiently navigated both in x - and y -coordinates via predecessor queries.

We next show how the computed data structures $D[e_1, e_2]$ enable us to answer restricted HIA queries efficiently.

Answering a restricted HIA query. We are now ready to present how to answer a restricted HIA query for a pair (v_1, v_2) of nodes and heavy trees B_1 and B_2 on the query paths from v_1 and v_2 . Let $(r_1, r_2) = \text{restricted-hia}(v_1, v_2, B_1, B_2, x_1, x_2)$ be a pair of ancestors of v_1 and v_2 within the trees B_1 and B_2 that are induced and have the maximum total weight. Recall that x_* is the lowest weak ancestor of v_* that is in B_* and let e_* be the branch containing x_* . Further, let ℓ be the label inducing (r_1, r_2) and let leaves ℓ_* share this label.

First, we show that we can find an induced pair of ancestors of v_1 and v_2 with the maximum combined weight using the structure $D[e_1, e_2]$ before postprocessing. Then, we show that after the postprocessing stage, we can still retrieve the correct answer but more efficiently, by performing predecessor queries for $\text{pre}(x_1)$ and $\text{pre}(x_2)$. Finally, we show that

8:12 Optimal Near-Linear Space Heaviest Induced Ancestors

we can call predecessor queries for $\text{pre}(v_1)$ and $\text{pre}(v_2)$ instead of $\text{pre}(x_1)$ and $\text{pre}(x_2)$. The last step is not important for the correctness or efficiency of a single restricted HIA query but improves the complexity of Algorithm 1. Indeed, as all predecessor queries are for one of $\text{pre}(v_1)$ or $\text{pre}(v_2)$, we can use fractional cascading. We explain this final component in detail in Section 3.2.

Recall that in Algorithm 2, we insert point $(\text{pre}(w_1), \text{pre}(w_2))$ to $D[e_1, e_2]$, where $w_\star = \text{LCA}(\ell_\star, \text{lowest node on } e_\star)$. In the proof of Property 8, we mention that w_\star always belongs to B_\star as the root of B_\star is an ancestor of both ℓ_\star and the lowest node on e_\star . There are two possible relative locations of w_\star and x_\star within a heavy tree:

- ℓ is *below* x_\star when w_\star is a (not necessarily proper) descendant of x_\star ;
- ℓ is *attached above* x_\star when w_\star is a proper ancestor of x_\star .

There are four cases for the relative locations of ℓ with respect to x_1 and x_2 :

- Case 1:** ℓ is attached above x_1 and x_2 ,
- Case 2:** ℓ is attached above x_1 and ℓ is below x_2 ,
- Case 3:** ℓ is attached above x_2 and ℓ is below x_1 ,
- Case 4:** ℓ is below x_1 and x_2 .

We next treat each of these cases. For each of them, we retrieve the pair of induced ancestors of x_1 and x_2 with the largest total weight among all pairs of ancestors induced by a label ℓ appropriately located with respect to x_1 and x_2 . Each of these variants gives us a candidate pair for the restricted heaviest induced ancestors of x_1 and x_2 . In the end, we return the candidate with the largest total weight. Similar case analysis was performed in previous solutions for the HIA problem, e.g., in [14].

► **Lemma 10.** *The answer to restricted-hia $(v_1, v_2, B_1, B_2, x_1, x_2)$ can be retrieved from the information stored in $D[e_1, e_2]$ before the postprocessing.*

Proof. By Property 8, for every two points $(\text{pre}(w_1), \text{pre}(w_2))$ and $(\text{pre}(w'_1), \text{pre}(w'_2))$ added to $D[e_1, e_2]$, we have that w_1 is either a weak ancestor or a descendant of w'_1 , and similarly for w_2 and w'_2 . Hence, the preorder numbers of the nodes correspond to their depths and we can check the ancestry relation by comparing them: for nodes u, u' on a path, u is a weak ancestor of u' if and only if $\text{pre}(u) \leq \text{pre}(u')$.

Using this property, we show how to reduce each of the four cases listed above to finding a specific point in a particular rectangular subset of points. For now, we ignore the efficiency of the queries (a trivial implementation takes linear time) and focus on showing that the correct answer to the restricted HIA query can be retrieved from $D[e_1, e_2]$ before the postprocessing.

Case 1: Every leaf ℓ that is attached above x_1 and x_2 in nodes w_1 and w_2 makes the pair (w_1, w_2) a candidate result of the restricted HIA query. Hence we need to find a point (x, y) in $D[e_1, e_2]$ such that $x < \text{pre}(x_1)$, $y < \text{pre}(x_2)$, and $\text{weight}(T_1[x]) + \text{weight}(T_2[y])$ is maximum. Then, $(T_1[x], T_2[y])$ is a restricted HIA candidate pair for (v_1, v_2) .

Case 2: We need to find a point $(x, y) \in D[e_1, e_2]$ such that $x < \text{pre}(x_1)$, $y \geq \text{pre}(x_2)$ and $\text{weight}(T_1[x])$ is maximized. Then, $(T_1[x], x_2)$ is a restricted HIA candidate pair for (v_1, v_2) .

Case 3: This case is symmetric to Case 2. We need to find a point $(x, y) \in D[e_1, e_2]$ such that $x \geq \text{pre}(x_1)$, $y < \text{pre}(x_2)$ and $\text{weight}(T_2[y])$ is maximized. Then, $(x_1, T_2[y])$ is a restricted HIA candidate pair for (v_1, v_2) .

Case 4: We need to check if there exists a point (x, y) such that $x \geq \text{pre}(x_1)$ and $y \geq \text{pre}(x_2)$. If so, the pair (x_1, x_2) is a restricted HIA candidate pair for (v_1, v_2) .

In each of the cases, we return a pair, if one exists, of induced ancestors of (x_1, x_2) and hence also of (v_1, v_2) . The label ℓ of leaves ℓ_1 and ℓ_2 that induces the pair (r_1, r_2) of heaviest induced ancestors of (v_1, v_2) in $B_1 \times B_2$ inserted the point to $D[e_1, e_2]$, since B_1 and B_2 are on the query paths from ℓ_1 and ℓ_2 , respectively. Hence, the pair (r_1, r_2) is found while considering one of the four cases. \blacktriangleleft

Now, we show that it suffices to run the above algorithm only for the points in $D[e_1, e_2]$ after the postprocessing stage.

► **Lemma 11.** *The answer to **restricted-hia** $(v_1, v_2, B_1, B_2, x_1, x_2)$ can be retrieved from the information stored in $D[e_1, e_2]$ after the postprocessing.*

Proof. As discussed in the proof of Lemma 10, for any two points $(\text{pre}(w_1), \text{pre}(w_2))$ and $(\text{pre}(w'_1), \text{pre}(w'_2))$ added to $D[e_1, e_2]$, w_i and w'_i lie on a single root-to-leaf path, so their preorder numbers are in the same order as their depths in the tree. Recall that the trees are monotonically weighted, that is, the weights along each root-to-leaf path are increasing, so we have that $\text{pre}(w_i) \leq \text{pre}(w'_i)$ implies $\text{weight}(w_i) \leq \text{weight}(w'_i)$.

Let $w = (\text{pre}(w_1), \text{pre}(w_2))$ be the point corresponding to the answer found by the algorithm presented in Lemma 10. Note that the returned induced pair of ancestors is not necessarily (w_1, w_2) , e.g., it can be (w_1, x_2) . Suppose that w was removed during the postprocessing phase. If so, it happened because there exists a point $w' = (\text{pre}(w'_1), \text{pre}(w'_2))$ where $\text{pre}(w'_1) \leq \text{pre}(w'_2)$ and $w \neq w'$. If w' is processed in a different case than w , then the pair of ancestors corresponding to w' has a larger total weight than the one returned, yielding a contradiction. If w' is processed in the same case as w , then the pair of ancestors corresponding to w' gives a pair of ancestors whose total weight is not smaller than that of the returned pair. Hence, the reduction presented in the proof of Lemma 10 still holds for the set $D[e_1, e_2]$ after the postprocessing. \blacktriangleleft

Next, we present how to implement each of the four cases in Lemma 10 efficiently using the fact that the points in $D[e_1, e_2]$ have been postprocessed.

► **Lemma 12.** *The answer to **restricted-hia** $(v_1, v_2, B_1, B_2, x_1, x_2)$ can be retrieved from the information stored in $D[e_1, e_2]$ after the postprocessing, with two predecessor queries: one for $\text{pre}(x_1)$ and one for $\text{pre}(x_2)$.*

Proof. By computing the predecessor of $\text{pre}(x_1)$ in $D_x[e_1, e_2]$, we obtain intervals $\mathcal{I}_x^{x < \text{pre}(x_1)}$ and $\mathcal{I}_x^{x \geq \text{pre}(x_1)}$ of $D_x[e_1, e_2]$. Similarly, intervals $\mathcal{I}_y^{y < \text{pre}(x_2)}$ and $\mathcal{I}_y^{y \geq \text{pre}(x_2)}$ of $D_y[e_1, e_2]$ are obtained by computing the predecessor of $\text{pre}(x_2)$ in $D_y[e_1, e_2]$. Note that by Property 9, points from $\mathcal{I}_y^{y \geq \text{pre}(x_2)}$ (resp. $\mathcal{I}_y^{y < \text{pre}(x_2)}$) of $D_y[e_1, e_2]$ correspond to points from the interval of $D_x[e_1, e_2]$ that we denote $\mathcal{I}_x^{y \geq \text{pre}(x_2)}$ ($\mathcal{I}_x^{y < \text{pre}(x_2)}$). Hence, we can translate each of the conditions on points in the cases of Lemma 10 to an intersection $\mathcal{I}_x^{\text{Case } i}$ of two intervals on $D_x[e_1, e_2]$. This reduces each of the four cases to:

Case 1: Find the point with maximum weight $\text{weight}(T_1[x]) + \text{weight}(T_2[y])$ in $\mathcal{I}_x^{\text{Case } 1}$ using an RMQ.

Case 2: By the monotonicity of weights with respect to x -coordinates, the point with maximum weight $\text{weight}(T_1[x])$ in $\mathcal{I}_x^{\text{Case } 2}$ is the rightmost element of $\mathcal{I}_x^{\text{Case } 2}$.

Case 3: By the monotonicity of weights with respect to y -coordinates and Property 9, the point with maximum weight $\text{weight}(T_2[y])$ in $\mathcal{I}_x^{\text{Case } 3}$ is the leftmost element of $\mathcal{I}_x^{\text{Case } 3}$.

Case 4: It suffices to check if $\mathcal{I}_x^{\text{Case } 4}$ is non-empty. \blacktriangleleft

► **Lemma 13.** *The answer to restricted-hia $(v_1, v_2, B_1, B_2, x_1, x_2)$ can be retrieved from the information stored in $D[e_1, e_2]$ after the postprocessing, with two predecessor queries: one for $\text{pre}(v_1)$ and one for $\text{pre}(v_2)$.*

Proof. Recall that in the approach presented in Lemma 12 we compute the predecessor of $\text{pre}(x_1)$ in $D_x[e_1, e_2]$ in order to divide $D_x[e_1, e_2]$ into $\mathcal{I}_x^{x < \text{pre}(x_1)}$ and $\mathcal{I}_x^{x \geq \text{pre}(x_1)}$ and that all elements in $D_x[e_1, e_2]$ are of the form $\text{pre}(w_1)$ for a node w_1 on the path from the lowest node on e_1 to the root of B_1 .

We consider only v_1 and x_1 as the analysis for v_2 and x_2 is symmetric. We can focus on the case where $v_1 \neq x_1$, as the other case is immediate. We clearly have that $\text{pre}(v_1) \geq \text{pre}(x_1)$ as v_1 is a descendant of x_1 . Recall that in the preprocessing stage, we reordered children of every node in such a way that children connected by a light edge are before those connected by a heavy edge, so if there is a child x'_1 of x_1 on e_1 we have $\text{pre}(x_1) \leq \text{pre}(v_1) < \text{pre}(x'_1)$, as v_1 is a descendant of a light child of x_1 (by the definition of x_1). Hence, the predecessor of $\text{pre}(v_1)$ is the same as the predecessor of $\text{pre}(x_1)$ in $D_x[e_1, e_2]$. ◀

This concludes the proof of Theorem 7.

Finally, by setting the value of b to $\lceil \log^\epsilon n \rceil$ for any constant $\epsilon > 0$, we obtain a data structure using $\mathcal{O}(n \log^{2+2\epsilon} n / (\log \log^\epsilon n)^2) = \mathcal{O}(1/\epsilon^2 \cdot n \log^{2+2\epsilon} n / (\log \log n)^2)$ space capable of answering restricted HIA queries in constant time plus the time required for answering two predecessor queries: one for $\text{pre}(v_1)$ and one for $\text{pre}(v_2)$. Algorithm 1 performs $\mathcal{O}(\log n / \log b)$ restricted HIA queries in order to answer an HIA query, so the total time required is $\mathcal{O}(1/\epsilon \cdot \log n)$. However, as all the predecessor queries ask about one of two values in different lists that are related to each other, we can make use of fractional cascading.

We omit the $1/\epsilon$ factor in further sections and use the $\mathcal{O}_\epsilon(\cdot)$ notation instead to indicate a dependency on ϵ .

3.2 Fractional Cascading

For most of the cases described in the previous subsection, our structures are issuing predecessor queries. This is the only reason why the time complexity of an HIA query with our approach is not yet $\mathcal{O}_\epsilon(\log n / \log \log n)$. We will exploit the fact that all these queries look for the same target value ($\text{pre}(v_1)$ for structures built for T_1 and $\text{pre}(v_2)$ for structures for T_2) but for different pairs of branches, which enables us to use fractional cascading.

We can think of creating two catalog graphs from $T_1 \times T_2$ with nodes representing pairs (e_1, e_2) of branches, storing the contents of $D_x[e_1, e_2]$ in one catalog graph and those of $D_y[e_1, e_2]$ in the other one. The execution of Algorithm 1 can be then seen as the traversal of a path in such a graph where, for a pair $(B_1[i], B_2[j])$ of heavy trees for which a restricted HIA query is performed by the algorithm, we query the catalogs of the nodes representing the pair of branches (e_1, e_2) that contain the lowest weak ancestors of (v_1, v_2) that are in $B_1[i]$ and $B_2[j]$, respectively. The problem with this direct approach is that it is not guaranteed that the degree of all vertices in each catalog graph is polylogarithmic: we might need to move from the node corresponding to two branches (e_1, e_2) to any node corresponding to two branches (e'_1, e'_2) , where the heavy tree containing e'_1 is attached to e_1 , and there could be even $\Omega(n)$ such branches e'_1 .

We need to create catalog graphs in which the length of the considered path for each HIA query is $\mathcal{O}_\epsilon(\log n / \log \log n)$, while the degree of each node is $\mathcal{O}(\text{polylog } n)$ in order to be able to apply the result of Shi and JáJá [21]. This would ensure that all predecessor queries in $D_x[\cdot, \cdot]$ and $D_y[\cdot, \cdot]$ take constant time, apart from the first ones, which take $\mathcal{O}(\log \log n)$

time using an x -fast trie. We describe how to build the appropriate catalog graphs below. As the shape of the graph for $D_x[\cdot, \cdot]$ and $D_y[\cdot, \cdot]$ is the same and only the contents of catalogs differ, we will only describe how to build one of them.

We preprocess T_1 and T_2 separately, first to compute trees $B(T_\star)$ and then to build catalog graphs $C(T_\star)$. From this, we build the catalog graph C that can be seen as a Cartesian product of $C(T_1)$ and $C(T_2)$. More precisely, each node in C is a pair (v, w) for $v \in C(T_1)$ and $w \in C(T_2)$. For an edge between nodes v_1 and v_2 in $C(T_1)$, in the final catalog graph, we create edges between nodes (v_1, w) and (v_2, w) for each $w \in C(T_2)$. Analogically, for an edge between nodes w_1 and w_2 in $C(T_2)$, in the final catalog graph we create edges between nodes (v, w_1) and (v, w_2) for each $v \in C(T_1)$. This way, if the degrees of $C(T_1)$ and $C(T_2)$ are polylogarithmic, so is the degree of C .

We now explain how to build tree $B(T_\star)$ from T_\star . $B(T_\star)$ contains nodes representing heavy trees (called *heavy tree nodes*) and nodes representing branches (called *branch nodes*):

- for each heavy tree H , we connect all branch nodes representing branches in H as children of the heavy tree node representing H ,
- for each heavy tree H , except the tree containing the root of T_\star , we connect the heavy tree node representing H as child of the branch node representing the branch containing the parent of the root of H .

► **Proposition 14.** *The depth of $B(T_\star)$ is $\mathcal{O}_\epsilon(\log n / \log \log n)$ and each heavy tree node has $\mathcal{O}(\log^\epsilon n)$ children.*

Recall that every heavy tree has $\mathcal{O}(b) = \mathcal{O}(\log^\epsilon n)$ branches, so every heavy tree node has $\mathcal{O}(\log^\epsilon n)$ children. Any root-to-leaf path in $B(T_\star)$ alternates between heavy tree nodes and branch nodes. For any root-to- v path p in T_\star , there is a corresponding path p' in $B(T_\star)$ that visits the heavy tree nodes that correspond to the heavy trees that intersect p and the branch nodes for which Algorithm 1 (when called for a pair of nodes containing v) could call predecessor queries for $D_x[\cdot, \cdot]$ and $D_y[\cdot, \cdot]$. For any p , p' is of length $\mathcal{O}_\epsilon(\log n / \log \log n)$ and can be found in $\mathcal{O}(|p'|)$ time by following the path from the heavy tree containing v to the root of $B(T_\star)$.

We now describe how to create a catalog graph $C(T_\star)$ from $B(T_\star)$. The construction is recursive and follows from the proof of Lemma 15 applied with $d = \mathcal{O}_\epsilon(\log n / \log \log n)$ and $b = \lfloor \log^\epsilon n \rfloor$. The idea is to replace the structure of children of branch nodes having too many children with appropriate gadgets that roughly preserve the structure of the tree, do not increase the depth of the tree asymptotically and reduce the degree of each node to $\mathcal{O}(\text{polylog } n)$. Due to Proposition 14, we do not need to alter the structure of children for heavy tree nodes.

► **Lemma 15.** *For any depth- d tree $B(T_\star)$ with $\mathcal{O}(n)$ nodes and parameter b , there is a tree $C(T_\star)$ satisfying all the following conditions:*

- $C(T_\star)$ has $\mathcal{O}(n)$ nodes,
- all nodes of $C(T_\star)$ have degree $\mathcal{O}(b)$,
- $C(T_\star)$ has depth $d + \mathcal{O}(\log n / \log b)$,
- for each simple path p in $B(T_\star)$, we can compute in $\mathcal{O}(|p'|)$ time a simple path p' in $C(T_\star)$, such that p is a subsequence of p' and $|p'| \leq d + \mathcal{O}(\log n / \log b)$.

Proof. Consider a (branch) node e of $B(T_\star)$ whose children, read left-to-right, are heavy tree nodes h_1, h_2, \dots, h_k for $k > b$. We replace this subgraph that contains $k + 1$ nodes with a gadget graph whose root is e and whose set of nodes is a superset of $\{e\} \cup \{h_i : i \in [k]\}$.

For each node u , let $s(u)$ be the number of nodes in the subtree of u in the considered tree. Let $s_0 = 0$ and, for $i \geq 1$, let s_i be the prefix sum $s(h_1) + s(h_2) + \dots + s(h_i)$. If there is an integer ℓ such that $s_{i-1} < \ell \cdot s(e)/b$ and $s_i \geq \ell \cdot s(e)/b$, we mark h_i . We call each set of consecutive unmarked nodes an *interval*. As $\ell \leq b$, there are $\mathcal{O}(b)$ marked nodes and $\mathcal{O}(b)$ intervals.

We create a gadget for e (and, recursively, for some other nodes created in the construction, as described later) as follows:

- We attach as a child of e every node that is either marked or is the only element of its interval.
- For each interval of more than one node, we create a new node i_j , called an *interval node*, attach it as a child of e , and attach all the nodes of the interval as children of i_j .

We recursively apply the same construction for any of the newly created interval nodes i_1, i_2, \dots, i_m whose degree is larger than b . See Figure 3 in Appendix B for an illustration.

From the construction, it follows that the degree of each node of $C(T_\star)$ is $\mathcal{O}(b)$ and that the size of $C(T_\star)$ is $\mathcal{O}(n)$, as all new nodes are of out-degree at least 2.

Let u be a node in T_\star . We now show that the depth for a node $e_u \in C(T_\star)$ representing a branch containing u is at most $d + \mathcal{O}(\log n / \log b)$ by considering the edges above e_u in $C(T_\star)$. The edges can be of two types:

- Edges that are incident to at least one node that is not an interval node. By the depth of $B(T_\star)$, we have at most d such edges.
- Edges between interval nodes. For each such edge (v, z) , we have $s(v) \geq s(z) \cdot b$. Thus, similarly to the proof of Lemma 5, there are $\mathcal{O}(\log n / \log b)$ such edges on the path from the root of $C(T_\star)$ to e_u .

This concludes the proof of the bound on the depth of $C(T_\star)$.

Each simple path p in $B(T_\star)$ naturally corresponds to a simple path p' in $C(T_\star)$. In particular, for each edge (v, w) in $B(T_\star)$, one can explicitly store a path $C(T_\star)$ to which (v, w) corresponds. The concatenation of all such paths for edges on p yields a simple path p' in $\mathcal{O}(|p'|)$ time. As the depth of $C(T_\star)$ is $d + \mathcal{O}(\log n / \log b)$, the bound on $|p'|$ follows. ◀

From $C(T_1)$ and $C(T_2)$ constructed as in Lemma 15, we create the catalog graph C as described earlier. Only nodes that represent pairs of branches contain non-empty original catalogs. After the original catalogs are filled, we run the preprocessing of fractional cascading and appropriate augmented catalogs are created for all nodes in C as described in [9, 21]. The $\mathcal{O}_\epsilon(\log n / \log \log n)$ predecessor queries coming from restricted HIA queries performed in Algorithm 1 are naturally reduced to a constant number of queries to the x -fast tries and the traversal of a path of length $\mathcal{O}_\epsilon(\log n / \log \log n)$ in C . This takes $\mathcal{O}_\epsilon(\log n / \log \log n)$ time in total and concludes the description of our data structure and the proof of Theorem 2.

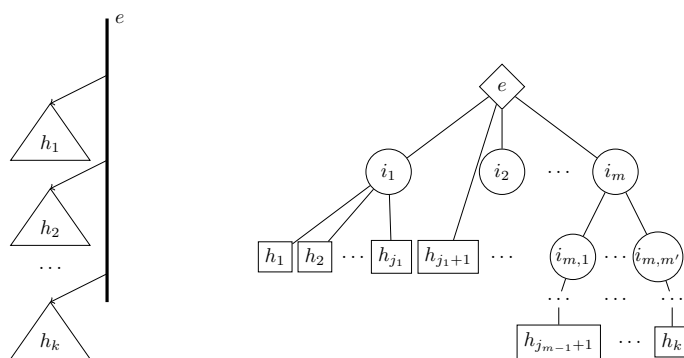
References

- 1 Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, 2022. doi:10.1007/s00453-022-00955-7.
- 2 S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings 39th Annual Symposium on Foundations of Computer Science*, pages 534–543, 1998. doi:10.1109/SFCS.1998.743504.
- 3 Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997. doi:10.1016/S0020-0190(97)00170-1.
- 4 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Proceedings*, pages 14–26, 2017. doi:10.1007/978-3-319-67428-5_2.

- 5 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.
- 6 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. doi:10.1007/s00224-013-9498-4.
- 7 Gerth Stølting Brodal, Rolf Fagerberg, Christian N. S. Pedersen, and Anna Östlin. The complexity of constructing evolutionary trees using experiments. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001*, pages 140–151, 2001. doi:10.1007/3-540-48224-5_12.
- 8 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic Longest Common Substring in Polylogarithmic Time. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*, pages 27:1–27:19, 2020. doi:10.4230/LIPIcs.ICALP.2020.27.
- 9 Bernard Chazelle and Leonidas Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, January 1986. doi:10.1007/BF01840440.
- 10 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.
- 11 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/s00453-012-9683-x.
- 12 Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.
- 13 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.
- 14 Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013*, 2013. URL: http://cccg.ca/proceedings/2013/papers/paper_29.pdf.
- 15 Davide Della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for centroid decomposition. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, pages 274–282, 2019. doi:10.1007/978-3-030-32686-9_20.
- 16 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CB09780511574931.
- 17 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- 18 Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968. doi:10.1145/321479.321481.
- 19 Mihai Pătrașcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.
- 20 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, pages 84–95, 2008. doi:10.1007/978-3-540-70575-8_8.
- 21 Qingmin Shi and Joseph JáJá. Novel transformation techniques using q-heaps with applications to computational geometry. *SIAM J. Comput.*, 34:1474–1492, January 2005. doi:10.1137/S0097539703435728.
- 22 Peter Weiner. Linear pattern matching algorithms. In *14th FOCS*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.
- 23 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:doi.org/10.1016/0020-0190(83)90075-3.

A Description of the $\mathcal{O}(\log n)$ -Query-Time Data Structure of [14]

As mentioned in the introduction, Gagie et al. [14] sketched an $\mathcal{O}(n \log^2 n)$ -size data structure that answers HIA queries in $\mathcal{O}(\log n)$ time, in the last paragraph of Subsection 2.1 of their work. They construct a data structure for each pair of heavy trees of T_1 and T_2 and reduce an HIA query for nodes u and v to a predecessor query in the data structure of each of $\mathcal{O}(\log n)$ pairs of heavy trees. The idea for improving the (fully-described) $\mathcal{O}(\log n \log \log n)$ -time procedure for answering queries with a more efficient one is similar to ours and involves fractional cascading. They would need to build a catalog graph with nodes being pairs of heavy paths, reduce its degree, and make sure that the predecessor queries have the same target (by asking for the preorder numbers of u and v). This is similar to what we describe in Section 3.2 for a different tree decomposition.



B Omitted Figure from Section 3.2

■ **Figure 3** On the left there is a branch e of T_* , with outgoing edges to h_1, h_2, \dots, h_k in $B(T_*)$. On the right, there is a catalog graph gadget created for e , which is part of $C(T_*)$. Circles denote interval nodes and rectangles heavy tree nodes. Some of the intervals are recursively replaced with the gadget to decrease their degree. Heavy tree nodes have degree $\mathcal{O}(b)$. In $C(T_*)$, e 's parent is the heavy tree to which it belongs, while the children of each h_i are the branches in h_i .



From Bit-Parallelism to Quantum String Matching for Labelled Graphs

Massimo Equi  

Department of Computer Science, University of Helsinki, Finland

Arianne Meijer-van de Griend  

Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen  

Department of Computer Science, University of Helsinki, Finland

Abstract

Many problems that can be solved in quadratic time have bit-parallel speed-ups with factor w , where w is the computer word size. A classic example is computing the edit distance of two strings of length n , which can be solved in $O(n^2/w)$ time. In a reasonable classical model of computation, one can assume $w = \Theta(\log n)$, and obtaining significantly better speed-ups is unlikely in the light of conditional lower bounds obtained for such problems.

In this paper, we study the connection of bit-parallelism to quantum computation, aiming to see if a bit-parallel algorithm could be converted to a quantum algorithm with better than logarithmic speed-up. We focus on *string matching in labeled graphs*, the problem of finding an exact occurrence of a string as the label of a path in a graph. This problem admits a quadratic conditional lower bound under a very restricted class of graphs (Equi et al. ICALP 2019), stating that no algorithm in the classical model of computation can solve the problem in time $O(|P||E|^{1-\epsilon})$ or $O(|P|^{1-\epsilon}|E|)$. We show that a simple bit-parallel algorithm on such restricted family of graphs (level DAGs) can indeed be converted into a realistic quantum algorithm that attains subquadratic time complexity $O(|E|\sqrt{|P|})$.

2012 ACM Subject Classification Theory of computation \rightarrow Quantum computation theory; Theory of computation \rightarrow Parallel algorithms; Theory of computation \rightarrow Pattern matching; Theory of computation \rightarrow Graph algorithms analysis

Keywords and phrases Bit-parallelism, quantum computation, string matching, level DAGs

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.9

Related Version *Preprint Version*: <https://arxiv.org/abs/2302.02848>

Funding *Massimo Equi*: Partially funded by the European Union’s Horizon 2020 research and innovation programme under the European Research Council (ERC) grant agreement No. 851093 (SAFE BIO) and by the Helsinki Institute for Information Technology (HIIT).

1 Introduction

Exact string matching problem is to decide if a pattern string P appears as a substring of a text string T . In the classical models of computation, this problem can be solved in $O(|P| + |T|)$ time [10]. Different quantum algorithms for this basic problem have been developed [13, 14, 16], resulting into different solutions, the best of which finds a match in $O(\sqrt{|T|}(\log^2 |T| + \log |P|))$ time [13] with high probability. These assume the pattern and text are stored in quantum registers, requiring thus $O(|P| + |T|)$ qubits to function. Moreover, these approaches may rely on applying a linear number of quantum gates in parallel on different qubits. For example, Niroula and Nam [13] perform $O(\log(|T|))$ rounds of parallel swaps, executing $O(|T|)$ swaps in parallel per round.



© Massimo Equi, Arianne Meijer-van de Griend, and Veli Mäkinen;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 9; pp. 9:1–9:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the classical models of computation, an analogy for these assumptions is to assume that the text has been preprocessed for subsequent queries. For example, one can build a Burrows-Wheeler transform -based index structure for the text in time $O(|T|)$ [4], assuming $T \in \{1, 2, \dots, \sigma\}^*$, where $\sigma \leq |T|$. Then, one can query the pattern from the index in $O(|P| \log \log \sigma)$ time [4, Theorem 6.2]. In this light, quantum models can offer only limited benefit over the classical models for exact string matching.

Motivated by this difficulty in improving linear-time solvable problems using quantum approaches, let us consider problems known to be solved in quadratic time. For example, approximate string matching problem is such a problem: decide if a pattern string P is within edit distance k from a substring of a text string T , where edit distance is the number of single symbol insertions, deletions, and substitution needed to convert a string to another. This problem can be solved using *bit-parallelism* in $O(\lceil |P|/w \rceil |T|)$ time [11], under the Random Access Memory (RAM) model with computer word size w . A reasonable assumption is that $w = \Theta(\log |T|)$, so that this model reflects the capacity of classical computers. Thus, when $|P| = |T| = n$, this bit-parallel algorithm for approximate string matching takes time at least $\Omega(n^{2-\epsilon})$ for all $\epsilon > 0$, as $\log n = o(n^\epsilon)$ for all $\epsilon > 0$. It is believed that this quadratic bound cannot be significantly improved, as there is a matching conditional lower bound saying that if approximate pattern matching could be solved in time $O(n^{2-\epsilon})$ with some $\epsilon > 0$, then the Orthogonal Vector Hypothesis (OVH) and thus the Strong Exponential Time Hypothesis (SETH) would not hold [2]. As these hypotheses are about classical models of computation, it is natural to ask if the quadratic barrier could be broken with quantum computation.

In this quest for breaking the quadratic barrier, we study another problem with a bit-parallel solution and a conditional lower bound. Consider exact pattern matching on a graph, that is, consider deciding if a pattern string $P \in \Sigma^*$ equals a labeled path in a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Here we assume the nodes v of the graph are labeled by $\ell(v) \in \Sigma$ and a path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_t$, $(v_i, v_{i+1}) \in E$ for $1 \leq i < t$, spells string $\ell(v_1)\ell(v_2)\dots\ell(v_t)$. There is an OVH lower bound conditionally refuting an $O(|P||E|^{1-\epsilon})$ or $O(|P|^{1-\epsilon}|E|)$ time solution [7]. This conditional lower bound holds even if graph G is a *level DAG*: for every two nodes u and v , holds the property that every path from u to v has the same length. On DAGs, this string matching on labeled graphs (SMLG) problem can be solved in $O(\lceil |P|/w \rceil |E|)$ time [15] in the bit-parallel model, so the status of this problem is identical to that of approximate pattern matching on strings. However, the simplicity of the bit-parallel solution for SMLG on level DAGs enables a connection to quantum computation. We consider a specific model of quantum computation, the Quantum Random Access Memory (QRAM) model [8], in which we have access to “quantum arrays”, and we assume that integer values like $|P|$, $|V|$ or $|E|$ fit into a (quantum) memory word. Under this model, we turn the bit-parallel solution into a quantum algorithm that solves SMLG on level DAGs with high probability in $O(|E|\sqrt{|P|})$ time, breaking through the classical quadratic conditional lower bound.

Classical conditional lower bounds are not new to be broken by quantum computing. For example, the quadratic Orthogonal Vectors problem itself can be solved in subquadratic time (linear using QRAM) using quantum computing. This is not the only problem to have a better-than-quadratic solution in the quantum realm [17]. Nevertheless, to the best of our knowledge, we are the first to propose a subquadratic time algorithm for SMLG, even if restricted to a specific class of graphs. Moreover, the translation of a bit-parallel strategy to a quantum-parallel one is an original technique, and we are not aware of any other work utilising it.

An earlier work [6] provided a quantum algorithm solving SMLG in time $O(\sqrt{|V||E|}|P|)$. When the graph is non-sparse, that is $|V| = O(\sqrt{|E|})$, the time complexity becomes $O(|E|^{\frac{3}{4}}|P|)$, which is an improvement over classical algorithms. We offer a different kind of trade-off, limiting ourselves to a special class of graphs, but obtaining a better time complexity. We also note that, even if no subquadratic classical algorithm exists for non-sparse graphs, the existing classical reduction from OV [7] produces a sparse level DAG, for which our quantum algorithm runs in subquadratic time.

As mentioned above, in some previous works [13, 14, 16] (and references in [16]) algorithms have been proposed to solve string matching in plain text in the QRAM model, under the assumption that a large number of quantum gates, possibly linear, can be applied in parallel when acting on different qubits. We find this assumption to be too restrictive, as even the classical RAM model does not adopt it, since in such a model of computation many operations would become trivial. Instead, our algorithm works without the need for such an assumption.

The paper is structured as follows. We revisit exact pattern matching and derive a simple quantum algorithm for it, in order to introduce the quantum machinery. Then we give a brute-force quantum algorithm for SMLG, which we later improve on level DAGs. This improvement is based on extending the Shift-And algorithm [3], whose quantum version we extend for level DAGs.

In what follows, we assume the reader is familiar with the basic notions in quantum computing as covered in textbooks [12].

2 Preliminaries

An *alphabet* Σ is a set of *characters*. Throughout the paper we assume Σ is ordered, i.e., for each $a, b \in \Sigma$ we can decide if $a < b$. A sequence $P \in \Sigma^n$ is called a *string* and its length is denoted $n = |P|$. We denote integers $i, i + 1, \dots, j$ as interval $[i..j]$ and represent a string P as an array $P[0..n - 1]$, where $P[i] \in \Sigma$ for $0 \leq i \leq n - 1$, as in this work all indexes start from 0. String $P[i..j]$ is called a *substring* and string $P[0..i]$ a *prefix* of P . With bit-vectors discussed next, we use 0-based indexing.

Let B be a w -bit integer interpreted as string $B[0..w - 1]$ from alphabet $\{0, 1\}$ such that $B = \sum_{i=0}^{w-1} B[i] \cdot 2^i$. We call B a *bit-vector*. Given two bit-vectors B and C , we define the following Boolean operations $A = B \wedge C$, $O = B \vee C$, and $N = \neg B$ as follows: $A[i] = 1$ iff $B[i] = C[i] = 1$, $O[i] = 1$ iff $B[i] = 1$ or $C[i] = 1$, and $N[i] = 1$ iff $B[i] = 0$. When bit-vector content is visualized, we list the most significant bit first, i.e., $B[w - 1]B[w - 2] \dots B[0]$. With this in mind, we define the left-shifts $L = B \ll k$ and right-shifts $R = B \gg k$ as follows: $L[i + k] = B[i]$ and $R[i] = B[i + k]$. Here values out of the domain of the bit-vectors are assumed to be 0. Logarithms are assumed to be in base two: $\log n = \log_2 n$.

In *directed labelled graph* (DAG) $G = (V, E, \ell)$, V is the set of nodes, E is the sets of vertices, and $\ell : V \rightarrow \Sigma$ is a labelling function that assigns a character of the alphabet to each node. We assume the nodes to be indexed as v_0, v_1, \dots, v_{n-1} in topological order, where $n = |V|$. For $v_i \in V$, $\ell(v_i)$ is its label. Set of nodes $in(v_i) = \{j \mid (v_j, v_i) \in E\}$ contains the indexes of the in-neighbours of v_i , and $D_i = |in(v_i)|$ is the in-degree of v_i . If, for $0 \leq d \leq D_i - 1$, v_k is the d -th in-neighbour of v_i according to the topological indexing that we defined above, we express this fact using notation $k = in_i(d)$, where $in_i : [0, D_i - 1] \rightarrow [0, n - 1]$.

In this work, we study the problem of *string matching in labelled graphs*, that consists in finding a match for a pattern string $P[0..m-1]$ in a labelled graphs G over alphabet Σ , where P has a *match* in G if there is a path v_1, \dots, v_k such that $P = \ell(v_1) \cdots \ell(v_k)$ (we also say that P *occurs* in G , and that v_1, \dots, v_k is an *occurrence* of P). Notice that if $|P| = 1$, a classic visit of the graph solves the problem in linear time, thus we always assume $|P| \geq 2$.

► **Problem 1** (String Matching in Labeled Graphs (SMLG)).

INPUT: A labeled graph $G = (V, E, L)$ and a pattern string P , both over an alphabet Σ .

OUTPUT: True if and only if there is at least one occurrence of P in G .

3 Quantum Notation and Preliminaries

In quantum computing, data is represented in quantum bits (qubits), the quantum analogue to classical bits. A qubit can be in two states, denoted as $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ but, unlike a classical bit, it can also be a linear combination of the two states, a *superposition*: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. The complex values α and β are called the *amplitudes* of $|\psi\rangle$. Measuring a qubit in superposition will result in either $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. Note that this notation can easily be generalised to integer states $|n\rangle$ using the tensor product between the quantum states of the binary representation on n : $|n\rangle = \bigotimes_{i \in \text{binary}(n)} |i\rangle$, and in this case we use the term *quantum register*. Throughout the paper, we will use notation $|q\rangle_Q$ to denote that qubit Q is in state $|q\rangle$. We use lower case letters for quantum states and capital letters for qubits.

In this work, we mainly use the NOT gate X , the controlled NOT CX , and the Toffoli gate CCX . We also apply an OR gate, that computes a logical *or* between two qubits and stores the results in a third qubit. This can easily be obtained with a simple combination of X gates with a Toffoli gate.

Furthermore, to define some quantum states, we use Kronecker's delta function $\delta_{x,y}$, which is $\delta_{x,y} = 1$ if $x = y$ and $\delta_{x,y} = 0$ otherwise. Given superposition $|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |i\rangle_I |\delta_{c,i}\rangle_Q$, the delta function specifies that qubit Q is in state $|1\rangle$ iff $i = c$, as in the following example

$$\sum_{i=0}^3 \alpha_i |i\rangle_I |\delta_{0,i}\rangle_Q = \alpha_0 |0\rangle_I |1\rangle_Q + \alpha_1 |1\rangle_I |0\rangle_Q + \alpha_2 |2\rangle_I |0\rangle_Q + \alpha_3 |3\rangle_I |0\rangle_Q$$

where I is a quantum register of at least two qubits.

We assume to have a *quantum random access memory* (QRAM) able to use a quantum register as an index to access classical data. Let m_0, m_1, \dots, m_{n-1} be the data stored in QRAM M . Given quantum register I , the operation that reads data from M into quantum register Q initialized to $|0\rangle$ using I as index is defined as follows [8]:

$$\sum_{i=0}^{n-1} \alpha_i |i\rangle_I |0\rangle_Q \xrightarrow{\text{QRAM read}} \sum_{i=1}^n \alpha_i |i\rangle_I |0 \oplus m_i\rangle_Q = \sum_{i=1}^n \alpha_i |i\rangle_I |m_i\rangle_Q.$$

Notice that this is a unitary operation, and thus reading the same data into the same register twice will reset such a register to the value it had before performing the reading operation. In terms of time complexity, the execution of the read operation is proportional to the number of qubits in quantum register I . Under the Word-QRAM model with memory-word size $O(\log n)$ for inputs of size n , we can assume to be able to perform a QRAM read operation in $O(1)$, because $O(\log n)$ qubits are enough for register I to index an input of size n . Indeed, this reflect the same assumption of the classical Word-RAM model, where operations on memory words are assumed to be constant.

4 String Matching in Plain Text

A quantum computer, with access to QRAM, can solve the problem of finding an exact match for a pattern string P into a text string T in time $O(|P|\sqrt{|T|})$, with high probability. We explain a simple solution to this problem. Let $|T| = n$ and $|P| = m$, then $T = t_0 t_1 \cdots t_{n-1}$ and $P = p_0 p_1 \cdots p_{m-1}$ are two strings defined over a binary alphabet, that is $t_i, p_j \in \{0, 1\}$ for $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$. We use qubits C_T and C_P initialized to $|0\rangle$ to track the current characters of T and P , and we assume to have the text and the pattern stored in qubits in the following way:

$$|0\rangle_{C_T} |0\rangle_{C_P} |t_0\rangle_{T_0} |t_1\rangle_{T_1} \cdots |t_{n-1}\rangle_{T_{n-1}} |p_0\rangle_{P_0} |p_1\rangle_{P_1} \cdots |p_{m-1}\rangle_{P_{m-1}}.$$

We also use auxiliary qubits $A_{-1}, A_0, A_1 \cdots A_{m-1}$, and quantum registers I, J , and Q , all three of $\log n$ qubits. We initialize A_{-1} and Q to $|1\rangle$, while $A_0, A_1 \cdots A_{m-1}, I$ and J are all initialized to $|0\rangle$. We prepare quantum register I in an equally balanced superposition spanning all the text positions, that is $|0\rangle_I \rightarrow 1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I$, assuming n to be a power of 2, without loss of generality. If this is not the case, we generate a superposition as large as the first power of two greater than n , then standard techniques can be applied to handle the additional substates, as explained in Appendix A.

Each individual state $|i\rangle$ in the superposition represents a computation starting at position i in the text. In each of these computations, we scan $T[i..i+m-1]$ and try to match each character with $P[0..m-1]$, storing the intermediate results of such comparisons in registers A_0, A_1, \dots, A_{m-1} . More precisely, at iteration j , $0 \leq j \leq m-1$, we compute a logical *xor* between t_{i+j} and p_j storing the result in C_P via a CX gate with control C_T and target C_P . Then, we apply a X gate to C_P , which now stores $|\neg(t_{i+j} \oplus p_j)\rangle_{C_P} = |t_{i+j} = p_j\rangle_{C_P}$. At this point, we apply a Toffoli gate with controls C_P and A_{j-1} , storing the value in target qubit A_j . We now reset C_T and C_P to $|0\rangle$ by applying to them the same gates again, but in reverse order. As last step in iteration j , we increase both I and J by 1 by performing transformation $1/\sqrt{n} \sum_{i=0}^{n-1} |1\rangle_Q |i\rangle_I |j\rangle_J \rightarrow 1/\sqrt{n} \sum_{i=0}^{n-1} |1\rangle_Q |i+1\rangle_I |j+1\rangle_J$ (this of course requires two separate addition operations), where the addition is intended to be modulo 2^n . This allows us to read the next character of the pattern at the next iteration.

After the last iteration, we can run Grover's operator [9] where the marked items are represented by $|a_{m-1,i}\rangle_{A_{m-1}} = |1\rangle$, and then measure register $|I\rangle$ to locate the ending position of a match. Of course, we do not know the exact number of marked items, and we address this issue by guessing the number of items and rerunning the whole algorithm a constant number of times. We illustrate the entire procedure in Algorithm 1.

The algorithm is correct because, after each iteration of the *for* loop, we correctly keep track of the positions of the text that are active matches for the current prefix of the pattern.

► **Lemma 1.** *After iteration j of the for loop of Algorithm 1, let qubits I and A_j be in superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |a_{j,i}\rangle_{A_j}$. Then, $|a_{j,i}\rangle_{A_j} = |1\rangle$ if and only if $T[i..i+j] = P[0..j]$, where $0 \leq j \leq m-1$ and $0 \leq i \leq n-1$.*

Proof. At iteration 0, after applying gates CX and X , C_P stores $|\neg(T[i] \oplus P[0])\rangle_{C_P}$ and A_{-1} stores $|1\rangle_{A_{-1}}$, thus the Toffoli gate simply copies value $\neg(T[i] \oplus P[0])$ to A_0 . Because we are working with a binary alphabet, $\neg(T[i] \oplus P[0])$ equals $T[i] = P[0]$, and thus we obtain superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |T[i] = P[0]\rangle_{A_0}$.

$$\begin{array}{c}
I \quad C_T \quad C_P \quad A_0 \quad A_1 \\
\frac{1}{2\sqrt{2}} |1\rangle |B\rangle |A\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |2\rangle |A\rangle |A\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |3\rangle |A\rangle |A\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |4\rangle |A\rangle |A\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |5\rangle |B\rangle |A\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |6\rangle |B\rangle |A\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |7\rangle |A\rangle |A\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |0\rangle |B\rangle |A\rangle |1\rangle |0\rangle
\end{array}
\longrightarrow
\begin{array}{c}
I \quad C_T \quad C_P \quad A_0 \quad A_1 \\
\frac{1}{2\sqrt{2}} |1\rangle |B\rangle |0\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |2\rangle |A\rangle |1\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |3\rangle |A\rangle |1\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |4\rangle |A\rangle |1\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |5\rangle |B\rangle |0\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |6\rangle |B\rangle |0\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |7\rangle |A\rangle |1\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |0\rangle |B\rangle |0\rangle |1\rangle |0\rangle
\end{array}
\longrightarrow
\begin{array}{c}
I \quad C_T \quad C_P \quad A_0 \quad A_1 \\
\frac{1}{2\sqrt{2}} |1\rangle |B\rangle |0\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |2\rangle |A\rangle |1\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |3\rangle |A\rangle |1\rangle |1\rangle |1\rangle + \\
\frac{1}{2\sqrt{2}} |4\rangle |A\rangle |1\rangle |1\rangle |1\rangle + \\
\frac{1}{2\sqrt{2}} |5\rangle |B\rangle |0\rangle |1\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |6\rangle |B\rangle |0\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |7\rangle |A\rangle |1\rangle |0\rangle |0\rangle + \\
\frac{1}{2\sqrt{2}} |0\rangle |B\rangle |0\rangle |1\rangle |0\rangle
\end{array}$$

■ **Figure 1** An example of the evolution of the superposition after one iteration of Algorithm 1. The first arrow represents the application of a CX gate with control T_1 and target P_1 , and the application of a X gate on P_1 . The second arrow represents the application of a Toffoli gate with controls P_1 and A_0 , and target A_1 . Intuitively, in the first step we are checking that $T[i+j] = P[j]$; in the second step we combine the result of this check with the contribution of the previous iteration(s). Characters A and B are to be considered binary values.

At iteration j , we assume by induction that register A_{j-1} stores $|a_{j-1,i}\rangle_{A_{j-1}} = |1\rangle$ if and only if $T[i \dots i+j-1] = P[0 \dots j-1]$. Gates CX and X compute $\neg(T[i+j] \oplus P[j])$ storing it in C_P . We then apply the Toffoli gate with controls C_P and A_{j-1} , and target A_j , obtaining superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |a_{j-1,i} \wedge T[i+j] = P[j]\rangle_{A_j}$. Thus, $|a_{j,i}\rangle_{A_j} = |a_{j-1,i} \wedge T[i+j] = P[j]\rangle_{A_j}$ is $|1\rangle$ if and only if $T[i \dots i+j] = P[0 \dots j]$. ◀

As mentioned above, we have to be careful in running Grover's search algorithm at the end of Algorithm 1. We defer these details to the full proof of Theorem 5 given in Appendix C. For now, we assume that we are able to retrieve with arbitrarily high probability $1 - (7/8)^c$ a marked substate representing a match. Combining this with Lemma 1, we obtain the claimed result.

► **Theorem 2.** *Given a text string T , pattern string P and integer $c > 0$, Algorithm 1 finds a match for P in T in time $O(c(|P|\sqrt{|T|}))$. If there is no match, the algorithm returns a negative answer with probability $p = 1$. If there is at least one match, the algorithm returns the index of the last position of a match with probability $p > 1 - (7/8)^c$.*

Proof. For the correctness, consider Lemma 1 where $j = m = |P|$, which is the number of times we run the *for* loop. In this case, $|a_{|P|,i}\rangle_{A_{|P|}} = |1\rangle$ if and only if $T[i \dots i+|P|-1] = P[0 \dots |P|-1]$. Thus, measuring these substates yields a correct solutions. The details of how to perform such a measurement respecting the time complexity and probability of success are deferred to the full proof of Theorem 5 in Appendix C. ◀

5 String Matching in Labeled Graphs

5.1 Quantum Brute-force Algorithm for SMLG

In SMLG we are given pattern string P with characters in alphabet Σ and a node-labeled graph $G = (V, E)$, with labelling function $\ell : V \rightarrow \Sigma$. We are asked to find a path (or, actually, a walk) $\pi = v_1, v_2, \dots, v_{|P|}$ in G such that $\ell(v_1) \circ \ell(v_2) \circ \dots \circ \ell(v_{|P|}) = P$, where \circ denotes string concatenation.

One could try to obtain a quantum algorithm for SMLG by generalizing the idea we presented for plain text. The idea would be to list all possible paths of length $|P|$ in the graph, and then mark those ones that are actual matches for P . Unfortunately, the superposition

■ **Algorithm 1** An algorithm for solving exact string matching in plain text that, using QRAM, achieves $O(|P|\sqrt{|T|})$ time complexity. The details of how to handle Grover's search at the end are given in Theorem 5, whose full proof is deferred to Appendix C.

Input: Text T stored as $|t_0\rangle_{T_0} |t_1\rangle_{T_1} \cdots |t_{n-1}\rangle_{T_{n-1}}$, pattern string P stored as $|p_0\rangle_{P_0} |p_1\rangle_{P_1} \cdots |p_{m-1}\rangle_{P_{m-1}}$, integer c

Output: A position of T where a match for P ends, if any

- 1 **for** c times **do**
- 2 Initialize quantum registers $I, J, A_0, A_1 \cdots A_{m-1}$ as $|0\rangle_I |0\rangle_J |0\rangle_{A_0} |0\rangle_{A_1} \cdots |0\rangle_{A_{m-1}}$;
- 3 Initialize quantum register A_{-1} and Q as $|1\rangle_{A_{-1}}$ and $|1\rangle_Q$;
- // Apply $H^{\otimes \log n}$ to register I
- 4 $|0\rangle_I \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I$;
- 5 **for** m times **do**
- // Read $T[i]$ in C_T and $P[j]$ in C_P using registers I and J as indexes
- 6 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |0\rangle_{C_T} |0\rangle_{C_P} \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i\rangle_{C_T} |p_j\rangle_{C_P}$;
- // Apply CX with control C_T and target C_P
- 7 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |p_j\rangle_{C_P} \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |t_i \oplus p_j\rangle_{C_P}$;
- // Apply X to C_P
- 8 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i \oplus p_j\rangle_{C_P} \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |\neg(t_i \oplus p_j)\rangle_{C_P} = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P}$;
- // Apply Toffoli with controls C_P and A_{j-1} , and target A_j
- 9 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P} |a_{j-1}\rangle_{A_{j-1}} |0\rangle_{A_j} \rightarrow$
 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P} |a_{j-1}\rangle_{A_{j-1}} |(t_i = p_j) \wedge a_{j-1}\rangle_{A_j}$;
- // Reset C_T and C_P to $|0\rangle$ via uncomputation
- 10 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |\neg(t_i \oplus p_j)\rangle_{C_P} \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i \oplus p_j\rangle_{C_P}$;
- 11 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |t_i \oplus p_j\rangle_{C_P} \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |p_j\rangle_{C_P}$;
- 12 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i\rangle_{C_T} |p_j\rangle_{C_P} \rightarrow$
 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i \oplus t_i\rangle_{C_T} |p_j \oplus p_j\rangle_{C_P} = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |0\rangle_{C_T} |0\rangle_{C_P}$;
- // Increment indexes I and J
- 13 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |1\rangle_Q |i\rangle_I |j\rangle_J \rightarrow \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |1\rangle_Q |i \oplus 1\rangle_I |j + 1\rangle_J$;
- 14 Apply gate Z to qubit R_{n-1} , so that the sign of the amplitude is flipped if $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$;
- 15 Choose $K \in [0, |P|]$ uniformly at random;
- 16 Run Grover's iterate operator the optimal number of times assuming to have K solutions, with the oracle function being lines 5–14 of this algorithm;
- 17 Measure R_{n-1} into classical register R_{cl} ;
- 18 **if** $R_{cl} = 1$ **then**
- 19 | Measure I into classical register I_{cl} and **return** I_{cl}
- 20 **return** no

would be as large as there are paths of length $|P|$, and thus the overall time complexity would be $O(|P|\sqrt{|V|^{|P|}})$. Moreover, an adjacency matrix would be needed to check the existence of edges between nodes in constant time, yielding a space complexity of $O(|V|^2)$ qubits. We conclude that more involved techniques are needed.

5.2 The Classical Shift-And Algorithm

We first introduce the classical *shift-and* algorithm [3] for matching a pattern against a text and generalize it to work on graphs. Then, we show how the bit-vector data structure of that algorithm can be represented as a superposition of a logarithmic number of qubits. This approach allows us to achieve better performances than the brute force algorithm.

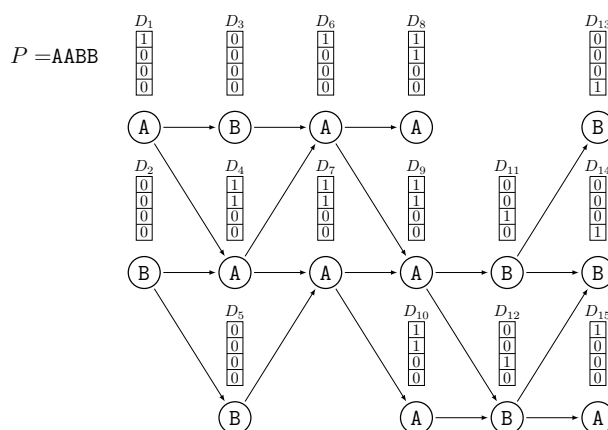
In the shift-and algorithm, we use bit vector B of the same length of pattern P to represent which of its prefixes are matching the text during the computation. Assuming integer-alphabet Σ , we also initialize bidimensional array M of size $|P| \times |\Sigma|$ so that $M[j][c] = 1$ if and only if $P[j] = c$, and $M[j][c] = 0$ otherwise. The algorithm starts by initializing vector B to zero and array M as specified above. Then, we scan whole text T performing the next four operations for each $T[i]$, $i \in [0, n - 1]$, where $M[*][c]$ represents the c -th column of M :

1. $B \leftarrow B + 1$;
2. $B \leftarrow B \wedge M[*][T[i]]$;
3. if $B[m - 1] = 1$, return *yes*;
4. $B \leftarrow B \ll 1$.

Operation 1 sets the least significant bit of B to 1, which is needed to test $P[0]$ against $T[i]$. Operation 2 computes a bit-wise *and* between B and the column of M corresponding to character $T[i]$. Remember that $M[j][T[i]] = 1$ means $P[j] = T[i]$, thus this operation leaves each bit $B[j]$ set to 1 if and only if it was already set to 1 before this step and the j -th character of the pattern matches the current character of the text. At this point, if bit $B[m - 1]$ is set to 1 we have found a match for P , and Operation 3 will return *yes*. For the other positions, if bit $B[j]$ is set to 1, then we know that prefix $P[0..j]$ matches $T[i - j + 1..i]$, and Operation 4 shifts the bits in B by one position, so that in the next iteration we will check whether $P[j + 1]$ matches $T[i + 1]$.

In labeled DAG $G = (V, E)$, each node $v_i \in V$ has a single-character label $\ell(v_i)$. We generalize the shift-and algorithm to labeled DAGs by computing a bit-vector B_i for each node $v \in V$, initializing them to zero. Consider a BFS visit of DAG G . When visiting node v_i , each bit-vector B_k of its in-neighbour $v_k \in in(v_i)$ represents a set of prefixes of P matching a path in the graph ending at v_k . Thus, we merge all of this information together by taking the bit-wise *or* of all of the in-neighbours of v_i , that is we replace Operation 1 with $B_i \leftarrow 1 + \bigvee_{v_k \in in(v_i)} B_k$. Operations 2, 3 and 4 are performed as before. An example of the state of the data structures after the execution of the algorithm is shown in Figure 2, and the body of the iteration now is:

1. $B_i \leftarrow 1 + \bigvee_{v_k \in in(v)} B_k$;
2. $B_i \leftarrow B_i \wedge M[*][T[i]]$;
3. if $B_i[m - 1] = 1$, return *yes*;
4. $B_i \leftarrow B_i \ll 1$.



■ **Figure 2** The adaptation of the classical algorithm for matching pattern P in level DAG G . Each bit-vector D_v represent the result after the merging of the bit-vectors of the in-neighbours of v and before the shifting.

5.3 Quantum Bit-Parallel Algorithm for Level DAGs

We make the classic techniques work in a quantum setting for a special class of DAGs, which we call *level DAGs*. A level DAG is a DAG such that, for every two nodes v and w , every path from v to w has the same length, as for the DAG in Figure 2. We also note that *degenerate strings* [1] can be represented as level DAGs. We use a function representing in-neighbours:

$$in_i(d) = \text{index of the } d\text{-th in-neighbour of } v_i$$

Our approach aims to represent each bit vector B_i with a single qubit V_i set up in a proper superposition, and translate the bit-wise operations to parallel operations across such superposition. In the algorithm, we use the following qubits and quantum registers. Quantum registers I and J store the index of a node and the position in the pattern, respectively. Qubit V_i represents, in superposition, the bit-vector of the node v_i , and qubit $E_{i,d}$ stores the contribution of edge $(v_{in_i(d)}, v_i) \in E$ in the update of qubit V_i , for, $0 \leq i \leq n - 1$, $0 \leq d \leq D_i - 1$ and $D_i = indeg(v_i)$. Quantum register C stores label $\ell(v_i)$ of the node in the current iteration, and is used to fetch the content of the corresponding matrix column, which we will store in qubit M . Occurrences of the pattern encountered during the execution of the algorithm are stored in qubit R_i . Qubits V'_i and R'_i are auxiliary qubits used to store intermediate results, and we also use auxiliary qubits A and B and auxiliary quantum register Q to implement necessary operations. Moreover, we assume to have access to QRAM.

5.3.1 The algorithm

Assume all the quantum registers and qubits to be initialized to $|0\rangle$, except Q initialized to $|1\rangle$. The algorithm starts by setting quantum register J in a balanced superposition, by applying the Hadamard gate on each one of its qubits. Then, we initialize qubits A so that $|a_j\rangle_A = |1\rangle$ for $j = 0$, and $|a_j\rangle_A = |0\rangle$ otherwise. We do the same with qubit B , with the difference that $|b_j\rangle_A = |1\rangle$ for $j = m - 1$, and $|b_j\rangle_A = |0\rangle$ otherwise. We can do these operations with two applications of a generalized Toffoli gate, using register J as control and qubits A and then B as targets. In the case of qubit A , we first apply an X gate to every qubit of register J , we then apply the Toffoli gate, and finally we undo the applications of

the X gate. The generalised Toffoli has a cost proportional to the number of qubits in J , that is logarithmic in the size of the input, and because this is an operation between a single quantum register and a qubit, we can assume it to be constant in the Word-QRAM model. We then initialize the qubits representing the bit-vectors of the nodes at level 0. This is done with the same operations described below for the main loop, the only difference being that these nodes do not have in-neighbours and thus we can simplify some operations. Specifically, we load each entry of the character matrix in superposition and we use it and qubit A as controls of a Toffoli gate which thus flips to $|1\rangle$ sub-state $|v_{i,j}\rangle_{V_i}$ if $P[0]$ matches $\ell(v_i)$.

The rest of the algorithm maintains almost the same overall structure, with the exception of one necessary adaptation. In a DAG of L levels where L_l is the set of nodes at level l , for $0 \leq l \leq L - 1$, we iterate over them one at the time, and for each level we process its nodes one after the other. As we will better explain later, we wait before applying the quantum equivalent of the shift operation once we scanned the whole level, not after processing every node. The overall idea is to translate the classical bit-parallel operations into analogous quantum operations that work across the superposition. This translation of bit-parallelism to superposition parallelism is the core of our technique, and we now describe how to apply it to each operation. The pseudocode of the entire procedure is given in Algorithm 1, where all the arithmetic operations are to be considered modulo $2^{|P|}$. We only omit the pseudocode for procedures `SourceNodesInit()`, `IncreaseI()` and `IncreaseJ()`, which is to be found in Appendix B. We also assume $|P|$ to be a power of two. If this is not the case, we generate a superposition as large as the first power of two greater than $|P|$, then standard techniques can be used to handle the additional substates, as explained in Appendix A.

Operation 1 (line 10) can be broken down into two simpler operations: computing the bit-wise *or* and adding 1. In our translation to quantum computing, each sub-state of superposition $\sum_{j=0}^{m-1} |j\rangle_J |v_{i,j}\rangle_{V_i}$ represents an entry of the classical bit-vector used in the Shift-And algorithm. Thus, what was a bit-wise *or* is now easily translated into the application of few quantum gates. Notice that, to compute the logical *or* between two generic qubits P and Q and store the result in qubit R , we can follow De Morgan's rules and apply an X gate to both P and Q , apply a Toffoli gate with controls P and Q and target R , apply an X gate to R , and finally apply an X gate to P and Q again to restore their initial values. In our case, at iteration i , we use qubit $E_{i,d}$ to store the *or* computed among the first $d + 1$ in-neighbours $v_{in_i(0)}, \dots, v_{in_i(d)}$ of node v_i , and we compute it in the following way. Let

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(0),j}\rangle_{V_{in_i(0)}} |v_{in_i(1),j}\rangle_{V_{in_i(1)}} \cdots |v_{in_i(d-1),j}\rangle_{V_{in_i(d-1)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}}$$

be such that

$$e_{i,d-1,j} = v_{in_i(0),j} \vee v_{in_i(1),j} \cdots \vee v_{in_i(d-1),j}.$$

We compute the value of $E_{i,d}$ from $E_{i,d-1}$ and $V_{in_i(d)}$ as

$$\begin{aligned} & \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(d),j}\rangle_{V_{in_i(d)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |0\rangle_{E_{i,d}} \rightarrow \\ & \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(d),j}\rangle_{V_{in_i(d)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |v_{in_i(d),j} \vee e_{i,d-1,j}\rangle_{E_{i,d}}. \end{aligned}$$

Once we processed the last in-neighbour, E_{i,D_i-1} stores the *or* computed among all in-neighbours, where D_i is the number of in-neighbours of node v_i .

We implement the classic operation of adding 1 by computing an *or* with qubit A and storing the result in V'_i . Since $|a_j\rangle_A = |\delta_{0,j}\rangle$, we obtain $|0\rangle_{V'_i} \rightarrow |v'_{i,j}\rangle_{V'_i}$ where $|v'_{i,j}\rangle_{V'_i} = |1\rangle$ for $j = 0$, while $|v'_{i,j}\rangle_{V'_i} = |e_{i,D_i-1,j}\rangle$ for $1 \leq j \leq m-1$.

Operation 2 (line 11) is implemented as a Toffoli-gate application with qubits M and V'_i as control and V_i as target.

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{j,\ell(v_i)}\rangle_M |v'_{i,j}\rangle_{V'_i} |0\rangle_{V_i} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{j,\ell(v_i)}\rangle_M |v'_{i,j}\rangle_{V'_i} |m_{j,\ell(v_i)} \wedge v'_{i,j}\rangle_{V_i}$$

Operation 3 (line 12) is replaced by storing in register R_i the presence of a match ending at node v_i . This requires an intermediate step in which we use qubit B to filter the content of V_i . In fact, qubit V_i now is in state $|v_{i,j}\rangle_{V_i} = |1\rangle$ for those values of j such that $P[0..j]$ has a match ending at v_i in the graph, and $|v_{i,j}\rangle_{V_i} = |0\rangle$ otherwise. Since we only care about potential full matches represented by $|v_{i,m-1}\rangle_{V_i}$, we use B , which is in state $|\delta_{m-1,j}\rangle_B$, as control qubit of a Toffoli gate, the other control qubit being V_i and the target qubit being R'_i .

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |0\rangle_{R'_i} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i}$$

Then, using the same technique as in Operation 1, we compute an *or* between R'_i and R_{i-1} , storing the result in R_i .

$$\begin{aligned} & \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i} |r_{i-1,j}\rangle_{R_{i-1}} |0\rangle_{R_i} \rightarrow \\ & \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i} |r_{i-1,j}\rangle_{R_{i-1}} |(v_{i,j} \wedge \delta_{m-1,j}) \vee r_{i-1,j}\rangle_{R_i} \end{aligned}$$

After this operation, $|r_{i,m-1}\rangle_{R_i}$ is turned to $|1\rangle$ if there is a full match of P ending at v_i , otherwise $|r_{i,m-1}\rangle_{R_i}$ is left unaltered.

Operation 4 (line 14) consists in shifting all bits of the classical bit-vector by one position. In the quantum setting, we can perform this operation by adding 1 to index register J and then reorganising the sum: $|1\rangle_{C_1} |j\rangle_J \rightarrow |1\rangle_{C_1} |j+1\rangle_J$. Notice that this changes value $|j\rangle_J$ in every term of the superposition to $|j+1\rangle_J$. This can be interpreted as “shifting” value k_j of generic register K from $|j\rangle_J |k_j\rangle_K$ to $|j+1\rangle_J |k_j\rangle_K$. Because this operation acts on every quantum register and qubit in this way, we have to reset qubits A and B to $|0\rangle$ before performing this operation and reinitialize their values afterwards, so that we prevent their values to be shifted. For the same reason, we also have to wait until having processed the whole level, otherwise we would shift the values of all the nodes at the previous level and compromise the computation.

As last step of the algorithm, we run Grover’s search that uses as oracle function the whole procedure described up to this point, and then applies a Z gate on qubit R . Thus, the marked sub-states are those such that $|r_{i,j}\rangle_{R_i} = |1\rangle$, which get mapped to $-|r_{i,j}\rangle_{R_i}$. Sub-states such that $|r_{i,j}\rangle_{R_i} = |0\rangle$ remain unaltered. As for the case of string matching in plain text, we rerun the whole algorithm a constant number of times to boost the probability of success, as explained in Theorem 5 and Appendix C. Algorithm 1 shows the entire procedure.

To prove the correctness of Algorithm 1, we formalise the key properties in the following lemmas. We start by ensuring that the shift operation provides the desired result. Let l and y be the total number of times that we started the execution of the middle *for*-loop

■ **Algorithm 2** Algorithm for testing whether pattern string P has a match in level DAG G , running in $O(|E|\sqrt{|P|})$.

```

1 time.
  Input: Graph  $G$ , pattern  $P$ , and constant  $c$ .
  Output: Returns yes if  $P$  occurs in  $G$ , otherwise no.
2 for  $c$  times do
3   Initialize quantum register  $Q$  to  $|1\rangle$ ;
4   Initialize quantum registers  $I, J, A, B, C, M, V_i, V'_i, R_i, R'_i, E_{i,d}$  to  $|0\rangle$  where
      $i \in [0, n-1]$  and  $d \in [0, D_i-1]$ ;
     // Apply Hadamard to  $J$ 
5    $|0\rangle_J \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J$ ;
6   SourceNodesInit( $I, J, A, C, V_0, \dots, V_{n-1}, V'_0, \dots, V'_{n-1}$ );
7   IncreaseJ( $J, A, B$ );
     //  $L$  is the number of levels
8   for  $l \in [1, L-1]$  do                                     // scan every level
9     for  $|L_l|$  times do                                       // scan every node in the level
10      OperationOne( $l, I, C, M, E_{i,0}, \dots, E_{i,D_i-1}, V'_i$ );
11      OperationTwo( $M, V'_i, V_i$ );
          // Invariant 1 holds here
12      OperationThree( $B, V_i, R'_i, R_i$ );
13      IncreaseI( $I, M, C$ );
14      OperationFour( $J, A, B$ );
          // Invariant 2 holds here
15      Apply gate  $Z$  to qubit  $R_{n-1}$ , so that the sign of the amplitude is flipped if
           $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$ ;
16      Choose  $K \in [0, |P|]$  uniformly at random;
17      Run Grover's iterate operator the optimal number of times assuming to have  $K$ 
          solutions, with the oracle function being lines 6–15 of this algorithm;
18      Measure  $R_{n-1}$  into classical register  $R_{cl}$ ;
19      if  $R_{cl} = 1$  then
20        return yes
21 return no

```

```

1 Function OperationOne( $l, I, C, M, E_{i,0}, \dots, E_{i,D_i-1}, V'_i$ ):
2   for  $|L_l|$  times do                                     // scan every node in the level
3     // scan every node in  $in(v_i)$ 
4      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C$ ;
5      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |0\rangle_M \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M$ ;
6      $k \leftarrow in_i(0)$ ;                                     // Classical operation
7      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |0\rangle_{E_{i,0}} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |v_{k,j}\rangle_{E_{i,0}}$ ;
8     for  $d \in [1, D_i - 1]$  do                               // scan every node in  $in(v_i)$ 
9        $k \leftarrow in_i(d)$ ;                                   // Classical operation
10      // Add the contribution of the current in-neighbour
11       $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |0\rangle_{E_{i,d}} \rightarrow$ 
12       $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |e_{i,d-1,j} \vee v_{k,j}\rangle_{E_{i,d}}$ ;
13      // Turn to  $|1\rangle$  the substate corresponding to  $j = 0$ 
14       $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |\delta_{0,j}\rangle_A |e_{i,D_i-1,j} \vee v_{k,j}\rangle_{E_{i,D_i-1}} |0\rangle_{V'_i} \rightarrow$ 
15       $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |\delta_{0,j}\rangle_A |e_{i,D_i-1,j} \vee v_{k,j}\rangle_{E_{i,D_i-1}} |e_{i,D_i-1,j} \vee v_{k,j} \vee \delta_{0,j}\rangle_{V'_i}$ ;

11 Function OperationTwo( $M, V'_i, V_i$ ):
12   // Compute the and with the column of the matrix
13    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |v'_{i,j}\rangle_{V'_i} |0\rangle_{V_i} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |v'_{i,j}\rangle_{V'_i} |m_{\ell(v_i),j} \wedge v'_{i,j}\rangle_{V_i}$ ;

13 Function OperationThree( $B, V_i, R'_i, R_i$ ):
14   // Set  $|r_{i,m-1}\rangle_{R_i} = |1\rangle$  if there is a match ending at  $v_i$ 
15   // Apply Toffoli on  $V_i, B$  and  $R'_i$ 
16    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |0\rangle_{R'_i} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i}$ ;
17   // Apply logic or on  $R'_i, R_{i-1}$  and  $R_i$ 
18    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i} |r_{i-1,j}\rangle_{R_{i-1}} |0\rangle_{R_i} \rightarrow$ 
19    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R'_i} |r_{i-1,j}\rangle_{R_{i-1}} |(v_{i,j} \wedge \delta_{m-1,j}) \vee r_{i-1,j}\rangle_{R_i}$ ;

16 Function OperationFour( $J, A, B$ ):
17   IncreaseJ( $J, A, B$ );

```

(lines 9–13) and of the outer *for*-loop (lines 8–14), respectively. That is, $y = x + \sum_{\lambda=1}^{l-1} |L_\lambda|$ for $l \geq 2$, where $x \in [0, |L_l|]$ is the number of times that we started the execution of the middle *for*-loop during the l -th iteration of the outer *for*-loop. Notice that $y = 0$ when $l = 0$, and $y = x$ when $l = 1$.

► **Lemma 3** (Invariant 1). *During the l -th execution of the outer *for*-loop (lines 8–14) and the y -th execution of the middle *for*-loop (lines 9–13) of Algorithm 1, but before the y -th execution of `OperationThree()` (line 14), Invariant 1 holds: for every qubit V_i such that $i \in L_l$ and $i \leq t$, we have substate $|v_{i,j}\rangle_{V_i} = |1\rangle$ if and only if there exists a path in G ending at v_i and matching $P[0, j]$, where $t = |L_0| + x - 1$ is the index of the last node v_t Algorithm 1 visited so far.*

Proof. We proceed by strong induction on y , defined as above.

Base case, $y = 0$. In this case, we executed the initialization but we have not run yet neither the outer nor the middle *for*-loop. Thus, $l = 0$, $t = |L_0| - 1$, and qubits V_i such that $i \in L_0$ and $i \leq t$ are those with in-degree zero, which are initialized by function `SourceNodesInit()`. For each such i , given that J is in state $\sum_{j=0}^{m-1} |j\rangle_J$, function `SourceNodesInit()` first loads character $\ell(v_i)$ in register C and matrix entry $m_{\ell(v_i), j}$ in register M , in superposition. Then, with regard to t , it performs transformation

$$\sum_{j=0}^{m-1} |m_{\ell(v_i), j}\rangle_M |\delta_{0,j}\rangle_A |0\rangle_{V_i} \rightarrow \sum_{j=0}^{m-1} |m_{\ell(v_i), j}\rangle_M |\delta_{0,j}\rangle_A |m_{\ell(v_i), j} \wedge \delta_{0,j}\rangle_{V_i},$$

where, by definition, $v_{t,j} = m_{\ell(v_i), j} \wedge \delta_{0,j}$. Thus, $|v_{t,j}\rangle = |0\rangle$ for every $j \neq 0$ because of $\delta_{0,j}$, and $|v_{t,j}\rangle_{V_i} = |m_{\ell(v_i), j}\rangle_{V_i}$ for $j = 0$, which in turn means that $|v_{t,0}\rangle_{V_i} = |1\rangle$ if and only if $P[0, 0] = \ell(v_i)$.

Inductive case, $y \geq 1$. We further divide our analysis in two sub-cases.

First sub-case, $x = |L_l|$. In this case, y is the last iteration of the inner *for*-loop during the l -th iteration of the outer *for*-loop. We assume the inductive hypothesis to hold after the execution of `OperationTwo()`. We execute `OperationThree()` and `IncreaseI()`, which do not change the state of any V_z , for any $z \in [0, |V| - 1]$. Now, we have to perform `OperationFour()` (line 14) before starting iteration $y + 1$ of the middle *for*-loop, which will start iteration $l + 1$ of the outer *for*-loop. Assuming the inductive hypothesis, the application of `OperationFour()` makes every V_z with $z \in L_l$ such that $|j'\rangle_J |v_{z,j}\rangle_{V_z} = |j'\rangle |1\rangle$, where $j' = j + 1$, if and only if there is a match for $P[0, j]$ in G ending at v_z , otherwise $|j'\rangle_J |v_{z,j}\rangle_{V_z} = |j'\rangle |0\rangle$. Then, we start iteration $y + 1$ ($l + 1$). Notice that we update V_i if and only if $i \in L_{l+1}$ and, in any previous iteration of the middle *for*-loop, this could have never been the case, thus every $|v_{i,j}\rangle_{V_i}$, $i \in L_{l+1}$, is currently set to $|0\rangle$. The same holds for every V'_i . The *for*-loop inside `OperationOne()` computes a logic *or* between all the qubits representing all the in-neighbours of v_i . Indeed, before running this *for*-loop, we have $|j'\rangle_J |v_{in_i(0), j}\rangle_{E_{i,0}}$. After one iteration, we have $|j'\rangle |v_{in_i(0), j} \vee v_{in_i(1), j}\rangle_{E_{i,1}}$. After two iteration, we have $|j'\rangle |v_{in_i(0), j} \vee v_{in_i(1), j} \vee v_{in_i(2), j}\rangle_{E_{i,2}}$. After $D_i - 1$ iterations, we have $|j'\rangle |e_{i, D_i-1, j'}\rangle_{E_{i, D_i-1}}$, where

$$e_{i, D_i-1, j'} = \bigvee_{d=0}^{D_i-1} v_{in_i(d), j}.$$

We store an intermediate result in V'_i , $|v'_{i,j'}\rangle_{V'_i}$, where $v'_{i,j'} = e_{i,D_i-1,j'}$ except for $j' = 0$, because we make sure that $|v'_{i,0}\rangle_{V'_i} = |1\rangle$ thanks to the *or* operation with qubit A , which stores $|\delta_{0,j'}\rangle_A$. Now we compute the logical *and* with the entry of the matrix, as in the base case, obtaining $|v_{i,j'}\rangle_{V_i}$, where

$$v_{i,j'} = m_{\ell(v_i),j'} \wedge (\delta_{0,j'} \vee \bigvee_{d=0}^{D_i-1} v_{in_i(d),j}).$$

Applying the inductive hypothesis, this translates to

$$\begin{aligned} v_{i,j+1} &= (P[j+1] = \ell(v_i)) \wedge \left((j+1=0) \vee \bigvee_{d=0}^{D_i-1} P[0..j] \text{ has a match ending at } v_{in_i(d)} \right) \\ &= P[0..j+1] \text{ has a match ending at } v_i \end{aligned}$$

Thus, the statement of the lemma holds for $y+1$.

Second sub-case, $x < |L_l|$. The reasoning is analogous to the previous case, the only difference being that j does not increase and thus we have to look back by $x+1$ iterations, when j was increased the last time. This requires to assume that the inductive hypothesis was holding for iteration $y-x$, that is correct because, by strong induction, we assume the inductive hypothesis to hold for every $y' \leq y$ while proving the statement for $y+1$. ◀

► **Lemma 4 (Invariant 2).** *After line 14 of Algorithm 1, Invariant 2 holds: if there exists at least one match for P in G ending at some v_i such that $i \leq t$, then there exists at least one j , $0 \leq j \leq m-1$, such that $|r_{t,j}\rangle_{R_i} = |1\rangle$, where v_t is the last node we visited in Algorithm 1 before line 14.*

Proof. We proceed by induction on the number l of times that we run the *for*-loop at lines 8–14.

Base case, $l = 0$. In this case, nodes v_t such that $t \in L_{l'}$, $l' \leq 0$ are those with in-degree zero, while the *for*-loop at lines 8–14 has never run. Since we are visiting only single-node paths and we are assuming that pattern P has length at least two, there can be no match for P ending at these nodes. Correctly, $|r_{i,j}\rangle_{R_i} = |0\rangle$ for every $0 \leq j \leq m-1$.

Inductive case, $1 \leq l \leq L-1$. By inductive hypothesis, we assume the statement of the lemma to be true right after running iteration l of the *for*-loop at lines 8–14, and thus right before executing `IncreaseJ()` at line 14. After the execution of `IncreaseJ()`, the new state is $\sum_{j=0}^{m-1} |j\rangle_J |r_{i,j}\rangle_{R_i}$, where $j' = j+1$ and v_i is the last node visited so far. Then, we start iteration $l+1$, processing $i' \in L_{l+1}$, $i' = t+1$. We execute `OperationOne()` `OperationTwo()`, which do not affect register $R_{i'}$. Then we run the operations at lines 12–12, obtaining $|r_{i',j'}\rangle_{R_i}$ where $r_{i',j'} = (v_{i',j'} \wedge \delta_{m-1,j'}) \vee r_{t,j}$. Let us consider the first time we run the middle *for*-loop during iteration $l+1$ of the outer *for*-loop. If P has a match ending at some v_z , $z < i'$, the inductive hypothesis guarantees $r_{t,j} = 1$ for some j . Otherwise, if P does not have any such match, then $r_{t,j} = 0$ for all j . In this second case, if P has a match ending at $v_{i'}$, we know by Lemma 3 that $v_{i',m-1} = 1$. This, combined with the fact that $\delta_{m-1,m-1} = 1$, correctly implies that $r_{i',m-1} = 1$, proving the statement for this specific i' and $j' = m-1$. If P has no match ending at $v_{i'}$, then $v_{i',m-1} = 0$, and $r_{i',j'} = 0$ for all j' , which must be the case when no match has been found yet. To conclude the proof, notice that the same reasoning applies for the subsequent iterations of the middle *for*-loop by using every time the

previous instance of this reasoning in place of the inductive hypothesis. That is, we use $r_{i',j'}$ when proving the statement for $r_{i'+1,j'}$ and so on, until we prove the statement for $r_{v',j'}$, where v' is the last node with index in L_{l+1} . At this point, we exit the middle *for*-loop and the statement of the lemma is proven for $l + 1$. ◀

The correctness of the algorithm follows from the previous lemma combined with few additional observations.

► **Theorem 5.** *Given pattern string P of length at least 2 and level DAG G , Algorithm 1 returns the right answer for the SMLG problem on P and G with probability $p > 1 - (7/8)^c$, for any given integer c .*

Proof. After running the outer *for*-loop of Algorithm 1 $L - 1$ times, we exit such a loop, and we know we have visited all the nodes (nodes in L_0 where visited during the initialization). If we consider Lemma 4 applied in the case of $t = n - 1$, we are considering all the nodes, which means that if P has no match ending in G , then no substate of register R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for any j . Instead, if P has a match in G , then at least one substate of R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for some j . We use standard techniques that consist in rerunning the algorithm a constant number of times to boost the probability of measuring such a state, and achieve the desired one. Appendix C provides a more detailed analysis. ◀

Finally, the time complexity of our algorithm is subquadratic in the size of the graph.

► **Theorem 6.** *The time complexity of Algorithm 1 is $O(|E|\sqrt{|P|})$ in the QRAM model, and the space complexity is $O(|E| + |V|)$.*

Proof. The algorithm uses $|V|$ qubits V_i , and the same amount of qubits V'_i, R_i, R'_i ; qubits $E_{i,d}$ are a total of $|E|$ qubits, and the rest are a constant number of qubits and registers. Thus, the space complexity is $O(|E| + |V|)$.

With the *for*-loop in function `SourceNodesInit()`, the algorithm visits the nodes in L_0 , which are at most $O(|V|)$. The iteration conditions at lines 8 and 9 make the algorithm visit every node. For each such iteration, we perform a constant number of operations except for the *for*-loop in `OperationOne()`. This *for*-loop visits all the in-neighbours of a node, each time performing a constant number of operations, and $\sum_{i=0}^{|V|-1} |in(i)| = |E|$. All of the aforementioned operations can be implemented with a constant number of quantum-gate applications, each affecting a constant number of qubits (3 at most), or by performing a load operation from the QRAM, assumed to require constant time. At the end of the algorithm, we run Grover's search procedure on a superposition of $2|P|$ states, using the entire algorithm as the oracle function.

Summing everything up, we spend $O(|V|)$ time for the initialization, $O(|V| + |E|)$ time in the *for*-loops, and $O(|E|\sqrt{|P|})$ time for Grover's search procedure. The total time complexity is thus dominated by $O(|E|\sqrt{|P|})$. ◀

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.
- 2 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.
- 3 Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992. doi:10.1145/135239.135243.

- 4 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2), March 2020. doi:10.1145/3381417.
- 5 Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998.
- 6 Parisa Darbari, Daniel Gibney, and Sharma V. Thankachan. Quantum time complexity and algorithms for pattern matching on labeled graphs. In *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 303–314. Springer, 2022. doi:10.1007/978-3-031-20643-6_22.
- 7 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPICs*, pages 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- 8 Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.
- 9 Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996. doi:10.1145/237814.237866.
- 10 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 11 Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999. doi:10.1145/316542.316550.
- 12 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi:10.1017/CB09780511976667.
- 13 Pradeep Niroula and Yunseong Nam. A quantum algorithm for string matching. *npj Quantum Information*, 7(1):1–5, 2021.
- 14 Hariharan Ramesh and V Vinay. String matching in $O(\sqrt{n} + \sqrt{m})$ quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. *Combinatorial Algorithms*. doi:10.1016/S1570-8667(03)00010-8.
- 15 Mikko Rautiainen, Veli Mäkinen, and Tobias Marschall. Bit-parallel sequence-to-graph alignment. *Bioinform.*, 35(19):3599–3607, 2019. doi:10.1093/bioinformatics/btz162.
- 16 Kapil Kumar Soni and Ashwini Kumar Malviya. Design and analysis of pattern matching algorithms based on quram processing. *Arabian Journal for Science and Engineering*, 46(4):3829–3851, 2021.
- 17 Jorg Van Renterghem. The implications of breaking the strong exponential time hypothesis on a quantum computer. Master’s thesis, Ghent University, 2019.

A Reductions to the power-of-two case

For string matching in plain text, if $|T|$ is not a power of two, in addition to quantum register I for indexing, we use also quantum register I' , of the same size. Let x be the only integer x such that $|T| < 2^x < 2|T|$. Generate superposition $\sum_{i'=0}^{2^x-1} |i'\rangle_{I'} |0\rangle_I$ and compute $\sum_{i'=0}^{2^x-1} |i'\rangle_{I'} |i' \bmod |T|\rangle_I$. Now run the algorithm using I as normal. This creates some redundant substates, but does not affect the correctness of the algorithm.

For SMLG in level DAGs, if $|P|$ is not a power of two, we generate a superposition of size 2^x , where x is the only integer such that $|P| < 2^x < 2|P|$. Then, it suffices to assume that every entry that we read from the QRAM to the additional substates between $|P|$ and 2^x

is always initialized to $|1\rangle$, because this is the neutral value in a logical *and*, and thus in the application of the Toffoli gate. Therefore, in these substates, a qubit R_i can and will be set to value $|1\rangle_{R_i}$ if and only if a previous “shift” carried $|1\rangle_{R_{i-1}}$.

Alternatively, if $|P|$ is not a power of two, we can classically reduce the problem to this case. We add new symbol $\$$ to the alphabet. Then, we pad P with as many $\$$ at the end as needed to reach the next power of two. For each level in the DAG, we add a new node with label $\$$, and we place an edge for every node in that level to the new node. We connect all this new nodes in a chain, and we also add a chain of $|P|$ such nodes after the last level (they create new levels consisting only of one node). The pattern now can overflow in these nodes after finding a proper match in the DAG. Finally, we apply the same binary encoding as in the plain text case, now replacing every node with a chain of two nodes, sending all the incoming edges to the first node and making all the outgoing edges leave from the second node. Overall, we add one new node per level, and one new edge per node, plus $|P|$ additional nodes and edges after the last level. This takes time $O(|E| + |P|)$.

B Additional pseudo-code

```

1 Function SourceNodesInit( $I, J, A, C, V_0, \dots, V_{n-1}, V'_0, \dots, V'_{n-1}$ ):
    // Initialize  $|a_j\rangle_A$  so that  $a_j = 1$  if  $j = 0$ ,  $a_j = 0$  otherwise
2      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_A \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{0,j}\rangle_A$ ;
    // Initialize  $|b_j\rangle_B$  so that  $b_j = 1$  if  $j = m - 1$ ,  $b_j = 0$  otherwise
3      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_B \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{m-1,j}\rangle_B$ ;
    //  $L_0$  is the set of nodes in level 0.
4     for  $|L_0|$  times do
        // Read node label  $\ell(v_i)$  in  $C$ 
5          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C$ ;
        // Read the matrix entries for character  $\ell(v_i)$  in  $M$ 
6          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |0\rangle_M \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M$ ;
        // Apply Toffoli to qubits  $M$ ,  $A$  and  $V_i$ 
7          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |0\rangle_{V_i} \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |m_{\ell(v_i),j} \wedge \delta_{0,j}\rangle_{V_i}$ ;
        // Reset  $M$  and  $C$ 
8          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j} \oplus m_{\ell(v_i),j}\rangle_M =$ 
             $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |0\rangle_M$ ;
9          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i) \oplus \ell(v_i)\rangle_C = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C$ ;
        // Increase  $I$  by one to visit the next node
10         $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |1\rangle_Q |i\rangle_I \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |1\rangle_Q |i+1\rangle_I$ ;
    
```

```

1 Function IncreaseI( $I, M, C$ ):
   // Reset  $M$  and  $C$ 
2    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j} \oplus m_{\ell(v_i),j}\rangle_M \rightarrow$ 
    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_{C_i} |0\rangle_M;$ 
3    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i) \oplus \ell(v_i)\rangle_C = \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C;$ 
   // Increase  $I$ 
4    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |1\rangle_Q |i\rangle_I \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |1\rangle_Q |i+1\rangle_I;$ 

```

```

1 Function IncreaseJ( $J, A, B$ ):
   // Reset  $A$  and  $B$ 
2    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{0,j}\rangle_A \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_A;$ 
3    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{m-1,j}\rangle_B \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_B;$ 
   // Increase  $J$ 
4    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |1\rangle_Q |j\rangle_J \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |1\rangle_Q |j+1\rangle_J;$ 
   // Reinitialize  $A$  and  $B$ 
5    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_A \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{0,j}\rangle_A;$ 
6    $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |0\rangle_B \rightarrow \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\delta_{m-1,j}\rangle_B;$ 

```

C Full proof of Theorem 5

Proof. After running the outer *for*-loop of Algorithm 1 $L - 1$ times, we exit such a loop, and we know we have visited all the nodes (nodes in L_0 where visited during the initialization). If we consider Lemma 4 applied in the case of $t = n - 1$, we are considering all the nodes, which means that if P has no match ending in G , then no substate of register R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for any j . Instead, if P has a match in G , then at least one substate of R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for some j .

The *for* loop that we run at the end of the algorithm ensures to achieve high probability of success. The probability of success p in Grover's search algorithm is the sinusoidal function $p(K) = \sin^2((2K + 1)\theta)$ [5], where $\theta = \sin^{-1}\left(\sqrt{\frac{M}{N}}\right)$, N is the search space, M is the number of good solutions and K is the number of iterations of the Grover's operator. This function has period $\lambda_M \approx \frac{\pi}{2} \sqrt{\frac{N}{M}} - 1$. Consider the case $M = 1$. If we choose a random number of iterations K between 1 and λ_1 , we have $p(K) \geq 1/2$ with probability $1/2$. This is because half of the material of the function is above the horizontal line of $1/2$. When $p(K) \geq 1/2$, the probability of measuring a wrong result is $p_{\text{top}} \leq 1/2$. When $p(K) \geq 1/2$, the probability of measuring a wrong result is greater than $1/2$, but anyway $p_{\text{bottom}} \leq 1$. If we run the process c times, the overall probability of failure (measuring a wrong result) p_f is then

9:20 From Bit-Parallelism to Quantum String Matching for Labelled Graphs

$$p_f = \left(p_{\text{top}} \frac{1}{2} + p_{\text{bottom}} \frac{1}{2} \right)^c \leq \left(\frac{1}{2} \frac{1}{2} + 1 \cdot \frac{1}{2} \right)^c = \left(\frac{3}{4} \right)^c$$

Thus, the probability of success (measuring a correct result) is $p_s = 1 - (3/4)^c$.

In the general case $1 < M \leq N$, the period λ_M of function $p(K)$ is smaller than period λ_1 of the case $M = 1$. We can still use the same random number of iterations K between 1 and λ_1 , as nearly half of the material of the function $p(K)$ is above the horizontal line of $1/2$: the worst case is when λ_M is little over half of λ_1 . In this case we know that $p(K)$ will be sampled uniformly over half of the range of period λ_1 , but the other half may have biased sampling. Namely, the other half of the function might have more material below $1/2$ than above. To have a safe estimate, we assume that the probability of returning the wrong result in the biased case is $p_{\text{biased}} = 1$. That is, if we run the process c times, the overall probability of failure (measuring a wrong result) p_f is then

$$p_f = \left(p_{\text{biased}} \frac{1}{2} + \left(p_{\text{top}} \frac{1}{2} + p_{\text{bottom}} \frac{1}{2} \right) \frac{1}{2} \right)^c \leq \left(1 \cdot \frac{1}{2} + \left(\frac{1}{2} \frac{1}{2} + 1 \cdot \frac{1}{2} \right) \frac{1}{2} \right)^c = \left(\frac{7}{8} \right)^c$$

Thus, the probability of success (measuring a correct result) is $p_s = 1 - (7/8)^c$. ◀

On the Impact of Morphisms on BWT-Runs

Gabriele Fici  

Department of Mathematics and Informatics, University of Palermo, Italy

Giuseppe Romana  

Department of Mathematics and Informatics, University of Palermo, Italy

Marinella Sciortino  

Department of Mathematics and Informatics, University of Palermo, Italy

Cristian Urbina  

Department of Computer Science, University of Chile, Santiago, Chile

Centre for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Abstract

Morphisms are widely studied combinatorial objects that can be used for generating infinite families of words. In the context of Information theory, injective morphisms are called (variable length) codes. In Data compression, the morphisms, combined with parsing techniques, have been recently used to define new mechanisms to generate repetitive words. Here, we show that the repetitiveness induced by applying a morphism to a word can be captured by a compression scheme based on the Burrows–Wheeler Transform (BWT). In fact, we prove that, differently from other compression-based repetitiveness measures, the measure r_{bwt} (which counts the number of equal-letter runs produced by applying BWT to a word) strongly depends on the applied morphism. More in detail, we characterize the binary morphisms that preserve the value of $r_{bwt}(w)$, when applied to any binary word w containing both letters. They are precisely the Sturmian morphisms, which are well-known objects in Combinatorics on words. Moreover, we prove that it is always possible to find a binary morphism that, when applied to any binary word containing both letters, increases the number of BWT-equal letter runs by a given (even) number. In addition, we derive a method for constructing arbitrarily large families of binary words on which BWT produces a given (even) number of new equal-letter runs. Such results are obtained by using a new class of morphisms that we call Thue–Morse-like. Finally, we show that there exist binary morphisms μ for which it is possible to find words w such that the difference $r_{bwt}(\mu(w)) - r_{bwt}(w)$ is arbitrarily large.

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

Keywords and phrases Morphism, Burrows–Wheeler transform, Sturmian word, Sturmian morphism, Thue–Morse morphism, Repetitiveness measure

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.10

Funding *Gabriele Fici*: Partly supported by MIUR project PRIN 2017 ADASCOML – 2017K7XPAN.

Giuseppe Romana: Partly supported by MIUR project PRIN 2017 ADASCOML – 2017K7XPAN.

Marinella Sciortino: Partly supported by the INdAM-GNCS Project (CUP E55F22000270001) and the PNRR project ITSERR (CUP B53C22001770006).

Cristian Urbina: Partly supported by ANID national doctoral scholarship – 21210580.

Acknowledgements This research was carried out during the visit of Cristian Urbina to the University of Palermo, Italy.

1 Introduction

The Burrows–Wheeler transform (BWT) is a reversible permutation of words, introduced in the Data compression field [5]. Such a transformation allows one to *boost* the effect of the run-length encoding with respect to the original word in input [27]. Due to its



© Gabriele Fici, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 10; pp. 10:1–10:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

myriad virtues, some of the well-known compressed text-indexes for pattern matching [11, 13] and the most used alignment tools in Bioinformatics [23, 21] are based on the BWT. The performance of the BWT is related to the repetitions of factors in the word, which is why the number of equal-letter runs of the BWT, denoted by r_{bwt} , is considered as a measure of repetitiveness [29]. Much attention has recently been paid to the measure r_{bwt} both for its crucial role in designing compressed indexing data structures for highly repetitive texts [13, 16, 30] and for its combinatorial properties [25, 14].

In Combinatorics on words, morphisms are a fundamental tool for generating repetitive sequences, with multiple applications. For instance, injective morphisms, known as codes, are widely used in the fields of Information theory, Data compression, and Cryptography [2]. Recently, morphisms have been used in conjunction with copy-paste mechanisms to define novel compressors and repetitiveness measures, called NU-systems [32]. Informally speaking, a morphism is a mechanism that transforms each letter in a given input word into a corresponding image word, thus producing an output that is likely to contain longer repeated factors. The relationship between morphisms and the measure r_{bwt} has been studied in the context of a subclass of infinite words generated by morphisms, i.e., the purely morphic words [4, 12].

Here, we focus the impact of morphism application on the number of BWT equal-letter runs of finite words.

In Section 3, we prove that a binary morphism is cyclic (i.e., the images of both letters are powers of the same word) if and only if the image of every word under this morphism has the same number of BWT equal-letter runs, regardless of the input word. We also prove other results relating morphisms and words sharing the same Parikh vector (i.e., having the same number of occurrences of each letter), which can be of independent interest.

Then, in Section 4 we find a novel characterization of Sturmian morphisms [3, 28] in terms of BWT equal-letter runs: they are exactly the binary morphisms that preserve the number of BWT equal-letter runs of every binary word containing both letters of the alphabet. This characterization is interesting from a combinatorial point of view, because Sturmian morphisms are a widely studied subject [3, 28]. It also builds another bridge between Combinatorics on words and Data compression.

Further, in Section 5 we show a wide class of morphisms, which we call Thue–Morse-like morphisms, that increase the number of BWT equal-letter runs by 2 on every binary word containing both letters of the alphabet. Moreover, for each even number $2k$, we can find a wide class of binary morphisms, obtained by composing Sturmian and Thue–Morse-like morphisms, that increase the BWT equal-letter runs of every binary word by exactly $2k$. Note that this is exhaustive for the binary alphabet. In fact, unless considering powers of a single letter, every binary word has an even number of BWT equal-letter runs. In addition, we can use the aforementioned morphisms to construct arbitrarily large families of binary words having all the same number of BWT equal-letter runs, for every fixed (even) number, and converging to an infinite aperiodic word.

At the other end of the spectrum, in Section 6 we show that there are binary morphisms (in particular, the so-called period-doubling morphism) that can highly increase the number of BWT equal-letter runs of binary words. We show that the increase in the number of BWT equal-letter runs can be $\Omega(\sqrt{n})$, where n is the length of the original word. In Section 7, we show that this degree of increase cannot occur in other relevant reachable repetitiveness measures, like the size of the Lempel–Ziv parsing [9, 22], or the size g of the smallest deterministic context-free grammar generating the word [18].

We conclude in Section 8 with some final remarks, and some open questions and conjectures.

2 Preliminaries

Basic terminology

Let $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$ be a finite sorted set of *letters* $a_1 < a_2 < \dots < a_\sigma$, which we call an *alphabet*. A *finite word* $w = w[1]w[2] \cdots w[|w|]$ is any finite sequence of letters where $w[i] \in \Sigma$, for $i \in [1, |w|]$, and $|w|$ is the *length* of the word. We denote by $\text{alph}(w)$ the set of the letters of Σ appearing in w . The *empty word*, denoted by ε , is the unique word of length 0. The set of all finite words (resp. all finite words of positive length) over the alphabet Σ is denoted by Σ^* (resp. Σ^+). If $u = u[1] \cdots u[n]$ and $v = v[1] \cdots v[m]$ are words, the *concatenation* uv of u and v is $uv = u[1] \cdots u[n]v[1] \cdots v[m]$. We use the notation $w[i, j]$ to denote the word $w[i]w[i+1] \cdots w[j]$, which we call a *factor* of w . If $i > j$, then we assume $w[i, j] = \varepsilon$. A factor of w is *proper* if it is different from w itself. The factor $w[i, j]$ is called a *prefix* when $i = 1$, and a *suffix* when $j = n$. We denote by $\Pi_{i=1}^k w_i$ the concatenation of the words w_1, w_2, \dots, w_k in that order. We denote by w^k the concatenation of the word w with itself k times. A *rotation* of the word $w = w[1]w[2] \cdots w[n]$ is a word of the form $w[i+1, n]w[1, i]$, for some $1 \leq i \leq n$, obtained by shifting i letters cyclically. We denote by $\mathcal{R}(w)$ the multiset of all the $|w|$ rotations of w . A factor of any word in $\mathcal{R}(w)$ is called a *circular factor* of w . A word is *primitive* if $w = u^k$ implies $k = 1$, or equivalently, if it cannot be written as uv for some non-empty words u and v such that $uv = vu$. A primitive word of length n has exactly n distinct rotations. If w is a *binary word* over the alphabet $\{a, b\}$, the *complement* of w , i.e., the word obtained by replacing all the a 's of w by b 's and all the b 's by a 's, is denoted by \bar{w} . If $w = w[1] \cdots w[n]$, the *reverse* of w is the word $w^R = w[n] \cdots w[1]$. Given a word $w \in \Sigma^*$ and $a \in \Sigma$, we denote by $|w|_a$ the number of occurrences of a in w . The *run-length encoding* of a word w , denoted by $\text{rle}(w)$, is a sequence of pairs (c_i, l_i) with $c_i \in \Sigma$ and $l_i > 0$, such that $w = c_1^{l_1} c_2^{l_2} \cdots c_r^{l_r}$ and $c_i \neq c_{i+1}$. The length $|\text{rle}(w)|$ is the number of *equal-letter runs* in w . The *Parikh vector* of w , denoted as $P(w)$, is the σ -tuple $(|w|_{a_1}, \dots, |w|_{a_\sigma})$. Given two words u and v having the same length, the *Hamming distance* between u and v , denoted as $d_H(u, v)$, is the number of positions at which the corresponding letters in u and v are different. An *infinite word* $x = x[1]x[2]x[3] \cdots$ is a non-ending sequence of elements of the alphabet Σ . An infinite word x is *ultimately periodic* if there exist $u \in \Sigma^*$ and $v \in \Sigma^+$ such that $x = uvv \cdots$; it is called *periodic* when $u = \varepsilon$; *aperiodic* if it is not ultimately periodic. If there is no ambiguity, finite words are simply called words.

Morphisms

Let Σ and Γ be two alphabets. A *morphism* is a map μ from Σ^* to Γ^* such that $\mu(uv) = \mu(u)\mu(v)$ for all words $u, v \in \Sigma^*$. Therefore, a morphism μ can be defined by specifying its action on the letters of Σ and can be denoted as $\mu \equiv (\mu(a_1), \dots, \mu(a_\sigma))$. When $\Sigma = \Gamma = \{a, b\}$, μ is called a *binary morphism*. A morphism μ is called *prolongable* on a letter $a \in \Sigma$ if $\mu(a) = au$ for some $u \in \Sigma^+$. If for all $a \in \Sigma$ it holds that $\mu(a) \neq \varepsilon$, then the morphism μ is called *non-erasing*. From now on, we will consider non-erasing morphisms, unless stated explicitly otherwise. If there exists k such that $|\mu(a)| = k$ for every $a \in \Sigma$, then the morphism is called *k-uniform*. A 1-uniform morphism is called a *coding*. Given a morphism μ prolongable on some letter $a \in \Sigma$, the family of words $\{a, \mu(a), \dots, \mu^i(a), \dots\}$ are prefixes of a unique infinite word $\mu^\infty(a) = \lim_{i \rightarrow \infty} \mu^i(a)$, that is a fixed point of μ . Such an infinite word is then called *purely morphic*. An infinite word is *morphic* if it is obtained by applying a coding to a purely morphic word. A morphism μ is *cyclic* if there exists $z \in \Gamma^*$ such that $\mu(a) \in z^*$, for each $a \in \Sigma$. Otherwise, it is called *acyclic*. Note that the fixed point of a cyclic morphism is periodic. In the case of a binary morphism, it is known that μ is cyclic if and only if $\mu(ab) = \mu(ba)$.

Sturmian words and Sturmian morphisms

Let $\Sigma = \{a, b\}$. A word $w \in \Sigma^*$ is called *balanced* if the difference of the number of a 's (or, equivalently, b 's) in every two factors of the same length of w is at most 1. An infinite word x is balanced if every finite factor of x is balanced. A finite word w is *circularly balanced* if each word in $\mathcal{R}(w)$ is balanced.

An infinite word over $\Sigma = \{a, b\}$ is a *Sturmian word* if it has exactly $n + 1$ distinct factors of length n for every $n \geq 0$. The theory of Sturmian words is very well studied (see [24] for a reference). For example, the following characterization is well known.

► **Theorem 1.** *An infinite word over $\Sigma = \{a, b\}$ is Sturmian if and only if it is balanced and aperiodic.*

A class of Sturmian words, called *characteristic Sturmian words*, can be constructed by using finite words, called *standard Sturmian words*, defined recursively as follows. Given an infinite sequence of integers (d_0, d_1, d_2, \dots) , with $d_0 \geq 0, d_i > 0$ for all $i > 0$, called *directive sequence*, the associated standard Sturmian words are defined by $s_0 = b, s_1 = a$, and $s_{i+1} = s_i^{d_i-1} s_{i-1}$, for $i \geq 1$. A characteristic Sturmian word is the limit of an infinite sequence of standard Sturmian words, i.e., $s = \lim_{i \rightarrow \infty} s_i$. Note that standard Sturmian words are finite words also appearing as extremal case for several algorithms and data structures [19, 7, 26, 37].

A *Sturmian morphism* is a morphism that maps infinite Sturmian words to infinite Sturmian words. Some combinatorial characterizations of Sturmian morphisms have been proved in [3]. In particular, a binary morphism μ is Sturmian if and only if it is acyclic and *balanced* (i.e., it maps balanced words to balanced words). Berstel and Séebold [3] also proved the following characterization:

► **Theorem 2.** *An acyclic morphism μ is Sturmian if and only if it is locally Sturmian, that is, there exists a Sturmian word s such that $\mu(s)$ is Sturmian.*

Let us denote the following morphisms:

$$E : \begin{cases} a \mapsto b \\ b \mapsto a \end{cases} \quad \varphi : \begin{cases} a \mapsto ab \\ b \mapsto a \end{cases} \quad \tilde{\varphi} : \begin{cases} a \mapsto ba \\ b \mapsto a \end{cases}$$

The morphism φ is called the *Fibonacci morphism*, since its fixed point is the Fibonacci word $abaababaabaababaab\dots$. The monoid $\{E, \varphi, \tilde{\varphi}\}^*$ generated by E, φ , and $\tilde{\varphi}$, by using the composition operator \circ , is known as the *Sturm monoid*. The following theorem, proved in [28], shows the combinatorial structure of Sturmian morphisms.

► **Theorem 3.** *A morphism is Sturmian if and only if it belongs to $\{E, \varphi, \tilde{\varphi}\}^*$.*

Burrows–Wheeler transform

The *Burrows–Wheeler transform* (BWT) of a word w , denoted by $\text{bwt}(w)$, is a permutation of w obtained by sorting all its rotations in lexicographical order and then concatenating the last symbol of each rotation. The original word can be recovered if one stores the position where it appears in the list of sorted rotations. If a word is highly repetitive, the number of equal-letter runs of the BWT tends to be small. In fact, Kempa and Kociumaka have shown that r_{bwt} is never too far from the size of the Lempel–Ziv parsing, a widely used repetitiveness measure [16]. Hence applying run-length encoding to the BWT is very effective. Because of this, the value $r_{\text{bwt}}(w) = |\text{rle}(\text{bwt}(w))|$ that counts the number of *BWT-runs* of w , i.e., equal-letter runs of $\text{bwt}(w)$, is used as a measure for capturing the repetitiveness of the word

w . To understand the particularities of the BWT of a word w , sometimes it is useful to think about the *BWT-matrix* of the sorted rotations of w . It is not difficult to see that, when w is a word such that $\text{alph}(w) = \{a, b\}$, then $r_{\text{bwt}}(w)$ is an even number.

The Burrows–Wheeler transform is strictly related to the notions of balance, Sturmian word and morphism, as shown in the following proposition.

► **Proposition 4.** *Let w be a word such that $\text{alph}(w) = \{a, b\}$. Then the following are equivalent:*

1. w is circularly balanced;
2. $w \in \mathcal{R}(s^\ell)$, for some standard Sturmian word s and for some $\ell > 0$;
3. $r_{\text{bwt}}(w) = 2$;
4. $w = (\mu(a))^\ell$ for a Sturmian morphism μ and for some $\ell > 0$.

Proof. The equivalence of 1, 2 and 3 is in [26, 35]. The equivalence with 4 is in [8] (see also Proposition 10 in [34]). ◀

3 Morphisms and sorted rotations of words

We start by introducing some definitions regarding the rotations of morphic images of words.

► **Definition 5.** *Let $\mu : \Sigma^* \mapsto \Gamma^*$ be a morphism. Then, we define the multisets*

$$\mathcal{I}_\mu(w) = \{\mu(w') \mid w' \in \mathcal{R}(w)\}$$

$$\mathcal{S}_\mu(w) = \{v\mu(w')u \mid u, v \in \Gamma^+, uv = \mu(a) \text{ for some } a \in \Sigma, \text{ and } aw' \in \mathcal{R}(w)\}.$$

The multiset $\mathcal{I}_\mu(w)$ corresponds to the rotations of $\mu(w)$ obtained by applying μ to the rotations of w . The multiset $\mathcal{S}_\mu(w)$ corresponds to all the remaining rotations of $\mu(w)$. We refer to the multiset $\mathcal{I}_\mu(w)$ as the *I-rotations* of $\mu(w)$, and to the multiset $\mathcal{S}_\mu(w)$ as the *S-rotations* of $\mu(w)$. These two multisets could have elements that end up being equal, as we show in the following example.

► **Example 6.** Let $\mu \equiv (a, bab)$, which is an acyclic binary morphism. Then, ab is primitive but $\mu(ab) = abab$ is not. Moreover, $\mathcal{I}_\mu(w) = \{abab, baba\} = \mathcal{S}_\mu(w)$.

We now prove some combinatorial properties of words having the same Parikh vector. By using such properties, we prove that, in the case of the binary alphabet, the lexicographic order among the rotations of a given word is either preserved or reversed, after a morphism is applied. This is a key point to show that the number of BWT-runs cannot decrease after the application of a binary morphism. This is no longer true for larger alphabets.

The following lemma shows that distinct words having the same Parikh vector must have Hamming distance of at least 2.

► **Lemma 7.** *Let $w_1, w_2 \in \Sigma^*$ be such that $w_1 \neq w_2$ and $P(w_1) = P(w_2)$. Then, $d_H(w_1, w_2) \geq 2$.*

Proof. By definition of d_H , we have that $d_H(w_1, w_2) = 0$ if and only if $w_1 = w_2$. So, let us suppose by contradiction that $d_H(w_1, w_2) = 1$. Then, there exist two finite words $u, v \in \Sigma^*$ and two distinct indices $i < j \in [1, \sigma]$ such that $w_1 = ua_i v$ and $w_2 = ua_j v$. It follows that the Parikh vectors of w_1 and w_2 are respectively

$$P(w_1) = (|u|_{a_1} + |v|_{a_1}, \dots, |u|_{a_i} + |v|_{a_i} + 1, \dots, |u|_{a_j} + |v|_{a_j}, \dots, |u|_{a_\sigma} + |v|_{a_\sigma})$$

and

$$P(w_2) = (|u|_{a_1} + |v|_{a_1}, \dots, |u|_{a_i} + |v|_{a_i}, \dots, |u|_{a_j} + |v|_{a_j} + 1, \dots, |u|_{a_\sigma} + |v|_{a_\sigma}).$$

Thus, we obtain that the $P(w_1) \neq P(w_2)$, a contradiction. ◀

10:6 On the Impact of Morphisms on BWT-Runs

Since all the words in the same conjugacy class share the same Parikh vector, we can derive the following

► **Corollary 8.** *Let $w \in \Sigma^*$ be a word. Then, for every word $w' \in \mathcal{R}(w)$ such that $w' \neq w$, one has $d_H(w, w') \geq 2$.*

Here, we introduce and study new properties of some classes of morphisms, which are related to the number of BWT-runs.

► **Definition 9.** *A morphism μ is abelian order-preserving if for every pair of distinct words x and y having the same Parikh vector, it holds that $x < y \iff \mu(x) < \mu(y)$.*

A morphism μ is abelian order-reversing if for every pair of distinct words x and y having the same Parikh vector, it holds that $x < y \iff \mu(x) > \mu(y)$.

In general, a morphism can be neither abelian order-preserving nor abelian order-reversing:

► **Example 10.** A cyclic morphism is trivially not abelian order-preserving nor abelian order-reversing. The acyclic morphism $\mu \equiv (b, a, c)$ is also neither of them. This can be verified on the rotations of the word abc .

However, all acyclic morphisms with a binary domain are either abelian order-preserving or abelian order-reversing, as we show in the following lemma.

► **Lemma 11.** *Let $\mu : \{a, b\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then, μ is either abelian order-preserving or abelian order-reversing.*

Proof. Let $\mu \equiv (\alpha, \beta)$ be an acyclic morphism (i.e., $\alpha\beta \neq \beta\alpha$). For the proof, we assume that $|\alpha| \leq |\beta|$, and the other case is treated symmetrically. Factorize μ as $(\alpha, \beta) = (\alpha, \alpha^k v)$, where $k \geq 0$ is as big as possible. This factorization is unique, and α is not a prefix of v , otherwise, k is not as big as possible. Also, $v \neq \varepsilon$ and $v \neq \alpha$ because the morphism μ is acyclic. Let $x = uaz_1$ and $y = ubz_2$ be two distinct binary words with the same Parikh vector. Note that a b has to appear in z_1 , since otherwise x has fewer b 's than y . Let $z_1 = a^t bz'_1$ for some $t \geq 0$ and $z'_1 \in \{a, b\}^*$. We can write $x = uaa^t bz'_1$. Then, $\mu(x) = \mu(u)\alpha^k \alpha^t v \mu(z'_1)$ and $\mu(y) = \mu(u)\alpha^k v \mu(z_2)$. We proceed by case analysis.

If v is not a prefix of α , then the order between $\mu(x)$ and $\mu(y)$ depends only on the order between α and v . The reason is that $\mu(x)$ and $\mu(y)$ share a common prefix $\mu(u)\alpha^k$, followed by α and v respectively, which differ at some position from left to right. Hence, if $\alpha < v$, we obtain $x < y \iff \mu(x) < \mu(y)$; if $v < \alpha$, then we obtain $x < y \iff \mu(x) > \mu(y)$.

If v is a proper prefix of α and $k > 0$, rewrite $\mu(y) = \mu(u)\alpha^k v \alpha z'_2$. We can do this because y has to have at least one letter after ub and both images α and β start with α (in the case of β because $k > 0$). We note that the common prefix $\mu(u)\alpha^k$ is followed by αv in $\mu(x)$ (αv is a prefix of $\alpha\alpha$), and by $v\alpha$ in the case of $\mu(y)$. The order between $\mu(x)$ and $\mu(y)$ is then completely determined by the order between αv and $v\alpha$. This happens because αv and $v\alpha$ are words of the same length which must be distinct, as implied by the inequality $\alpha\beta = \alpha\alpha^k v \neq \beta\alpha = \alpha^k v\alpha$. Hence, if $\alpha v < v\alpha$, we obtain $x < y \iff \mu(x) < \mu(y)$; if $v\alpha < \alpha v$, then we obtain $x < y \iff \mu(x) > \mu(y)$.

No other case is possible. By construction, α is not a prefix of v . Also, $\alpha \neq v$, so if v is a prefix of α , it has to be a proper prefix. If this is the case, as $|\alpha| \leq |\alpha^k v|$ and $|v| < |\alpha|$, k has to be at least 1. ◀

Using Lemma 11 we can easily derive the following corollary.

► **Corollary 12.** *Let w be a binary word and let μ be an acyclic morphism. Then, for all pairs of rotations u, v of w , either $u < v \iff \mu(u) < \mu(v)$ (when μ is abelian order-preserving), or $u < v \iff \mu(u) > \mu(v)$ (when μ is abelian order-reversing).*

We introduce new measures to study how the action of a morphism affects the BWT-runs.

► **Definition 13.** *Let μ be a morphism and w a word. We define*

$$\Delta_{\mu}^{+}(w) = r_{\text{bwt}}(\mu(w)) - r_{\text{bwt}}(w)$$

and

$$\Delta_{\mu}^{\times}(w) = \frac{r_{\text{bwt}}(\mu(w))}{r_{\text{bwt}}(w)}.$$

Acyclic binary morphisms cannot decrease the number of BWT-runs of any word.

► **Theorem 14.** *Let $\mu : \{a, b\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then $\Delta_{\mu}^{+}(w) \geq 0$ for every $w \in \{a, b\}^*$.*

Proof. Let $\mu \equiv (\alpha, \beta)$. Since $r_{\text{bwt}}(w) = r_{\text{bwt}}(w^m)$ for every $w \in \Sigma^*$ and $m > 1$, let us assume that w is primitive. For the proof, we assume that $|\alpha| \geq |\beta|$, and the other case is treated symmetrically. First, let us consider the case where β is not a suffix of α . Let moreover $x \in \Sigma^*$ be the longest common suffix between α and β . It follows that there exist $\alpha', \beta' \in \Sigma^+$ such that $\alpha = \alpha'x$ and $\beta = \beta'x$, and that the last symbol of α' is different from the last of β' (otherwise x would be longer). Let $\mathcal{R}_x(\mu(w))$ denote the multiset of rotations of $\mu(w)$ with x as a prefix. Note that if $x = \varepsilon$, then $\mathcal{R}_x(\mu(w)) = \mathcal{I}_{\mu}(w)$. Since x appears in both α and β , it follows that $|\mathcal{R}_x(\mu(w))| \geq |w|$. Specifically, for each $i \in [1, |w|]$, there exists $t_i \in \mathcal{R}_x(\mu(w))$ such that $t_i = x\mu(w[i+1, |w|] \cdot w[1, i-1])v$, where v is either α' or β' , depending on whether $w[i]$ is a or b respectively. The lexicographical order of these $|w|$ rotations of $\mu(w)$ with the same prefix correspond to the lexicographical order of the rotations in $\mathcal{I}_{\mu}(w)$, since by Corollary 8 the words $\bigcup_{i=1}^{|w|} \{\mu(w[i+1, |w|] \cdot w[1, i-1])\}$ must differ in at least one position. By Corollary 12 this is either in the same or in the reverse order with respect to the sorting of the rotations of w . Thus, there exists an injective coding $\lambda : \{a, b\}^* \mapsto \Sigma'^* \subseteq \Sigma$ such that either $\lambda(\text{bwt}(w))$ or $\lambda(\text{bwt}(w)^R)$ is a subsequence of $\text{bwt}(\mu(w))$, and therefore $r_{\text{bwt}}(\mu(w)) \geq r_{\text{bwt}}(w)$.

Let us now consider the case where β is suffix of α . Then, there exists a primitive word $u \in \Sigma^+$ and two integers $p \geq q \geq 1$ such that $\beta = u^q$, and $\alpha = \alpha'u^p$, with $\alpha' \in \Sigma^+$ that does not have u as suffix. Note that $\alpha' \neq \varepsilon$, otherwise we would have $\alpha\beta = u^p u^q = u^q u^p = \beta\alpha$, i.e. μ would not be acyclic. Let x be the longest common suffix between α' and u . If $x \neq \alpha'$, from analogous arguments to the case where β is not a suffix of α , we have at least $r_{\text{bwt}}(w)$ equal-letter runs in $\mathcal{R}_{x u^p}(\mu(w))$. Otherwise, if $x = \alpha'$, let us consider the word $y \in \Sigma^+$ such that $u = yx$. We can then consider the longest common suffix x' between xy and yx , which must be a proper suffix (otherwise u would not be primitive), and apply the same reasoning over the set $\mathcal{R}_{x' x u^p}(\mu(w))$ and the thesis follows. ◀

The following example shows that Theorem 14 does not hold in the case of larger alphabets.

► **Example 15.** Consider the acyclic morphism $\mu \equiv (b, a, c)$. Then, $\text{bwt}(bcba) = bcab$ and $\text{bwt}(\mu(bcba)) = \text{bwt}(acab) = cbaa$.

An immediate consequence of Theorem 14 is the following.

10:8 On the Impact of Morphisms on BWT-Runs

► **Corollary 16.** *Let $\mu : \{a, b\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then, $\Delta_\mu^\times(w) \geq 1$, for every $w \in \{a, b\}^*$.*

The following theorem provides a characterization of cyclic morphisms in terms of the number of BWT-runs.

► **Theorem 17.** *A morphism $\mu : \{a, b\}^* \mapsto \Sigma^*$ is cyclic if and only if there exists $k > 0$ such that $r_{\text{bwt}}(\mu(w)) = k$ for all $w \in \{a, b\}^*$.*

Proof. If $\mu \equiv (\alpha, \beta)$ is cyclic then there exists a primitive word $u \in \Sigma^*$ such that $\alpha = u^p$ and $\beta = u^q$, for some $p, q \geq 0$. Therefore, for each word $w \in \{a, b\}^*$, we have $r_{\text{bwt}}(\mu(w)) = r_{\text{bwt}}(u^{p \cdot |w|_a + q \cdot |w|_b}) = r_{\text{bwt}}(u)$. The other implication is a consequence of Theorem 14. In fact, by contraposition for each $k > 0$ we can find a word w such that $r_{\text{bwt}}(w) > k$ (for instance, the i -th Thue–Morse finite word such that $i > \frac{k}{2}$ [4]), which leads to $r_{\text{bwt}}(\mu(w)) \geq r_{\text{bwt}}(w) > k$ as well. ◀

4 Binary morphisms preserving r_{bwt}

This section is devoted to characterizing binary morphisms such that the number of BWT equal-letter runs is preserved after the action of the morphism on any binary word. First, we show with an example that this property is not trivial.

► **Example 18.** Let $\theta \equiv (ab, aa)$ be the period-doubling morphism. It can be verified that $\Delta_\theta^+(ab) = 0$, $\Delta_\theta^+(aab) = 2$, and $\Delta_\theta^+(aaabbaabab) = 4$.

Next, we show that every Sturmian morphism fixes the number of BWT-runs. From the definition of E , φ , and $\tilde{\varphi}$, and by Lemma 11, we derive the following.

► **Lemma 19.** *Let $w \in \{a, b\}^*$ be a binary word. Then, for all pairs of rotations u and v of w , and for each $\chi \in \{E, \varphi, \tilde{\varphi}\}$, it holds that $u < v$ if and only if $\chi(u) > \chi(v)$.*

We prove that the number of BWT-runs is preserved by the morphisms that are the generators of the Sturmian morphisms. Note that from the following lemma a method can be derived to construct $\text{bwt}(\mu(w))$ starting from $\text{bwt}(w)$, for every Sturmian morphism μ and every binary word w .

► **Lemma 20.** *Let $w \in \{a, b\}^*$ be a binary word with $|\text{alph}(w)| = 2$. Then, for all $\chi \in \{E, \varphi, \tilde{\varphi}\}$, one has $r_{\text{bwt}}(w) = r_{\text{bwt}}(\chi(w))$. More in detail, one has $\text{bwt}(E(w)) = \overline{\text{bwt}(w)}^R$ and $\text{bwt}(\varphi(w)) = \text{bwt}(\tilde{\varphi}(w)) = \overline{\text{bwt}(w)}^R \cdot a^{|w|_a}$.*

Proof. Since for each word w and each integer $k > 0$ we have $r_{\text{bwt}}(w) = r_{\text{bwt}}(w^k)$, let us assume that w is a primitive word. From Lemma 19, the case $\chi = E$ is trivial: in fact, from it follows that $\text{bwt}(E(w)) = \overline{\text{bwt}(w)}^R$, and therefore $r_{\text{bwt}}(w) = r_{\text{bwt}}(E(w))$.

For the case $\chi = \varphi$ one can observe that every b that occurs in $\varphi(w)$ is obtained from $\varphi(a)$, and therefore it is always preceded by an a . Thus, the rotations of $\varphi(w)$ left to cover are all those starting with an a , which therefore must also start with either $\varphi(a)$ or $\varphi(b)$. By Lemma 19, and by observing that $\varphi(a)$ ends with a b and $\varphi(b)$ ends with an a , we have that $\text{bwt}(\varphi(w)) = \overline{\text{bwt}(w)}^R \cdot a^{|w|_a}$. Thus, we need to check if the run of a 's at the end merges with the last symbol of $\overline{\text{bwt}(w)}^R$. This is equivalent to checking that the first symbol of $\text{bwt}(w)$ is a b , and by contradiction if the first rotation in lexicographical order is ua for some $u \in \{a, b\}^{n-1}$, then au is a conjugate of w and $au < ua$ for each binary word w , a contradiction.

For the case $\chi = \tilde{\varphi}$, one can see for any binary word $w = w_1w_2 \cdots w_n$ we have that $\varphi(w) = \varphi(w_1w_2 \cdots w_n) = av_1av_2 \cdots av_n$, where for each $i \in [1, n]$ we have $v_i = b$ if $w_i = a$, or $v_i = \varepsilon$ if $w_i = b$. On the other hand, for the same word w we have $\tilde{\varphi}(w) = \tilde{\varphi}(w_1w_2 \cdots w_n) = v_1av_2 \cdots av_n a$, where analogously to the previous case $v_i = b$ if $w_i = a$, or $v_i = \varepsilon$ if $w_i = b$. One can notice that $\varphi(w)$ and $\tilde{\varphi}(w)$ are conjugate, and the thesis follows. ◀

A graphical interpretation of Lemma 20 is shown in Figure 1.

$M(w)$	$M(\varphi(w))$	$M(\tilde{\varphi}(w))$
$abba$	a.a.ab.a.ab.a	$a.$ $a.a.ba.a.ba.$ b
$abaab$	a.ab.a.ab.ab.	a. a.ba.a.ba.b a.
$abbab$	a.ab.ab.a.a.a	$a.$ $a.ba.ba.a.a.$ b
$baabb$	ab.a.a.ab.a.a	$a.$ $ba.a.a.ba.a.$ b
$babaa$	ab.a.ab.ab.a.	a. ba.a.ba.ba. a.
$bbaba$	ab.ab.a.a.ab.	a. ba.ba.a.a.b a.
	$b.a.a.ab.a.ab.$ a	b a.a.a.ba.a.b a.
	$b.a.ab.ab.a.a.$ a	b a.a.ba.ba.a. a.
	$b.ab.a.a.ab.a.$ a	b a.ba.a.a.ba. a.

■ **Figure 1** From left to right, the BWT-matrix for the words $w = abbaba$, $\varphi(w)$, and $\tilde{\varphi}(w)$ respectively. For $M(\varphi(w))$ and $M(\tilde{\varphi}(w))$, we separate with dots the images of symbols from w . The rotations in bold of $M(\varphi(w))$ and $M(\tilde{\varphi}(w))$ correspond to the words in $\mathcal{I}_\varphi(w)$ and $\mathcal{I}_{\tilde{\varphi}}(w)$ respectively. The block of rotations in gray at the end of both $M(\varphi(w))$ and $M(\tilde{\varphi}(w))$ are in correspondence of the equal-letter run of a 's of length $|w|_a$, which occurs for every $w \in \{a, b\}^*$. One can see that $\text{bwt}(\varphi(w)) = \text{bwt}(\tilde{\varphi}(w)) = \overline{\text{bwt}(w)^R} \cdot a^{|w|_a}$.

The following theorem shows a new characterization of Sturmian morphisms.

▶ **Theorem 21.** *Let μ be a binary morphism. Then, the following are equivalent:*

1. $\Delta_\mu^+(w) = 0$ for every word w with $|\text{alph}(w)| = 2$;
2. μ is a Sturmian morphism.

Proof. By Theorem 3 and Lemma 20, all Sturmian morphisms preserve the number of BWT-runs. Conversely, suppose that μ preserves the number of BWT-runs. By Theorem 17, such a morphism must be acyclic. Let $s = \lim s_i$ be a characteristic Sturmian word. For every i , the word $\mu(s_i)$ has 2 runs in its BWT, hence it is circularly balanced (Proposition 4). Let us consider the word $\mu(s) = \lim \mu(s_i)$. It is balanced and aperiodic, since it is obtained by applying an acyclic morphism to a Sturmian word [6]. Then, $\mu(s)$ is Sturmian by using Theorem 1, whence μ is a Sturmian morphism by applying Theorem 2. ◀

5 Binary morphisms increasing r_{bwt} by a constant

The next step after characterizing Sturmian morphisms as those fixing BWT equal-letter runs on binary words, is to find other binary morphisms that increase the number of BWT-runs always by the same fixed constant. Remind that if such a constant exists, it has to be an even integer because the BWT of any binary word starts with b and ends with a .

We show that for every $k > 0$, we can find a morphism increasing the BWT-runs of any binary word by exactly $2k$. We do so by showing a family of binary morphisms that increase the BWT-runs always by 2, which then we can compose as we want. This family is formed by binary morphisms that are similar to the famous Thue–Morse morphism $\tau \equiv (ab, ba)$. The

10:10 On the Impact of Morphisms on BWT-Runs

structure of the BWT of Thue–Morse words has been studied before and it is well understood [4, 10]. We generalize such results by showing how to derive $\text{bwt}(\mu(w))$ from $\text{bwt}(w)$ for every Thue–Morse-like morphism μ and every binary word w .

► **Definition 22.** *A binary morphism is Thue–Morse-like if it has the form $\tau_{p,q} \equiv (ab^p, ba^q)$ for some $p, q > 0$.*

We prove the following proposition, which is crucial to obtain the main result of this section. Figure 2 highlights the key aspects of the proof.

► **Proposition 23.** *For every binary word w such that $\text{alph}(w) = \{a, b\}$, the I-rotations of $\tau_{p,q}(w)$ are contiguous in the BWT-matrix of $\tau_{p,q}(w)$, and their last letters spell $\overline{\text{bwt}(w)}$.*

Proof. Let w be a binary word of length n such that $\text{alph}(w) = \{a, b\}$. Observe that $\tau_{p,q} \equiv (ab^p, ba^q)$ is abelian order-preserving, so the I-rotations of $\tau_{p,q}(w)$ maintain their relative order. Because $\tau_{p,q}(a)$ ends with b and $\tau_{p,q}(b)$ ends with a , if we consider only the I-rotations of $\tau_{p,q}(w)$ and take the last letter of each, we obtain $\overline{\text{bwt}(w)}$, which starts with a and ends with b . It remains to show that all the I-rotations of $\tau_{p,q}(w)$ are contiguous in its BWT-matrix.

If $p > 1$, each S-rotation starting with a , has to start either with $a^i b$ for some $2 \leq i \leq q+1$, or with aba^q , and both of these prefixes are smaller than ab^p . If $p = 1$, an S-rotation starting with a is smaller than the word $a(ba^q)^{n-1}ba^{q-1}$, which is smaller than a rotation having $(ab)^i ba$ as a prefix for some $0 < i < n$. The I-rotations that start with a have prefixes of such type. In both cases, we obtain that the S-rotations starting with a are smaller than the I-rotations starting with a . A symmetric argument shows that S-rotations starting with b are greater than the I-rotations starting with b . Thus, the I-rotations are contiguous and the thesis holds. ◀

Now we are ready to show that Thue–Morse-like morphisms increase the number of BWT-runs of binary words always by 2.

► **Lemma 24.** *For every binary word w such that $\text{alph}(w) = \{a, b\}$, it holds that*

$$\text{bwt}(\tau_{p,q}(w)) = b^{|w|_b} a^{(q-1)|w|_b} \cdot \overline{\text{bwt}(w)} \cdot b^{(p-1)|w|_a} a^{|w|_a},$$

and that $r_{\text{bwt}}(\tau_{p,q}(w)) = r_{\text{bwt}}(w) + 2$.

Proof. We show that the block of $\text{bwt}(\tau_{p,q}(w))$ that corresponds to the S-rotations starting with the letter a is equal to $b^{|w|_b} a^{(q-1)|w|_b}$. If $q = 1$, all the S-rotations starting with a end with the letter b . If $q > 1$, the only S-rotations that start with a and end with b have as a prefix either $a^{q+1}b$ or $a^q ba^q$. The smallest S-rotation starting with a and ending with a starts with $a^q b^p ab$ or $a^q b^p ba$. Hence, S-rotations starting with a and ending with b appear before those ending with a .

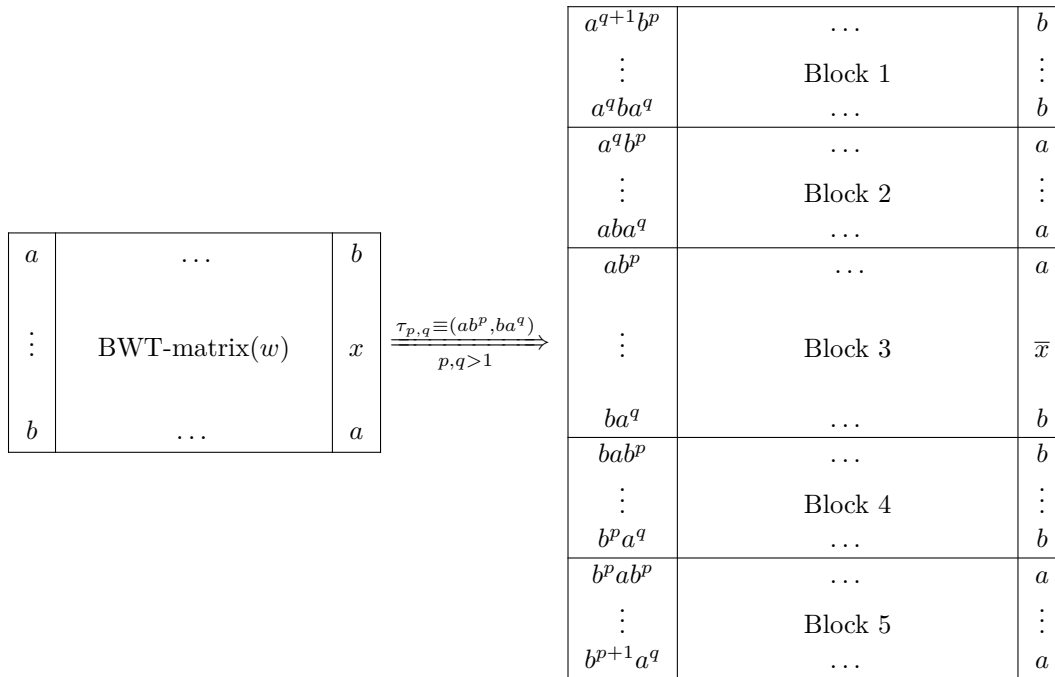
It follows that the block of $\text{bwt}(\tau_{p,q}(w))$ defined by the S-rotations starting with a spells $b^{|w|_b} a^{(q-1)|w|_b}$, because of their order, and because each of these rotations is in correspondence with some specific a inside $\tau_{p,q}(b)$ for some specific b of w . Only one of these a 's per image produces a rotation ending with b , and the other $q - 1$ a 's yield rotations ending with a .

Showing that the block of $\text{bwt}(\tau_{p,q}(w))$ corresponding to the S-rotations starting with the letter b equals $b^{(p-1)|w|_a} a^{|w|_a}$ is handled symmetrically.

By using Proposition 23, we obtain

$$\text{bwt}(\tau_{p,q}(w)) = b^{|w|_b} a^{(q-1)|w|_b} \cdot \overline{\text{bwt}(w)} \cdot b^{(p-1)|w|_a} a^{|w|_a}.$$

As $\overline{\text{bwt}(w)}$ starts with a and ends with b , we have that $r_{\text{bwt}}(\tau_{p,q}(w)) = r_{\text{bwt}}(w) + 2$, and the thesis holds. ◀



■ **Figure 2** Scheme showing the action of a Thue–Morse-like morphism $\tau_{p,q} \equiv (ab^p, ba^q)$ with $p, q > 1$ on a binary word w with $\text{alph}(w) = \{a, b\}$. At the left is the BWT-matrix of w . At the right is the BWT-matrix of $\tau_{p,q}(w)$. The cases where $p = 1$ or $q = 1$ are similar with Block 2 or Block 4 omitted.

As a consequence of Theorem 21 and Lemma 24, we obtain the following corollary.

► **Corollary 25.** *Given a non-negative even integer $2t$, there exists a binary morphism μ such that $\Delta_\mu^+(w) = 2t$ and $\Delta_\mu^\times(w) \leq t + 1$, for every word w with $|\text{alph}(w)| = 2$.*

Proof. We can construct the morphism $\mu \in (\{E, \varphi, \bar{\varphi}\} \cup \{(ab^p, ba^q) \mid p, q > 0\})^*$ such that μ is obtained by composing, in any order, exactly t morphisms taken in the set $\{(ab^p, ba^q) \mid p, q > 0\}$ and an arbitrary number of Sturmian morphisms. By Theorem 21 and Lemma 24, it holds that $\Delta_\mu^+(w) = 2t$. The value of the function $\Delta_\mu^\times(w) = (r_{\text{bwt}}(w) + 2t) / r_{\text{bwt}}(w) = 1 + 2t / r_{\text{bwt}}(w)$ is maximized when $r_{\text{bwt}}(w) = 2$. This maximum is $\Delta_\mu^\times(w) = t + 1$. ◀

We conclude this section by showing a simple algorithm that allows us to construct an arbitrarily large family of words w_1, w_2, \dots with exactly $2t$ BWT-runs each. In Algorithm 1, a morphism μ such that $\Delta_\mu^+(w) = 2(t - 1)$ for every binary word is required. Note that Corollary 25 assures that such a morphism exists.

Moreover, each word w_i is a prefix of the next word w_{i+1} , so that the infinite word $w = \lim_{i \rightarrow \infty} w_i$ is well defined, and it is aperiodic. This is given because it holds for the (implicit) standard Sturmian words s_i for $i \in [1, k]$ being used, which are circularly balanced (i.e., $r_{\text{bwt}}(w) = 2$ on them, as reported in Proposition 4), and their limit is a characteristic Sturmian word, which is aperiodic.

■ **Algorithm 1** Algorithm for constructing words with $2t$ BWT-runs.

Require: A morphism μ with $\Delta_\mu^+(w) = 2(t-1)$. A sequence of positive integers d_1, \dots, d_k .

Ensure: A sequence of words w_1, w_2, \dots, w_k where $r_{bwt}(w_i) = 2t$ for any $1 \leq i \leq k$.

```

 $w_{-1} \leftarrow \mu(b)$ 
 $w_0 \leftarrow \mu(a)$ 
for  $i \in [1, k]$  do
     $w_i \leftarrow w_{i-1}^{d_i} w_{i-2}$ 
end for
return  $w_1, \dots, w_k$ 

```

6 Morphisms with an unbounded increase on r_{bwt}

There exist morphisms that do not behave as well as Sturmian and Thue–Morse-like morphisms with respect to r_{bwt} . If we consider an alphabet of size greater than 2, we can always find a morphism μ such that the values $\Delta_\mu^+(w)$ and $\Delta_\mu^\times(w)$ are arbitrarily large.

► **Lemma 26.** *Let $\Sigma = \{c_1, \dots, c_k, a, b\}$ with $k \geq 1$. Let $\varphi \equiv (ab, a)$ be the Fibonacci morphism. Then, $r_{bwt}(w) = k + 3$ if w belongs to $\{\varphi^{2i}(a)c_1c_2 \dots c_k \mid i \geq 1\}$.*

Proof. We prove the result by induction on $k \geq 1$. Observe that the words $\varphi^{2i}(a)$ for $i \geq 1$ are Fibonacci words ending with the letter a . It is known that in these words, if we append the letter c_1 smaller than a at the end, then the number of runs becomes 4 [31, Theorem 11]. For the inductive step, suppose that $r_{bwt}(\varphi^{2i}(a)c_1 \dots c_{k-1}) = k + 2$. When appending c_k at the end, the rotations that do not start with c_k keep their relative order, and the rotation that originally ended with c_{k-1} now ends with c_k . Hence, they define the same number of runs as before. The rotation starting with c_k can be found after the rotation starting with c_{k-1} , which does not end with b , and before the first rotation starting with a , which ends with b . Hence, the number of runs increases by 1. Thus, $r_{bwt}(\varphi^{2i}(a)c_1 \dots c_k) = k + 3$. ◀

► **Lemma 27.** *Let $\Sigma = \{c_1, \dots, c_k, a, b\}$ with $k \geq 1$. Let $\varphi \equiv (ab, a)$ be the Fibonacci morphism, and $\mu \equiv (c_1, c_2, \dots, c_k, ab, a)$ be a morphism on the alphabet Σ . Then, $r_{bwt}(w) = \Omega(\log n)$ for every $w \in \{\mu(\varphi^{2i}(a)c_1c_2 \dots c_k) \mid i \geq 1\}$.*

Proof. The morphism μ maps a Fibonacci word ending with a having $c_1 \dots c_k$ appended at the end, to the next Fibonacci word, which ends with b , having $c_1 \dots c_k$ appended at the end. For $k = 1$, it is known that the number of runs in this family is $\Omega(\log n)$ [15]. In a similar way to Lemma 26, it is possible to prove by induction that appending c_k at the end of $\mu(\varphi^{2i}(a)c_1c_2c_{k-1})$ adds 2 runs when $k = 2$ and exactly 1 new run when $k > 2$. ◀

► **Proposition 28.** *For each alphabet Σ with size greater than 2 there exist a morphism μ , satisfying that for every k , there is a word $w \in \Sigma^*$ such that $\Delta_\mu^+(w) \geq k$ and $\Delta_\mu^\times(w) \geq k$.*

Proof. This is immediate from Lemma 26 together with Lemma 27. ◀

Finding examples like the previous ones for binary morphisms is trickier, but at least in the case of Δ_μ^+ , it is possible. An example of a binary morphism for which the value $\Delta_\mu^+(w)$ can be arbitrarily large is the *period-doubling morphism* denoted by θ and defined by the rules $\theta(a) = ab$ and $\theta(b) = aa$.

► **Lemma 29.** *Let θ be the period-doubling morphism. For any positive integer k there exist a word w such that $\Delta_\theta^+(w) > k$.*

Proof. W.l.o.g assume that $k > 2$. For $i \in [2, k]$ define the words

$$s_i = ab^i a \cdot u_i \text{ and } e_i = ab^i a \cdot u_i^R, \text{ where } u_i = a^{2k-i} b a^{i-2}.$$

We say that s_i is a *starting factor*, and e_i is an *ending factor*. Observe that s_i (resp. u_i) is always smaller than e_i (resp. u_i^R). Moreover, it holds that if $i < j$, then $u_i < u_j < u_j^R < u_i^R$. We define the word $w_k = (\prod_{i=2}^k s_i e_i) a^k$ and show that $\Delta_\theta^+(w_k) = 2k$. Figure 3 shows the structure of both BWTs and highlights the increase in the number of runs.

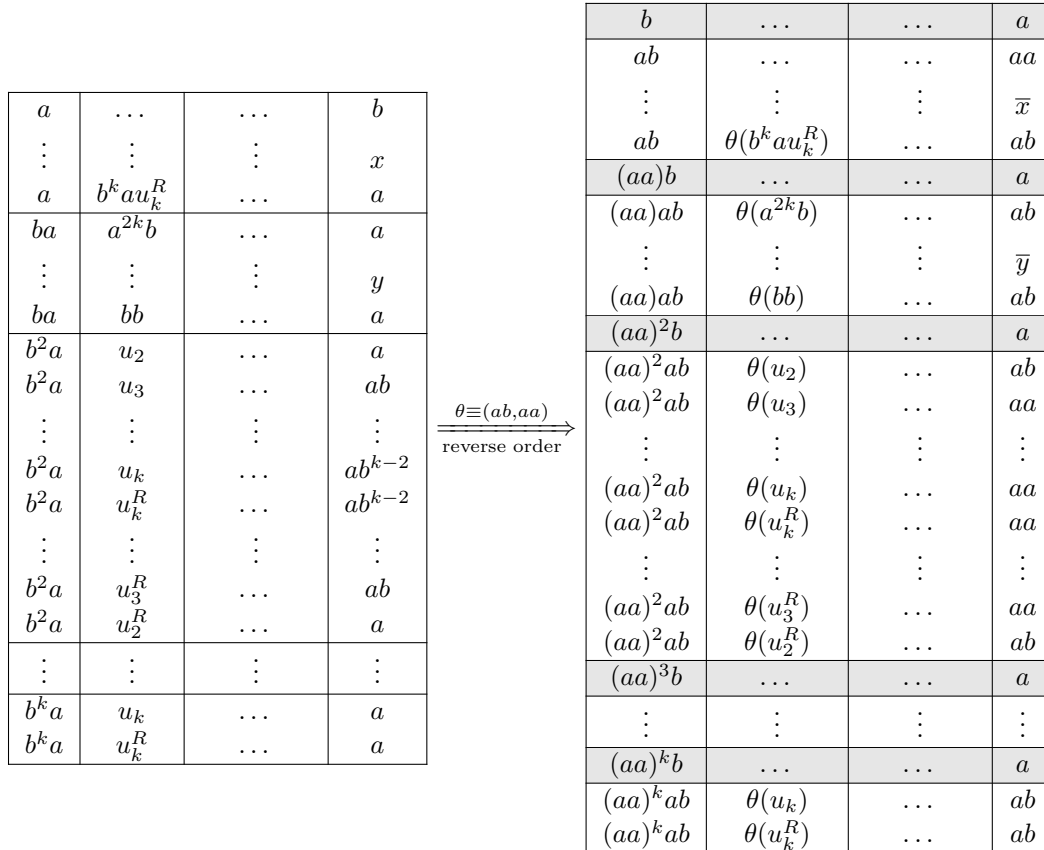


Figure 3 To the left is the BWT-matrix of w_k . To the right is the BWT-matrix of $\theta(w_k)$, here displayed in reverse order. Each gray row represents a block of rotations from $\mathcal{S}_\theta(w_k)$ starting with the same prefix, highlighted in the first column. Each one of these block except the first one yields 2 extra runs on $\text{bwt}(\theta(w_k))$. The words x and y correspond to the concatenation of the last letters of blocks of the BWT-matrix of w_k whose form is unknown, but do not play a role in the increase on $r_{\text{bwt}}(\theta(w_k))$.

Consider the rotations of w_k starting with $b^i a$ with $1 < i \leq k$. The left shift of the unique rotation starting with the i -th starting factor, and the left shift of the unique rotation starting with the i -th ending factor, are the smallest and greater, respectively. Both of them end with the letter a . The remaining rotations starting with $b^i a$ (if any) have to end with b because in them the prefix $b^i a$ corresponds to a suffix of a longer run of b 's followed by an a .

In the case of the rotations of w_k starting with ba , the one starting in the last b of e_k , has $ba^k a^k$ as a prefix, so it is the smallest of them. Also, this rotation is preceded by the factor $ab^k a a^{k-2}$, which ends in a . The greatest rotation starting with ba is the one starting with the b preceding e_2 , which is followed by abb and preceded also by an a . In the case of the rotations of w_k starting with a , the smallest of them ends with the letter b as in any binary word. The greatest is the rotation starting with $ab^k a u_k^R$ which is preceded by the letter a .

10:14 On the Impact of Morphisms on BWT-Runs

With the general structure of the BWT of w_k in mind, now we analyse the BWT of $\theta(w_k)$. The morphism θ is order-reversing and all the I-rotations of $\theta(w_k)$ start with the letter a . S-rotations of $\theta(w_k)$ starting with the letter b are always preceded by an a , and it is easy to see that this run of a 's merges with the last a in the greatest I-rotation. The S-rotations starting with an a have an even number of a 's before the first b appears, and also end with the letter a . This implies that they appear grouped after all the I-rotations of the form $(aa)^i ab$ for some $1 \leq i \leq k$, and before all the I-rotations starting with $(aa)^{i-1} ab$. As the smallest and greatest rotations of each of these blocks of I-rotations end with b (because of the action of θ), it follows that the group of S-rotations starting with $(aa)^i b$ increases the number of runs of the BWT of $\theta(w_k)$ by 2 with respect to the BWT of w_k . This happens for $1 \leq i \leq k$, so the overall increase in r_{bwt} after applying the morphism θ is exactly $2k$. ◀

From the lemma above we can deduce that there are binary morphisms that can greatly increase the number of BWT-runs of some words. We define the sensitivity of BWT-runs to morphism application in a similar way to how Akagi et al. define the sensitivity of repetitiveness measures to edit operations [1].

► **Definition 30.** *The BWT additive sensitivity and BWT multiplicative sensitivity for a morphism μ are respectively, the functions*

$$AS_\mu(n) = \max_{w \in \Sigma^n} (\Delta_\mu^+(w)) \text{ and } MS_\mu(n) = \max_{w \in \Sigma^n} (\Delta_\mu^\times(w))$$

► **Proposition 31.** *Let θ be the period-doubling morphism. It holds that $AS_\theta(n) = \Omega(\sqrt{n})$.*

Proof. The length of the words w_k in Lemma 29 is $n = \Theta(k^2)$. We showed that $\Delta_\theta^+(w_k) = 2k = \Theta(\sqrt{n})$ on these words. For values of n in between $|w_k|$ and $|w_{k+1}|$, it is easy to see that for the word $w_k a^j$ for $0 < j < |w_{k+1}| - |w_k|$, it still holds that $\Delta_\theta^+(w_k a^j) = 2k$, as none of the key aspects of the proof of Lemma 29 changes. Thus, the claim is true. ◀

7 On the impact of morphisms on other repetitiveness measures

Morphisms behave very differently when other repetitiveness measures are considered. For a general survey on repetitiveness measures see [29]. For instance, any morphism μ increases the size of the Lempel-Ziv parsing [22] of any word by at most an additive constant depending only on μ . This holds for any alphabet size, as shown by Constantinescu and Ilie [9, Lemma 8].

► **Lemma 32.** *Let $\mu : \Sigma^* \rightarrow \Gamma^*$ be any morphism. For every word w , it holds that $z(\mu(w)) \leq z(w) + k$ where k is a constant depending only on μ .*

The result of Constantinescu and Ilie can easily be extended to the LZ parsing without overlaps [22], the optimal (not the greedy) LZ-end parsing [20], and bidirectional macroschemes [38]. We can show a similar result for the measure $g(w)$ defined as the size of the smallest deterministic context-free grammar generating only w [18]. This can be further generalized to the size of the smallest run-length context-free grammar [33], and also to the size of the smallest collage system [17].

► **Lemma 33.** *Let $\mu : \Sigma^* \rightarrow \Gamma^*$ be any (possible erasing) morphism. For every word w , it holds that $g(\mu(w)) \leq g(w) + k$ where k is a constant depending only on μ .*

Proof. Given a deterministic context-free grammar G of size $|G|$ generating w , we construct a grammar generating $\mu(w)$. For each occurrence of a terminal symbol a in any rule of the grammar, replace it with a new non-terminal A_a . For each terminal symbol add the rule

$A_a \rightarrow \mu(a)$. The size of the resulting grammar is $g' \leq |G| + k$ where $k = \sum_{a \in \Sigma} |\mu(a)|$. Let G be the smallest grammar generating w , and then the thesis holds. If the resulting grammar has erasing rules, we can delete them, and replace the occurrences of those erasing variables in other variables by ε . We repeat this recursively. The size of the resulting grammar can only decrease, so the thesis still holds. ◀

If for some fixed measure and morphism, this morphism increases the value of the measure always by at most a fixed constant, then we can derive an easy upper bound for the family of words obtained by iterating that morphism.

► **Proposition 34.** *Let ρ be a repetitiveness measure and μ be a morphism. Suppose that for every word w it holds that $\rho(\mu(w)) \leq \rho(w) + k$ for a constant k depending only on ρ and μ . Then, $\rho = O(i)$ in the family $\{\mu^i(w) \mid i \geq 0\}$.*

Proof. Let $k' = \rho(\mu(w))$. We show by induction that $\rho(\mu^i(w)) \leq ki + k'$ for any $i \geq 1$. For $i = 1$, clearly $\rho(\mu(w)) \leq k + k'$. Let $i > 1$ and suppose the claim is true for $i - 1$. Then, $\rho(\mu^i(w)) \leq \rho(\mu^{i-1}(w)) + k \leq (k(i-1) + k') + k \leq ki + k'$. ◀

The families on the proposition above are known as *D0L-sequences* [36]. As a direct consequence of Lemma 33 and Proposition 34, it holds that all repetitiveness measures upper-bounded by g are $O(i)$ on the family of words belonging to a fixed D0L-sequence. In fact, the result we obtain is even more general because we can apply any morphism to words obtained from a D0L-sequence increasing the size of the grammar only by a fixed constant.

► **Proposition 35.** *For every (possibly erasing) morphisms μ and λ , and every word w , it holds that $g = O(i)$ in the family $\{\lambda \circ \mu^i(w) \mid i \geq 0\}$.*

Proof. By Lemma 33 and Proposition 34, it holds that $g(\mu^i(w)) = O(i)$ for every (possibly erasing) morphism μ . By Lemma 33, one has $g(\lambda \circ \mu^i(w)) \leq g(\mu^i(w)) + k$ for every (possibly erasing) morphism λ , and a constant k depending on λ . Thus, $g(\lambda \circ \mu^i(w)) = O(i)$. ◀

It is unknown if an analogous result is true for r_{bwt} . In fact, even for the restricted case of purely morphic words, this is known to hold only for the binary case [12].

8 Conclusions and further work

In this work, we have studied the impact of morphism application on the number of BWT equal-letter runs of finite words.

Firstly, we characterized Sturmian morphisms as the binary morphisms preserving the number of BWT-runs for all words w such that $|\text{alph}(w)| = 2$. Besides being interesting on its own, when this characterization is in conjunction with the rest of our results, it allows us to construct binary words with any possible number of BWT-runs, and morphisms with known behavior. This can have practical applications, for instance, in experimentation. In fact, we showed an infinite family of binary morphisms called Thue–Morse-like morphisms, which increase the number of BWT-runs of binary words by 2. As a consequence, we have extended the results of Brlek et al. [4] on the number of BWT-runs of words generated by iterating the composition of the Fibonacci morphism with the Thue–Morse morphism to any composition of Sturmian morphisms and Thue–Morse-like morphisms. Also, we are able to construct infinite sequences of words of increasing length, having all exactly $2k$ BWT-runs, and converging to an aperiodic infinite word at their limit. While the result on Sturmian morphisms is a complete characterization, it is unknown if the compositions of Thue–Morse-like and Sturmian morphisms are the only binary morphisms increasing the number of BWT-runs exactly by 2. The following question is left open.

► **Question 36.** *What is a sufficient and necessary condition for a binary morphism μ , to have $\Delta_\mu^+(w) \leq 2k$ (where $k > 0$) for every binary word w ?*

We showed that when the alphabet size of the domain is $\sigma > 2$, the values $\Delta_\mu^+(w)$ and $\Delta_\mu^\times(w)$ can be arbitrarily large for some morphisms. In the case of the binary alphabet, we went further and showed that there exists morphisms where $AS_\mu(w) = \Omega(\sqrt{n})$. We plan to extend such a result by studying morphisms where all the images of letters are primitive words. On the other hand, it is unknown if the value $\Delta_\mu^\times(w)$ can be unbounded for some morphism μ with binary domain. We conjecture that this is not the case.

► **Conjecture 37.** *For every morphism $\mu : \{a, b\}^* \mapsto \Sigma^*$, we can find a constant k such that $\Delta_\mu^\times(w) \leq k$, for every word $w \in \{a, b\}^*$.*

If Conjecture 37 were true, the following conjecture on images of standard Sturmian words would also be true.

► **Conjecture 38.** *For every morphism $\mu : \{a, b\}^* \mapsto \Sigma^*$ and every sequence of standard Sturmian words $(s_i)_{i \in \mathbb{N}}$, it holds that $r_{\text{bwt}}(\mu(s_i)) = \Theta(1)$.*

Finally, we showed that the impact of morphism application on BWT-runs, is quite different from the impact of morphisms on other repetitiveness measures based on popular compression schemes, like context-free grammars and LZ factorizations. In these measures, the additive increase after morphism application is bounded by a constant depending only on the morphism and the measure. This raises the following question, which is true in the case of smallest grammars and (some) variants of the LZ parsing, but unknown in the case of BWT equal-letter runs.

► **Question 39.** *Does it hold that $r_{\text{bwt}}(w) = O(i)$ when $w \in \{\lambda \circ \mu^i(a) \mid i \geq 0\}$, for every pair of morphisms $\mu : \{a, b\}^* \mapsto \{a, b\}^*$ and $\lambda : \{a, b\}^* \rightarrow \Sigma^*$?*

We are working on proving or refuting these questions and conjectures. In the future, we plan to study how to extend the results on morphism fixing BWT-runs, and morphisms increasing BWT-runs by a fixed natural number, to alphabets of size greater than 2.

References


- 1 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Inf. Comput.*, 291:104999, 2023.
- 2 Jean Berstel, Dominique Perrin, and Christophe Reutenauer. *Codes and Automata*, volume 129 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2010.
- 3 Jean Berstel and Patrice Séébold. A characterization of sturmian morphisms. In *MFCS*, volume 711 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 1993.
- 4 Srečko Brlek, Andrea Frosini, Ilaria Mancini, Elisa Pergola, and Simone Rinaldi. Burrows-wheeler transform of words defined by morphisms. In *IWOCA*, volume 11638 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2019.
- 5 Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 6 Julien Cassaigne. Sequences with grouped factors. In *DLT*, pages 211–222. Aristotle University of Thessaloniki, 1997.
- 7 Giusi Castiglione, Antonio Restivo, and Marinella Sciortino. Circular Sturmian words and Hopcroft’s algorithm. *Theor. Comput. Sci.*, 410(43):4372–4381, 2009.
- 8 Wai-Fong Chuan. Sturmian morphisms and alpha-words. *Theor. Comput. Sci.*, 225(1-2):129–148, 1999.


- 9 Sorin Constantinescu and Lucian Ilie. The lempel–ziv complexity of fixed points of morphisms. *SIAM J. Discret. Math.*, 21(2):466–481, 2007.
- 10 Maxime Crochemore, Thierry Lecroq, and Wojciech Rytter. *125 Problems in Text Algorithms: with Solutions*. Cambridge University Press, 2021.
- 11 Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000.
- 12 Andrea Frosini, Iaria Mancini, Simone Rinaldi, Giuseppe Romana, and Marinella Sciortino. Logarithmic equal-letter runs for BWT of purely morphic words. In *DLT*, volume 13257 of *Lect. Notes Comput. Sc.*, pages 139–151. Springer, 2022.
- 13 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM*, 67(1):2:1–2:54, 2020.
- 14 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello. Novel results on the number of runs of the burrows-wheeler-transform. In *SOFSEM*, volume 12607 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2021.
- 15 Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Bit catastrophes for the Burrows-Wheeler Transform. In *DLT*, volume 13911 of *Lect. Notes Comput. Sc.* Springer, 2023.
- 16 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler Transform Conjecture. *Commun. ACM*, 65(6):91–98, 2022.
- 17 Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 298(1):253–272, 2003.
- 18 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.
- 19 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
- 20 Sebastian Kreft and Gonzalo Navarro. LZ77-Like Compression with Fast Random Access. In *DCC*, pages 239–248. IEEE, 2010.
- 21 Ben Langmead and Steven Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9:357–359, March 2012.
- 22 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976.
- 23 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinform.*, 26(5):589–595, 2010.
- 24 M. Lothaire. *Algebraic Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge Univ. Press, New York, NY, USA, 2002.
- 25 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theoret. Comput. Sci.*, 698:79–87, 2017.
- 26 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows–Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003. doi:10.1016/S0020-0190(02)00512-4.
- 27 Giovanni Manzini. An analysis of the Burrows–Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
- 28 Filippo Mignosi and Patrice Séebold. Morphismes sturmiens et règles de Rauzy. *Journal de théorie des nombres de Bordeaux*, 5(2):221–233, 1993.
- 29 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.
- 30 Gonzalo Navarro. The compression power of the BWT: technical perspective. *Commun. ACM*, 65(6):90, 2022.
- 31 Gonzalo Navarro, Carlos Ochoa, and Nicola Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021.


10:18 On the Impact of Morphisms on BWT-Runs


- 32 Gonzalo Navarro and Cristian Urbina. On stricter reachable repetitiveness measures. In *SPIRE*, volume 12944 of *Lect. Notes Comput. Sci.*, pages 193–206. Springer, 2021.
- 33 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *MFCs*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72:1–72:15, 2016.
- 34 Geneviève Paquin. On a generalization of Christoffel words: epichristoffel words. *Theor. Comput. Sci.*, 410(38-40):3782–3791, 2009. doi:10.1016/j.tcs.2009.05.014.
- 35 Antonio Restivo and Giovanna Rosone. Balancing and clustering of words in the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 412(27):3019–3032, 2011. doi:10.1016/j.tcs.2010.11.040.
- 36 Grzegorz Rozenberg and Arto Salomaa. *The mathematical theory of L systems*. Academic press, 1980.
- 37 Marinella Sciortino and Luca Q. Zamboni. Suffix automata and standard sturmian words. In *DLT*, volume 4588 of *Lect. Notes Comput. Sc.*, pages 382–398. Springer, 2007.
- 38 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, !month = oct, 29(4):928–951, 1982.

Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications

Esteban Gabory 
CWI, Amsterdam, The Netherlands


Nadia Pisanti 
University of Pisa, Italy

Jakub Radoszewski 
Institute of Informatics,
University of Warsaw, Poland

Wiktor Zuba 
CWI, Amsterdam, The Netherlands

Moses Njagi Mwaniki 
University of Pisa, Italy

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

Michelle Sweering 
CWI, Amsterdam, The Netherlands

Abstract

An elastic-degenerate (ED) string T is a sequence of n sets $T[1], \dots, T[n]$ containing m strings in total whose cumulative length is N . We call n , m , and N the length, the cardinality and the size of T , respectively. The language of T is defined as $\mathcal{L}(T) = \{S_1 \cdots S_n : S_i \in T[i] \text{ for all } i \in [1, n]\}$. ED strings have been introduced to represent a set of closely-related DNA sequences, also known as a pangenome. The basic question we investigate here is: Given two ED strings, how fast can we check whether the two languages they represent have a nonempty intersection? We call the underlying problem the ED STRING INTERSECTION (EDSI) problem. For two ED strings T_1 and T_2 of lengths n_1 and n_2 , cardinalities m_1 and m_2 , and sizes N_1 and N_2 , respectively, we show the following:

- There is no $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time algorithm, thus no $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time algorithm and no $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time algorithm, for any constant $\epsilon > 0$, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis is false.
- There is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm, for any constant $\epsilon > 0$ and any function f , for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication conjecture is false.
- An $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ -time algorithm for outputting a compact (RLE) representation of the intersection language of two unary ED strings. In the case when T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
- An $\mathcal{O}(N_1 m_2 + N_2 m_1)$ -time algorithm for EDSI.
- An $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$ -time algorithm for EDSI, where ω is the exponent of matrix multiplication; the $\tilde{\mathcal{O}}$ notation suppresses factors that are polylogarithmic in the input size.

We also show that the techniques we develop have applications outside of ED string comparison.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases elastic-degenerate string, sequence comparison, languages intersection, pangenome, acronym identification

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.11

Funding The work in this paper is supported in part: by the PANGAIA and ALPACA projects that have received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively; by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003; and by PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 “Precision medicine & personalized healthcare”, funded by the European Commission under the NextGeneration EU programme.

Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.



© Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 11; pp. 11:1–11:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Sequence (or string) comparison is a fundamental task in computer science, with numerous applications in computational biology [24], signal processing [16], information retrieval [7], file comparison [25], pattern recognition [4], security [36], and elsewhere [37]. Given two or more sequences and a distance function, the task is to compare the sequences in order to infer or visualize their (dis)similarities [15].

Many sequence representations have been introduced over the years to account for *unknown* or *uncertain* letters, a phenomenon that often occurs in data that comes from experiments [8]. In the context of computational biology, for example, the IUPAC notation [27] is used to represent locations of a DNA sequence for which several alternative nucleotides are possible. This gives rise to the notion of *degenerate string* (or *indeterminate string*): a sequence of finite sets of *letters* [2]. When all sets are of size 1, we are in the special case of a *standard string* (or *deterministic string*). Degenerate strings can encode the consensus of a population of DNA sequences [17] in a gapless multiple sequence alignment (MSA). Iliopoulos et al. generalized this notion to also encode insertions and deletions (gaps) occurring in MSAs by introducing the notion of *elastic-degenerate string*: a sequence of finite sets of *strings* [26].

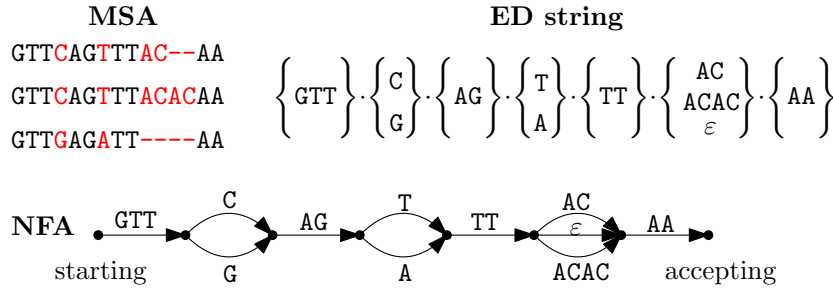
The main motivation to consider elastic-degenerate (ED) strings is that they can be used to represent a *pangenome*: a *collection* of closely-related genomic sequences that are meant to be analyzed together [42]. Several other, more powerful, pangenome representations have been proposed in the literature, mostly graph-based ones; see the comprehensive survey by Carletti et al. [12] or by Baaijens et al. [5]. Compared to these more powerful representations, ED strings have at least two algorithmic advantages, as they support: (i) fast and simple on-line string matching [23, 13]; and (ii) (deterministic) subquadratic string matching [3, 9, 10].

Our main goal here is to give the first algorithms and lower bounds for comparing two pangenomes represented by two ED strings.¹ We consider the most basic notion of matching, namely, to decide whether two ED strings, each encoding a language, have a nonempty intersection. Like with standard strings, algorithms for pairwise ED string comparison can serve as computational primitives for many analysis tasks (e.g., phylogeny reconstruction); lower bounds for pairwise ED string comparison can serve as meaningful lower bounds for more powerful pangenome representations such as, for instance, variation graphs [12].

Let us start with some basic definitions and notation. An *alphabet* Σ is a finite nonempty set of elements called *letters*. By Σ^* we denote the set of all strings over Σ including the *empty string* ε of length 0. For a string $S = S[1] \cdots S[n]$ over Σ , we call $n = |S|$ its *length*. The fragment $S[i..j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We also say that P occurs at *position* i in S . A *prefix* of S is a fragment of S of the form $S[1..j]$ and a *suffix* of S is a fragment of S of the form $S[i..n]$. An *elastic-degenerate string* (ED string, in short) T is a sequence $T = T[1] \cdots T[n]$ of n finite sets, where $T[i]$ is a subset of Σ^* . The total size of T is defined as $N = N_\varepsilon + \sum_{i=1}^n \sum_{S \in T[i]} |S|$, where N_ε is the total number of empty strings in T . By m we denote the total number of strings in all $T[i]$, i.e., $m = \sum_{i=1}^n |T[i]|$. We say that T has *length* $n = |T|$, *cardinality* m and *size* $N = ||T||$. An ED string T can be treated as a compacted nondeterministic finite automaton (NFA) with $n + 1$ states, numbered $1, \dots, n + 1$, and m transitions labeled by strings in Σ^* . State 1 is *starting* and state $n + 1$ is *accepting*. For each index $i \in [1, n]$ and string $S \in T[i]$, there is a

¹ Pangenome comparison is one of the central goals of two large EU funded projects on computational pangenomics: PANGAIA (<https://www.pangenome.eu/>) and ALPACA (<https://alpaca-itn.eu/>).

transition from state i to state $i + 1$ with label S ; inspect also Figure 1 for an example. The language $\mathcal{L}(T)$ generated by the ED string T is the language accepted by this compacted NFA. That is, $\mathcal{L}(T) = \{S_1 \cdots S_n : S_i \in T[i] \text{ for all } i \in [1, n]\}$.



■ **Figure 1** An example of an MSA and its corresponding (non-unique) ED string T of length $n = 7$, cardinality $m = 11$ and size $N = 20$, and the compacted NFA for T . The compacted NFA can be seen as a special case of an edge-labeled directed acyclic graph.

We next define the main problem in scope; inspect also Figure 2 for an example.

ED STRING INTERSECTION (EDSI)

Input: Two ED strings, T_1 of length n_1 , cardinality m_1 and size N_1 , and T_2 of length n_2 , cardinality m_2 and size N_2 .

Output: YES if $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection, NO otherwise.

ED strings	Parameters	$\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$
$T_1 = \left\{ \begin{matrix} \text{a} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{b} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{c} \\ \text{bcd} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{de} \\ \text{cde} \end{matrix} \right\}$	$n_1 = 4$ $m_1 = 8$ $N_1 = 13$	abcde accde
$T_2 = \left\{ \begin{matrix} \text{a} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{d} \\ \text{bcd} \\ \text{cc} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{e} \\ \text{de} \end{matrix} \right\}$	$n_2 = 3$ $m_2 = 6$ $N_2 = 10$	abcdde ade

■ **Figure 2** An example of two ED strings T_1 and T_2 with their parameters and the intersection of their languages. In this instance, we see that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection.

Our Results. We assume that the ED strings are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$. We make the following specific contributions:

1. In Section 2.1, we give several conditional lower bounds. In particular, we show that there is no $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time algorithm, thus no $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time algorithm and no $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time algorithm, for any constant $\epsilon > 0$, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis (SETH) [11] is false.
2. In Section 2.2, we present other conditional lower bounds. In particular, we show that there is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm, for any constant $\epsilon > 0$ and any function f , for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication (BMM) conjecture [1] is false.

3. In Section 3, we show an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ -time algorithm for outputting a compact (RLE) representation of the intersection language of two unary ED strings. In the case when T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
4. In Section 4.1, we show an $\mathcal{O}(N_1 m_2 + N_2 m_1)$ -time algorithm for EDSI.
5. In Section 4.2, we show an $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$ -time algorithm for EDSI, where ω is the matrix multiplication exponent.

Interestingly, we show that the techniques we develop here have applications outside of ED string comparison. Given a sequence $P = P_1, \dots, P_n$ of n standard strings, we define an *acronym* of P as a string $A = A_1 \cdots A_n$, where A_i is a possibly empty prefix of P_i , for all $i \in [1, n]$. In the ACRONYM GENERATION (AG) problem, we are given a set D of k strings of total length K and a sequence P of n strings of total length N , and we are asked to say YES if and only if some acronym of P belongs to D . In Section 5, we show how our techniques for EDSI can be modified to solve AG in $\mathcal{O}(nK + N)$ time.

Related Work. Apart from its applications to pangenome comparison, EDSI is interesting theoretically on its own as a special case of regular expression (regex) matching. Regex is a basic notion in formal languages and automata theory. Regular expressions are commonly used in practical applications to define search patterns. Regex matching and membership testing are widely used as computational primitives in programming languages and text processing utilities (e.g., the widely-used `agrep`). The classic algorithm for solving these problems constructs and simulates an NFA corresponding to the regex, which gives an $\mathcal{O}(MN)$ running time, where M is the length of the pattern and N is the length of the text. Unfortunately, significantly faster solutions are unknown and unlikely [6]. However, much faster algorithms exist for many special cases of the problem: dictionary matching; wildcard matching; subset matching; and the word break problem; see [6] and references therein.

Special cases of EDSI have also been studied. First, let us consider the case when both T_1 and T_2 are degenerate strings. In this case, the problem is trivial: EDSI has a positive answer if and only if for every i , $T_1[i] \cap T_2[i]$ is nonempty. Alzamel et al. [2] studied the case when T_1 and T_2 are *generalized degenerate strings*: for any $i \in [1, n_1]$ and $j \in [1, n_2]$ all strings in $T_1[i]$ have the same length $\ell_{1,i}$ and all strings in $T_2[j]$ have the same length $\ell_{2,j}$. In the case of generalized degenerate strings, they showed that deciding if $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection can be done in $\mathcal{O}(N_1 + N_2)$ time. If T_2 is a standard string, i.e., an ED string with $m_2 = n_2 = 1$, then we can resort to the results of Bernardini et al. [10] for ED string matching. In particular: there is no combinatorial algorithm² for EDSI working in $\mathcal{O}(n_1 N_2^{1.5-\epsilon} + N_1)$ time unless the BMM conjecture is false; and we can solve EDSI in $\tilde{\mathcal{O}}(n_1 N_2^{\omega-1} + N_1)$ time. Moreover, Gawrychowski et al. [21] provided a systematic study of the complexity of degenerate string comparison under different notions of matching: Cartesian tree matching; order-preserving matching; and parameterized matching.

Similar to ED strings (and to generalized degenerate strings) is the representation of pangenomes via *founder graphs*. The idea behind founder graphs is that a multiple alignment of few *founder sequences* can be used to approximate the input MSA, with the feature that each row of the MSA is a recombination of the founders. Like founder graphs, ED strings support the recombination of different rows of the MSA between consecutive columns. Unlike

² The notion of “combinatorial algorithm” is informal but widely used in the literature. Typically, we call an algorithm “combinatorial” if it does not call an oracle for ring matrix multiplication.

ED strings, for which no efficient index is probable [22] (and indeed their value is to enable fast on-line string matching), some subclasses of founder graphs are indexable, and a recent research line is devoted to constructing and indexing such structures [18, 35, 38, 39].

2 Conditional Lower Bounds

In this section, we show several conditional lower bounds for the EDSI problem. Bounds in the first group (see Section 2.1) are based on the popular Strong Exponential-Time Hypothesis (SETH) [11]; the second group of bounds (see Section 2.2) is based on another popular conjecture, the Boolean Matrix Multiplication (BMM) conjecture [1].

2.1 Lower Bounds Based on SETH

We are going to reduce the ORTHOGONAL VECTORS (OV, in short) problem to the EDSI problem. In the OV problem we are given a set $V = \{v^1, \dots, v^k\}$ of k binary vectors, each of length d , and we are asked to decide whether or not there are any two vectors in V which are orthogonal; i.e., the dot product of the two vectors is zero. The OV conjecture, implied by SETH (see [44]), is the following.

► **Conjecture 1** (OV conjecture [44]). *The OV problem for k binary vectors, each of length $d = \Theta(\log k)$, cannot be solved in $\mathcal{O}(k^{2-\epsilon})$ time, for any constant $\epsilon > 0$.*

We show the following reduction.

► **Theorem 2.** *Given any set $V = \{v^1, \dots, v^k\}$ of k binary vectors of length d , we can construct in linear time two ED strings T_1 and T_2 over a binary alphabet such that:*

- T_1 has length, cardinality, and size $\Theta(kd)$;
- T_2 has length $\Theta(\log k)$, cardinality $\Theta(k)$ and size $\Theta(kd)$; and
- V contains two orthogonal vectors if and only if T_1 and T_2 have a nonempty intersection.

Proof. Let $u^i = 1^d - v^i$ for all $i \in [1, k]$. For a length- d vector v and $j \in \{1, \dots, d\}$, by v_j we denote the j th component of v . We construct T_1 and T_2 as follows (see Example 3):³

$$T_1 = \prod_{i=1}^k \prod_{j=1}^d \{0, u_j^i\}, \quad T_2 = \prod_{i=0}^{\lfloor \log_2 k \rfloor} \{0^{d \cdot 2^i}, \varepsilon\} \cdot V \cdot \prod_{i=0}^{\lfloor \log_2 k \rfloor} \{0^{d \cdot 2^i}, \varepsilon\}.$$

We now show that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection if and only if there exists a pair of orthogonal vectors in V .

- Suppose v^a and v^b are orthogonal. Then for all $j \in [1, d]$, $v_j^b \in \{0, u_j^a\}$ and hence $v^b \in \prod_j \{0, u_j^a\}$. It follows that

$$0^{(a-1)d} v^b 0^{(k-a)d} \in 0^{(a-1)d} \prod_j \{0, u_j^a\} 0^{(k-a)d} \subseteq \mathcal{L}(T_1).$$

By decomposing $a-1 = \sum_{i \in S_{a-1}} 2^i$ and $k-a = \sum_{i \in S_{k-a}} 2^i$, where for any integer p , the set S_p contains the positions with a 1 in the binary representation of p , we find that

$$0^{(a-1)d} v^b 0^{(k-a)d} \in \prod_{i \in S_{a-1}} 0^{d \cdot 2^i} \cdot V \cdot \prod_{i \in S_{k-a}} 0^{d \cdot 2^i} \subseteq \mathcal{L}(T_2).$$

We conclude that $0^{(a-1)d} v^b 0^{(k-a)d} \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$.

³ By the \prod notation we denote a sequence of concatenations of segments in an ED string.

- Conversely, suppose that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection and consider a string $S \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$. Let v^b be the vector from V which is chosen in T_2 when constructing S . The strings in the sets of T_2 all have length divisible by d . Thus v^b starts at an index $(a-1)d+1$ of string S for some integer a . Since $S \in \mathcal{L}(T_1)$, we have $v^b \in \prod_{j=1}^d \{0, u_j^a\}$. This implies that v^a and v^b are orthogonal.

Therefore, solving the orthogonal vectors problem for V is equivalent to checking whether $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection. ◀

► **Example 3.** Let $k = 3$, $d = 2$ and $V = \{v^1 = (1, 0), v^2 = (0, 1), v^3 = (1, 1)\}$.

$$\text{We have that } T_1 = \{0\} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \{0\} \cdot \{0\} \cdot \{0\}$$

$$\text{and } T_2 = \begin{Bmatrix} 00 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 0000 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 10 \\ 01 \\ 11 \end{Bmatrix} \cdot \begin{Bmatrix} 00 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 0000 \\ \varepsilon \end{Bmatrix}.$$

One can observe that each string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ corresponds to a pair of orthogonal vectors from V . For example, the string 010000 is in $\mathcal{L}(T_2)$ because $v^2 = (0, 1) \in V$. Since the vector $v^1 = (1, 0) \in V$ is orthogonal to v^2 , one also has $010000 \in \mathcal{L}(T_1)$. This is because the two first segments of T_1 are constructed to encode any vector which is orthogonal to v^1 .

Note that when $d = \Theta(\log k)$, the length n_1 , the cardinality m_1 and the size N_1 of T_1 are $\mathcal{O}(k \log k)$, whereas T_2 has length $n_2 = \mathcal{O}(\log k)$, cardinality $m_2 = \mathcal{O}(k)$ and size $N_2 = \mathcal{O}(k \log k)$. Moreover, both ED strings are over a binary alphabet $\Sigma = \{0, 1\}$. This implies various hardness results for EDSI. For example, we can see that, for any constant $\epsilon > 0$, and an alphabet Σ of size at least 2 the problem cannot be solved in

$$\mathcal{O}((N_1 + N_2 + n_1 + n_2)^{2-\epsilon} \cdot \text{poly}(n_2))$$

time, conditional on the OV conjecture. By using the fact that $n_1 \leq m_1 \leq N_1$ and $n_2 \leq m_2 \leq N_2$, we obtain the following bounds.

- **Corollary 4.** *For any constant $\epsilon > 0$, there exists no*
- $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time
 - $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time
 - $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time
- algorithm for the EDSI problem, unless the OV conjecture is false.*

2.2 Combinatorial Lower Bounds Based on BMM Conjecture

In the TRIANGLE DETECTION (TD, in short) problem we are given three $D \times D$ Boolean matrices A, B, C and are to check if there are three indices $i, j, k \in [0, D)$ such that $A[i, j] = B[j, k] = C[k, i] = 1$. It is known that Boolean Matrix Multiplication (BMM) and TD either both have truly subcubic combinatorial algorithms or none of them do [45]. The BMM conjecture is stated as follows.

► **Conjecture 5** (BMM conjecture [1]). *Given two $D \times D$ Boolean matrices, there is no combinatorial algorithm for BMM working in $\mathcal{O}(D^{3-\epsilon})$ time, for any constant $\epsilon > 0$.*

Our construction is based on the construction of Bernardini et al. from [10] for ED string matching.

► **Theorem 6.** *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f , then there exists a truly subcubic combinatorial algorithm for TD.*

Proof. Let D be a positive integer and let A , B , and C be three $D \times D$ Boolean matrices. Further let $s \leq D$ be a positive integer to be set later. In the rest of the proof, we can assume that s divides D , up to adding α rows and columns containing only 0's to all three matrices, where α is the smallest non-negative representative of the equivalence class $-D \bmod s$.

Let us first construct an ED string $T_1 = X_1 X_2 X_3$ over a large alphabet with $n_1 = 3$, where each X_p , $p \in [1, n_1]$, contains a string for each occurrence of value 1 in A , B and C , respectively. Below i iterates over $[0, D)$, j and k over $[0, \frac{D}{s})$, and x and y iterate over $[0, s)$. Moreover, $x, y \in [0, s)$, v_i for $i \in [0, D)$, and $a, \$$ are all letters.

- If $A[i, x \cdot \frac{D}{s} + j] = 1$, then X_1 contains the string $v_i x a^j$;
- If $B[x \cdot \frac{D}{s} + j, y \cdot \frac{D}{s} + k] = 1$, then X_2 contains the string $a^{\frac{D}{s}-j} x \$ \$ y a^{\frac{D}{s}-k}$;
- If $C[y \cdot \frac{D}{s} + k, i] = 1$, then X_3 contains the string $a^k y v_i$;

The length of each string in each X_p is $\mathcal{O}(D/s)$ and the total number m_1 of strings is up to $3D^2$. Overall, $N_1 = \mathcal{O}(D^3/s)$.

We construct an ED string T_2 with $n_2 = 1$ containing the following strings:

$$P(i, x, y) = v_i x a^{\frac{D}{s}} x \$ \$ y a^{\frac{D}{s}} y v_i \quad \text{for every } x, y \in [0, s) \text{ and } i \in [0, D).$$

Each string has length $\mathcal{O}(D/s)$ and there are $m_2 = Ds^2$ strings, so $N_2 = \mathcal{O}(D^2 s)$.

We use the following fact.

► **Fact 7** ([10]). $P(i, x, y) \in \mathcal{L}(T_1)$ if and only if the following holds for some $j, k \in [0, D/s)$:

$$A[i, x \cdot \frac{D}{s} + j] = B[x \cdot \frac{D}{s} + j, y \cdot \frac{D}{s} + k] = C[y \cdot \frac{D}{s} + k, i] = 1.$$

We choose $s = \lfloor \sqrt{D} \rfloor$; then $N_1, N_2 = \mathcal{O}(D^{2.5})$ and $n_1, n_2 = \mathcal{O}(1)$. Then indeed an $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm for EDSI would yield an $\mathcal{O}(D^{3-2.5\epsilon})$ -time algorithm for the TD problem.

Note also that even though the size of the alphabet used above is $\Theta(s + D) = \Theta(D)$, we can encode all letters by equal-length binary strings blowing N_1 and N_2 up only by a factor of $\Theta(\log D)$ and, hence, obtain the same lower bound for a binary alphabet. ◀

Both m_1 and m_2 in the reduction are $\mathcal{O}(D^2)$, thus an $\mathcal{O}((m_1 + m_2)^{1.5-\epsilon} f(n_1, n_2))$ -time algorithm would yield an $\mathcal{O}(D^{3-2\epsilon})$ -time algorithm for TD; hence we obtain the following.

► **Corollary 8.** *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1^{1.2} + N_2^{1.2} + m_1^{1.5} + m_2^{1.5})^{1-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f , then there exists a truly subcubic combinatorial algorithm for TD.*

3 EDSI: The Unary Case

An ED string is called *unary* if it is over an alphabet of size 1. In this special case, if both T_1 and T_2 are over the same alphabet $\Sigma = \{a\}$, EDSI boils down to checking whether there exists any $b \geq 0$ such that a^b belongs to both $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$.

Let T be a unary ED string of length n over alphabet $\Sigma = \{a\}$. We define the *compact representation* $R(T)$ of T as the following sequence of sets of integers:

$$\forall i \in [1, n] \quad R(T)[i] = \{b_{i,1}, b_{i,2}, \dots, b_{i,m_i}\} \iff T[i] = \{a^{b_{i,1}}, a^{b_{i,2}}, \dots, a^{b_{i,m_i}}\},$$

where $b_{i,j} \geq 0$ for all $i \in [1, n]$ and $j \in [1, m_i]$, the cardinality of T is $m = \sum_{i=1}^n m_i$, and its size is $N = N_\epsilon + \sum_{i=1}^n \sum_{j=1}^{m_i} b_{i,j}$, where N_ϵ is the total number of empty strings in T .

► **Theorem 9.** *If T_1 and T_2 are unary ED strings and each is given in a compact representation, the problem of deciding whether $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty is NP-complete.*

Proof. The problem is clearly in NP, as it is enough to guess a single element for each set in both T_1 and T_2 , and then simply check if the sums match in linear time. We show the NP-hardness through a reduction from the SUBSET SUM problem, which takes n integers b_1, b_2, \dots, b_n and an integer c , and asks whether there exist $x_i \in \{0, 1\}$, for all $i \in [1, n]$, such that $\sum_{i=1}^n x_i b_i = c$. SUBSET SUM is NP-complete [30] also for non-negative integers. For any instance of SUBSET SUM, we set $R(T_1)[i] = \{b_i, 0\}$ for all $i \in [1, n]$, $n_2 = 1$ and $R(T_2)[1] = \{c\}$. Then the answer to the SUBSET SUM instance is YES if and only if $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty. ◀

In what follows, we provide an algorithm which runs in polynomial time in the size of the two unary ED strings when the latter are given uncompact.

The set $\mathcal{L}(T)$ can be represented as a set $L(T) \subset \mathbb{N}$ such that $\mathcal{L}(T) = \{a^\ell : \ell \in L(T)\}$. The set $L(T)$ will be stored as a list (without repetitions). We will show how to efficiently compute $L(T_1)$ and $L(T_2)$. Then one can compute $L(T_1) \cap L(T_2)$ in $\mathcal{O}(N_1 + N_2)$ time, which allows, in particular, to check if $L(T_1) \cap L(T_2) = \emptyset$ (which is equivalent to $\mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \emptyset$).

We show the computation for $L(T_1)$. The workhorse is an algorithm from the following Lemma 10 that allows to compute the set $L(X_1 X_2)$ of concatenation of two ED strings based on their sets $L(X_1), L(X_2)$.

► **Lemma 10.** *Let X_1 and X_2 be ED strings. Given $L(X_1)$ and $L(X_2)$ such that $t_1 = \max L(X_1)$ and $t_2 = \max L(X_2)$, we can compute $L(X_1 X_2)$ in $\mathcal{O}((t_1 + t_2) \log(t_1 + t_2))$ time.*

Proof. For two sets $A, B \subset \mathbb{N}$, by $A + B$ we denote the set $\{a + b : a \in A, b \in B\}$. We then have $L(X_1 X_2) = L(X_1) + L(X_2)$. Fast Fourier Transform (FFT) [14] can be used directly to compute $L(X_1) + L(X_2)$ in $\mathcal{O}((t_1 + t_2) \log(t_1 + t_2))$ time. ◀

► **Lemma 11.** *$L(T_1)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1)$ time.*

Proof. We apply the recursive algorithm described in Algorithm 1 to T_1 .

■ **Algorithm 1** Compute- $L(T[1] \cdots T[k])$.

```

if  $k = 1$  then
    Compute  $L(T[1])$  naïvely
 $i \leftarrow \lfloor k/2 \rfloor$ 
 $L_1 \leftarrow$  Compute- $L(T[1] \cdots T[i])$ 
 $L_2 \leftarrow$  Compute- $L(T[i+1] \cdots T[k])$ 
return  $L_1 + L_2$ 
    
```

Let $N_{1,i} = \sum_{x \in L(T_1)[i]} x$ and $t_{1,i} = \max L(T_1[i])$ for $i \in [1, n_1]$. Obviously, $t_{1,i} \leq N_{1,i}$.

We analyze the complexity of the recursion by levels. For the bottom level, $L(T_1[i])$ can be computed in $\mathcal{O}(N_{1,i})$ time for each $i \in [1, n_1]$, which sums up to $\mathcal{O}(N_1)$. For the remaining levels, we notice that $\max L(T_1[i] \cdots T_1[j]) = t_{1,i} + \dots + t_{1,j}$. On each level, the fragments of T_1 that are considered are disjoint. Thus, the complexity on each level via Lemma 10 is $\mathcal{O}((\sum_{i=1}^{n_1} t_{1,i}) \log(\sum_{i=1}^{n_1} t_{1,i})) = \mathcal{O}(N_1 \log N_1)$. The number of levels of recursion is $\mathcal{O}(\log n_1)$; the complexity follows. ◀

► **Theorem 12.** *If T_1 and T_2 are unary ED strings, then $L(T_1) \cap L(T_2)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ time.*

Proof. We use Lemma 11 to compute $L(T_1)$ and $L(T_2)$ in the required complexity. Then $L(T_1) \cap L(T_2)$ can be computed via bucket sort. ◀

4 EDSI: General Case

Assuming that the two ED strings, T_1 and T_2 , of total size $N_1 + N_2$ are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$, we can sort the suffixes of all strings in $T_1[i]$, for all $i \in [1, n_1]$, and the suffixes of all strings in $T_2[j]$, for all $j \in [1, n_2]$, in $\mathcal{O}(N_1 + N_2)$ time [19].

By $\text{LCP}(X, Y)$ let us denote the length of the longest common prefix of two strings X and Y . Given a string S over an integer alphabet, we can construct a data structure over S in $\mathcal{O}(|S|)$ time, so that when $i, j \in [1, |S|]$ are given to us on-line, we can determine $\text{LCP}(S[i..|S|], S[j..|S|])$ in $\mathcal{O}(1)$ time [15].

4.1 Compacted NFA Intersection

In this section we show an algorithm for computing a representation of the intersection of the languages of two ED strings using techniques from formal languages and automata theory.

► **Definition 13** (NFA). *A nondeterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is an alphabet; $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ is a transition function, where $\mathcal{P}(Q)$ is the power set of Q ; $q_0 \in Q$ is the starting state; and $F \subseteq Q$ is the set of accepting states.*

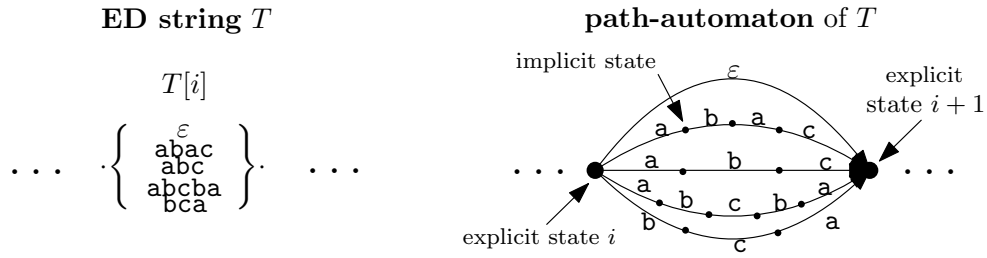
Using the folklore product automaton construction, one can check whether two NFA have a nonempty intersection in $\mathcal{O}(N_1 \cdot N_2)$ time, where N_1 and N_2 are the sizes of the two NFA [33]. We use a different, compacted representation of automata, which in some special cases allows a more efficient algorithm for computing and representing the intersection.

► **Definition 14** (Compacted NFA). *An extended transition is a transition function of the form $\delta^{ext} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$, where Q is a finite set of states, Σ^* is the set of strings over alphabet Σ , and $\mathcal{P}(Q)$ is the power set of Q . A compacted NFA is an NFA in which we allow extended transitions. Such an NFA can also be represented by a standard (uncompacted) NFA, where each extended transition is subdivided into standard one-letter transitions (and ε -transitions), $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$. The states of the compacted NFA are called explicit, while the states obtained due to these subdivisions are called implicit.*

Given a compacted NFA A with V explicit states and E extended transitions, we denote by V^u and E^u the number of states and transitions, respectively, of its uncompacted version A^u . Henceforth we assume that in the given NFA every state is reachable, and hence we have $V^u = \mathcal{O}(E^u)$ and $V = \mathcal{O}(E)$.

► **Lemma 15.** *Given two compacted NFA A_1 and A_2 , with V_1 and V_2 explicit states and E_1 and E_2 extended transitions, respectively, a compacted NFA representing the intersection of A_1 and A_2 with $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$ explicit states and $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ extended transitions can be computed in $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ time.*

Proof. We start by constructing an LCP data structure over the concatenation of all the labels of extended transitions of both NFA of total size $\mathcal{O}(E_1^u + E_2^u)$. It requires $\mathcal{O}(E_1^u + E_2^u)$ -time preprocessing and allows answering LCP queries on any two substrings of such labels in $\mathcal{O}(1)$ time.



■ **Figure 3** On the left: an ED string; on the right: the corresponding path-automaton.

We construct B a compacted NFA representing the intersection of A_1 and A_2 .

Every state of B is composed of a pair: an explicit state of one automaton and any explicit or implicit state of the other automaton (or equivalently a state of the uncompact version of the automaton). Thus the total number of explicit states of B is $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$.

We need to compute the extended transitions of B . For a state (u, v) we check every string pair (P, Q) , where P iterates over all extended transitions going out of u and Q iterates over all extended transitions going out of v (a transition going out of an implicit state is represented by a suffix of the transition it belongs to). For every pair (P, Q) we ask an $\text{LCP}(P, Q)$ query. If $\text{LCP}(P, Q)$ is equal to one of $|P|$, $|Q|$ (possibly both), we create an extended transition between (u, v) and the pair of states reachable through those transitions (if one of the transitions is strictly longer, we prune it to the right length, ending it at an implicit state of its input NFA). Otherwise such a transition does not lead to any explicit state of B and thus cannot be used to reach the accepting state; hence we ignore it.

Finally, the starting (resp. accepting) state of B corresponds to a pair of starting (resp. accepting) states of A_1 and A_2 .

Since any pair representing an explicit state of B contains an explicit state of A_1 or A_2 , the number of such transition pair checks (and hence also the number of the extended transitions of B) is $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$. Since each such check takes $\mathcal{O}(1)$ time, the construction complexity follows. Note that NFA B may contain unreachable states; such states can be removed afterwards in linear time. The algorithms' correctness follows from the observation that B^u is in fact the standard intersection automaton of A_1^u and A_2^u with some states, that do not belong to any path between the starting and the accepting states, removed. ◀

We next define the path-automaton of an ED string (inspect Figure 3 for an example).

► **Definition 16** (Path-automaton). *Let T be an ED string of length n , cardinality m , and size N . The path-automaton of T is the compacted NFA consisting of:*

- $V = n + 1$ explicit states, numbered from 1 through $n + 1$. State 1 is the starting state and state $n + 1$ is the accepting state. State $i \in [2, n]$ is the state in-between $T[i - 1]$ and $T[i]$.
- m_i extended transitions from state i to state $i + 1$ labeled with the strings in $T[i]$, for all $i \in [1, n]$, where $E = m = \sum_i m_i$.

The path-automaton of T accepts exactly $\mathcal{L}(T)$. The uncompact version of this path-automaton has $V^u = \mathcal{O}(N)$ states and $E^u = N$ transitions.

Lemma 15 thus implies the following result.

► **Corollary 17.** *The compacted NFA representing the intersection of two path-automata with $\mathcal{O}(N_1 n_2 + N_2 n_1)$ explicit states and $\mathcal{O}(N_1 m_2 + N_2 m_1)$ extended transitions can be constructed in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time.*

► **Theorem 18.** *EDSI can be solved in $\mathcal{O}(N_1m_2 + N_2m_1)$ time. If the answer is YES, a witness can be reported within the same time complexity.*

Proof. The path-automaton of an ED string of size N can be constructed in $\mathcal{O}(N)$ time. Given two ED strings, we can construct their path-automata in linear time and apply Corollary 17. By finding any path from the starting to the accepting state in linear time (if it exists), we obtain the result. ◀

Notice that the path-automata representing ED strings, as well as their intersection, are always acyclic, but may contain ε -transitions. In the following we are only interested in the graph underlying the path-automaton, that is the directed acyclic graph (DAG), where every *node* represents an explicit state and every labeled directed *edge* represents an extended transition of the path-automaton (inspect also Figure 1).

4.2 An $\tilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ -time Algorithm for EDSI

In this section, we start by showing a construction of the *intersection graph* computed by means of Lemma 15 in the case when the input is a pair of path-automata that allows an easier and more efficient implementation. The construction is then adapted to obtain an $\tilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ -time algorithm for solving the EDSI problem.

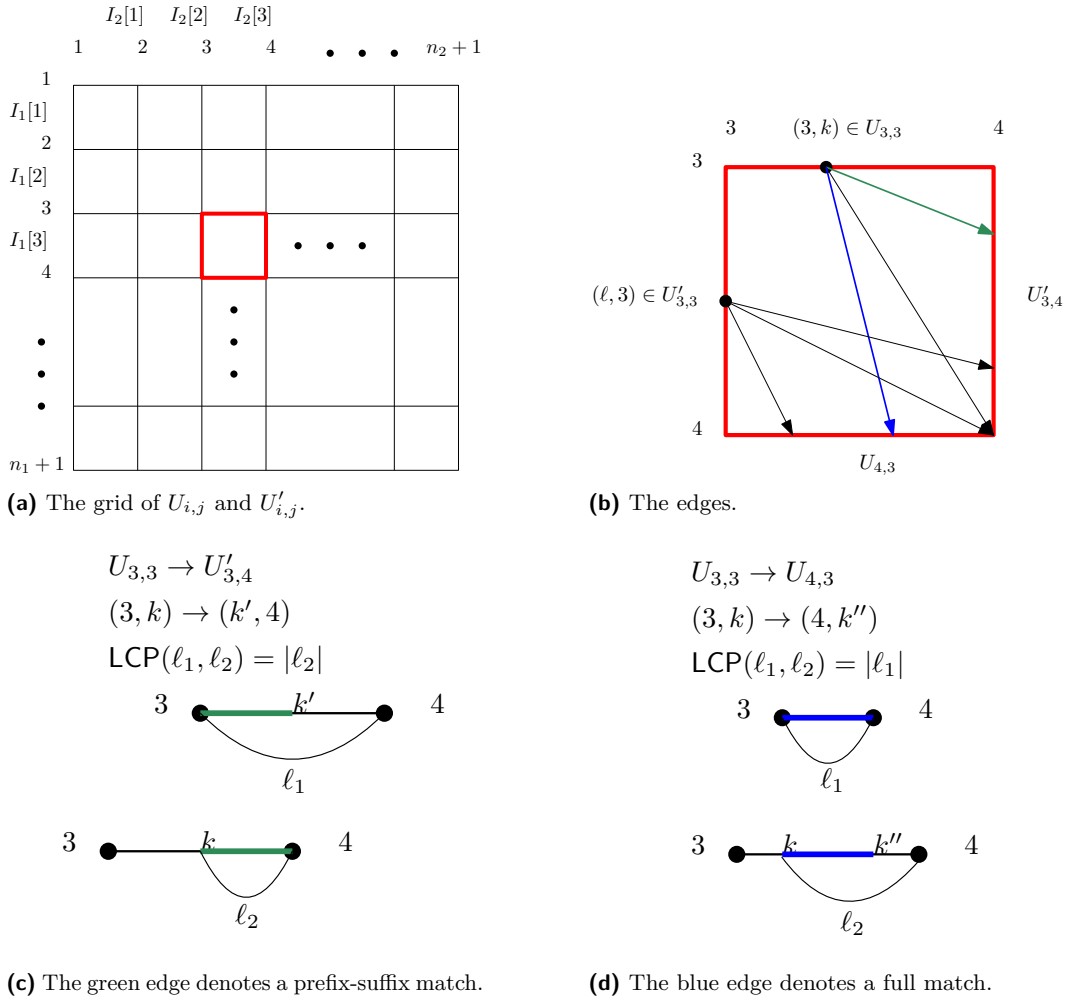
For $x \in \{1, 2\}$ by A_x we denote the compacted NFA (henceforth, graph A_x) representing the ED string T_x . By $I_x[i]$ we denote the set of implicit states (henceforth, implicit nodes) appearing on the extended transitions (henceforth, edges) between explicit states (henceforth, explicit nodes) i and $i + 1$. For convenience, the implicit nodes in the sets $I_x[1], \dots, I_x[n_x]$ can be numbered consecutively starting from $n_x + 2$.

Let $U_{i,j} = \{(i, k) : k \in \{j\} \cup I_2[j]\}$ and $U'_{i,j} = \{(k, j) : k \in \{i\} \cup I_1[i]\}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. As in the construction of Lemma 15, the union of all $U_{i,j}$ and $U'_{i,j}$ is the set of explicit nodes of the intersection graph that we construct; this can be represented graphically by a grid, where the horizontal and vertical lines correspond to $U_{i,j}$ and $U'_{i,j}$, respectively (inspect Figure 4a). In particular, we would like to compute the edges between these explicit nodes (inspect Figure 4b) in $\mathcal{O}(N_1m_2 + N_2m_1)$ time.

Consider an explicit node of the intersection graph; this node is represented by a pair of nodes: one from A_1 and one from A_2 . We need to consider two cases: explicit *vs* explicit node; or explicit *vs* implicit node. Without loss of generality, we consider the first node to be explicit. Let us denote this pair by $(i, k) \in U_{i,j}$, where i is an explicit node of A_1 and k is a node of A_2 . Let us further denote by ℓ_1 the label of one of the edges going from node i to node $i + 1$. For k , we have two cases. If k is explicit (i.e., $k = j$) then we denote by ℓ_2 the label of one of the edges going from k to $k + 1$. Otherwise (k is implicit), we denote by ℓ_2 the path label (concatenation of labels) from node k to node $j + 1$.

As noted in the proof of Lemma 15, an edge is constructed only if $\text{LCP}(\ell_1, \ell_2) = \min(|\ell_1|, |\ell_2|)$. If $\text{LCP}(\ell_1, \ell_2) = |\ell_2| < |\ell_1|$ (a prefix of a string in $T_1[i]$ is equal to the suffix of a string in $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U'_{i,j+1}$ (Figure 4c). If $\text{LCP}(\ell_1, \ell_2) = |\ell_1| < |\ell_2|$ (a whole string from $T_1[i]$ occurs in a string from $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U_{i+1,j}$ (Figure 4d). Otherwise ($\text{LCP}(\ell_1, \ell_2) = |\ell_1| = |\ell_2|$; the two strings are equal) the edge ends in $(i + 1, j + 1)$. Symmetrically (i.e., the second node is explicit), the edge going out of a node from $U'_{i,j}$ ends at a node from the same set $U'_{i,j+1} \cup U_{i+1,j} \cup \{(i + 1, j + 1)\}$ (inspect Figure 4b).

11:12 Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications



■ **Figure 4** An overview of the edges computed by the algorithm.

We next show how to construct the intersection graph by computing all such edges going out of $U_{i,j}$ or $U'_{i,j}$ in a *single batch* using suffix trees (inspect Figure 5 in Appendix A for an example). This construction allows an easier and more efficient implementation in comparison to the LCP data structure used in the general NFA intersection construction. Let us recall that $\|T\|$ denotes the size of the ED string T .

► **Lemma 19.** *For any $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$, we can construct all $K_{i,j}$ edges going out of nodes in $U_{i,j}$ in $\mathcal{O}(N_{1,i} + N_{2,j} + K_{i,j})$ time, where $N_{1,i} = \|T_1[i]\|$ and $N_{2,j} = \|T_2[j]\|$, using the generalized suffix tree of the strings in $T_2[j]$.*

Proof. We first construct the generalized suffix tree of the strings in $T_2[j]$ in $\mathcal{O}(N_{2,j})$ time [19]. We also mark each node corresponding to a suffix of a string in $T_2[j]$ with a T_2 -label. Each such node is also decorated with one or multiple starting positions, respectively, from one or multiple elements of $T_2[j]$ sharing the same suffix. For each branching node of the suffix tree, we construct a hash table, to ensure that any outgoing edge can be retrieved in constant time based on the first letter (the key) of its label. This can be done in $\mathcal{O}(N_{2,j})$ time with perfect hashing [20]. We next spell each string from $T_1[i]$ from the root of the suffix tree

making implicit nodes explicit or adding new ones if necessary to create the compacted trie of all those strings; and, finally, we mark the reached nodes of the suffix tree with a T_1 -label. Spelling all strings from $T_1[i]$ takes $\mathcal{O}(N_{1,i})$ time.

Every pair of different labels marking two nodes in an ancestor-descendant relationship corresponds to exactly one outgoing edge of the nodes in $U_{i,j}$: (i) if a node marked with a T_2 -label is an ancestor of a node marked with a T_1 -label, then the suffix of a string from $T_2[j]$ matches a prefix of a string from $T_1[i]$ forming an edge ending in $U'_{i,j+1}$; (ii) if a node marked with a T_1 -label is an ancestor of a node marked with a T_2 -label, then a string from $T_1[i]$ occurs in a string from $T_2[j]$ extending its prefix and forming an edge ending in $U_{i+1,j}$; (iii) if a node is marked with a T_1 -label and with a T_2 -label, then the suffix of a string from $T_2[j]$ matches a string from $T_1[i]$ forming an edge ending in $(i+1, j+1)$. After constructing the generalized suffix tree of $T_2[j]$ and spelling the strings from $T_1[i]$, it suffices to make a DFS traversal on the annotated tree to output all $K_{i,j}$ such pairs of nodes. ◀

► **Theorem 20.** *We can construct the intersection graph of T_1 and T_2 in $\mathcal{O}(N_1m_2 + N_2m_1)$ time using the suffix tree data structure and tree search traversals.⁴*

Proof. We apply Lemma 19 for $U_{i,j}$ and $U'_{i,j}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. We have that the total number of nodes is $\sum_{i,j} \mathcal{O}(N_{1,i} + N_{2,j}) = \mathcal{O}(N_1n_2 + N_2n_1)$, and then the sum of all output edges is bounded by $\mathcal{O}(N_1m_2 + N_2m_1)$ by Corollary 17. ◀

Note that if we are interested only in checking whether the intersection is nonempty, and not in the computation of its graph representation, it suffices to check which of the nodes are *reachable* from the starting node, which may be more efficient as there are $\mathcal{O}(N_1n_2 + N_2n_1)$ explicit nodes in this graph.

Let X be the set of nodes of $U_{i,j}$ that are reachable from the starting node. From this set of nodes we need to compute two types of edges (inspect Figure 4b). The first type of edges, namely, the ones from X to $U'_{i,j+1} \cup \{(i+1, j+1)\}$ (green edges in Figure 4b) are computed by means of Lemma 21, which is similar to Lemma 19. For the second type of edges, namely, the ones from X to $U_{i+1,j} \cup \{(i+1, j+1)\}$ (blue edges in Figure 4b), we use a reduction to the so-called *active prefixes extension* problem [10] (Lemma 23).

► **Lemma 21.** *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U'_{i,j+1} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\mathcal{O}(N_{1,i} + N_{2,j})$ time.*

Proof. In Lemma 19, the edges from nodes of $U_{i,j}$ to nodes of $U'_{i,j+1}$ come from a pair of nodes in the generalized suffix tree of $T_2[j]$: one marked with a T_1 -label and its ancestor marked with a T_2 -label. Notice that the T_2 -labels are in a correspondence with the elements of $U_{i,j}$ (the labels on a proper suffix of a string in T_2 are in a one-to-one correspondence with $U_{i,j} \setminus \{(i, j)\}$, and (i, j) corresponds to whole strings in $T_2[j]$), and hence we can trivially remove the T_2 -labels that do not correspond to the elements of X . Furthermore, we are not interested in the set of starting positions decorating a node with a T_2 -label; we are interested only in whether a node is T_2 -labeled or not (i.e., we do not care from which node of X the edge originates). Since the nodes marked with a T_1 -label have in total $N_{1,i}$ ancestors (including duplicates), we can compute the result of this case in $\mathcal{O}(N_{1,i} + N_{2,j})$ time in total. Finally, the node $(i+1, j+1)$ is reachable when a single node is marked with both a T_1 -label and a T_2 -label. This can be checked within the same time complexity. ◀

⁴ Our implementation of this algorithm can be found at <https://github.com/urbanslug/junctions>.

11:14 Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications

The remaining edges (blue edges in Figure 4b) are dealt with via a reduction to the following problem:

ACTIVE PREFIXES EXTENSION (APE)

Input: A string P of length m , a bit vector W of size m , and a set \mathcal{S} of strings of total length N .

Output: A bit vector V of size m with $V[p] = 1$ if and only if there exists $P' \in \mathcal{S}$ and $p' \in [1, m]$, such that $P[1..p' - 1] \cdot P' = P[1..p - 1]$ and $W[p'] = 1$.

Bernardini et al. have shown the following result in [10].

► **Lemma 22** ([10]). *The APE problem can be solved in $\tilde{\mathcal{O}}(m^{\omega-1}) + \mathcal{O}(N)$ time, where ω is the matrix multiplication exponent.*

► **Lemma 23.** *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U_{i+1,j} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\tilde{\mathcal{O}}(N_{1,i} + N_{2,j}^{\omega-1})$ time.*

Proof. The problems of computing the subset of $U_{i+1,j}$ reachable from X and the APE problem can be reduced to one another in linear time.

For the forward reduction, let us set $\mathcal{S} = T_1[i]$ and $P = \prod_{S \in T_2[j]} \S , where $\$$ is a letter outside of the alphabet of T_1 . This means that we order the strings in $T_2[j]$, in an arbitrary but fixed way. For a single string $\$S$ (where $S \in T_2[j]$), the positions from $S[1..|S| - 1]$ correspond to the implicit nodes (along the path spelling S) of $I_2[j]$, while the position with $\$$ corresponds to the explicit node j of A_2 and the one with $S[|S|]$ to the explicit node $j + 1$ of A_2 . Through this correspondence, we can construct two bit vectors W and V , each of them of size $|P|$, and whose positions are in correspondence with $\{j\} \cup I_2[j] \cup \{j + 1\}$ (note that this correspondence is not a bijection, as the explicit nodes j and $j + 1$ have several preimages when $|T_2[j]| \geq 2$). As $U_{i,j} \cup \{(i, j + 1)\}$ and $U_{i+1,j} \cup \{(i + 1, j + 1)\}$ are copies of $\{j\} \cup I_2[j] \cup \{j + 1\}$, we use the same correspondence to match positions between W and $U_{i,j} \cup \{(i, j + 1)\}$ and between V and $U_{i+1,j} \cup \{(i + 1, j + 1)\}$. Finally, we set $W[k] = 1$ if and only if the corresponding node of $U_{i,j}$ belongs to X (for k corresponding to $(i, j + 1)$, we set $W[k] = 0$ as such a node cannot belong to X). After solving APE, we have $V[k] = 1$ for some⁵ k corresponding to a node of $U_{i+1,j} \cup \{(i + 1, j + 1)\}$ if and only if this node is reachable from X .

In more detail, observe that since $\$$ does not belong to the alphabet of T_1 , a string S from $T_1[i]$ has to match a fragment of a string from $T_2[j]$ to set $V[k]$ to 1. This happens only if additionally $W[k - |S|] = 1$; both things happen at the same time exactly when: (i) there exists a node $(i, \ell) \in X$; (ii) there exists an edge from (i, ℓ) to $(i + 1, \ell')$; and (iii) the positions $k - |S|$ and k in P correspond to ℓ, ℓ' , respectively.

In the above reduction we have $|P| = \sum_{S \in T_2[j]} |S| + 1 = \mathcal{O}(N_{2,j})$, and $|\mathcal{S}| = N_{1,j}$, hence the lemma statement follows by Lemma 22.

For the reverse reduction, given an instance of APE, we encode it by setting $T_1[i] = \mathcal{S}$, $T_2[j] = \{P\}$ ($N_{1,i} = |\mathcal{S}|$, $N_{2,j} = |P|$) and X containing the nodes corresponding to positions k where $W[k] = 1$ (the last element of such X is potentially $(i + 1, j)$, but we do not care about this corner case of extending the prefix which is already the full string P).

This reduction shows that a more efficient solution to the problem of finding the endpoints of edges originating in X would result in a more efficient solution to the APE problem. ◀

⁵ Here, note that if the node is $(i + 1, j)$ or $(i + 1, j + 1)$, then a corresponding k is not unique, but at least one of them satisfy $V[k] = 1$.

► **Theorem 24.** *We can solve EDSI in $\tilde{O}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ time, where ω is the matrix multiplication exponent. If the answer is YES, we can output a witness within the same time complexity.*

Proof. It suffices to set the starting node $(1, 1)$ as reachable, apply Lemmas 21 and 23, and their symmetric versions for $U'_{i,j}$, for each value of $(i, j) \in [1, n_1 + 1] \times [1, n_2 + 1]$ in lexicographical order, with X equal to the set of reachable nodes of $U_{i,j}$ (respectively of $U'_{i,j}$); and, finally, check whether node $(n_1 + 1, n_2 + 1)$ is set as reachable. We bound the total time complexity of the algorithm by:

$$\sum_{i,j} \tilde{O}(N_{1,i}^{\omega-1} + N_{2,j}^{\omega-1}) = \tilde{O}(n_2 \sum_i N_{1,i}^{\omega-1} + n_1 \sum_j N_{2,j}^{\omega-1}) \leq \tilde{O}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1).$$

If $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty, that is, if the node $(n_1 + 1, n_2 + 1)$ is set as reachable from node $(1, 1)$, then we can additionally output a witness of the intersection – a single string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ – within the same time complexity. To do that we mimic the algorithm on the graph with reversed edges. This time, however, we do not mark all of the reachable nodes; we rather choose a single one that was also reachable from $(1, 1)$ in the forward direction. This way, the marked nodes form a single path from $(1, 1)$ to $(n_1 + 1, n_2 + 1)$. The witness is obtained by reading the labels on the edges of this path. ◀

5 Acronym Generation

In this section we study a problem on standard strings. Given a sequence $P = P_1, \dots, P_n$ of n strings we define an *acronym* of P as a string $A = A_1 \dots A_n$, where A_i is a (possibly empty) prefix of P_i , $i \in [1, n]$. We next formalize the ACRONYM GENERATION problem.

ACRONYM GENERATION (AG)

Input: A set D of k strings of total length K and a sequence $P = P_1, \dots, P_n$ of n strings of total length N .

Output: YES if some acronym of P is an element of D , NO otherwise.

The AG problem is underlying real-world information systems (e.g., see <https://acronymify.com/> or <https://acronym-generator.com/>) and existing approaches rely on brute-force algorithms or heuristics to address different variants of the problem [41, 40, 32, 34, 28, 43, 29, 31]. These algorithms usually accept a sequence P of $n \leq n_{\max}$ strings, for some small integer n_{\max} , which highlights the lack of efficient exact algorithms for generating acronyms. Here we show an exact polynomial-time algorithm to solve AG for any n .

We can encode AG by means of EDSI and modify the developed methods. Let $T_1[i]$, $i \in [1, n]$, be the set of all prefixes of P_i and further let $T_2[1] = D$. By using Theorem 18 or Corollary 17 we obtain an $\mathcal{O}(\sum_i |P_i|^2 k + KN) = \mathcal{O}(N^2 k + KN)$ -time algorithm, while using Theorem 24 we obtain an $\tilde{O}(N^{2\omega-2} + K^{\omega-1}n)$ -time algorithm, for solving the AG problem.

Since, however, all elements of set $T_1[i]$ are prefixes of a single string (P_i), we can obtain a more efficient graph representation of T_1 by joining nodes i and $i + 1$ with a single path labeled with P_i , with an additional ε edge between every (implicit) node of the path and node $i + 1$. As the size of the graph for T_1 is smaller ($\mathcal{O}(N)$ nodes and edges), by using Lemma 15 we obtain an $\mathcal{O}(Nk + KN) = \mathcal{O}(NK)$ -time algorithm for solving the AG problem.

The considered ED strings have additional strong properties however. $T_1[i]$'s are not just sets of prefixes of single strings, but sets of all their prefixes, while the length n_2 of T_2 is equal to 1. By employing these two properties we obtain the following improved result.

► **Theorem 25.** *AG can be solved in $\mathcal{O}(nK + N)$ time.*

Proof. The algorithm of Theorem 24 is based on finding out which elements of sets $U_{i,j}, U'_{i,j}$ are reachable; however, since $n_2 = 1$, the sets $U'_{i,j}$ are trivialized: by definition, a node from the middle of $T_1[i]$ cannot correspond to the starting or accepting node of the graph of T_2 (reading a letter in the first graph means also moving out of the starting node in the second one), hence the only possible reachable node of $U'_{i,j}$ is the explicit node (i, j) , which is also an element of $U_{i,j}$. More formally, a reachable node $(k, 1) \in U'_{i,1}$ must be equal to $(i, 1)$ as other such nodes can only be reached using some edge with a nonempty label. By symmetry, nodes from $U'_{i,2}$ other than $(i, 2)$ are not backwards reachable from the accepting node.

In Lemma 23, to compute the reachable nodes of $U_{i+1,j}$ knowing the reachable nodes of $U_{i,j}$, fast matrix multiplication is employed (Lemma 22), but in this special case a simpler method will be more effective. Let W_k be the string read between nodes k and 2 in the path-graph of T_2 . The crucial observation is: the edges going out of node $(i, k) \in U_{i,1} \cup \{(i, 2)\}$ for $k \neq 1$ end in nodes $(i+1, k')$ for $k' \in [k, k+l]$, where $l = \text{LCP}(P_i, W_k)$ as the strings from $T_1[i]$ matching the prefix of W_k are exactly all the prefixes of P_i of length at most l .

Hence to compute the reachable subset of $U_{i+1,1} \cup \{(i+1, 2)\}$, we can handle the edges going out of $(i, 1)$ separately in $\mathcal{O}(K + |P_i|)$ time by letter comparisons, then compute the $\text{LCP}(P_i, W_k)$ for all the reachable nodes (i, k) either using the LCP data structure, or with the use of the generalized suffix tree of $T_2[1] = D$ in $\mathcal{O}(K + |P_i|)$ total time, and finally, using a sweep line approach, compute the union of the obtained intervals in $\mathcal{O}(K)$ time. We answer YES if and only if node $(n+1, 2)$ is reachable.

Over all i values this gives an algorithm running in $\sum_i \mathcal{O}(K + |P_i|) = \mathcal{O}(nK + N)$ total time. Furthermore one is allowed to choose, for each $i \in [1, n]$, the minimal length x_i of the prefix of P_i (including length $x_i = 0$ if one wants to allow empty prefixes) used in the acronym (some strings should not be completely excluded from the acronym). The only modification to the algorithm in such a generalized case is replacing intervals $[k, k+l]$ by $[k+x_i, k+l]$, which does not influence the claimed complexity. ◀

► **Corollary 26.** *If the answer to the instance of the AG problem is YES, we can output all strings in D which are acronyms of P within $\mathcal{O}(nK + N)$ time.*

Proof. In the algorithm employed by Theorem 25 the reachable nodes of $U_{i,1} \cup \{(i, 2)\}$ are found. When the node $(i+1, 2)$ is the endpoint of an edge starting in node (i, k) for $k \neq 1$, then the path of the path-graph of T_2 containing node k is an acronym of P . If node $(i+1, 2)$ is reached directly from reachable node $(i, 1)$, then the whole prefix of P_i used to do that is in D , and hence is a standalone acronym of P . If for a path neither of the two cases qualifies, then it cannot be used to reach node $(n+1, 2)$, and hence is not an acronym of P .

If the generalization with minimal lengths of prefixes is applied, then the values of i used here are restricted to $[i', n]$, where i' is the largest value of i with a restriction $x_i > 0$: node $(i'-1, 2)$ cannot have an edge to node $(i', 2)$, and hence does not lie on a path from $(1, 1)$ to $(n+1, 2)$. ◀

Let us remark that although the main focus of real-world acronym generation systems is on the natural language parsing and interpretation of acronyms, our new algorithmic solution may inspire practical improvements in such systems or further algorithmic work.

References

- 1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
- 2 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundamenta Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.
- 3 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, July 2-4, 2018 – Qingdao, China*, volume 105 of *LIPICs*, pages 9:1–9:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CPM.2018.9.
- 4 Lorraine A. K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. doi:10.1016/j.patrec.2017.01.018.
- 5 Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangénomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.
- 6 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 457–466. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.56.
- 7 Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval – the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011. URL: <http://www.mir2ed.org/>.
- 8 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.
- 9 Giulia Bernardini, Paweł 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, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPICs*, pages 21:1–21:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICALP.2019.21.
- 10 Giulia Bernardini, Paweł Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM Journal on Computing*, 51(3):549–576, 2022. doi:10.1137/20M1368033.
- 11 Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2009. doi:10.1007/978-3-642-11269-0_6.
- 12 Vincenzo Carletti, Pasquale Foggia, Erik Garrison, Luca Greco, Pierluigi Ritrovato, and Mario Vento. Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition – 12th IAPR-TC-15 International Workshop, GBRPR 2019, Tours, France, June 19-21, 2019, Proceedings*, volume 11510 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 2019. doi:10.1007/978-3-030-20081-7_23.

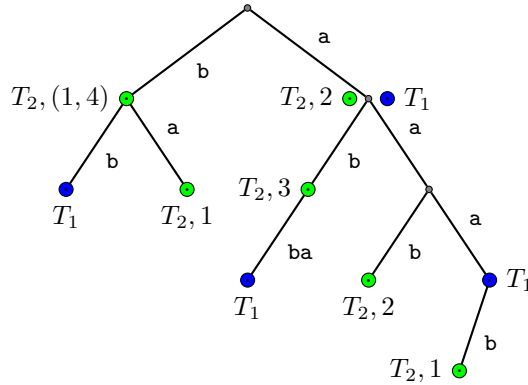
- 13 Aleksander Cisłak, Szymon Grabowski, and Jan Holub. SOPanG: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018. doi:10.1093/bioinformatics/bty506.
- 14 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 15 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
- 16 N. Rex Dixon and Thomas B. Martin. *Automatic Speech and Speaker Recognition*. John Wiley & Sons, Inc., USA, 1979.
- 17 Esteban Domingo, Carlos García-Crespo, and Celia Perales. Historical perspective on the discovery of the quasispecies concept. *Annual Review of Virology*, 8(1):51–72, 2021. PMID: 34586874. doi:10.1146/annurev-virology-091919-105900.
- 18 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing elastic founder graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPICs*, pages 20:1–20:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ISAAC.2021.20.
- 19 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS 1997, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
- 20 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.
- 21 Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate strings matching. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPICs*, pages 14:1–14:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.CPM.2020.14.
- 22 Daniel Gibney. An efficient elastic-degenerate text index? Not likely. 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 76–88. Springer, 2020. doi:10.1007/978-3-030-59212-7_6.
- 23 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, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPICs*, pages 9:1–9:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.9.
- 24 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 25 Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978. doi:10.1145/359460.359467.
- 26 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Information and Computation*, 279:104616, 2021. doi:10.1016/j.ic.2020.104616.
- 27 IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027, 1970. doi:10.1016/0022-2836(71)90319-6.

- 28 Kayla Jacobs, Alon Itai, and Shuly Wintner. Acronyms: identification, expansion and disambiguation. *Annals of Mathematics and Artificial Intelligence*, 88(5-6):517–532, 2020. doi:10.1007/s10472-018-9608-8.
- 29 Katrin Kirchhoff and Anne M. Turner. Unsupervised resolution of acronyms and abbreviations in nursing notes using document-level context models. In Cyril Grouin, Thierry Hamon, Aurélie Névél, and Pierre Zweigenbaum, editors, *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis, Louhi@EMNLP 2016, Austin, TX, USA, November 5, 2016*, pages 52–60. Association for Computational Linguistics, 2016. doi:10.18653/v1/W16-6107.
- 30 Jon M. Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.
- 31 Divesh R. Kubal and Apurva Nagvenkar. Effective ensembling of transformer based language models for acronyms identification. In Amir Pouran Ben Veyseh, Franck Deroncourt, Thien Huu Nguyen, Walter Chang, and Leo Anthony Celi, editors, *Proceedings of the Workshop on Scientific Document Understanding co-located with 35th AAAI Conference on Artificial Intelligence, SDU@AAAI 2021, Virtual Event, February 9, 2021*, volume 2831 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: <http://ceur-ws.org/Vol-2831/paper24.pdf>.
- 32 Cheng-Ju Kuo, Maurice H. T. Ling, Kuan-Ting Lin, and Chun-Nan Hsu. BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature. *BMC Bioinformatics*, 10(S-15):7, 2009. doi:10.1186/1471-2105-10-S15-S7.
- 33 Mark V. Lawson. *Finite Automata*. Chapman and Hall/CRC, 2004.
- 34 Jie Liu, Caihua Liu, and Yalou Huang. Multi-granularity sequence labeling model for acronym expansion identification. *Information Sciences*, 378:462–474, 2017. doi:10.1016/j.ins.2016.06.045.
- 35 Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. 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 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.WABI.2020.7.
- 36 Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994. doi:10.1016/0020-0190(94)00032-8.
- 37 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 38 Nicola Rizzo and Veli Mäkinen. Indexable elastic founder graphs of minimum height. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPICs*, pages 19:1–19:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.19.
- 39 Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. In Cristina Bazgan and Henning Fernau, editors, *Combinatorial Algorithms – 33rd International Workshop, IWOCA 2022, Trier, Germany, June 7-9, 2022, Proceedings*, volume 13270 of *Lecture Notes in Computer Science*, pages 480–493. Springer, 2022. doi:10.1007/978-3-031-06678-8_35.
- 40 Ariel S. Schwartz and Marti A. Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, and Teri E. Klein, editors, *Proceedings of the 8th Pacific Symposium on Biocomputing, PSB 2003, Lihue, Hawaii, USA, January 3-7, 2003*, pages 451–462, 2003. URL: <http://psb.stanford.edu/psb-online/proceedings/psb03/schwartz.pdf>.
- 41 Kazem Taghva and Jeff Gilbreth. Recognizing acronyms and their definitions. *International Journal on Document Analysis and Recognition*, 1(4):191–198, 1999. doi:10.1007/s100320050018.
- 42 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.

- 43 Amir Pouran Ben Veyseh, Franck Dernoncourt, Quan Hung Tran, and Thien Huu Nguyen. What does this acronym mean? Introducing a new dataset for acronym identification and disambiguation. In Donia Scott, Núria Bel, and Chengqing Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 3285–3301. International Committee on Computational Linguistics, 2020. doi:10.18653/v1/2020.coling-main.292.
- 44 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.
- 45 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 645–654. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.67.

A Omitted Figure

Figure 5 illustrates an example of the algorithm underlying Lemma 19.



■ **Figure 5** The annotated compacted trie constructed for $T_1[i] = \left\{ \begin{matrix} abba \\ aaa \\ bb \\ a \end{matrix} \right\}$ and $T_2[j] = \left\{ \begin{matrix} ba \\ aaab \\ b \end{matrix} \right\}$

in Lemma 19. The node corresponding to **b** has two T_2 labels and is an ancestor of the node corresponding to **bb** with a T_1 label; hence two corresponding edges to $U'_{i,j+1}$ are constructed. The node corresponding to **aaa** has a T_1 label and is an ancestor of the node corresponding to **aaab** with a T_2 label; hence a corresponding edge to $U_{i+1,j}$ is constructed. The node corresponding to **a** has both a T_1 and a T_2 label; hence a corresponding edge to $(i + 1, j + 1)$ is constructed.

Compressed Indexing for Consecutive Occurrences

Paweł Gawrychowski ✉

Institute of Computer Science, University of Wrocław, Poland

Garance Gourdel ✉

DI/ENS, PSL Research University, IRISA Inria Rennes, France

Tatiana Starikovskaya ✉

DI/ENS, PSL Research University, Paris, France

Teresa Anna Steiner ✉

DTU Compute, Technical University of Denmark, Lyngby, Denmark

Abstract

The fundamental question considered in algorithms on strings is that of indexing, that is, preprocessing a given string for specific queries. By now we have a number of efficient solutions for this problem when the queries ask for an exact occurrence of a given pattern P . However, practical applications motivate the necessity of considering more complex queries, for example concerning near occurrences of two patterns. Recently, Bille et al. [CPM 2021] introduced a variant of such queries, called gapped consecutive occurrences, in which a query consists of two patterns P_1 and P_2 and a range $[a, b]$, and one must find all consecutive occurrences (q_1, q_2) of P_1 and P_2 such that $q_2 - q_1 \in [a, b]$. By their results, we cannot hope for a very efficient indexing structure for such queries, even if $a = 0$ is fixed (although at the same time they provided a non-trivial upper bound). Motivated by this, we focus on a text given as a straight-line program (SLP) and design an index taking space polynomial in the size of the grammar that answers such queries in time optimal up to polylog factors.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Pattern matching

Keywords and phrases Compressed indexing, two patterns, consecutive occurrences

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.12

Related Version *Full Version:* <https://arxiv.org/abs/2304.00887>

Funding *Paweł Gawrychowski:* Partially supported by the Bekker programme of the Polish National Agency for Academic Exchange (PPN/BEK/2020/1/00444) and the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).

Garance Gourdel: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).

Tatiana Starikovskaya: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).

Teresa Anna Steiner: Supported by a research grant (VIL51463) from VILLUM FONDEN.

1 Introduction

In the indexing problem, the goal is to preprocess a string for locating occurrences of a given pattern. For a string of length N , structures such as the suffix tree [36] or the suffix array [31], use space linear in N and allow for answering such queries in time linear in the length of the pattern m . By now, we have multiple space- and time-efficient solutions for this problem (both in theory and in practice). We refer the reader to the excellent survey by Lewenstein [29] that provides an overview of some of the approaches and some of its extensions, highlighting its connection to orthogonal range searching.



© Paweł Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 12; pp. 12:1–12:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, from the point of view of possible applications, it is desirable to allow for more general queries than just locating an exact match of a given pattern in the preprocessed text, while keeping the time sublinear in the length of the preprocessed string. A very general query is locating a substring matching a regular expression. Very recently, Gibney and Thankachan [19] showed that if the Online Matrix-Vector multiplication conjecture holds, even with a polynomial preprocessing time we cannot answer regular expression query in sublinear time. A more reasonable and yet interesting query could concern occurrences of two given patterns that are closest to each other, or just close enough.

Preprocessing a string for queries concerning two patterns has been first studied in the context of document retrieval, where the goal is to preprocess a collection of strings. There, in *the two patterns document retrieval problem* the query consists of two patterns P_1 and P_2 , and we must report all documents containing both of them [32]. In *the forbidden pattern query problem* we must report all documents containing P_1 but not P_2 [15]. For both problems, the asymptotically fastest linear-space solutions need as much as $\Omega(\sqrt{N})$ time to answer a query, where N is the total length of all strings [23, 22]. That is, the complexity heavily depends on the length of the strings. Larsen et al. [28] established a connection between Boolean matrix multiplication and the two problems, thus providing a conditional explanation for the high $\Omega(\sqrt{N})$ query complexity. Later, Kopelowitz et al. [27] provided an even stronger argument using a connection to the 3SUM problem. Even more relevant to this paper is the question considered by Kopelowitz and Krauthgamer [26], who asked for preprocessing a string for computing, given two patterns P_1 and P_2 , their occurrences that are closest to each other. The main result of their paper is a structure constructible in $O(N^{1.5} \log^\epsilon N)$ time that answers such queries in $O(|P_1| + |P_2| + \sqrt{N} \log^\epsilon N)$, for a string of length N , for any $\epsilon > 0$. They also established a connection between Boolean matrix multiplication and this problem, highlighting a difficulty in removing the $O(\sqrt{N})$ from both the preprocessing and query time at the same time.

The focus of this paper is the recently introduced variant of the indexing problem, called *gapped indexing for consecutive occurrences*, in which a query consists of two patterns P_1 and P_2 and a range $[a, b]$, and one must find the pairs of consecutive occurrences of P_1, P_2 separated by a distance in the range $[a, b]$. Navarro and Thankachan [33] showed that for $P_1 = P_2$ there is a $O(n \log n)$ -space index with optimal query time $O(m + \text{occ})$, where $m = |P_1| = |P_2|$ and occ is the number of pairs to report, but in conclusion they noticed that extending their solution to the general case of two patterns might not be possible. Bille et al. [4] provided an evidence of hardness of the general case and established a (conditional) lower bound for gapped indexing for consecutive occurrences, by connecting its complexity to that of set intersection. This lower bound suggests that, at least for indexes of size $\tilde{O}(N)$, achieving query time better than $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$ would contradict the Set Disjointness conjecture, even if $a = 0$ is fixed. In particular, obtaining query time depending mostly on the lengths of the patterns (perhaps with some additional logarithms), arguably the whole point of string indexing, is unlikely in this case.

Motivated by the (conditional) lower bound for gapped indexing for consecutive occurrences, we consider the compressed version of this problem for query intervals $[0, b]$. For exact pattern matching, there is a long line of research devoted to designing the so-called compressed indexes, that is, indexing structures with the size being a function of the length of the compressed representation of the text, see e.g. the entry in the Encyclopedia of Algorithms [30] or the Encyclopedia of Database Systems [13]. This suggests the following research direction: can we design an efficient compressed gapped index for consecutive occurrences?

The answer of course depends on the chosen compression method. With a goal to design an index that uses very little space, we focus on the most challenging setting when the compression is capable of describing a string of exponential length (in the size of its representation). An elegant formalism for such a compression method is that of straight-line programs (SLP), which are context-free grammars describing exactly one string. SLPs are known to capture the popular Lempel–Ziv compression method up to a logarithmic factor [7, 35], and at the same time provide a more convenient interface, and in particular, allow for random access in $O(\log N)$ time [5].

By now it is known that pattern matching admits efficient indexing in SLP-compressed space. Assuming a string S of length N described by an SLP with g productions, Claude and Navarro [9] designed an $O(g)$ -space index for S that allows retrieving all occurrences of a pattern of length m in time $O(m^2 \log \log N + \text{occ} \log g)$. Recently, several results have improved the query time bound while still using a comparable $O(g \log N)$ amount of space: Claude, Navarro and Pacheco [10] showed an index with query time $O((m^2 + \text{occ}) \log g)$; Christiansen et al. [8] used strings attractors to further improve the time bound to $O(m + \text{occ} \log^c N)$; and Díaz-Domínguez et al. [12] achieved $O((m \log m + \text{occ}) \log g)$ query time.

However it is not always the case that a highly compressible string is easier to preprocess. On the negative side, Abboud et al. [1] showed that, for some problems on compressed strings, such as computing the LCS, one cannot completely avoid a high dependency on the length of the uncompressed string and that for other problems on compressed strings, such as context-free grammar parsing or RNA folding, one essentially cannot hope for anything better than just decompressing the string and working with the uncompressed representation! This is also the case for some problems related to linear algebra [2]. Hence, it was not clear to us if one can avoid a high dependency on the length of the uncompressed string in the gapped indexing for consecutive occurrences problem.

In this work, we address the lower bound of Bille et al. [4] and show that, despite the negative results by Abboud et al. [1], one can circumvent it assuming that the text is very compressible:

► **Theorem 1.** *For an SLP of size g representing a string S of length N , there is an $O(g^5 \log^5 N)$ -space data structure that maintains the following queries: given two patterns P_1, P_2 both of length $O(m)$, and a range $[0, b]$, report all occ consecutive occurrences of P_1 and P_2 separated by a distance $d \in [0, b]$. The query time is $O(m \log N + (1 + \text{occ}) \cdot \log^4 N \log \log N)$.*

While achieving $O(g)$ space and $O(m + \text{occ})$ query time would contradict the Set Disjointness conjecture by the reduction of Bille et al. [4], one might wonder if the space can be improved without increasing the query time and what is the true complexity of the problem when a is not fixed (recall that $[a, b]$ is the range limiting the distance between co-occurrences to report). While we leave improvement on space and the general case as an interesting open question, we show that in the simpler case $a = 0, b = N$ (i.e. when there is no bound on the distance between the starting positions of P_1 and P_2), our techniques do allow for $O(g^2 \log^4 N)$ space complexity, see Corollary 11¹.

Throughout the paper we assume a unit-cost RAM model of computation with word size $\Theta(\log N)$. All space complexities refer to the number of words used by a data structure.

¹ Note that the conditional lower bound of Bille et al. [4] does not hold for this simpler case.

2 Preliminaries

A *string* S of length $|S| = N$ is a sequence $S[0]S[1]\dots S[N-1]$ of characters from an alphabet Σ . We denote the *reverse* $S[N-1]S[N-2]\dots S[0]$ of S by $\text{rev}(S)$. We define $S[i\dots j]$ to be equal to $S[i]\dots S[j]$ which we call a *substring* of S if $i \leq j$ and to the empty string otherwise. We also use notations $S[i\dots j)$ and $S(i\dots j]$ which naturally stand for $S[i]\dots S[j-1]$ and $S[i+1]\dots S[j]$, respectively. We call a substring $S[0\dots i]$ a *prefix* of S and use a simplified notation $S[\dots i]$, and a substring $S[i\dots N-1]$ a *suffix* of S denoted by $S[i\dots]$. We say that X is a *substring* of S if $X = S[i\dots j]$ for some $0 \leq i \leq j \leq N-1$. The index i is called an *occurrence* of X in S .

An occurrence q_1 of P_1 and an occurrence q_2 of P_2 form a *consecutive occurrence* (*co-occurrence*) of strings P_1, P_2 in a string S if there are no occurrences of P_1, P_2 between q_1 and q_2 , formally, there should be no occurrences of P_1 in $(q_1, q_2]$ and no occurrences of P_2 in $[q_1, q_2)$. For brevity, we say that a co-occurrence is *b-close* if $q_2 - q_1 \leq b$.

An integer π is a *period* of a string S of length N , if $S[i] = S[i+\pi]$ for all $i = 0, \dots, N-1-\pi$. The smallest period of a string S is called *the period* of S . We say that S is *periodic* if the period of S is at most $N/2$. We exploit the well-known corollary of the Fine and Wilf's periodicity lemma [14]:

► **Corollary 2.** *If there are at least three occurrences of a string Y in a string X , where $|X| \leq 2|Y|$, then the occurrences of Y in X form an arithmetic progression with a difference equal to the period of Y .*

2.1 Grammars

► **Definition 3** (Straight-line program [25]). *A straight-line program (SLP) G is a context-free grammar (CFG) consisting of a set of non-terminals, a set of terminals, an initial symbol, and a set of productions, satisfying the following properties:*

- *A production consists of a left-hand side and a right-hand side, where the left-hand side is a non-terminal A and the right-hand side is either a sequence BC , where B, C are non-terminals, or a terminal;*
- *Every non-terminal is on the left-hand side of exactly one production;*
- *There exists a linear order $<$ on the non-terminals such that $A < B$ whenever B occurs on the right-hand side of the production associated with A .*

A *run-length straight-line program* (RLSLP) [34] additionally allows productions of form $A \rightarrow B^k$ for positive integers k , which correspond to concatenating k copies of B . If A is associated with a production $A \rightarrow a$, where a is a terminal, we denote $\text{head}(A) = a$, $\text{tail}(A) = \varepsilon$ (the empty string); if A is associated with a production $A \rightarrow BC$, we denote $\text{head}(A) = B$, $\text{tail}(A) = C$; and finally if A is associated with a production $A \rightarrow B^k$, then $\text{head}(A) = B$, $\text{tail}(A) = B^{k-1}$.

The *expansion* \bar{S} of a sequence of terminals and non-terminals S is the string that is obtained by iteratively replacing non-terminals by the right-hand sides in the respective productions, until only terminals remain. We say that G *represents* the expansion of its initial symbol.

► **Definition 4** (Parse tree). *The parse tree of a SLP (RLSLP) is a rooted tree defined as follows:*

- The root is labeled by the initial symbol;
- Each internal node is labeled by a non-terminal;
- If S is the expansion of the initial symbol, then the i th leaf of the parse tree is labeled by a terminal $S[i]$;
- A node labeled with a non-terminal A that is associated with a production $A \rightarrow BC$, where B, C are non-terminals, has 2 children labeled by B and C , respectively. If A is associated with a production $A \rightarrow a$, where a is a terminal, then the node has one child labeled by a .
- (RLSLP only) A node labeled with non-terminal A that is associated with a production $A \rightarrow B^k$, where B is a non-terminal, has k children, each labeled by B .

The *size* of a grammar is its number of productions. The *height* of a grammar is the height of the parse tree. We say that a non-terminal A is an *ancestor* of a non-terminal B if there are nodes u, v of the parse tree labeled with A, B respectively, and u is an ancestor of v . For a node u of the parse tree, denote by $\text{off}(u)$ the number of leaves to the left of the subtree rooted at u .

► **Definition 5** (Relevant occurrences). *Let A be a non-terminal associated with a production $A \rightarrow \text{head}(A)\text{tail}(A)$. We say that an occurrence q of a string P in \bar{A} is relevant with a split s if $q = |\overline{\text{head}(A)}| - s \leq |\overline{\text{head}(A)}| \leq q + |P| - 1$.*

For example, in Fig. 1 the occurrence $q = 3$ of $P = \text{cab}$ is a relevant occurrence in \bar{C} with a split $s = 1$ but \bar{A} contains no relevant occurrences of P .

▷ **Claim 6.** Let q be an occurrence of a string P in a string S . Consider the parse tree of an RLSLP representing S , and let w be the lowest node containing leaves $S[q], S[q+1], \dots, S[q+|P|-1]$ in its subtree, then either

1. The label A of w is associated with a production $A \rightarrow BC$, and $q - \text{off}(w)$ is a relevant occurrence in \bar{A} ; or
2. The label A of w is associated with a production $A \rightarrow B^r$ and $q - \text{off}(w) = q' + r'|\bar{B}|$ for some $0 \leq r' \leq r$, where q' is a relevant occurrence of P in \bar{A} .

Proof. Assume first that A is associated with a production $A \rightarrow BC$. We then have that the subtree rooted at the left child of w (that corresponds to \bar{B}) does not contain $S[q+|P|-1]$ and the subtree rooted at the right child of w (that corresponds to \bar{C}) does not contain $S[q]$. As a consequence, $q - \text{off}(w)$ is a relevant occurrence in \bar{A} .

Consider now the case where A is associated with a production $A \rightarrow B^r$. The leaves labeled by $S[q]$ and $S[q+|P|-1]$ belong to the subtrees rooted at different children of A . If $S[q]$ belongs to the subtree rooted at the $(r'+1)$ -th child of A , then $q' = q - \text{off}(w) - |\bar{B}| \cdot r'$ is a relevant occurrence of P in \bar{A} . ◁

► **Definition 7** (Splits). *Consider a non-terminal A of an RLSLP G . If it is associated with a production $A \rightarrow BC$, define*

$$\text{Splits}(A, P) = \text{Splits}_{\text{rev}}(A, P) = \{s : q \text{ is a relevant occurrence of } P \text{ in } \bar{A} \text{ with a split } s\}.$$

If A is associated with a rule $A \rightarrow B^k$, define

$$\begin{aligned} \text{Splits}(A, P) &= \{s : q \text{ is a relevant occurrence of } P \text{ in } \bar{A} \text{ with a split } s\}; \\ \text{Splits}_{\text{rev}}(A, P) &= \{|P| - s : q \text{ is a relevant occurrence of } \text{rev}(P) \text{ in } \text{rev}(\bar{A}) \text{ with split } s\}. \end{aligned}$$

Define $\text{Splits}(G, P)$ ($\text{Splits}_{\text{rev}}(G, P)$) to be the union of $\text{Splits}(A, P)$ ($\text{Splits}_{\text{rev}}(A, P)$) over all non-terminals A in G , and $\text{Splits}'(G, P) = \text{Splits}(G, P) \cup \text{Splits}_{\text{rev}}(G, P)$.

We need the following lemma, which can be derived from Gawrychowski et al. [18]:

► **Lemma 8.** *Let G be an SLP of size g representing a string S of length N , where $g \leq N$. There exists a Las Vegas algorithm that builds a RLSLP G' of size $g' = O(g \log N)$ of height $h = O(\log N)$ representing S in time $O(g \log N)$ with high probability. This RLSLP has the following additional property: For a pattern P of length m , we can in $O(m \log N)$ time provide a certificate that P does not occur in S , or compute the set $\text{Splits}'(G', P)$. In the latter case, $|\text{Splits}'(G', P)| = O(\log N)$.*

2.2 Compact Tries

We assume the reader to be familiar with the definition of a compact trie (see e.g. [21]). Informally, a trie is a tree that represents a lexicographically ordered set of strings. The edges of a trie are labeled with strings. We define the label $\lambda(u)$ of a node u to be the concatenation of labels on the path from the root to u and an interval $I(u)$ to be the interval of the set of strings starting with $\lambda(u)$. From the implementation point of view, we assume that a node u is specified by the interval $I(u)$. The *locus* of a string P is the minimum depth node u such that P is a prefix of $\lambda(u)$.

The standard tree-based implementation of a trie for a generic set of strings $\mathcal{S} = \{S_1, \dots, S_k\}$ takes $\Theta\left(\sum_{i=1}^k |S_i|\right)$ space. Given a pattern P of length m and $\tau > 0$ suffixes Q_1, \dots, Q_τ of P , the trie allows retrieving the ranges of strings in (the lexicographically-sorted) \mathcal{S} prefixed by Q_1, \dots, Q_τ in $O(m^2)$ time. However, in this work, we build the tries for very special sets of strings only, which allows for a much more efficient implementation based on the techniques of Christiansen et al. [8], the proof is given in Appendix A:

► **Lemma 9.** *Given an RLSLP G of size g and height h . Assume that every string in a set \mathcal{S} is either a prefix or a suffix of the expansion of a non-terminal of G or its reverse. The trie for \mathcal{S} can be implemented in space $O(|\mathcal{S}|)$ to maintain the following queries in $O(m + \tau \cdot (h + \log m))$ time: Given a pattern P of length m and suffixes Q_i of P , $1 \leq i \leq \tau$, find, for each i , the interval of strings in the (lexicographically sorted) \mathcal{S} prefixed by Q_i .*

3 Relevant, extremal, and predecessor occurrences in a non-terminal

In this section, we present a data structure that allows various efficient queries, which we will need to prove Theorem 1. We also show how it can be leveraged for an index in the simpler case of consecutive occurrences ($a = 0, b = N$). Recall that the text S is a string of length N represented by an SLP G of size g . By applying Lemma 8, we transform G into an RLSLP G' of size $g' = O(g \log N)$ and depth $h = O(\log N)$ representing S , which we fix from now on. We start by showing that G' can be processed in small space to allow multiple efficient queries:

► **Theorem 10.** *There is a $O(g^2 \log^4 N)$ -space data structure for G' that given a pattern P of length m can preprocess it in $O(m \log N + \log^2 N)$ time to support the following queries for a given non-terminal A of G' :*

1. Report the sorted set of relevant occurrences of P in \bar{A} in $O(\log N)$ time;
2. Decide whether there is an occurrence of P in \bar{A} in $O(\log N \log \log N)$ time;
3. Report the leftmost and the rightmost occurrences of P in \bar{A} , $\overline{\text{head}(A)}$, and $\overline{\text{tail}(A)}$ in $O(\log^2 N \log \log N)$ time;
4. Given a position p , find the rightmost (leftmost) occurrence $q \leq p$ ($q \geq p$) of P in \bar{A} in $O(\log^3 N \log \log N)$ time (predecessor/successor).

Before we proceed to the proof, let us derive a data structure to report all consecutive occurrences (co-occurrences) of a given pair of patterns.

► **Corollary 11.** *For an SLP of size g representing a string S of length N , there is an $O(g^2 \log^4 N)$ -space data structure that supports the following queries: given two patterns P_1, P_2 both of length $O(m)$, report all occ co-occurrences of P_1 and P_2 in S . The query time is $O(m \log N + (1 + \text{occ}) \cdot \log^3 N \log \log N)$.*

Proof. We exploit the data structure of Theorem 10 for G' . To report all co-occurrences of P_1, P_2 in S , we preprocess P_1, P_2 in $O(m \log N + \log^2 N)$ time and then proceed as follows. Suppose that we want to find the leftmost co-occurrence of P_1 and P_2 in the string $S[i \dots]$, where at the beginning $i = 0$. We find the leftmost occurrence q'_1 of P_1 with $q'_1 \geq i$ (if it exists) by a successor query on the initial symbol of G' (the expansion of which is the entire string S). Then we find the leftmost occurrence q_2 of P_2 with $q_2 \geq q'_1$ (if it exists) by a successor query and the rightmost occurrence q_1 of P_1 with $q_1 \leq q_2$ by a predecessor query. If either q'_1 or q_2 do not exist, then there are no more co-occurrences in $S[i \dots]$. Otherwise, clearly, (q_1, q_2) is a co-occurrence, and there can be no other co-occurrences starting in $S[i \dots q_2]$. In this case, we return (q_1, q_2) and set $i = q_2 + 1$. The running time of the retrieval phase is $O(\log^3 N \log \log N \cdot (\text{occ} + 1))$, since we use at most three successor/predecessor queries to either output a new co-occurrence or decide that there are no more co-occurrences. ◀

3.1 Proof of Theorem 10

The data structure consists of two compact tries T_{pre} and T_{suf} defined as follows. For each non-terminal A , we store $\text{rev}(\text{head}(A))$ in T_{pre} and $\text{tail}(A)$ in T_{suf} . We augment T_{pre} and T_{suf} by computing their heavy path decomposition:

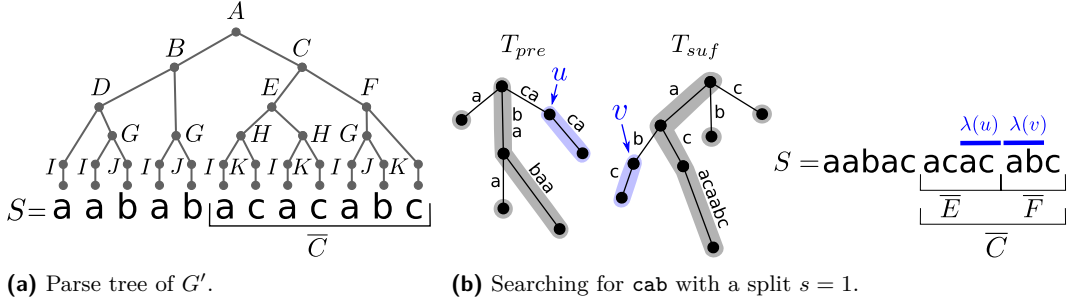
► **Definition 12.** *The heavy path of a trie T is the path that starts at the root of T and at each node v on the path branches to the child with the largest number of leaves in its subtree (heavy child), with ties broken arbitrarily. The heavy path decomposition is a set of disjoint paths defined recursively, namely it is defined to be a union of the singleton set containing the heavy path of T and the heavy path decompositions of the subtrees of T that hang off the heavy path.*

For each non-terminal A of G' , a heavy path h_{pre} in T_{pre} , and a heavy path h_{suf} in T_{suf} , we construct a multiset of points $\mathcal{P}(A, h_{pre}, h_{suf})$. For every non-terminal A' and nodes $u \in h_{pre}, v \in h_{suf}$ the multiset contains a point $(|\lambda(u)|, |\lambda(v)|)$ iff A', u, v satisfy the following properties:

1. A is an ancestor of A' ;
2. $I(u)$ contains $\overline{\text{head}(A')}$ and $I(v)$ contains $\overline{\text{tail}(A')}$.
3. u, v are the lowest nodes in h_{pre}, h_{suf} , respectively, satisfying Property 2.

(See Fig. 1.) The set $\mathcal{P}(A, h_{pre}, h_{suf})$ is stored in a two-sided 2D orthogonal range emptiness data structure [29, 6] which occupies $O(|\mathcal{P}(A, h_{pre}, h_{suf})|)$ space. Given a 2D range of the form $[\alpha, \infty] \times [\beta, \infty]$, it allows to decide whether the range contains a point in $\mathcal{P}(A, h_{pre}, h_{suf})$ in $O(\log \log N)$ time.

▷ **Claim 13.** The data structure occupies $O(g^2 \log^4 N)$ space.



■ **Figure 1** A string $S = \text{aababacacabc}$ is generated by an SLP G' . Nodes u and v are the loci of c and ab in T_{pre} and T_{suf} respectively. The heavy paths h_{pre} in T_{pre} and h_{suf} in T_{suf} are shown in blue. We have $(2, 2) \in \mathcal{P}(A, h_{pre}, h_{suf})$ corresponding to C, u, v .

Proof. Each non-terminal A' has at most g' distinct ancestors and each root-to-leaf path in T_{pre} or T_{suf} crosses $O(\log g')$ heavy paths (as each time we switch heavy paths, the number of leaves in the subtree of the current node decreases by at least a factor of two). As a corollary, each non-terminal creates $O(g' \log^2 g') = O(g \log^3 N)$ points across all orthogonal range emptiness data structures. \triangleleft

When we receive a pattern P , we compute $\text{Splits}'(G', P)$ via Lemma 8 in $O(m \log N)$ time or provide a certificate that P does not occur in S , in which case there are no occurrences of P in the expansions of the non-terminals of G' . Recall that $|\text{Splits}'(G', P)| \in O(\log N)$. We then sort $\text{Splits}'(G', P)$ in $O(\log^2 N)$ time (a technicality which will allow us reporting relevant occurrences sorted without time overhead). Finally, we compute, for each $s \in \text{Splits}'(G', P)$, the interval of strings in T_{pre} prefixed by $\text{rev}(P[\dots s])$ (which is the interval $I(u)$ for the locus u of $\text{rev}(P[\dots s])$ in T_{pre}) and the interval of strings in T_{suf} prefixed by $P(s\dots)$ (which is the interval $I(v)$ for the locus v of $P(s\dots)$ in T_{suf}). By Lemma 9, with $\tau = |\text{Splits}'(G', P)| = O(\log N)$ and $h = O(\log N)$, this step takes $O(m + \log^2 N)$ time.

Reporting relevant occurrences is easy: by definition, each relevant occurrence q of P in \overline{A} is equal to $|\overline{\text{head}(A)}| - s$ for some $s \in \text{Splits}'(G', P)$ such that $\text{rev}(P[\dots s])$ is a prefix of $\text{rev}(\overline{\text{head}(A)})$ and $P(s\dots)$ is a prefix of $\overline{\text{tail}(A)}$. As we already know the intervals of the strings in T_{suf} and T_{pre} starting with $\text{rev}(P[\dots s])$ and $P(s\dots)$, respectively, both conditions can be checked in constant time per split, or in $O(|\text{Splits}'(G', P)|) = O(\log N)$ time overall. Note that since $\text{Splits}'(G', P)$ are sorted, the relevant occurrences are reported sorted as well.

We now explain how to answer emptiness queries on a non-terminal:

\triangleright **Claim 14.** Let A be a non-terminal labeling a node in the parse tree of G' . We can decide whether \overline{A} contains an occurrence of P in $O(\log N \log \log N)$ time.

Proof. Below we show that P occurs in \overline{A} iff there exists a split $s \in \text{Splits}'(G', P)$ such that for u being the locus of $\text{rev}(P[\dots s])$ in T_{pre} and v the locus of $P(s\dots)$ in T_{suf} , for h_{pre} the heavy path containing u in T_{pre} and h_{suf} the heavy path containing v in T_{suf} , the rectangle $[|\lambda(u)|, +\infty) \times [|\lambda(v)|, +\infty)$ contains a point from $\mathcal{P}(A, h_{pre}, h_{suf})$. Before we proceed to the proof, observe that by the bound on $|\text{Splits}'(G', P)|$ this allows us to decide whether P occurs in \overline{A} in $O(\log N)$ range emptiness queries, which results in $O(\log N \log \log N)$ query time.

Assume that $[|\lambda(u)|, +\infty) \times [|\lambda(v)|, +\infty)$ contains a point $(x, y) \in \mathcal{P}(A, h_{pre}, h_{suf})$ corresponding to a non-terminal A' . By construction, A is an ancestor of A' , the subtree of u contains a leaf corresponding to $\overline{\text{head}(A')}$ and the subtree of v contains a leaf corresponding to $\overline{\text{tail}(A')}$. Consequently, $\overline{A'}$ contains an occurrence of P , which implies

that \bar{A} contains an occurrence of P . To show the reverse direction, let $\ell = \text{off}(u) + 1$ and $r = \text{off}(u) + |\bar{A}|$, i.e. $S[\ell \dots r] = \bar{A}$. The string \bar{A} contains an occurrence $\bar{A}[q \dots q + |P|]$ of P iff $S[\ell + q \dots \ell + q + |P|]$ is an occurrence of P in S . From Claim 6 it follows that if w is the lowest node in the parse tree of G' that contains leaves $S[\ell + q], \dots, S[\ell + q + |P| - 1]$ in its subtree and A' is its label, then there exists a split $s \in \text{Splits}'(G', P)$ such that $\text{rev}(P[\dots s])$ is a prefix of $\text{rev}(\text{head}(A'))$ and $P(s \dots)$ of $\text{tail}(A')$. By definition of u and v , the leaf of T_{pre} labeled with $\text{rev}(\text{head}(A'))$ belongs to $I(u)$ and the leaf of T_{suf} labeled with $\text{tail}(A')$ belongs to $I(v)$. Let h_{pre} (h_{suf}) be the heavy path in T_{pre} (T_{suf}) containing u (v) and (x, y) be the point in $\mathcal{P}(A, h_{pre}, h_{suf})$ created for A' . As $|\lambda(u)| \leq x$ and $|\lambda(v)| \leq y$, the rectangle $[|\lambda(u)|, +\infty) \times [|\lambda(v)|, +\infty)$ is not empty. \triangleleft

It remains to explain how to retrieve the leftmost/rightmost occurrences in a non-terminal, as well as to answer predecessor/successor queries. The main idea for all four types of queries is to start at any node of the parse tree of G' labeled by A and recurse down via emptiness queries and case inspection. Since the length of the expansion decreases each time we recurse from a non-terminal to its child and the height of G' is $h = O(\log N)$, this allows to achieve the desired query time. We provide full details in Appendix B.

4 Compressed Indexing for Close Co-occurrences

In this section, we show our main result, Theorem 1. Recall that S is a string of length N represented by an SLP G of size g . We start by applying Lemma 8 to transform G into an RLSLP G' of size $g' = O(g \log N)$ and height $h = O(\log N)$ representing S .

The query algorithm uses the following strategy: first, it identifies all non-terminals of G' such that their expansion contains a b -close relevant co-occurrence, where a relevant co-occurrence is defined similarly to a relevant occurrence:

► **Definition 15** (Relevant co-occurrence). *Let A be a non-terminal of G' . We say that a co-occurrence (q_1, q_2) of P_1, P_2 in \bar{A} is relevant if $q_1 \leq |\text{head}(A)| \leq q_2 + |P_2| - 1$.*

Second, it retrieves all b -close relevant co-occurrences in each of those non-terminals, and finally, reports all b -close co-occurrences by traversing the (pruned) parse tree of G' , which is possible due to the following claim:

▷ **Claim 16.** Assume that P_2 is not a substring of P_1 , and let (q_1, q_2) be a co-occurrence of P_1, P_2 in a string S . In the parse tree of G' , there exists a unique node u such that either

1. Its label A is associated with a production $A \rightarrow BC$, and $(q_1 - \text{off}(u), q_2 - \text{off}(u))$ is a relevant co-occurrence of P_1, P_2 in \bar{A} ;
2. Its label A is associated with a production $A \rightarrow B^k$, $q_1 - \text{off}(u) = q'_1 + k'|\bar{B}|$, $q_2 - \text{off}(u) = q'_2 + k'|\bar{B}|$ for some $0 \leq k' \leq k$, where (q'_1, q'_2) is a relevant co-occurrence of P_1, P_2 in \bar{A} .

Proof. Let A be the label of the lowest node u in the parse tree that contains leaves $S[q_1], S[q_1 + 1], \dots, S[q_2 + |P_2| - 1]$ in its subtree. Because P_2 is not a substring of P_1 , A cannot be associated with a production $A \rightarrow a$. By definition, $S[\text{off}(u) + 1]$ is the leftmost leaf in the subtree of this node.

Assume first that A is associated with a production $A \rightarrow BC$. We then have that the subtree rooted at the left child of u (labeled by B) does not contain $S[q_2 + |P_2| - 1]$ and the subtree rooted at the right child of u (labeled by C) does not contain $S[q_1]$. As a consequence, $(q_1 - \text{off}(u), q_2 - \text{off}(u))$ is a relevant co-occurrence of P_1, P_2 in \bar{A} .

Consider now the case where A is associated with a production $A \rightarrow B^k$. The leaves labeled by $S[q_1]$ and $S[q_2 + |P_2| - 1]$ belong to the subtrees rooted at different children of A . If $S[q_1]$ belongs to the subtree rooted at the k' -th child of A , then $(q_1 - \text{off}(u) - |\overline{B}| \cdot (k' - 1), q_2 - \text{off}(u) - |\overline{B}| \cdot (k' - 1))$ is a relevant co-occurrence of P_1, P_2 in \overline{A} . \triangleleft

4.1 Combinatorial observations

Informally, we define a set of $O(g^2)$ strings and show that for any patterns P_1, P_2 there are two strings S_1, S_2 in the set with the following property: whenever the expansion of a non-terminal A in G' contains a pair of occurrences P_1, P_2 forming a relevant co-occurrence, there are occurrences of S_1, S_2 in the proximity. This will allow us to preprocess the non-terminals of G' for occurrences of the strings in the set and use them to detect b -close relevant co-occurrences of P_1, P_2 .

Consider two tries, T_{pre} and T_{suf} : For each production of G' of the form $A \rightarrow BC$, we store \overline{C} in T_{suf} and $\text{rev}(\overline{B})$ in T_{pre} . For each production of the form $A \rightarrow B^k$, we store \overline{B} , $\overline{B^2}$, $\overline{B^{k-2}}$, and $\overline{B^{k-1}}$ in T_{suf} and the reverses of those strings in T_{pre} . For $j \in \{1, 2\}$ and $s \in \text{Splits}'(G', P_j)$ define $S_j(s) = \text{rev}(UV)$, where U is the label of the locus of $\text{rev}(P_j[\dots s])$ in T_{pre} and V is the label of the locus of $P_j(s\dots)$ in T_{suf} . Let $l_j(s) = |\text{rev}(U)|$ and $\Delta_j(s) = l_j(s) - s$.

Consider a non-terminal A such that its expansion \overline{A} contains a relevant co-occurrence (q_1, q_2) of P_1, P_2 .

\triangleright **Claim 17.** There exists $s \in \text{Splits}'(G', P_2)$ such that $p_2 = q_2 - \Delta_2(s)$ is an occurrence of $S_2(s)$ in \overline{A} and $[p_2, p_2 + |S_2(s)|] \supseteq [q_2, q_2 + |P_2|]$.

Proof. Below we show that there exists a descendant A' of A and a split $s \in \text{Splits}'(G', P_2)$ such that either $\text{rev}(P_2[\dots s])$ is a prefix of $\text{rev}(\text{head}(A'))$ and $P_2(s\dots)$ is a prefix of $\text{tail}(A')$, or A' is associated with a rule $A' \rightarrow (B')^k$, $\text{rev}(P_2[\dots s])$ is a prefix of $\text{rev}(\overline{(B')^2})$ and $P_2(s\dots)$ is a prefix of $\overline{(B')^{k-2}}$. The claim follows by the definition of T_{pre}, T_{suf} , and $S_2(s)$.

If q_2 is relevant in \overline{A} , there exists a split $s \in \text{Splits}'(G', P_2)$ such that $\text{rev}(P_2[\dots s])$ is a prefix of $\text{rev}(\text{head}(A))$ and $P_2(s\dots)$ is a prefix of $\text{tail}(A)$ by definition. If q_2 is not relevant, then $q_2 \geq |\text{head}(A)|$ by the definition of a co-occurrence. By Claim 6, there is a descendant A' of A corresponding to a substring $\overline{A}[\ell\dots r]$ for which either $(q_2 - \ell)$ is relevant (and then we can repeat the argument above), or A' is associated with a rule $A' \rightarrow (B')^k$ and $(q_2 - \ell) - k' \cdot |\overline{B'}|$ is relevant, for some $0 \leq k' \leq k$. Consider the latter case. If $A' = A$, then $k' = 1$, as otherwise $q_1 < q'_2 = q_2 - |\overline{B'}| < q_2$ is an occurrence of P_2 in \overline{A} contradicting the definition of a co-occurrence (recall that (q_1, q_2) is a relevant co-occurrence and hence by definition $q_1 < |\text{head}(A)|$), and therefore $s = \frac{|\overline{(B')^2}| - q_2 + \ell}{k} \in \text{Splits}'(G', P_2)$, $\text{rev}(P_2[\dots s])$ is a prefix of $\text{rev}(\overline{(B')^2})$ and $P_2(s\dots)$ is a prefix of $\overline{(B')^{k-2}}$. If $A' \neq A$, then we can analogously conclude that $k' = 0$, which implies $s = \frac{|\overline{B'}| - q_2 + \ell}{k} \in \text{Splits}'(G', P_2)$, $\text{rev}(P_2[\dots s])$ is a prefix of $\text{rev}(B')$ and $P_2(s\dots)$ is a prefix of $\overline{(B')^{k-1}}$. \triangleleft

As the definition of a co-occurrence is not symmetric, q_1 does not enjoy the same property. However, a similar claim can be shown:

\blacktriangleright **Lemma 18.** *There exists $s \in \text{Splits}'(G', P_1)$ and an occurrence p_1 of $S_1(s)$ in \overline{A} such that $[p_1, p_1 + |S_1(s)|] \supseteq [q_1, q_1 + |P_1|]$ and at least one of the following holds:*

1. $q_1 - \Delta_1(s)$ is an occurrence of $S_1(s)$;
2. q_2 is a relevant occurrence of P_2 in \overline{A} , the period of $S_1(s)$ equals the period π_1 of P_1 , and there exists an integer k such that $p_1 = q_1 - \Delta_1(s) - \pi_1 \cdot k$ and $q_2 + \pi_1 - 1 \leq p_1 + |S_1(s)| - 1 \leq q_2 + |P_2| - 1$.

Proof. If q_1 is a relevant occurrence of P_1 in A with a split $s \in \text{Splits}'(G', P_1)$, then $\text{rev}(P_1[\dots s])$ is a prefix of $\text{rev}(\overline{\text{head}(A)})$ and $P_1(s\dots)$ is a prefix of $\overline{\text{tail}(A)}$ and therefore the first case holds by the definition of T_{pre} and T_{suf} .

Otherwise, by Claim 6, there is a descendant A' of $\text{head}(A)$ corresponding to a substring $\overline{A}[\ell\dots r]$ for which either $(q_1 - \ell)$ is relevant (and then we can repeat the argument above), or A' is associated with a rule $A' \rightarrow (B')^k$ and $(q_1 - \ell) - k' \cdot |\overline{B'}|$, for some $0 \leq k' \leq k$, is a relevant occurrence of P_1 in $\overline{A'}$ with a split $s \in \text{Splits}'(G', P_1)$. Consider the latter case. We must have (1) $q_1 + |P_1| - 1 + |\overline{B'}| \geq r$ or (2) $q_1 + |\overline{B'}| - 1 \geq q_2$, because if both inequalities do not hold, then $q_1 < q_1 + |\overline{B'}| \leq q_2$ is an occurrence of P_1 in \overline{A} , which contradicts the definition of a co-occurrence. Additionally, if (1) holds, then by definition there exists a split $s' \in \text{Splits}'(G', P_1)$ (which might be different from the split s above) such that $\text{rev}(P_1[\dots s'])$ is a prefix of $\text{rev}(\overline{(B')^{r-1}})$ and $P_1(s'\dots)$ is a prefix of $\overline{B'}$ and we fall into the first case of the lemma.

From now on, assume that (2) holds and (1) does not. Since $q_1 + |\overline{B'}| \leq r \leq |\overline{\text{head}(A)}|$ and (q_1, q_2) is a relevant co-occurrence, q_2 must be a relevant occurrence of P_2 in \overline{A} . If $|P_1| - s \leq |\overline{(B')^2}|$, then $\text{rev}(P_1[\dots s])$ is a prefix of $\text{rev}(\overline{B'})$ and $P_1(s\dots)$ is a prefix of $\overline{(B')^2}$ and therefore $q_1 - \Delta_1(s)$ is an occurrence of $S_1(s)$. Otherwise, by Fine and Wilf's periodicity lemma [14], the periods of $\overline{A'}$, P_1 , and $S_1(s)$ are equal, since P_1 and hence $S_1(s)$ span at least two periods of $\overline{A'}$. By periodicity, $S_1(s)$ occurs at positions $q_1 - \Delta_1(s) - |\overline{B'}| \cdot k$ of \overline{A} . Let p_1 be the leftmost of these positions which satisfies $p_1 + |S_1(s)| - 1 \geq q_1 + |P_1| - 1$. This position is well-defined as (1) does not hold, and furthermore $[q_1, q_1 + |P_1|] \subseteq [p_1, p_1 + |S_1(s)|]$ as $s \leq l_1(s)$ and $|S_1(s)| - l_1(s) \geq |P_1| - s$. We have $p_1 = q_1 - \Delta_1(s) - \pi_1 \cdot k''$ for some integer k'' (as $|\overline{B'}|$ is a multiple of π_1), and $q_2 + \pi_1 - 1 \leq q_1 + 2|\overline{B'}| - 1 \leq q_1 + |P_1| - 1 \leq p_1 + |S_1(s)| - 1 \leq r < q_2 + |P_2| - 1$, where the last inequality holds as q_2 is a relevant occurrence in \overline{A} . The claim of the lemma follows. \blacktriangleleft

We summarize Claim 17 and Lemma 18:

► **Corollary 19.** *Let (q_1, q_2) be a co-occurrence of P_1, P_2 in the expansion of a non-terminal A . There exist splits $s_1 \in \text{Splits}'(G', P_1), s_2 \in \text{Splits}'(G', P_2)$ and occurrences p_1 of $S_1(s_1)$ and p_2 of $S_2(s_2)$, where $[p_1, p_1 + |S_1(s_1)|] \supseteq [q_1, q_1 + |P_1|]$ and $[p_2, p_2 + |S_2(s_2)|] \supseteq [q_2, q_2 + |P_2|]$, such that at least one of the following holds:*

1. *The occurrence p_1 is either relevant or $p_1 + |S_1(s_1)| - 1 \leq |\overline{\text{head}(A)}|$. The occurrence p_2 is either relevant or $p_2 > |\overline{\text{head}(A)}|$. Additionally, $p_1 = q_1 - \Delta_1(s_1)$ and $p_2 = q_2 - \Delta_2(s_2)$.*
2. *The occurrence p_2 is relevant and $p_1 \leq |\overline{\text{head}(A)}|$. Additionally, $p_2 = q_2 - \Delta_2(s_2)$, the period of $S_1(s)$ equals the period π_1 of P_1 , and there exists an integer k such that $p_1 = q_1 - \Delta_1(s_1) - \pi_1 \cdot k$ and $p_2 + \pi_1 - 1 \leq p_1 + |S_1(s_1)| - 1 \leq p_2 + |S_2(s_2)| - 1$.*

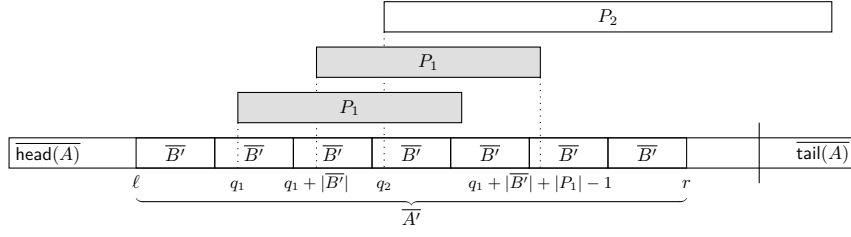
The reverse observation holds as well:

► **Observation 20.** *If p_j is an occurrence of $S_j(s)$ in \overline{A} , $j = 1, 2$, then $q_j = p_j + \Delta_j(s)$ is an occurrence of P_j . Furthermore, if $S_1(s)$ is periodic with period π_1 , then $q_1 + \pi_1 \cdot k$, $0 \leq k \leq \lfloor (|S_1(s)| - q_1 - |P_1|) / \pi_1 \rfloor$, are occurrences of P_1 in \overline{A} .*

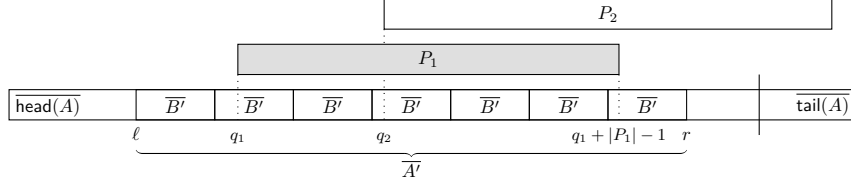
Finally, the following trivial observation will be important for upper bounding the time complexity of our query algorithm:

► **Observation 21.** *If a string contains a pair of occurrences (q_1, q_2) of P_1 and P_2 such that $0 \leq q_2 - q_1 \leq b$, then it contains a b -close co-occurrence of P_1 and P_2 .*

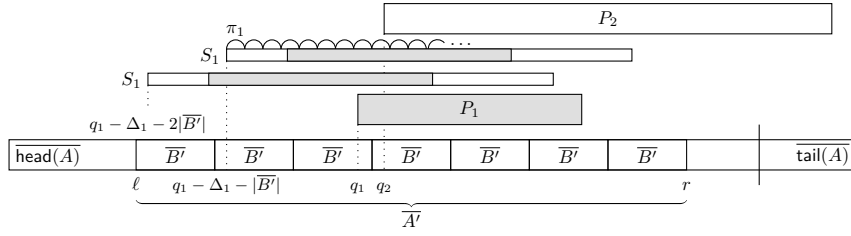
12:12 Compressed Indexing for Consecutive Occurrences



(a) If neither (1) nor (2), then (q_1, q_2) is not consecutive.



(b) (1) holds and (2) does not.



(c) (2) holds, (1) does not, and $|P_1| - s \geq \overline{(B')^2}$.

■ **Figure 2** Subcases of Lemma 18.

4.2 Index

The first part of the index is the data structure of Theorem 10 and the index of Christiansen et al. [8]:

► **Fact 22** ([8, Introduction and Theorem 6.12]). *There is a $O(g \log^2 N)$ -space data structure that can find the occ occurrences of any pattern $P[1 \dots m]$ in S in time $O(m + \text{occ})$.*

The second part of the index are the tries T_{pre} and T_{suf} , augmented as explained below. Consider a quadruple (u_1, u_2, v_1, v_2) , where u_1 and u_2 are nodes of T_{pre} and v_1 and v_2 are nodes of T_{suf} . Let U_1, U_2, V_1, V_2 be the labels of u_1, u_2, v_1, v_2 , respectively. Define $S_1 = \text{rev}(U_1)V_1$ and $S_2 = \text{rev}(U_2)V_2$, and let $l_1 = |\text{rev}(U_1)|$ and $l_2 = |\text{rev}(U_2)|$.

First, we store a binary search tree $\mathcal{T}_1(u_1, u_2, v_1, v_2)$ that for each non-terminal A contains at most six integers $d = p_2 - p_1$, where p_1, p_2 are occurrences of S_1, S_2 in \overline{A} , satisfying at least one of the below:

1. p_1 is the rightmost occurrence of S_1 such that $p_1 + |S_1| - 1 < \overline{\text{head}(A)}$ and p_2 is the leftmost occurrence of S_2 such that $p_2 \geq \overline{\text{head}(A)}$;
2. p_1 is a relevant occurrence of S_1 with a split l_1 and p_2 is the leftmost occurrence of S_2 such that $p_2 \geq \overline{\text{head}(A)}$;
3. p_1 is a relevant occurrence of S_1 with a split l_1 , p_2 is a relevant occurrence of S_2 with a split l_2 ;
4. p_2 is a relevant occurrence of S_2 with a split l_2 and p_1 is the rightmost occurrence of S_1 such that $p_1 + |S_1| - 1 < p_2$;
5. p_2 is a relevant occurrence of S_2 with a split l_2 and p_1 is the leftmost or second leftmost occurrence of S_1 in $\overline{\text{head}(A)}$ such that $p_1 < p_2 \leq p_1 + |S_1| - 1 < p_2 + |S_2| - 1$.

Second, we store a list of non-terminals $\mathcal{L}(u_2, v_2)$ such that their expansion contains a relevant occurrence of S_2 with a split l_2 . Additionally, for every $k \in [0, \log N]$, we store, if defined:

1. The rightmost occurrence p_1 of S_1 in S_2 such that $p_1 + (|S_1| - 1) \leq l_2 - 2^k$;
2. The leftmost occurrence p'_1 of S_1 in S_2 such that $p'_1 \leq l_2 - 2^k \leq p'_1 + |S_1| - 1$;
3. The rightmost occurrence p''_1 of S_1 in S_2 such that $p''_1 \leq l_2 - 2^k \leq p''_1 + |S_1| - 1$.

Finally, we compute and memorize the period π_1 of S_1 . If the period is well-defined (i.e., S_1 is periodic), we build a binary search tree $\mathcal{T}_2(u_1, u_2, v_1, v_2)$. Consider a non-terminal A containing a relevant occurrence p_2 of S_2 with a split l_2 . Let p_1 be the leftmost occurrence of S_1 such that $p_1 \leq p_2 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$ and p'_1 the rightmost. If p_1 and p'_1 exist (p_1 might be equal to p'_1) and $p'_1 + |S_1| - 1 \geq p_2 + \pi_1 - 1$, we add an integer $(p'_1 - p_1)/\pi_1$ to the tree and associate it with A . We also memorize a number $ov(S_1, S_2) = p_2 - p'_1$, which does not depend on A by Corollary 2 and therefore is well-defined (it corresponds to the longest prefix of S_2 periodic with period π_1).

▷ **Claim 23.** The data structure occupies $O(g^5 \log^5 N)$ space.

Proof. The data structure of Theorem 10 occupies $O(g^2 \log^4 N)$ space. The index of Christiansen et al. occupies $O(g \log^2 N)$ space. The tries, by Lemma 9, use $O(g') = O(g \log N)$ space. There are $O((g')^4)$ quadruples (u_1, u_2, v_1, v_2) and for each of them the trees take $O(g')$ space. The arrays of occurrences of S_1 in S_2 use $O(\log N)$ space. Therefore, overall the data structure uses $O(g^5 \log^5 N)$ space. ◁

4.3 Query

Recall that a query consists of two strings P_1, P_2 of length at most m each and an integer b , and we must find all b -close co-occurrences of P_1, P_2 in S , let occ be their number.

We start by checking whether P_2 occurs in P_1 using a linear-time and constant-space pattern matching algorithm such as [11]. If it is, let q_2 be the position of the first occurrence. If $q_2 > b$, then there are no b -close co-occurrences of P_1, P_2 in S . Otherwise, to find all b -close co-occurrences of P_1, P_2 in S (that *always* consist of an occurrence of P_1 in S and the first occurrence of P_2 in P_1), it suffices to find all occurrences of P_1 in S , which we do using the index of Christiansen et al. [8] in time $O(|P_1| + \text{occ}) = O(m + \text{occ})$.

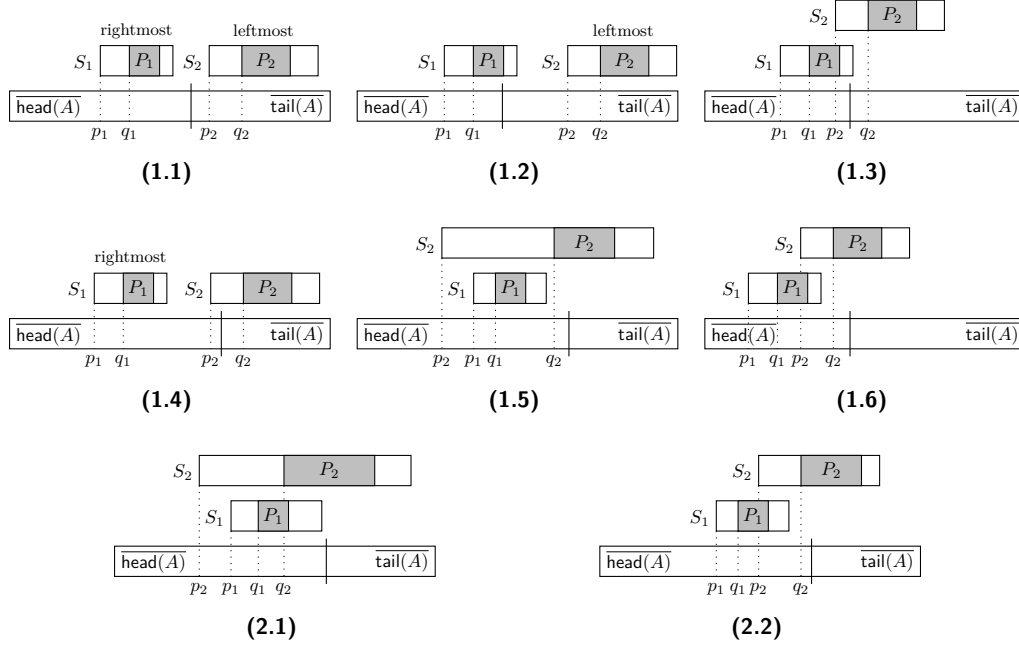
From now on, assume that P_2 is not a substring of P_1 . Let \mathcal{N} be the set of all non-terminals in G' such that their expansion contains a relevant b -close co-occurrence of P_1, P_2 . By Claim 16, $|\mathcal{N}| \leq \text{occ}$.

► **Lemma 24.** *Assume that P_2 is not a substring of P_1 . One can retrieve in $O(m + (1 + \text{occ}) \log^3 N)$ time a set $\mathcal{N}' \supset \mathcal{N}$, $|\mathcal{N}'| = O(\text{occ} \log N)$.*

Proof. We start by computing $\text{Splits}'(G', P_1)$ and $\text{Splits}'(G', P_2)$ via Lemma 8 in $O((|P_1| + |P_2|) \log N) = O(m \log N)$ time (or providing a certificate that either P_1 or P_2 does not occur in S , in which case there are no co-occurrences of P_1, P_2 in S and we are done). Recall that $|\text{Splits}'(G', P_1)|, |\text{Splits}'(G', P_2)| \in O(\log N)$. For each fixed pair of splits $s_1 \in \text{Splits}'(G', P_1)$, $s_2 \in \text{Splits}'(G', P_2)$ and $j \in \{1, 2\}$, we compute the interval of strings in T_{pre} prefixed by $\text{rev}(P_j[\dots s_j])$, which corresponds to the locus u_j of $\text{rev}(P_j[\dots s_j])$ in T_{pre} and the interval of strings in T_{suf} prefixed by $P_j(s_j \dots)$, which corresponds to the locus v_j of $P_j(s_j \dots)$ in T_{suf} . Computing the intervals takes $O(m + \log^2 N)$ time for all the splits by Lemma 9.

Consider the strings $S_1 = \text{rev}(U_1)V_1$ and $S_2 = \text{rev}(U_2)V_2$, where U_1, U_2, V_1, V_2 are the labels of u_1, v_1, u_2, v_2 , respectively. Let $l_1 = |\text{rev}(U_1)|$, $\Delta_1 = l_1 - s_1$, $l_2 = |\text{rev}(U_2)|$, $\Delta_2 = l_2 - s_2$, and $\Delta = \Delta_1 - \Delta_2$.

Consider a relevant co-occurrence (q_1, q_2) of P_1, P_2 in the expansion of a non-terminal A . By Corollary 19, q_1, q_2 imply existence of occurrences p_1, p_2 of S_1, S_2 such that $[p_1, p_1 + |S_1|] \supseteq [q_1, q_1 + |P_1|]$ and $[p_2, p_2 + |S_2|] \supseteq [q_2, q_2 + |P_2|]$. Our index must treat both cases of Corollary 19. We consider eight subcases defined in Fig. 3, which describe all possible locations of p_1 and p_2 .



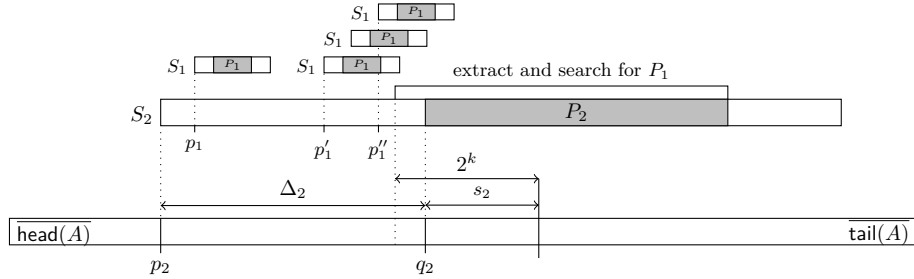
■ **Figure 3** Assume that S_1 does not contain S_2 . The figure shows all possible locations of occurrences p_1, p_2 of S_1, S_2 in \bar{A} . **In Case 1 of Corollary 19**, there are six subcases: (1.1) $p_1 + |S_1| - 1 \leq |\text{head}(A)|$, $p_2 > |\text{head}(A)|$; (1.2) p_1 is a relevant occurrence of S_1 , $p_2 > |\text{head}(A)|$; (1.3) p_1, p_2 are relevant; (1.4) p_2 is relevant, $p_1 + |S_1| - 1 \leq p_2$; (1.5) p_2 is relevant, $p_2 < p_1 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$; (1.6) p_2 is relevant, $p_1 < p_2 < p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$. By the definition of a co-occurrence and by Observation 20, in Subcases (1.1) and (1.4) p_1 must be as far to the right as possible, and in Subcases (1.1) and (1.2) p_2 must be as far to the left as possible. **In Case 2**, there are two subcases: (2.1) p_2 is relevant and $p_2 \leq p_1 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$; (2.2) p_2 is relevant and $p_1 < p_2 < p_2 + \pi_1 - 1 \leq p_1 + |S_1| - 1$, where π_1 is the period of S_1 . In all subcases, $q_2 = p_2 + \Delta_2$. In Subcases (1.1)-(1.6) $q_1 = p_1 + \Delta_1$ and in Subcases (2.1) and (2.2) $q_1 = p_1 + \Delta_1 + k \cdot \pi_1$ for some integer k .

Subcases (1.1)–(1.4). To retrieve the non-terminals, we query $\mathcal{T}_1(u_1, u_2, v_1, v_2)$ to find all integers that belong to the range $[\Delta, \Delta + b]$ (and the corresponding non-terminals). Recall that, for each non-terminal A , the tree stores an integer $d = p_2 - p_1$, where p_1 is the starting position of an occurrence of S_1 in \bar{A} and p_2 of S_2 . By Observation 20, $p_1 + \Delta_1$ is an occurrence of P_1 and $p_2 + \Delta_2$ is an occurrence of P_2 . The distance between them is in $[0, b]$ iff $d \in [\Delta, \Delta + b]$. By Observation 21, each retrieved non-terminal contains a close co-occurrence of (q_1, q_2) . On other other hand, if \bar{A} contains a co-occurrence (q_1, q_2) corresponding to one Subcases (1.1)-(1.4), then by Corollary 19, $p_1 = q_1 - \Delta_1$ is an occurrence of S_1 and $p_2 = q_2 - \Delta_2$ is an occurrence of S_2 and by construction $\mathcal{T}_1(u_1, u_2, v_1, v_2)$ stores an integer $d = p_2 - p_1$. Therefore, the query retrieves all non-terminals corresponding to Subcases (1.1)-(1.4).

Subcases (1.5) and (2.1). We must decide whether an occurrence of P_1 in S_2 forms a b -close co-occurrence with the occurrence Δ_2 of P_2 in S_2 , and if so, report all non-terminals such that their expansion contains a relevant co-occurrence of S_2 with a split l_2 , which are exactly the non-terminals stored in the list $\mathcal{L}(u_2, v_2)$. Let $k = \lceil \log(s_2) \rceil$. Recall that the index stores the following information for k :

1. p_1 , the rightmost occurrence of S_1 in S_2 such that $p_1 + (|S_1| - 1) \leq l_2 - 2^k$;
2. p'_1 , the leftmost occurrence of S_1 in S_2 such that $p'_1 \leq l_2 - 2^k \leq p_1 + (|S_1| - 1)$;
3. p''_1 , the rightmost occurrence of S_1 in S_2 such that $p''_1 \leq l_2 - 2^k \leq p''_1 + (|S_1| - 1)$.

(See Fig. 4). By Observation 20, the occurrence p_1 of S_1 induces an occurrence $q_1 = p_1 + \Delta_1$ of P_1 . Furthermore, if S_1 is periodic with period π_1 , then $q_1 + \pi_1 \cdot k$, $0 \leq k \leq \lfloor (|S_1| - q_1 - |P_1|) / \pi_1 \rfloor$, are also occurrences of P_1 . One can decide whether the distance from any of these occurrences to q_2 is in $[0, b]$ in constant time, and if yes, then there S_2 contains a b -close co-occurrence of P_1, P_2 by Observation 21. Second, by Corollary 2, if S_1 is not periodic, then there are no occurrences of S_1 between p'_1 and p''_1 and p'_1, p''_1 by Observation 20 induce occurrences $p'_1 + \Delta_1, p''_1 + \Delta_1$ of P_1 . Otherwise, there are occurrences of P_1 in every position $p'_1 + \Delta_1 + k \cdot \pi_1$, $0 \leq k \leq \lfloor (|S_1| + p''_1 - |P_1| - p'_1) / \pi_1 \rfloor$. Similarly, we can decide whether the distance from any of them to the occurrence Δ_2 of P_2 in S_2 is in $[0, b]$ in constant time. Finally, let q_1 be the rightmost occurrence of P_1 in S_2 in the interval $[l_2 - 2^k + 1, \Delta_2]$. We extract $S_2(l_2 - 2^k, \Delta_2 + |P_2|)$ via Fact 28 and search for q_1 using a linear-time pattern matching algorithm for P_1 , which takes $O(|P_1| + |P_2|) = O(m)$ time. If $0 \leq \Delta_2 - q_1 \leq b$, then there is a b -close co-occurrence of P_1, P_2 in S_2 . Correctness follows from Corollary 19, Observation 20 and Observation 21.



■ **Figure 4** Query algorithm for Subcases (1.5) and (2.1).

Subcase (2.2). Let π_1 be the period of S_1 . We retrieve the non-terminals associated with the integers $q \in \mathcal{T}_2(u_1, u_2, v_1, v_2)$ such that the intersection of an interval $I = [a, b]$ and $[\ell, q]$ is non-empty, where $a = \lceil (\Delta - \text{ov}(S_1, S_2)) / \pi_1 \rceil$, $b = \lfloor (\Delta - \text{ov}(S_1, S_2) + b) / \pi_1 \rfloor$ and $\ell = -\lfloor (|S_1| - |P_1| - \Delta_1) / \pi_1 \rfloor$ (See the description of the index for the definition of $\text{ov}(S_1, S_2)$). As ℓ is fixed, we can implement the query via at most one binary tree search: If $b \leq \ell$, the output is empty, if $a \leq \ell \leq b$, we must output all integers, and if $\ell \leq a$, we must output all $q \geq b$. Let us now explain why the algorithm is correct. Consider a non-terminal A for which $\mathcal{T}_2(u_1, u_2, v_1, v_2)$ stores an integer q . By construction, \overline{A} contains a relevant occurrence of S_2 with a split l_2 . A position $p_1 = \lceil \text{head}(A) \rceil - l_2 - \text{ov}(S_1, S_2) - q \cdot \pi_1$ is the leftmost occurrence of S_1 in \overline{A} such that $p_1 \leq p_2 \leq p_1 + |S_1| - 1$ and $p_2 = \lfloor \text{head}(A) \rfloor - l_2 - \text{ov}(S_1, S_2)$ the rightmost. Consequently, there is an occurrence $q_1 = \lceil \text{head}(A) \rceil - l_2 - \text{ov}(S_1, S_2) - q' \cdot \pi_1 + \Delta_1$ of P_1 for each $-\lfloor (|S_1| - |P_1| - \Delta_1) / \pi_1 \rfloor \leq q' \leq q$. The occurrence of S_2 implies that $q_2 = \lceil \text{head}(A) \rceil - s_2$ is an occurrence of P_2 . We have $0 \leq q_2 - q_1 = q' \cdot \pi_1 + \text{ov}(S_1, S_2) - \Delta \leq b$ iff $\Delta - \text{ov}(S_1, S_2) \leq q' \cdot \pi_1 \leq \Delta - \text{ov}(S_1, S_2) + b$, which is equivalent to $[\ell, q] \cap I \neq \emptyset$. It

follows that we retrieve every non-terminal corresponding to Subcase (2.2). On the other hand, by Observation 21, the expansion of each retrieved non-terminal contains a b -close co-occurrence of P_1, P_2 .

Subcase (1.6). We argue that we have already reported all non-terminals corresponding to this subcase and there is nothing left to do. Consider a non-terminal A such that its expansion contains a relevant occurrence p_2 of S_2 . If there are at most two occurrences p_1 of S_1 such that $p_1 \leq p_2 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$, we will treat them when we query $\mathcal{T}_1(u_1, u_2, v_1, v_2)$ (Subcases (1.1)-(1.4)). Otherwise, by Corollary 2, S_1 is periodic and there is an occurrence p'_1 of S_1 such that $p'_1 \leq p_2 < p_2 + \pi_1 \leq p_1 + |S_1| - 1 < p_2 + |S_2| - 1$. The non-terminals corresponding to this case are reported when we query $\mathcal{T}_2(u_1, u_2, v_1, v_2)$ (Subcase (2.2)).

Time complexity. As shown above, the algorithm reports a set $\mathcal{N}' \supset \mathcal{N}$ of non-terminals and each non-terminal in \mathcal{N}' contains a b -close co-occurrence. By Claim 16 and since the height of G' is $h = O(\log N)$, we have $|\mathcal{N}'| = O(\text{occ} \log N)$. Furthermore, for a fixed pair of splits of P_1, P_2 , each non-terminal in \mathcal{N}' can be reported a constant number of times. Since $|\text{Splits}'(G', P_1)| \cdot |\text{Splits}'(G', P_2)| = O(\log^2 N)$, the total size of the output is $|\mathcal{N}'| \cdot O(\log^2 N) = O(\text{occ} \cdot \log^3 N)$. We therefore obtain that the running time of the algorithm is $O(m + \log^3 N + \text{occ} \log^3 N) = O(m + (1 + \text{occ}) \log^3 N)$ as desired. \blacktriangleleft

Once we have retrieved the set \mathcal{N}' , we find all b -close relevant co-occurrences for each of the non-terminals in \mathcal{N}' using Theorem 10. In fact, our algorithm acts naively and computes *all* relevant co-occurrences for a non-terminal in \mathcal{N}' , and then selects those that are b -close. By case inspection, one can show that a relevant co-occurrence for a non-terminal A always consists of an occurrence of P_2 that is either relevant or the leftmost in $\text{tail}(A)$, and a preceding occurrence of P_1 . Intuitively, this allows to compute all relevant co-occurrences efficiently and guarantees that their number is small. Formally, we show the following claim:

► Lemma 25. *Assume that P_2 is not a substring of P_1 . After $O(m \log N + \log^2 N)$ -time preprocessing, the data structure of Theorem 10 allows to compute all b -close relevant co-occurrences of P_1, P_2 in the expansion of a given non-terminal A in time $O(\log^3 N \log \log N)$.*

A part of the index of Christiansen et al. [8] is a pruned copy of the parse tree of G' . They showed how to traverse the tree to report all occurrences of a pattern, given its relevant occurrences in the non-terminals. By using essentially the same algorithm, we can report all b -close co-occurrences in amortized constant time per co-occurrence, which concludes the proof of Theorem 1. (See Appendix C, Lemma 33.)

References

- 1 Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In *Proc. 58th FOCS*, pages 192–203, 2017.
- 2 Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Impossibility results for grammar-compressed linear algebra. In *Proc. 34th NeurIPS*, pages 8810–8823, 2020.
- 3 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proc. 18th ESA*, pages 427–438, 2010.
- 4 Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner. Gapped indexing for consecutive occurrences. In *Proc. 32nd CPM*, pages 10:1–10:19, 2021.

- 5 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiro Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.
- 6 Timothy M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. *ACM Trans. Algorithms*, 9(3):22:1–22:22, 2013.
- 7 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and Abhi Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. 34th STOC*, pages 792–801, 2002.
- 8 Anders Roy Christiansen, Mikko Berggren Ettienné, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021.
- 9 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, pages 180–192, 2012.
- 10 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *J. Comput. Syst. Sci.*, 118:53–74, 2021.
- 11 Maxime Crochemore. Constant-space string-matching. In *Proc. 8th FSTTCS*, pages 80–87, 1988.
- 12 Diego Díaz-Domínguez, Gonzalo Navarro, and Alejandro Pacheco. An LMS-based grammar self-index with local consistency properties. In *Proc. 28th SPIRE*, 2021.
- 13 Paolo Ferragina and Rossano Venturini. Indexing compressed text. In *Encyclopedia of Database Systems (2nd ed.)*. Springer, 2018.
- 14 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proc. Am. Math. Soc.*, 16(1):109–114, 1965.
- 15 Johannes Fischer, Travis Gagie, Tsvi Kopelowitz, Moshe Lewenstein, Veli Mäkinen, Leena Salmela, and Niko Välimäki. Forbidden patterns. In *Proc. 10th LATIN*, pages 327–337, 2012.
- 16 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, pages 731–742, 2014.
- 17 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th SODA*, pages 1459–1477, 2018.
- 18 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *Proc. 29th SODA*, pages 1509–1528, 2018.
- 19 Daniel Gibney and Sharma V. Thankachan. Text indexing for regular expression matching. *Algorithms*, 14(5):133, 2021.
- 20 Leszek Gąsieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.
- 21 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 22 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. String retrieval for multi-pattern queries. In *Proc. 17th SPIRE*, pages 55–66, 2010.
- 23 Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Document listing for queries with excluded pattern. In *Proc. 23rd CPM*, pages 185–195, 2012.
- 24 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
- 25 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.
- 26 Tsvi Kopelowitz and Robert Krauthgamer. Color-distance oracles and snippets. In *Proc. 27th CPM*, pages 24:1–24:10, 2016.
- 27 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In *Proc. 27th SODA*, pages 1272–1287, 2016.
- 28 Kasper Green Larsen, J. Ian Munro, Jesper Sindahl Nielsen, and Sharma V. Thankachan. On hardness of several string indexing problems. *Theor. Comput. Sci.*, 582:74–82, 2015.

- 29 Moshe Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302, 2013.
- 30 Veli Mäkinen and Gonzalo Navarro. Compressed text indexing. In *Encyclopedia of Algorithms*, pages 394–397. Springer New York, 2016.
- 31 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- 32 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th SODA*, pages 657–666, 2002.
- 33 Gonzalo Navarro and Sharma V. Thankachan. Reporting consecutive substring occurrences under bounded gap constraints. *Theor. Comput. Sci.*, 638:108–111, 2016.
- 34 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st MFCS*, volume 58, pages 72:1–72:15, 2016.
- 35 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
- 36 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.

A Proofs omitted from Section 2

► **Lemma 9.** *Given an RLSLP G of size g and height h . Assume that every string in a set \mathcal{S} is either a prefix or a suffix of the expansion of a non-terminal of G or its reverse. The trie for \mathcal{S} can be implemented in space $O(|\mathcal{S}|)$ to maintain the following queries in $O(m + \tau \cdot (h + \log m))$ time: Given a pattern P of length m and suffixes Q_i of P , $1 \leq i \leq \tau$, find, for each i , the interval of strings in the (lexicographically sorted) \mathcal{S} prefixed by Q_i .*

Proof. Let us first recall the definition of the Karp–Rabin fingerprint.

► **Definition 26 (Karp–Rabin fingerprint).** *For a prime p and an $r \in \mathbb{F}_p^*$, the Karp–Rabin fingerprint [24] of a string X is defined as a tuple $(r^{|X|-1} \bmod p, r^{-|X|+1} \bmod p, \varphi_{p,r}(X))$, where $\varphi_{p,r}(X) = \sum_{k=0}^{|X|-1} S[k]r^k \bmod p$.*

We use the following result of Christiansen et al. [8], which builds on Belazzougui et al. [3] and Gagie et al. [16, 17].

► **Fact 27 ([8, Lemma 6.5]).** *Let \mathcal{S} be a set of strings and assume we have a data structure supporting extraction of any length- l prefix of strings in \mathcal{S} in time $f_e(l)$ and computing the Karp–Rabin fingerprint φ of any length- l prefix of a string in \mathcal{S} in time $f_h(l)$. We can then build a data structure that uses $O(|\mathcal{S}|)$ space and supports the following queries in $O(m + f_e(m) + \tau(f_h(m) + \log m))$ time: Given a pattern P of length m and $\tau > 0$ suffixes Q_1, \dots, Q_τ of P , find the intervals of strings in (the lexicographically-sorted) \mathcal{S} prefixed by Q_1, \dots, Q_τ .*

It should be noted that despite using a hash function, the query algorithm is deterministic: the proof shows that p and r can be chosen during the construction time to ensure that there are no collisions on the substrings of the strings in \mathcal{S} .

To bound f_e , we use [8, Lemma 6.6] which builds on Gąsieniec et al. [20] and Claude and Navarro [9].

► **Fact 28 ([8, Lemma 6.6]).** *Given an RLSLP of size $O(g)$, there exists a data structure of size $O(g)$ such that any length- l prefix or suffix of \bar{A} can be obtained from any non-terminal A in time $f_e(l) = O(l)$.*

To bound $f_h(l)$, we introduce a simple construction based on the following well-known fact:

► **Fact 29.** *Consider strings X, Y, Z where $XY = Z$. Given the Karp–Rabin fingerprints of two of the three strings, one can compute the fingerprint of the third string in constant time.*

▷ **Claim 30.** Given a RLSP G of size g and height h , there exists a data structure of size $O(g)$ that given a non-terminal A and an integer l allows to retrieve the Karp–Rabin fingerprints of the length- l prefix and suffix of $\overline{A^r}$ and $\text{rev}(\overline{A^r})$ in time $f_h(l) = O(h + \log l)$.

Proof. The claim for $\text{rev}(\overline{A^r})$ follows for the claim for $\overline{A^r}$ by considering the grammar G_{rev} , where the order of the non-terminals in each production is reversed. Below we focus on extracting the fingerprints for $\overline{A^r}$, and we further restrict our attention to prefixes of $\overline{A^r}$, the algorithm for suffixes being analogous.

The data structure consists of two sets. The first set contains the lengths of the expansions of all non-terminals in the grammar, and the second one their fingerprints.

By Fact 29 and doubling, it suffices to show an algorithm for computing the fingerprint of the length- l prefix of \overline{A} . Assume that A associated with a rule $A \rightarrow BC$. If the length of \overline{A} is smaller than l , we return error. Otherwise, to compute the fingerprint of the length- l prefix of \overline{A} , we consider two cases. If $l \leq |\overline{B}|$, we recurse on B to retrieve the fingerprint of the l -length prefix of \overline{B} . Otherwise, we recurse on C to retrieve the fingerprint of $\overline{C}[\dots l - |\overline{B}|)$ and then compute the fingerprint of the l -length prefix of \overline{A} from the fingerprints of \overline{B} and $\overline{C}[\dots l - |\overline{B}|)$ in constant time by Fact 29.

For a non-terminal A associated with a rule $A \rightarrow B^r$, we compute the fingerprint analogously. If the length of \overline{A} is smaller than l , we return error. Otherwise, let q be such that $q \cdot |\overline{B}| \leq l < (q + 1) \cdot |\overline{B}|$. We compute the fingerprint of \overline{B}^q from the fingerprint of \overline{B} by applying Fact 29 $O(1 + \log q)$ times, and the fingerprint of $\overline{B}[\dots l - q \cdot |\overline{B}|)$ recursively. We can then apply Fact 29 to compute the fingerprint of the length- l prefix of \overline{A} in constant time. Note that in this case, the length of the prefix decreases by a factor at least q .

If we are in a terminal A , the calculation takes $O(1)$ time (the prefix must be equal to A itself).

In total, we spend $O(h + \log l)$ time as we recurse $O(h)$ times, and whenever we spend more than constant time in a symbol, we charge it on the decrease in the length. The fingerprints of length- l suffixes are computed analogously. ◁

By substituting the bounds for $f_e(l)$ (Fact 29) and $f_h(l)$ (Claim 28) into Fact 27, we obtain the claim of the lemma. ◀

B Proofs omitted from Section 3

▷ **Claim 31.** Given a non-terminal A of G' , we can find the leftmost and the rightmost occurrences of P in \overline{A} and as a corollary in $\overline{\text{head}(A)}$ and $\overline{\text{tail}(A)}$ in $O(\log^2 N \log \log N)$ time.

Proof. We explain how to find the leftmost occurrence of P in \overline{A} , the rightmost one can be found analogously. We first check whether \overline{A} contains an occurrence of P via Claim 14 in $O(\log N \log \log N)$ time. If it does not, we can stop immediately. Below we assume that there is an occurrence of P in \overline{A} . Next, we check whether $\overline{\text{head}(A)}$ contains an occurrence of P via Claim 14 in $O(\log N \log \log N)$ time. If it does, the leftmost occurrence of P in \overline{A} is the leftmost occurrence of P in $\overline{\text{head}(A)}$ and we can find it by recursing on $\overline{\text{head}(A)}$. If $\overline{\text{head}(A)}$ does not contain an occurrence of P , but \overline{A} contains relevant occurrences of P , then the leftmost occurrence of P in \overline{A} is the leftmost relevant occurrence of P in \overline{A} and we can

find it in $O(|\text{Splits}'(G', P)|) = O(\log N)$ time. Finally, if P neither occurs in $\overline{\text{head}(A)}$ nor has relevant occurrences in \overline{A} , then the leftmost occurrence of P in \overline{A} is the leftmost occurrence of P in $\overline{\text{tail}(A)}$. If $\overline{\text{tail}(A)}$ is a non-terminal C , we recurse on C to find it. If $\overline{\text{tail}(A)} = \overline{B^{r-1}}$ for a non-terminal B , $\overline{\text{tail}(A)}$ cannot contain an occurrence of P because \overline{B} does not contain P and there are no relevant occurrences in A . We recurse down at most $h = O(\log N)$ levels, and spend $O(\log N \log \log N)$ time per level. The claim follows. \triangleleft

► **Lemma 32.** *Let A be a non-terminal of G' . For any position p , we can find the rightmost occurrence $q \leq p$ of P in \overline{A} and the leftmost occurrence $q' \geq p$ of P in \overline{A} in $O(\log^3 N \log \log N)$ time.*

Proof. First we describe how to locate q . Consider a node u of the parse tree of G' labeled by A . The algorithm starts at u and recurses down. Let A' be the label of the current node. It computes the leftmost and rightmost occurrences in $\overline{A'}$, $\overline{\text{head}(A')}$ and $\overline{\text{tail}(A')}$ as well as all relevant occurrences via Claim 31. If the leftmost occurrence of P in $\overline{A'}$ is larger than p , the search result is empty. Otherwise, consider two cases.

1. A' is associated with a rule $A' \rightarrow B'C'$, i.e. $\overline{\text{head}(A')} = \overline{B'}$, $\overline{\text{tail}(A')} = \overline{C'}$.
 - a. If $p \leq |\overline{B'}|$, recurse on $\overline{B'}$.
 - b. Assume now that $p > |\overline{B'}|$. If the leftmost occurrence of P in $\overline{C'}$ is smaller than p , recurse on $\overline{C'}$. Otherwise, return the rightmost relevant occurrence of P in $\overline{A'}$ if it exists else the rightmost occurrence of P in $\overline{B'}$.
2. A' is associated with a rule $A \rightarrow (B')^r$, i.e. $\overline{\text{head}(A')} = \overline{B'}$, $\overline{\text{tail}(A')} = \overline{(B')^{r-1}}$. Let an integer k be such that $(k-1) \cdot |\overline{B'}| + 1 \leq p \leq k \cdot |\overline{B'}|$. The desired occurrence of P is the rightmost one of the following ones:
 - a. The rightmost occurrence $q \leq p$ of P which crosses the border between two copies of $\overline{B'}$. To compute q , we compute all relevant occurrences of P in $\overline{A'}$ and then shift each of them by the maximal possible shift $r' \cdot |\overline{B'}|$, where r' is an integer, which guarantees that it starts before p and ends before $|\overline{A'}|$ and take the rightmost of the computed occurrences to obtain q .
 - b. The rightmost occurrence q of P such that for some integer k' , we have $(k'-1) \cdot |\overline{B'}| \leq q \leq q + |P| - 1 \leq k' \cdot |\overline{B'}|$ (i.e. the occurrence fully belongs to some copy of $\overline{B'}$). In this case, q is either the rightmost occurrence of P in the $(k-1)$ -th copy of $\overline{B'}$, or the rightmost occurrence of P in the k -th copy of $\overline{B'}$ that is smaller than p . In the second case, we compute q by recursing on $\overline{B'}$.

We recurse down at most h levels. On each level we spend $O(\log^2 N \log \log N)$ time to compute the leftmost, the rightmost, and relevant occurrences and respective shifts for a constant number of non-terminals via Claim 31. Therefore, in total we spend $O(h \cdot \log^2 N \log \log N) = O(\log^3 N \log \log N)$ time.

Locating q' is very similar and differs only in small technicalities. The algorithm starts at the node u and recurses down. Let A' be the label of the current node. We compute the leftmost and rightmost occurrences in $\overline{A'}$, $\overline{\text{head}(A')}$ and $\overline{\text{tail}(A')}$ as well as all relevant occurrences via Claim 31. If the rightmost occurrence of P in $\overline{A'}$ is smaller than p , the search result is empty. Otherwise, consider two cases.

1. A' is associated with a rule $A' \rightarrow B'C'$, i.e. $\overline{\text{head}(A')} = \overline{B'}$, $\overline{\text{tail}(A')} = \overline{C'}$.
 - a. If $p > |\overline{B'}|$, recurse on $\overline{C'}$.
 - b. Assume now that $p \leq |\overline{B'}|$. If the rightmost occurrence of P in $\overline{B'}$ is larger than p , recurse on $\overline{B'}$. Otherwise, return the leftmost relevant occurrence q satisfying $q \geq p$, if it exists, and otherwise the leftmost occurrence of P in $\overline{C'}$.

2. A' is associated with a rule $A \rightarrow (B')^r$, i.e. $\text{head}(A') = B'$, $\text{tail}(A') = (B')^{r-1}$. Let an integer k be such that $(k-1) \cdot |\overline{B'}| + 1 \leq p \leq k \cdot |\overline{B'}|$. The desired occurrence of P is the leftmost one of the following ones:
 - a. The leftmost occurrence $q' \geq p$ of P which crosses the border between two copies of $\overline{B'}$. To compute q' , we compute all relevant occurrences of P in $\overline{A'}$ and then shift each of them by the minimal possible shift $r' \cdot |\overline{B'}|$, where r' is an integer, which guarantees that it starts after p and ends before $|\overline{A'}|$ (if it exists) and take the leftmost of the computed occurrences to obtain q .
 - b. The leftmost occurrence q' of P such that for some integer k' , we have $(k'-1) \cdot |\overline{B'}| \leq q' \leq q' + |P| - 1 \leq k' \cdot |\overline{B'}|$ (i.e. the occurrence fully belongs to some copy of $\overline{B'}$). In this case, q' is either the leftmost occurrence of P in the $(k+1)$ -st copy of $\overline{B'}$, or the leftmost occurrence of P in the k -th copy of $\overline{B'}$ that is larger than p . In the second case, we compute q' by recursing on B' .

The time complexities are the same as for computing q . ◀

C Proofs omitted from Section 4

► **Lemma 25.** *Assume that P_2 is not a substring of P_1 . After $O(m \log N + \log^2 N)$ -time preprocessing, the data structure of Theorem 10 allows to compute all b -close relevant co-occurrences of P_1, P_2 in the expansion of a given non-terminal A in time $O(\log^3 N \log \log N)$.*

Proof. We preprocess P_1, P_2 in $O(m \log N + \log^2 N)$ time as explained in Theorem 10. Upon receiving a non-terminal A , we compute the leftmost and the rightmost occurrences of P_1, P_2 in $\overline{\text{head}(A)}$ and $\overline{\text{tail}(A)}$, as well as a set Π_1 of all relevant occurrences of P_1 in \overline{A} and a set Π_2 of all relevant occurrences of P_2 in \overline{A} via Claim 31. We will compute all relevant co-occurrences in \overline{A} , selecting those of them that are b -close is then trivial. As $q_1 \leq q_2$ by definition, each relevant co-occurrence (q_1, q_2) of P_1, P_2 in \overline{A} falls under one of the following categories:

1. q_1 is a relevant occurrence of P_1 in \overline{A} and q_2 is a relevant occurrence of P_2 in \overline{A} (i.e. $q_1 \in \Pi_1, q_2 \in \Pi_2$). To check whether a pair $q_1 \in \Pi_1, q_2 \in \Pi_2$ forms a co-occurrence of P_1, P_2 in \overline{A} , we must check whether there is an occurrence q of either P_1 or P_2 between q_1 and q_2 . The occurrence q can only be the rightmost occurrence r_q of P_2 in $\overline{\text{head}(A)}$, the leftmost occurrence l_q of P_1 in $\overline{\text{tail}(A)}$, or an occurrence in $\Pi_1 \cup \Pi_2$. Consequently, we can find all co-occurrences in this category by merging two (sorted) sets: $\Pi_1 \cup \{l_q\}$ and $\{r_q\} \cup \Pi_2$, which can be done in $O(2 + |\Pi_1 \cup \Pi_2|)$ time.
2. $1 \leq q_1 \leq q_1 + |P_1| - 1 \leq |\overline{\text{head}(A)}|$ and $|\overline{\text{head}(A)}| < q_2 \leq q_2 + |P_2| - 1$. In this case, q_1 must be the rightmost occurrence of P_1 in $\overline{\text{head}(A)}$ and q_2 the leftmost occurrence in $\overline{\text{tail}(A)}$, $q_1 \leq q_2$, and there must be no occurrence $q \in \Pi_1 \cup \Pi_2$ such that $q_1 \leq q \leq q_2$. Therefore, if there is a co-occurrence in this category, we can retrieve it in $O(|\Pi_1 \cup \Pi_2|)$ time.
3. q_1 is a relevant occurrence of P_1 in \overline{A} (i.e. $q_1 \in \Pi_1$) and $|\overline{\text{head}(A)}| < q_2 \leq q_2 + |P_2| - 1$. In this case, q_1 must be the rightmost occurrence in Π_1 and q_2 the leftmost occurrence of P_2 in $\overline{\text{tail}(A)}$, and there should be no occurrence from Π_2 between q_1 and q_2 . Therefore, if there is a co-occurrence in this category, we can find it in $O(|\Pi_1 \cup \Pi_2|)$ time.
4. $q_1 \leq q_1 + |P_1| - 1 \leq |\overline{\text{head}(A)}|$ and q_2 is a relevant occurrence of P_2 in \overline{A} (i.e. $q_2 \in \Pi_2$). First, consider the leftmost occurrence in $q_2 \in \Pi_2$. We find the rightmost occurrence $q_1 \leq q_2$ of P_1 in \overline{A} via a predecessor query. The pair (q_1, q_2) is a co-occurrence iff the rightmost occurrence of P_2 in $\overline{\text{head}(A)}$ is smaller than q_1 , which can be checked in constant time. Second, we consider the remaining occurrences in Π_2 . Let q'_2 be the leftmost one.

We begin by computing the preceding occurrence q'_1 of P_1 via a predecessor query and if $q_2 \leq q'_1$, output the resulting co-occurrence. If $\Pi_2 = \{q_2, q'_2\}$, we are done. Otherwise, by Corollary 2, the occurrences in $\Pi_2 \setminus \{q_2\}$ form an arithmetic progression with difference equal to the period of P_2 (as all of them contain the position $|\text{head}(A)|$). Furthermore, as P_1 does not contain P_2 , the occurrence of P_1 preceding q'_2 belongs to the periodic region formed by the relevant occurrences of P_2 . Therefore, all the remaining co-occurrences can be obtained from the co-occurrence for q'_2 by shifting them by the period. In total, this step takes $O(|\Pi_2| + \log^3 N \log \log N)$ time. ◀

► **Lemma 33.** *Assume that P_2 is not a substring of P_1 . One can compute all b -close co-occurrences of P_1, P_2 in S in time $O(m + (1 + \text{occ}) \cdot \log^4 N \log \log N)$.*

Proof. During the preprocessing, we prune the parse tree: First, for each non-terminal B , all but the first node labeled by B in the preorder is converted into a leaf and its subtree is pruned. For each node v labeled by a non-terminal B , we store $\text{anc}(v)$, the nearest ancestor u of v labeled by A such that u is the root or A labels more than one node in the pruned tree. Second, for every node labeled by a non-terminal A associated with a rule $A \rightarrow B^k$, we replace its $k - 1$ rightmost children with a leaf labeled by B^{k-1} . We call the resulting tree *the pruned parse tree* and for each node v labeled by a non-terminal B store $\text{next}(v)$, the next node labeled by B in preorder, if there is one. As every non-terminal labels at most one internal node of the pruned parse tree and every node has at most two children, it occupies $O(g')$ space.

When the algorithm of Lemma 24 outputs $A \in \mathcal{N}'$, we compute all relevant co-occurrences (q_1, q_2) in \bar{A} in time $O(\log^3 N \log \log N)$ using Lemma 25 and select those which satisfy $q_2 - q_1 \leq b$.

Fix a b -close relevant co-occurrence (q_1, q_2) in \bar{A} . If A is associated with a rule $A \rightarrow BC$, construct a set $\text{occ}(A) := \{(q_1, q_2)\}$, and otherwise if A is associated with a rule $A \rightarrow B^k$,

$$\text{occ}(A) := \{(q_1 + i \cdot |\bar{B}|, q_2 + i \cdot |\bar{B}|) : 0 \leq i \leq \lfloor (|\bar{A}| - q_2 - |P_2| + 1) / |\bar{B}| \rfloor\}$$

Suppose that A labels nodes v_1, v_2, \dots, v_k of the unpruned parse tree of G' (by construction v_1 is not pruned and we assimilate it to the corresponding node in the pruned parse tree). If W is a set of co-occurrences, denote for brevity $W + \delta = \{(q_1 + \delta, q_2 + \delta) : (q_1, q_2) \in W\}$. Below we show an algorithm that generates a set $\mathcal{S} = \cup_i \text{occ}(A) + \text{off}(v_i)$ that contains all secondary b -close co-occurrences due to (q_1, q_2) .

We traverse the pruned parse tree, while maintaining a priority queue. The queue is initialized to contain the first node in the preorder labeled by A together with $\text{occ}(A)$. Until the priority queue is empty, pop a node v and a set W of co-occurrences of P_1, P_2 in the expansion of its label, and perform the following steps:

- **Reporting step:** If v is the root, report W ;
- **Next node step:** If $\text{next}(v)$ is defined, push $(\text{next}(v), W + \text{off}(\text{next}(v)) - \text{off}(v))$;
- **Sibling step:** If v is labeled by a non-terminal B and its sibling by B^k , for some integer k , then $W := \cup_{0 \leq i \leq k} W + i \cdot |\bar{B}|$
- **Ancestor step:** Push to the queue $(\text{anc}(v), W + \text{off}(\text{anc}(v)) - \text{off}(v))$.

By construction and as every node is connected with the root by a path of anc links, the algorithm generates each co-occurrence in \mathcal{S} exactly once. The time complexity follows: The algorithm of Lemma 24 takes $O(m + (1 + \text{occ}) \cdot \log^3 N)$ time; applying Lemma 25 to every non-terminal in \mathcal{N}' takes $O(\text{occ} \cdot \log^4 N \log \log N)$ time; and maintaining the queue and reporting the co-occurrences takes $O(\text{occ})$ time as at every step we can charge the time needed to update the queue on newly created co-occurrences. ◀

Order-Preserving Squares in Strings

Paweł Gawrychowski ✉ 

Institute of Computer Science, University of Wrocław, Poland

Samah Ghazawi ✉

Department of Computer Science, University of Haifa, Israel

Gad M. Landau

Department of Computer Science, University of Haifa, Israel

Department of Computer Science and Engineering, NYU Tandon School of Engineering, New York University, Brooklyn, NY, USA

Abstract

An order-preserving square in a string is a fragment of the form uv where $u \neq v$ and u is order-isomorphic to v . We show that a string w of length n over an alphabet of size σ contains $\mathcal{O}(\sigma n)$ order-preserving squares that are distinct as words. This improves the upper bound of $\mathcal{O}(\sigma^2 n)$ by Kociumaka, Radoszewski, Rytter, and Waleń [TCS 2016]. Further, for every σ and n we exhibit a string with $\Omega(\sigma n)$ order-preserving squares that are distinct as words, thus establishing that our upper bound is asymptotically tight. Finally, we design an $\mathcal{O}(\sigma n)$ time algorithm that outputs all order-preserving squares that occur in a given string and are distinct as words. By our lower bound, this is optimal in the worst case.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases repetitions, distinct squares, order-isomorphism

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.13

Funding *Samah Ghazawi*: partially supported by the Israel Science Foundation grant 1475/18, and Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF).

Gad M. Landau: partially supported by the Israel Science Foundation grant 1475/18, and Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF).

1 Introduction

A natural definition of repetitions in strings is that of squares, which are fragments of the form uu , where u is a string. The study of repetitions in strings goes back at least to the work of Thue from 1906 [28], who constructed an infinite square-free word over the ternary alphabet. Since then, multiple definitions of repetitions have been proposed and studied, with the basic question being focused on analyzing how many such repetitions a string of length n can contain. Of course, any even-length fragment of the string \mathbf{a}^n is a square, therefore we would like to count distinct squares. Using a combinatorial result of Crochemore and Rytter [5], Fraenkel and Simpson [10] proved that a string of length n contains at most $2n$ distinct squares (also see a simpler proof by Ilie [17]). They also provided an infinite family of strings of length n with $n - o(n)$ distinct squares. For many years, it was conjectured that the right upper bound is actually n . Interestingly, a proof of the conjecture for the binary alphabet would imply it for any alphabet [24]. Very recently, after a series of improvements on the upper bound [7, 18, 23, 27], the conjecture has been finally resolved by Brlek and Li [1], who showed an upper bound of $n - \sigma + 1$, where σ is the size of the alphabet.

For many of the applications, it seems more appropriate to work with different definitions of equality, giving us different notions of squares. Three interesting examples are (1) Abelian squares [6, 8, 9, 16, 19–21, 26] (also called Jumbled squares) are of interest in natural language



© Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 13; pp. 13:1–13:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

processing applications and in other domains where the classifications strongly depend on feature sets distribution, as opposed to feature sequences distributions. (2) Parameterized squares [20] are considered in applications for finding identical sections of code. (3) Order-preserving squares [4, 13, 20] could be used in applications of stock price analysis and musical melody matching.

The combinatorial properties of the three types of squares were studied by Kociumaka et al. [20]. Given a string of length n over an alphabet of size σ , first the authors bounded the number of abelian squares that are distinct as words by $\Theta(n^2)$. Second, bounded the number of parameterized squares that are distinct as words by $\mathcal{O}((\sigma!)^2 n)$ and bounded the number of nonequivalent parameterized squares (see definition within) by $\mathcal{O}(\sigma! n)$. Third, the authors provided $\mathcal{O}(\sigma^2 n)$ bound for the number of order-preserving squares that are distinct as words.

From an algorithmic perspective, various algorithms were proposed for computing abelian squares and order-preserving squares in a string of length n . Cummings and Smyth [6] proposed an $\Theta(n^2)$ time algorithm for computing all substrings that consist of a concatenation of two or more abelian-equivalent substrings. Kociumaka et al. [21] proposed an algorithm for computing the longest, the shortest, and the number of all abelian squares in $\mathcal{O}(n^2 / \log^2 n)$ time using linear space. Gourdel et al. [13] proved that all nonshiftable order-preserving squares (see definition within) can be computed in $\mathcal{O}(n \log n)$ time. Additionally, Crochemore et al. [4] proposed the *incomplete* order-preserving suffix tree (see details within), denoted by T , that enables order-preserving pattern matching queries in time proportional to the pattern length. The suffix tree T can be constructed in $\mathcal{O}(n \log \log n)$ expected time and $\mathcal{O}(n \log^2 \log n / \log \log \log n)$ worst-case time. Moreover, the authors proved that using T , all occurrences of order-preserving squares can be computed in $\mathcal{O}(n \log n + occ)$ time, where occ is the total number of occurrences of order-preserving squares. Note that, the number of all occurrences of order-preserving squares might be unreasonably high. In particular, every regular square is considered to be an order-preserving square, hence \mathbf{a}^n contains $\Theta(n^2)$ occurrences of order-preserving squares. Henceforth, a more natural approach is to generate only order-preserving squares that are distinct as words.

Our results. In this paper, we focus on order-preserving squares. Same-length strings u and v over an ordered alphabet are order-isomorphic, denoted $u \approx v$, when the order between the characters at the corresponding positions is the same in u and v . For example, the strings $u = \mathbf{acb}$ and $v = \mathbf{azd}$ are order-isomorphic, assuming $\mathbf{a} < \mathbf{b} < \mathbf{c} < \mathbf{d} < \mathbf{z}$. In this paper, order-preserving squares are strings of the form uv , where $u \approx v$ and additionally $u \neq v$.

The main result of our paper is that the number of order-preserving squares in a string of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$. This improves the bound of $\mathcal{O}(\sigma^2 n)$ by Kociumaka et al. [20]. We stress that in our definition of an order-preserving square, we require that $u \neq v$, while Kociumaka et al. [20] counted fragments of the form uv , where $u \approx v$, that are distinct as words. We believe that our definition is more natural in the context of this paper. At the same time, by the result of Brlek and Li [1] a string of length n contains less than n fragments of the form uu that are distinct as words, thus our result implies that the number of fragments uv such that $u \approx v$ that are distinct as words is also $\mathcal{O}(\sigma n)$. We complement our upper bound by designing, for each σ , an infinite family of strings of length n over an alphabet of size σ containing $\Theta(\sigma n)$ such fragments. We begin with describing the lower bound in Section 3, and then present the upper bound in Section 4.

► **Theorem 1.** *The number of order-preserving squares in a string of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$, and this bound is asymptotically tight even if we only consider order-preserving squares that are distinct as words.*

Next, we design an algorithm for reporting all order-preserving squares in a given string of length n over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time, which (by our lower bound) is asymptotically optimal in the worst case. We again stress that in our definition of an order-preserving square, we require that $u \neq v$. However, all fragments of the form uu that are distinct as words can be reported in $\mathcal{O}(\sigma n)$ time using the algorithm of Gusfield and Stoye [14]¹. Thus, for $\sigma = o(\log n)$, this resolves one of the open questions by Crochemore et al. [4], who asked if there is an $o(n \log n)$ time algorithm for finding the longest order-preserving square. This is described in Section 5.

► **Theorem 2.** *All order-preserving squares in a string of length n over an alphabet of size σ can be found in $\mathcal{O}(\sigma n)$ time.*

High-level description of our techniques. For the lower bound, first, we consider the increasing string $w = 123 \dots n$ where $\sigma = n$. Clearly, any even-length fragment is an order-preserving square thus producing the maximum number, i.e. $\Omega(n^2) = \Omega(\sigma n)$, of order-preserving squares in a string of length n . To decrease the size of the alphabet σ , we replace w with a non-decreasing string $w = 11 \dots 122 \dots 2 \dots \sigma \sigma \dots \sigma$, where each character is repeated the same number of times. We exhibit $\Omega(\sigma n)$ order-preserving squares in w that are distinct as words. See Section 3 for more details.

For the upper bound, we build on the insight by Kociumaka et al. [20], where the high-level strategy is to consider each suffix of w separately. For each suffix and an alphabet character, they considered the leftmost occurrence of this character within the suffix. Thus, there are at most σ leftmost occurrences in each suffix. For a fixed suffix, they considered all of its prefixes as possible order-preserving squares uv . Next, they showed that, because $u \neq v$, the order-preserving square uv is defined by a pair (or pairs) of leftmost occurrences such that one occurrence belongs to u , and the other one belongs to v at the same relative position, where the length of uv is twice the difference between the leftmost occurrences. For example, let **acbadxyz** be the suffix, then the pair of positions 2 and 5 are leftmost occurrences defining the order-preserving square **acbadx** of length 6 that is a prefix of the given suffix. Note that, also 3 and 6 are leftmost occurrences defining the same order-preserving square **acbadx**. Thus, as a result, they upper bounded the number of order-preserving squares being a prefix of the considered suffix by $\binom{\sigma}{2}$, so $\binom{\sigma}{2}n$ in total.

In this paper, we adopt a similar approach, by separately upper bound the number of order-preserving squares that are prefixes of a suffix of the input string w . However, our goal is to show that there are only $\mathcal{O}(\sigma)$ such prefixes, so $\mathcal{O}(\sigma n)$ in total. To this end, we first partition the order-preserving squares into groups. Let O_k the set of all order-preserving squares uv such that $2^k \leq |uv| < 2^{k+1}$. Similarly, we partition the leftmost occurrences into groups. Let L_k the set of all leftmost occurrences i such that $2^k \leq i < 2^{k+1}$. Now, our strategy is to show that if $|O_k|$ is larger than some fixed constant then $|O_k| = \mathcal{O}(|L_{k-2}|)$. The structure of the argument is as follows. We first observe that two order-preserving squares uv and $u'v'$ imply that $|u| - \Delta$, where $\Delta = |u'| - |u|$, is a so-called order-preserving border of u . We write $u = b_1 b_2 \dots b_f b_{f+1}$, where $|b_1| = |b_2| = \dots = |b_f| = \Delta$ and $|b_{f+1}| < \Delta$, and by carefully choosing uv and $u'v'$ from O_k conclude that b_2 contains a leftmost occurrence and f is proportional to $|O_k|$. Then, we argue that b_2 containing a leftmost occurrence implies

¹ They only claim $\mathcal{O}(n)$ time for fixed alphabets, however a closer look at the algorithm reveals that there are 3 phases: the first phase takes $\mathcal{O}(n)$ time (Theorem 6 and Lemma 7), the second phase also takes $\mathcal{O}(n)$ time (Section 4), and the third phase takes $\mathcal{O}(\sigma n)$ time (Lemma 11). Additionally, the algorithm assumes that the suffix tree is constructed in $\mathcal{O}(n)$ time, for larger alphabets this increases to $\mathcal{O}(\sigma n)$.

that, in fact, every b_j contains a leftmost occurrence, and thus $|O_k| = \mathcal{O}(|L_{k-2}|)$. Summing this over all k , and separately considering all k such that $|O_k|$ is less than the fixed constant, we are able to conclude that $\sum_k |O_k| = \sum_k \mathcal{O}(|L_k|) < \mathcal{O}(\sigma)$. See Section 4 for more details.

To obtain an efficient algorithm for reporting all order-preserving squares, we apply the order-preserving suffix tree as defined by Crochemore et al. [4]. This structure allows us to check if $w[i..i + 2\ell - 1]$ is an order-preserving square by checking if the LCA of two leaves is at string depth at least ℓ . First, we need to show how to construct the order-preserving tree in $\mathcal{O}(\sigma n)$ time. Second, we extend the above reasoning to efficiently generate only $\mathcal{O}(\sigma n)$ fragments that are then tested for being an order-preserving square in constant time each. While the underlying argument is essentially the same as when bounding the number of order-preserving squares, it needs to be executed differently for the purpose of an efficient implementation. See Section 5 for more details.

2 Preliminaries

Let $\Sigma = \{1, \dots, \sigma\}$ be a fixed finite alphabet of size σ . Let $|s|$ denote the length of a string s . For a string s , the character at position i of s is denoted by $s[i]$, and $s[i..j]$ is the fragment of s starting at position i and ending at position j . We call two strings u and v order-isomorphic, denoted by $u \approx v$, when $|u| = |v|$ and, for each i, j , we have $u[i] \leq u[j]$ if and only if $v[i] \leq v[j]$. The concatenation of two strings u and v is denoted by uv . A string of the form uv is called an order-preserving square, or op-square, when $u \neq v$ and $u \approx v$. We call u its left arm and v its right arm. We stress that a regular square, that is, a string of the form xx , is not an op-square. Two op-squares uv and $u'v'$ are *distinct as words* if and only if $uv \neq u'v'$.

A trie is a rooted tree, with every edge labeled with a single character and edges outgoing from the same node having distinct labels. A node u of a trie represents the string obtained by reading the labels on the path from the root to u . A compacted trie is obtained from a trie by replacing maximal paths consisting of nodes with exactly one child with single edges labeled by the concatenation of the labels of the edges on the path. A suffix tree T of a string w is a compacted trie whose leaves correspond to the suffixes of $w\$$. The string depth of a node u of T is the length of the string that it corresponds to. An explicit node of T is simply a node of T . An implicit node of T is a node of the non-compacted trie corresponding to T , or in other words a location on an edge of T .

Next, we need some definitions specific to order-isomorphism. Following Kubica et al. [22], we call b an op-border of a string $s[1..n]$ when $s[1..b] \approx s[n - b + 1..n]$. Following Gourdel et al. [13] (and Matsuoka et al. [25]), we call p an (initial) op-period of $s[1..n]$ when $s = b_1 b_2 \dots b_f b_{f+1}$ with $|b_1| = |b_2| = \dots = |b_f| = p$ and $|b_{f+1}| < p$ (so $f = \lfloor n/p \rfloor$), $b_1 \approx b_2 \approx \dots \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \dots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$. b_1, b_2, \dots, b_f are called the *blocks* defined by p in s , while b_{f+1} (possibly empty) is called the *incomplete block*. While in the classical setting p is a period of $s[1..n]$ if and only if $n - p$ is a border of $s[1..n]$, in the order-preserving setting, we only have an implication in one direction. For example, the string `aficdgbbeh` has an op-period 3 while 6 is not its op-border.

► **Proposition 3.** *If b is an op-border of $s[1..n]$ then $n - b$ is an initial op-period of $s[1..n]$.*

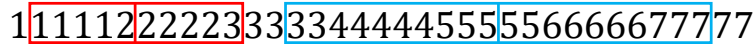
Proof. Let $p = n - b$ and $f = \lfloor n/p \rfloor$. We represent $s[1..n]$ as $s = b_1 b_2 \dots b_f b_{f+1}$ with $|b_1| = |b_2| = \dots = |b_f| = p$ and $|b_{f+1}| < p$. By b being an op-border of $s[1..n]$, we have $s[1..b] \approx s[n - b + 1..n]$, so $s[1..n - p] \approx s[p + 1..n]$. We observe that $s[1..n - p] = b_1 b_2 \dots b_{f-1} b_f[1..|b_{f+1}|]$ and $s[p + 1..n] = b_2 b_3 \dots b_f b_{f+1}$. Then, $b_1 b_2 \dots b_{f-1} b_f[1..|b_{f+1}|] \approx$

$b_2b_3 \dots b_f b_{f+1}$ implies $b_i \approx b_{i+1}$, for every $i = 1, 2, \dots, f-1$, and $b_i[1..|b_{f+1}|] \approx b_{i+1}[1..|b_{f+1}|]$, for every $i = 1, 2, \dots, f$. Hence, we obtain $b_1 \approx b_2 \approx b_3 \approx \dots \approx b_{f-1} \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \dots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$, so $p = n - b$ is indeed an initial op-period of $s[1..n]$. ◀

Due to Proposition 3, if b is an op-border of $s[1..n]$ then $s[1..n] = b_1b_2 \dots b_f b_{f+1}$, where $b_1b_2 \dots b_f[1..|b_{f+1}|] \approx b_2b_3 \dots b_f b_{f+1}$, $|b_1| = |b_2| = \dots = |b_f| = n - b$ and $|b_{f+1}| < p$ (so $f = \lfloor n/(n - b) \rfloor$), $b_1 \approx b_2 \approx \dots \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \dots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$. We will say that these blocks are defined by b .

3 Lower Bound

Recall that $\Sigma = \{1, \dots, \sigma\}$. We define a string $w = 11 \dots 122 \dots 2 \dots \sigma \sigma \dots \sigma$, that is, a concatenation of σ blocks, each consisting of k repetitions of the same character. We note that $|w| = \sigma k$. For $i = 1, 2, \dots, \lfloor \sigma/2 \rfloor$, we consider all fragments of w of length $2ik$ starting at positions $j = 1, 2, \dots, |w| - 2ik + 1$. For $j = 1 \pmod k$, the fragment is a concatenation of $2i$ blocks, each block consisting of k repetitions of the same character. For $j \neq 1 \pmod k$, the fragment starts with $r \in [1, k - 1]$ repetitions of the same character, then $2i - 1$ blocks, each block consisting of k repetitions of the same character, and finally $\ell = k - r$ repetitions of the same character. See Figure 1.



■ **Figure 1** The red box corresponds to an op-square of length 10 containing 3 different characters. The blue box corresponds to an op-square of length 20 containing 5 different characters.

Each such fragment is an op-square. For $j = 1 \pmod k$, both the left and the right arm consist of i blocks consisting of k repetitions of character $c, c + 1, \dots, c + i - 1$. For $j \neq 1 \pmod k$, both the left and the right arm consist of first r repetitions of character c , then $i - 1$ blocks consisting of k repetitions of characters $c + 1, c + 2, \dots, c + i - 2$, and then finally ℓ repetitions of character $c + i - 1$. Thus, the left and the right arm are always order-isomorphic. Further, for every choice of i and the starting position we obtain a different word, as two such fragments of the same length either start with different characters or differ in the length of the first block of the same character.

Now, we analyze the number of such op-squares in w . By considering every $1 \leq i \leq \lfloor \sigma/2 \rfloor$ and starting position $1, 2, \dots, |w| - 2ik + 1$, we obtain that the number of op-squares in w is at least:

$$\begin{aligned} \sum_{i=1}^{\lfloor \sigma/2 \rfloor} (|w| - 2ik + 1) &= \sum_{i=1}^{\lfloor \sigma/2 \rfloor} (\sigma k - 2ik + 1) = \lfloor \sigma/2 \rfloor \cdot (\sigma k - k(\lfloor \sigma/2 \rfloor + 1) + 1) \\ &\geq \lfloor \sigma/2 \rfloor \cdot (k(\lceil \sigma/2 \rceil - 1) + 1). \end{aligned}$$

For $\sigma \geq 3$, this is at least $\sigma^2 k/12 = \sigma n/12$ for any k . For $\sigma = 1, 2$, we additionally assume $k \geq 2$ and count op-squares of the form 1^{2i} , there are $\lfloor k/2 \rfloor \geq n/6 \geq \sigma n/12$ of them. Thus, in either case for every $k \geq 2$ we obtain a string of length $n = k\sigma$ over Σ containing $\sigma n/12$ op-squares that are distinct as words.

▶ **Theorem 4.** *For any alphabet $\Sigma = \{1, 2, \dots, \sigma\}$, there exists an infinite family of strings of length $n = k\sigma$ over Σ containing $\Omega(\sigma n)$ op-squares distinct as words.*

4 Upper bound

Our goal in this section is to upper bound the number of op-squares in a given string w of length n over the alphabet $\Sigma = \{1, \dots, \sigma\}$. Recall that uv is an op-square when $u \neq v$ and $u \approx v$. We will show that this number is $\mathcal{O}(\sigma n)$. As explained in the introduction, by the result of Brlek and Li [1], the number of regular squares, that is, fragments of the form uu that are distinct as words, is less than n . Thus, our result in fact allows us to upper bound the number of fragments of the form uv , where $u \approx v$, that are distinct as words by $\mathcal{O}(\sigma n)$.

We consider each suffix of w separately. For each suffix $w[i..n]$, we will upper bound the number of prefixes of $w[i..n]$ that are op-squares by $\mathcal{O}(\sigma)$. Therefore, to avoid cumbersome notation in the remaining part of this section we will assume that we have a string s of length m over the alphabet $\Sigma = \{1, \dots, \sigma\}$, and we want to upper bound the number of op-squares uv that are prefixes of s by $\mathcal{O}(\sigma)$. See Figure 2.



Figure 2 Green prefixes of s are op-squares.

Kociumaka et al. [20] observed that every op-square uv that is a prefix of s can be obtained as follows (recall that in our definition $u \neq v$). We call position i a leftmost occurrence and $s[i]$ a leftmost character when $s[j] \neq s[i]$ for every $j < i$. Then, there exists i and j such that both i and j are leftmost occurrences, where i belongs to u and j belongs to v , and further $|u| = j - i$. More formally:

► **Proposition 5** ([20, Lemma 4.2 and Corollary 4.3]). *We can construct an injective function g mapping op-squares that are prefixes of s to 2-element subsets of the alphabet as follows. We choose the smallest i belonging to v such that $s[i] = a$ does not occur in u , and let $s[i - |u|] = b$ be its counterpart in u , then set $g(uv) = \{a, b\}$. Both i and $i - |u|$ are leftmost occurrences.*

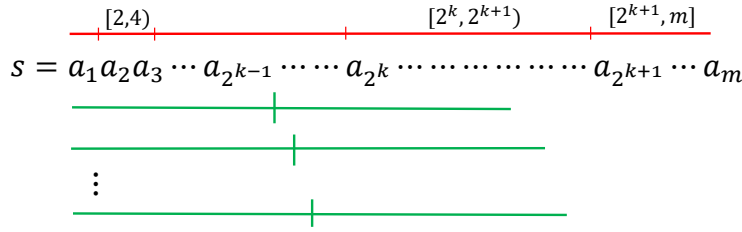
We split all op-squares that are prefixes of s into groups. Let O_k denote the group of op-squares that are prefixes of s having length at least 2^k and at most $2^{k+1} - 1$:

► **Definition 6.** $O_k = \{uv \mid u \neq v \text{ and } u \approx v \text{ and } 2^k \leq |uv| < 2^{k+1}\}$ for $0 \leq k \leq \log m$.

In other words, we split s into consecutive ranges of exponentially increasing lengths, such that the k -th range is of length 2^{k-1} , starts at position 2^k and ends at position $2^{k+1} - 1$ in s (where $0 \leq k \leq \log m$ and the final range may not be complete when $m < 2^{k+1} - 1$). Then, the set O_k consists of op-squares that end in the k -th range. See Figure 3.

The number of op-squares uv that are prefixes of s is $\sum_{k=0}^{\log m} |O_k|$. In order to upper bound the sum, we will separately upper bound the size of each group. We first need some propositions.

► **Proposition 7.** *For any $\{uv, u'v'\} \in O_k$ such that $|u| < |u'|$ and $\Delta = |u'| - |u|$, $|u| - \Delta$ is an op-border of both u and v .*

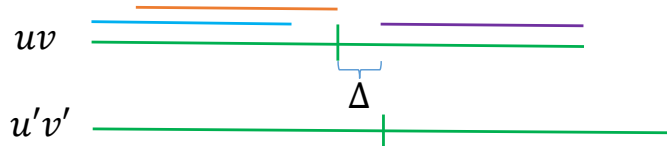


■ **Figure 3** Green prefixes of s are op-squares ending in the k -th range. The red line illustrates the ranges.

Proof. Because $u \approx v$ it is enough to show that $|u| - \Delta$ is an op-border of u . By the assumption that both uv and $u'v'$ are op-squares we have:

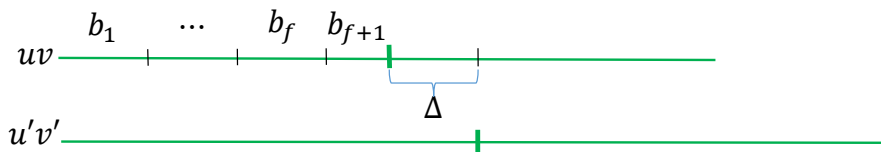
$$u[1..|u| - \Delta] = u'[1..|u| - \Delta] \approx v'[1..|u| - \Delta] = v[\Delta + 1..|u|] \approx u[\Delta + 1..|u|].$$

See Figure 4. ◀



■ **Figure 4** The green lines correspond to uv and $u'v'$. The blue line corresponds to $u[1..|u| - \Delta]$. The orange line corresponds to $u[\Delta + 1..|u|]$. The purple line corresponds to $v[\Delta + 1..|u|]$.

In the remaining part of this section, we will often consider $\{uv, u'v'\} \in O_k$ such that $|u| < |u'|$ and $\Delta = |u'| - |u|$. Then, by Proposition 7, we know that $|u| - \Delta$ is an op-border of u , and thus by Proposition 3 Δ is an initial op-period of u . Hence, u can be represented as a concatenation of $f = \lfloor |u|/\Delta \rfloor$ blocks b_1, b_2, \dots, b_f and one incomplete block b_{f+1} , where $|b_1| = |b_2| = \dots = |b_f| = \Delta$ and $|b_{f+1}| < \Delta$, such that $b_1 b_2 \dots b_f [1..|b_{f+1}|] \approx b_2 b_3 \dots b_f b_{f+1}$, $b_1 \approx b_2 \approx \dots \approx b_f$ and $b_1 [1..|b_{f+1}|] \approx b_2 [1..|b_{f+1}|] \approx \dots \approx b_f [1..|b_{f+1}|] \approx b_{f+1}$. See Figure 5. For brevity, in the remaining part of the paper we will describe this situation by saying that $\{uv, u'v'\} \in O_k$ define blocks $b_1, b_2, \dots, b_f, b_{f+1}$.



■ **Figure 5** Blocks defined by $\{uv, u'v'\}$ in u .

► **Proposition 8.** *If $|O_k| \geq 3$ then there exist $\{uv, u'v', u''v''\} \in O_k$ such that $0 < |u'| - |u|, |u''| - |u'| < 2^k / (|O_k| - 2)$.*

Proof. The length of every op-square in O_k belongs to $[2^k, 2^{k+1})$, thus the length of its left arm falls within $[2^{k-1}, 2^k)$. Let $O_k = \{u_1 v_1, u_2 v_2, \dots, u_\ell v_\ell\}$ with $|u_1| < |u_2| < \dots < |u_\ell|$. Then, for some $i \in \{1, 2, \dots, \lfloor (\ell - 1)/2 \rfloor\}$ we must have $|u_{2i+1}| < |u_{2i-1}| + 2^{k-1} / \lfloor (\ell - 1)/2 \rfloor$ (as otherwise we would have $u_\ell \geq u_1 + 2^{k-1}$). The sought op-squares are $u_{2i-1} v_{2i-1}, u_{2i} v_{2i}, u_{2i+1} v_{2i+1}$ because:

13:8 Order-Preserving Squares in Strings

$$\begin{aligned} |u_{2i}| - |u_{2i-1}|, |u_{2i+1}| - |u_{2i}| &< |u_{2i+1}| - |u_{2i-1}| < 2^{k-1}/\lfloor(\ell-1)/2\rfloor \\ &\leq 2^{k-1}/(\ell/2-1) = 2^k/(|O_k|-2). \end{aligned} \quad \blacktriangleleft$$

With all the propositions in hand, we are now ready for the technical lemmas. Our goal is to upper bound $\sum_k |O_k|$ by the number of leftmost occurrences. To this end, we need to show that, if some O_k is large then there are many leftmost occurrences in some range. This will be done by applying the following reasoning to the three op-squares chosen by applying Proposition 8. In the following, whenever we refer to a leftmost occurrence in block b we mean a leftmost occurrence falling within the positions in block b .

► **Lemma 9.** *If $|O_k| \geq 3$ then for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines b_1, \dots, b_f, b_{f+1} and $\{u'v', u''v''\}$ defines $b'_1, \dots, b'_f, b'_{f+1}$ there is a leftmost occurrence in block b_j such that $j \neq 1$ or there is a leftmost occurrence in block $b'_{j'}$ such that $j' \neq 1$.*

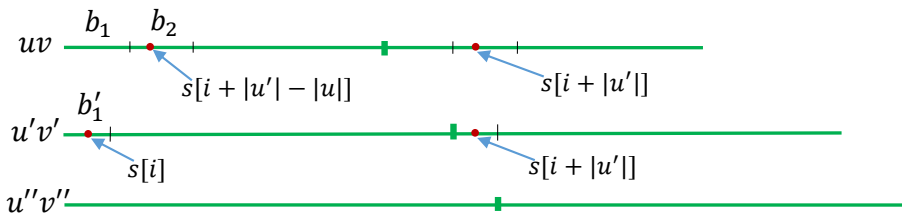
Proof. Let $\Delta = |u'| - |u|$ be the length of every block b_j and $\Delta' = |u''| - |u'|$ be the length of every block $b'_{j'}$. By Proposition 5, we know that there must be a leftmost occurrence i that falls within u' and its corresponding leftmost occurrence $i + |u'|$ that falls within v' . If $s[i]$ belongs to a block b'_j with $j \neq 1$ then we are done. Thus, we assume that i belongs to b'_1 . We claim that the leftmost occurrence $i + |u'|$ falls within v . To verify this, we calculate:

$$i + |u'| \leq \Delta' + |u'| = |u''| - |u'| + |u'| = |u''| < 2^k \leq |uv|.$$

We have established that $i + |u'|$ is a leftmost occurrence and falls within v . Thus, $s[i'] \neq s[i + |u'|]$ for every $i' \in [1, i + |u'|)$. Because $u \approx v$, this then implies that $s[i' - |u|] \neq s[i + |u'| - |u|]$ for every $i' \in [|u| + 1, i + |u'|)$. Thus, $i + |u'| - |u|$ is also a leftmost occurrence. We claim that $s[i + |u'| - |u|]$ cannot belong to b_1 . To verify this, we calculate:

$$i + |u'| - |u| \geq 1 + |u'| - |u| = 1 + \Delta.$$

Thus, we have found a leftmost occurrence $i + |u'| - |u|$ that falls within u and belongs to a block b_j with $j \neq 1$. See Figure 6. ◀

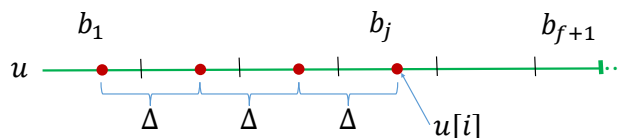


■ **Figure 6** The red points correspond to the leftmost occurrences considered in the proof of Lemma 9.

Next, we show that if $\{uv, u'v'\} \in O_k$ define blocks $b_1, b_2, \dots, b_f, b_{f+1}$ such that there is a leftmost occurrence in block b_j for some $j \neq 1$ then, in fact, there is a leftmost occurrence in every block b_j . This reasoning is done in two steps.

► **Lemma 10.** *Let b be an op-border of $u = s[1..|u|]$ that defines blocks $b_1, b_2, \dots, b_f, b_{f+1}$, and assume that there is a leftmost occurrence in block b_j , for some $j \in [1, f + 1]$. Then there is a leftmost occurrence in every block b_1, b_2, \dots, b_j .*

Proof. Let $\Delta = |b_1| = |b_2| = \dots = |b_f|$ and $|b_{f+1}| < \Delta$. By induction, it is enough to show that if there is a leftmost occurrence in block b_j for some $j \geq 2$ then there is a leftmost occurrence in block b_{j-1} . Let i be a leftmost occurrence that belongs to b_j . Then $u[i'] \neq u[i]$ for every $i' \in [1, i)$. Because $b_1 b_2 \dots b_{f-1} b_f [1..|b_{f+1}|] \approx b_2 b_3 \dots b_f b_{f+1}$, this implies $u[i' - \Delta] \neq u[i - \Delta]$ for every $i' \in [\Delta + 1, i)$. But then $i - \Delta$ is also a leftmost occurrence, and it belongs to b_{j-1} as required. See Figure 7. ◀



■ **Figure 7** Each red point corresponds to a leftmost character at the same relative position in every block b_1, b_2, \dots, b_j .

► **Lemma 11.** *Let b be an op -border of $u = s[1..|u|]$ that defines blocks $b_1, b_2, \dots, b_f, b_{f+1}$, and assume that there is a leftmost character in block b_2 . Then there is a leftmost occurrence in every block b_1, b_2, \dots, b_f .*

Proof. Let $\Delta = |b_1| = |b_2| = \dots = |b_f|$ and $|b_{f+1}| < \Delta$. By assumption, there is a leftmost character $s[i]$ in block b_2 , where $i \in [\Delta + 1, 2\Delta]$. Our goal is to show that there is a leftmost occurrence in every block b_1, b_2, \dots, b_f .

Because $b_1 b_2 \dots b_f [1..|b_{f+1}|] \approx b_2 b_3 \dots b_f b_{f+1}$, each position $x \in [1..|\Delta]$ satisfies exactly one of the following possibilities:

1. $s[x + p \cdot \Delta]$ is the same, for all integers $p \in [0, f)$,
2. $s[x + p \cdot \Delta] < s[x + (p + 1) \cdot \Delta]$ for all integers $p \in [0, f - 1)$,
3. $s[x + p \cdot \Delta] > s[x + (p + 1) \cdot \Delta]$ for all integers $p \in [0, f - 1)$.

Note that $i_0 = i - \Delta$ satisfies (2) or (3), because $s[i]$ is different than $s[1], s[2], \dots, s[i - 1]$, so in particular $s[i - \Delta] \neq s[i]$. By reversing the order of the alphabet, it is enough to establish the lemma assuming that i_0 satisfies (2). To this end, we choose some positions i_1, i_2, \dots, i_ℓ in b_1 as follows. Let C be the set of characters that appear in b_1 . The position $i_1 \in [1, \Delta]$ is chosen so that $s[i_1]$ is the strict successor of $s[i_0]$ in C , then $i_2 \in [1, \Delta]$ is chosen so that $s[i_2]$ is the strict successor of $s[i_1]$ in C , and so on. If there are multiple choices for the next $i_j \in [1, \Delta]$ then we take the smallest. We stop when one of the following two possibilities holds:

- (a) $s[i_{\ell+1}]$ is not defined, i.e. $s[i_\ell]$ is the largest character in b_1 .
- (b) $i_{\ell+1}$ satisfies (1) or (3).

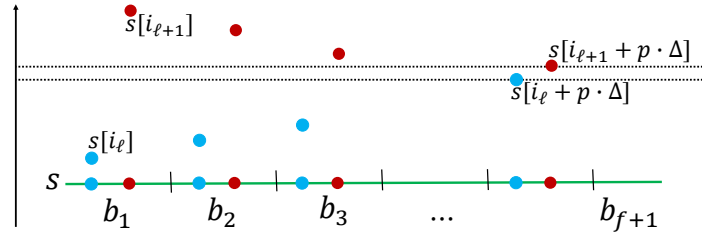
Notice that, by definition, the positions i_0, i_1, \dots, i_ℓ all satisfy (2). Further, $s[i_0]$ is a leftmost character because $s[i]$ is a leftmost character, so $s[i'] \neq s[i]$ for every $i' \in [1, i)$, and $b_1 \approx b_2$ so $s[i' - \Delta] \neq s[i - \Delta]$ for every $i' \in [\Delta + 1, i)$. Next, we note that $s[i_1], \dots, s[i_\ell]$ are all leftmost characters because we are always choosing the smallest i_j such that $s[i_j]$ is equal to a specific character, for $j = 1, 2, \dots, \ell$.

We summarize the situation so far. For every integer $p \in [0, f)$, the fragment $s[i_\ell + p \cdot \Delta]$ belongs to block b_{p+1} , and we want to show that it is a leftmost character. We know that $s[i_\ell]$ is a leftmost character, thus by $b_1 \approx b_{p+1}$ we obtain that $s[i_\ell + p \cdot \Delta]$ does not occur earlier in b_{p+1} . We need to establish that it also does not occur earlier in b_1, b_2, \dots, b_p . We separately consider the two possible cases (a) and (b).

13:10 Order-Preserving Squares in Strings

- (a) $s[i_{\ell+1}]$ is not defined, i.e. $s[i_{\ell}]$ is the largest character in b_1 . We know that i_{ℓ} satisfies (2), so $s[i_{\ell}] < s[i_{\ell} + \Delta] < \dots < s[i_{\ell} + (p-1) \cdot \Delta] < s[i_{\ell} + p \cdot \Delta]$. For all integers $q \in [0, p)$, by $b_1 \approx b_{q+1}$ we obtain that $s[i_{\ell} + q \cdot \Delta]$ is the largest character in b_{q+1} . So in fact $s[i_{\ell} + p \cdot \Delta]$ is larger than all characters in the whole block b_{q+1} , for every integer $q \in [0, p)$, making $i_{\ell} + p \cdot \Delta$ a leftmost occurrence.
- (b) $i_{\ell+1}$ is defined and satisfies (1) or (3), so $s[i_{\ell+1}] \geq s[i_{\ell+1} + \Delta] \geq \dots \geq s[i_{\ell+1} + (p-1) \cdot \Delta] \geq s[i_{\ell+1} + p \cdot \Delta]$. See Figure 8. We know that i_{ℓ} satisfies (2), so $s[i_{\ell}] < s[i_{\ell} + \Delta] < \dots < s[i_{\ell} + (p-1) \cdot \Delta] < s[i_{\ell} + p \cdot \Delta]$. Recall that $s[i_{\ell+1}]$ is a strict successor of $s[i_{\ell}]$ in b_1 . Thus, for every $i' \in [1, \Delta]$ we have that $s[i']$ does not belong to the interval $(s[i_{\ell}], s[i_{\ell+1}])$. Because we have $b_1 \approx b_{q+1}$, for every integer $q \in [0, p)$, this implies $s[i' + q \cdot \Delta]$ does not belong to the interval $(s[i_{\ell} + q \cdot \Delta], s[i_{\ell+1} + q \cdot \Delta])$. As observed earlier, $s[i_{\ell} + q \cdot \Delta] < s[i_{\ell} + p \cdot \Delta]$ and $s[i_{\ell+1} + q \cdot \Delta] \geq s[i_{\ell+1} + p \cdot \Delta]$. We conclude that, for every $i' \in [1, \Delta]$, we have that $s[i' + q \cdot \Delta]$ does not belong to the interval $[s[i_{\ell} + p \cdot \Delta], s[i_{\ell+1} + p \cdot \Delta])$ (the interval is non-empty, as both positions belong to the same block b_q , and by $b_{q+1} \approx b_1$ we have that $s[i_{\ell+1} + q \cdot \Delta]$ is a strict successor of $s[i_{\ell} + q \cdot \Delta]$ in b_q). In particular, $s[i' + q \cdot \Delta] \neq s[i_{\ell} + p \cdot \Delta]$, so $s[i_{\ell} + p \cdot \Delta]$ does not occur in b_{q+1} , making it a leftmost character.

Hence, for every integer $p \in [0, f)$, position $i_{\ell} + p \cdot \Delta$ is a leftmost occurrence. \blacktriangleleft



■ **Figure 8** The red points correspond to $s[i_{\ell+1}], \dots, s[i_{\ell+1} + p \cdot \Delta]$. The blue points correspond to $s[i_{\ell}], \dots, s[i_{\ell} + p \cdot \Delta]$. The black arrow illustrates the character's axis.

By combining the above lemmas we obtain the following conclusion.

► **Lemma 12.** *If $|O_k| \geq 3$ then for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines b_1, \dots, b_f, b_{f+1} and $\{u'v', u''v''\}$ defines $b'_1, \dots, b'_f, b'_{f+1}$ there is a leftmost occurrence in every block b_1, b_2, \dots, b_f or there is a leftmost occurrence in every block b'_1, b'_2, \dots, b'_f .*

Proof. Recall that by Proposition 7, $|u'| - |u|$ is an op-border of $u = s[1..|u|]$ while $|u''| - |u'|$ is an op-border of $u' = s[1..|u'|]$. By Lemma 10 there is a leftmost occurrence in block b_2 or in block b'_2 . Then, by Lemma 11 applied either to the blocks $b_1, b_2, \dots, b_f, b_{f+1}$ defined by the op-border $|u'| - |u|$ or the blocks $b'_1, b'_2, \dots, b'_f, b'_{f+1}$ defined by the op-border $|u''| - |u'|$, there is a leftmost occurrence in every block b_1, b_2, \dots, b_f or in every block b'_1, b'_2, \dots, b'_f . \blacktriangleleft

We are now ready to upper bound $\sum_k |O_k|$ by the number of leftmost characters. We will show that, if some O_k is large then there are many leftmost characters in some range. To this end, we define groups of leftmost occurrences. Let L_k be the set of the leftmost occurrences i such that $i \in [2^k, 2^{k+1})$:

► **Definition 13.** $L_k = \{i \mid i \in [2^k, 2^{k+1}) \wedge \forall_{j \in [1, i)} s[i] \neq s[j]\}$ for $0 \leq k \leq \log m$.

Note that the groups are disjoint, i.e. $L_k \cap L_{k'} = \emptyset$ for any k and k' . Thus $\sum_{k=0}^{\log m} |L_k| \leq \sigma$. With this definition in hand, we are ready to show the main technical lemma.

► **Lemma 14.** *The number of op-squares that are prefixes of s is $\mathcal{O}(\sigma)$.*

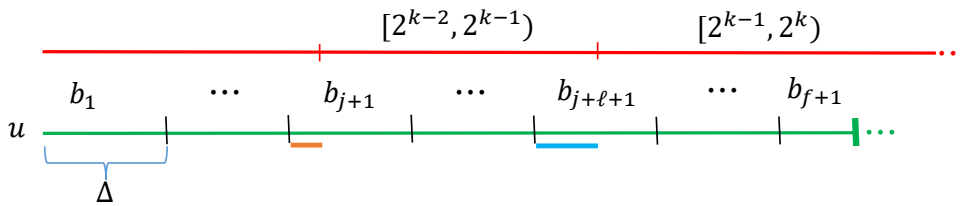
Proof. To establish the lemma we want to connect $|O_k|$ with $|L_k|$, and then sum over all possible values of k . $k = 0, 1$ will be considered separately, and for larger k we apply different arguments for $|O_k| \geq 11$ and $|O_k| \leq 10$.

We first consider $k \geq 2$ such that $|O_k| \geq 11$. In particular, when $|O_k| \geq 3$, so by Proposition 8, there exist $\{uv, u'v', u''v''\} \in O_k$ such that $0 < |u'| - |u|, |u''| - |u'| < 2^k / (|O_k| - 2)$. By Lemma 12, for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines b_1, \dots, b_f, b_{f+1} in u and $\{u'v', u''v''\}$ defines $b'_1, \dots, b'_{f'}, b'_{f'+1}$ in u' either there is a leftmost occurrence in every block b_1, b_2, \dots, b_f or there is a leftmost occurrence in every block $b'_1, b'_2, \dots, b'_{f'}$. In either case, we have found $\{uv, u'v'\} \in O_k$ with $0 < \Delta < 2^k / (|O_k| - 2)$, where $\Delta = |u'| - |u|$, such that $\{uv, u'v'\}$ defines b_1, \dots, b_f, b_{f+1} with $f = \lfloor |u|/\Delta \rfloor$ and there is a leftmost occurrence in every block b_1, b_2, \dots, b_f . We want to establish a lower bound on the number of leftmost occurrences in L_{k-2} . To this end, it is enough to show a lower bound on the number of blocks b_i that are fully contained in the range $[2^{k-2}, 2^{k-1})$. Recall that $|u| \in [2^{k-1}, 2^k)$, and $u = b_1 b_2 \dots b_f b_{f+1}$. Thus, the fragment $u[2^{k-2}..2^{k-1} - 1]$ consists of a suffix (possibly empty) of some b_j , then $b_{j+1}, b_{j+2}, \dots, b_{j+\ell}$, and then a prefix of $b_{j+\ell+1}$ (where $b_{j+\ell+1}$ might be the incomplete block b_{f+1} that should not be counted in the lower bound). Thus, the number of blocks b_i that are fully contained in the range $[2^{k-2}, 2^{k-1})$ is at least $\lfloor 2^{k-2}/\Delta \rfloor - 1$. See Figure 9. Combining this with the upper bound on Δ , we obtain the following inequality:

$$|L_{k-2}| \geq \left\lfloor \frac{2^{k-2}}{\Delta} \right\rfloor - 1 \geq \frac{2^{k-2}}{\Delta} - 2 > \frac{|O_k| - 2}{4} - 2 = \frac{|O_k| - 10}{4}.$$

Using the assumption $|O_k| \geq 11$, we conclude that $|L_{k-2}| > |O_k|/44$. Hence:

$$\sum_{k \geq 2: |O_k| \geq 11} |O_k| < \sum_{k \geq 2} 44 \cdot |L_{k-2}| \leq 44 \sum_k |L_k| \leq 44 \cdot \sigma.$$



■ **Figure 9** The orange line corresponds to the suffix of b_j . The blue line corresponds to the prefix of $b_{j+\ell+1}$. The red line illustrates the ranges.

Next, we consider $k \geq 2$ such that $|O_k| \leq 10$. Of course, we have the trivial upper bound $\sum_{k: |O_k| \leq 10} |O_k| \leq \sum_{k=0}^{\log m} 10 = \mathcal{O}(\log m)$. As in the previous case, we want to use the leftmost occurrences to improve the bound. Recall that, by Proposition 5, every op-square $uv \in O_k$ is defined by a pair of leftmost occurrences i and j , where i belongs to u and j belongs to v . Because $|uv| \in [2^k, 2^{k+1})$, we conclude that j falls within the range $[2^{k-1} + 1, 2^{k+1})$, so must belong to $L_{k-1} \cup L_k$. Hence, the set O_k can be non-empty only when L_{k-1} or L_k is non-empty. Hence:

13:12 Order-Preserving Squares in Strings

$$\begin{aligned} \sum_{k \geq 2: |O_k| \leq 10} |O_k| &\leq \sum_{k \geq 2: |O_k| > 0} 10 \leq \sum_{k \geq 2: |L_{k-1}| > 0} 10 + \sum_{k \geq 2: |L_k| > 0} 10 \\ &\leq 10 \sum_{k \geq 2} |L_{k-1}| + 10 \sum_{k \geq 2} |L_k| \leq 20\sigma. \end{aligned}$$

To upper bound $\sum_k |O_k|$, we split the sum into three parts. For $k = 0, 1$, we have $|O_0| \leq 1$ and $|O_1| \leq 2$. Then, for $k \geq 2$ we separately consider all k with $|O_k| \geq 11$ and $|O_k| \leq 10$ and plug in the above upper bounds. Overall, we obtain:

$$\sum_k |O_k| \leq 1 + 2 + 44 \cdot \sigma + 20 \cdot \sigma = \mathcal{O}(\sigma).$$

Thus, the number of op-squares that are prefixes of s is $\mathcal{O}(\sigma)$. \blacktriangleleft

We conclude the section with the main theorem.

► **Theorem 15.** *The number of op-squares in a string w of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$.*

Proof. We consider each suffix of w separately. For each suffix $w[i..n]$, we apply Lemma 14 to conclude that the number of op-squares that are prefixes of $w[i..n]$ is upper bounded by $\mathcal{O}(\sigma)$. Thus, summing over all i we obtain that the number of op-squares in w is $\mathcal{O}(\sigma n)$. \blacktriangleleft

5 Algorithm

In this section, we describe the algorithm that reports all occurrences of op-squares in a string $w[1..n]$ over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time.

The high-level idea of the algorithm is to generate $\mathcal{O}(\sigma n)$ candidates for op-squares and then test each of them in constant time, see the pseudocode in Algorithm 1. To this end, we first describe a mechanism for checking if $w[i..i + \ell - 1] \approx w[i + \ell..i + 2\ell - 1]$ in constant time. This can be implemented with an LCA query on the order-preserving suffix tree of w , as explained in [3]. However, we need to explain how to construct this structure in $\mathcal{O}(\sigma n)$ time.

Order-preserving suffix tree. Following [3], for a string $w[1..n]$ we define $\text{code}(w)$ as $(\phi(w, 1), \phi(w, 2), \dots, \phi(w, n))$, where $\phi(w, i) = (\text{prev}_<(w, i), \text{prev}_=(w, i))$ and $\text{prev}_<(w, i) = \{k < i : w[k] < w[i]\}$, $\text{prev}_=(w, i) = \{k < i : w[k] = w[i]\}$. We observe that $\text{code}(w) = \text{code}(w')$ if and only if $w \approx w'$. Then, the order-preserving suffix tree of $w[1..n]$ is the compacted trie of all strings of the form $\text{code}(w[i..n])\$,$ for $i = 1, 2, \dots, n$. It is easy to see that $w[i..i + \ell - 1] \approx w[i + \ell..i + 2\ell - 1]$ if and only if the lowest common ancestor of the leaves corresponding to $\text{code}(w[i..n])\$$ and $\text{code}(w[i + \ell..n])\$$ is at string depth at least ℓ . Therefore, assuming that we have already built the order-preserving suffix tree of $w[1..n]$, such a test can be implemented in constant time after $\mathcal{O}(n)$ preprocessing for LCA queries [15]. It remains to explain how to construct the order-preserving suffix tree. We stress that while [3] does provide an efficient $\mathcal{O}(n \log n / \log \log n)$ time construction algorithm (in fact, the full version [4] further improves the time complexity to $\mathcal{O}(n \sqrt{\log n})$), such complexity is incompatible with our goal. Due to the lack of space, the proof is moved to the appendix.

► **Lemma 16.** *Given a string $w[1..n]$ over an alphabet of size σ , we can construct its order-preserving suffix tree in $\mathcal{O}(\sigma n)$ time and space.*

The main part of the algorithm is efficiently generating $\mathcal{O}(\sigma n)$ candidates for op-squares. Then, each of them is tested in constant time as explained above, assuming the preprocessing from Lemma 16.

■ **Algorithm 1** Report all occurrences of op-squares in a string $w[1..n]$ over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time.

```

1 Preprocess  $w[1..n]$  for retrieving the characters of any  $\text{code}(w[i..n])\$$ 
2 Construct the order-preserving suffix tree  $T$ 
3 Preprocess  $T$  for LCA queries
4  $i \leftarrow n$ 
5 while  $i > 0$  do
6    $s \leftarrow w[i..n]$ 
7   Retrieve the leftmost occurrences  $x_1, x_2, \dots, x_t$  in  $s$ 
8   foreach  $x_j$  that is the smallest or the largest of its group  $L_k$  do
9      $s' \leftarrow s[1..2^{k-1}]$ 
10    foreach fragment  $s[y..y + 2^{k-1} - 1] \approx s'$  such that  $x_j \in [y, y + 2^{k-1} - 1]$  do
11      | Store  $s[1..2(y-1)]$  as a candidate in  $R[x_j][k][i]$ 
12    end
13  end
14  foreach candidate  $s[1..2(y-1)]$  in  $R[x_j][k][i]$  do
15     $v_1 \leftarrow$  the leaf corresponding to  $\text{code}(w[i..n])\$$  in  $T$ 
16     $v_2 \leftarrow$  the leaf corresponding to  $\text{code}(w[i + (y-1)..n])\$$  in  $T$ 
17    if the string depth of  $\text{LCA}_T(v_1, v_2)$  is at least  $y-1$  then
18      | Report  $s[1..2(y-1) - 1]$  as an op-square
19    end
20  end
21   $i \leftarrow i - 1$ 
22 end

```

Leftmost occurrences. As in the proof of the $\mathcal{O}(\sigma n)$ upper bound on the number of op-squares, we will consider the suffixes of the input string $w[1..n]$ one-by-one. For $i = n, n-1, \dots, 1$ in this order, let $s = w[i..n]$ be the currently considered suffix, and $x_1 < x_2 < \dots < x_t$ be the leftmost occurrences in s . By spending $\mathcal{O}(\sigma)$ time per each suffix, we can assume that the positions x_1, x_2, \dots, x_t are known, as after moving from $w[i..n]$ to $w[i-1..n]$ we only have to insert the new leftmost occurrence $i-1$ and possibly remove the previous leftmost occurrence i' such that $w[i-1] = w[i']$ (unless $w[i-1]$ has not been seen before), which can be done in $\mathcal{O}(t) = \mathcal{O}(\sigma)$ time. By Proposition 5, every prefix of s that is an op-square can be obtained by choosing two leftmost characters at positions x_q and x_j , where $q < j$, and setting the length of the possible square to be $2(x_j - x_q)$. This gives us $\mathcal{O}(\sigma^2)$ candidates for prefixes that could be op-squares. However, our goal is to generate only $\mathcal{O}(\sigma)$ such candidates. To achieve this goal, we first provide some combinatorial properties in Lemma 17, Lemma 18, and Proposition 19.

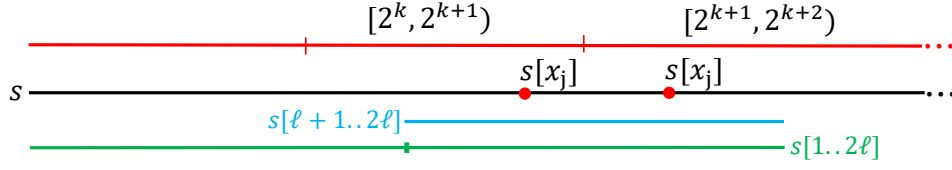
Recall that all leftmost occurrences are partitioned into groups L_0, L_1, \dots . Next, we show that it is enough to consider x_j that is the smallest or the largest element in its group.

► **Lemma 17.** *Consider an op-square $s[1..2\ell]$. Then there exists $q < j$ such that the leftmost occurrences x_q and x_j satisfy $x_j - x_q = \ell$, $x_j \in [\ell + 1, 2\ell]$ and x_j is either the smallest or the largest element of its group.*

The proof is described in the appendix. Figure 10 illustrates the scenario of the lemma.

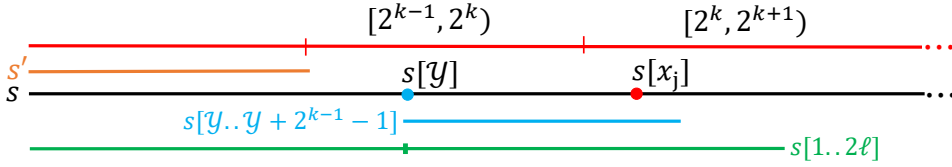
To generate the candidates, we iterate over all j such that x_j is the smallest or largest element of its group L_k . Consider $q < j$ such that $x_j - x_q = \ell$ and $x_j \in [\ell + 1, 2\ell]$ for an op-square $s[1..2\ell]$. Then, because $x_j \in [2^k, 2^{k+1})$, $\ell \geq 2^{k-1}$. To avoid clutter, let $s' = s[1..2^{k-1}]$.

13:14 Order-Preserving Squares in Strings



■ **Figure 10** The op-square $s[1..2\ell]$ is colored in green. The red points are the two possibilities for $s[x_j]$. $s[\ell+1..2\ell]$ is colored in blue. The red line illustrates the ranges.

Because $s[1..2\ell]$ is assumed to be an op-square, we have $s[\ell.. \ell + 2^{k-1} - 1] \approx s'$. This suggests the following natural strategy to generate the candidates: we iterate over all fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$, and output $s[1..2(y-1)]$ as a possible op-square (as explained earlier, each such candidate is then tested in constant time). See Figure 11. We first bound the number of such fragments by $\mathcal{O}(1 + |L_{k-2}|)$, and then explain how to generate them in the same time complexity.



■ **Figure 11** The op-square $s[1..2\ell]$, the fragment $s[y..y + 2^{k-1} - 1]$, and s are colored in green, blue, and black, respectively. s' is colored in orange. The red line illustrates the ranges.

► **Lemma 18.** *The number of fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ is upper bounded by $\mathcal{O}(1 + |L_{k-2}|)$.*

The proof of the lemma relies on showing that ℓ' is an op-border of s' and thus we can define blocks of length $\Delta = 2^{k-1} - \ell'$ in s' and then apply Lemma 10 and Lemma 11 to achieve the desired bound. The full proof is described in the appendix. Hence, for every k such that L_k is non-empty, we generate $\mathcal{O}(1 + |L_{k-2}|)$ candidates. The overall number of candidates generated by following the above strategy is $\sum_{k:L_k \neq \emptyset} \mathcal{O}(1 + |L_{k-2}|) = \mathcal{O}(\sigma + \sum_k |L_{k-2}|) = \mathcal{O}(\sigma)$ as promised. It remains to show how to access all fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ in time proportional to their number.

Accessing candidates. We will solve a more general problem, and show how to ensure that, when considering $s = w[i..n]$, for every leftmost occurrence x_j in s we have access to a list of all fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s[1..2^{k-1} - 1] \approx s[y..y + 2^{k-1} - 1]$, where $2^k \leq x_j < 2^{k+1}$. We call this list the *result* for i and x_j .

Recall that $s = w[i..n]$, and we consider $i = n, n-1, \dots, 1$ in this order. When we consider $s = w[i..n]$, position i becomes a leftmost occurrence and remains to be so until we reach $s = w[\text{prev}_i..n]$ such that $w[i] = w[\text{prev}_i]$ (possibly, it is a leftmost occurrence till the very end of the scan). We can calculate prev_i for every i in $\mathcal{O}(\sigma n)$ time by maintaining a list of leftmost occurrences as described earlier. We say that a position i is k -active at position i' when $i' \in [\text{prev}_i, i]$ and $2^k \leq i - i' + 1 < 2^{k+1}$. We observe that, as we consider longer and longer suffixes of w , position i is first 0-active, then 1-active, and so on until it becomes

k_i -active, and then it is never active again. Further, indices i' such that i is k -active at i' form a contiguous range $[\text{begin}_{i,k}, \text{end}_{i,k}]$ (the length of each such range is 2^k , except possibly for $k = k_i$ when it is shorter). The total length of these ranges is small as shown below.

► **Proposition 19.** $\sum_{i,k:k \leq k_i} 2^k = \mathcal{O}(\sigma n)$

Proof. For $k \geq 1$ we can upper bound 2^k by $2 \cdot |[\text{begin}_{i,k-1}, \text{end}_{i,k-1}]|$. Then the sum becomes:

$$\sum_{i,k:k \leq k_i} 2^k = n + 2 \cdot \sum_{i,k:1 \leq k \leq k_i} |[\text{begin}_{i,k-1}, \text{end}_{i,k-1}]| \leq n + 2 \cdot \sum_{i,k:k \leq k_i} |[\text{begin}_{i,k}, \text{end}_{i,k}]|.$$

We observe that every position $i' \in [\text{begin}_{i,k}, \text{end}_{i,k}]$ in the suffix $w[i..n]$ corresponds to the relative position $i - i' + 1$ being a leftmost occurrence in the suffix $w[i'..n]$. Because there are at most σ leftmost characters in any suffix $w[i'..n]$, this allows us to upper bound the sum by $\mathcal{O}(\sigma n)$. ◀

Storing candidates. This allows us to physically store the results as follows. For every leftmost occurrence x , we have an array indexed by $k \leq k_i$, denoted $R[x]$. Each entry of this array is an array indexed by $i' \in [\text{begin}_{i,k}, \text{end}_{i,k}]$, denoted $R[x][k]$. Finally, each entry of that array, denoted $R[x][k][i']$, is a pointer to a list of ys such that $x \in [y, y + 2^{k-1} - 1]$ and $w[i'..i' + 2^{k-1} - 1] \approx w[y..y + 2^{k-1} - 1]$ (note that it is a pointer to a list and not its physical copy). The arrays allow us to access the result for every x_j , $k \leq k_{x_j}$ and i' in constant time, by retrieving the pointer $R[x_j][k][i']$ (where we first verify that $i' \in [\text{begin}_{i,k}, \text{end}_{i,k}]$). The total length of all arrays $R[x]$ is only $\mathcal{O}(\sigma n)$ by Proposition 19. Further, the total length of all lists of occurrences that we need to prepare (assuming that we store every $R[x][k][i]$ as a pointer to such a list and not their physical copies) is also $\mathcal{O}(\sigma n)$ by the following argument. Consider i and $k \leq k_i$. Then, we need a list of positions y such that $i \in [y, y + 2^{k-1} - 1]$ and $w[y..y + 2^{k-1} - 1]$ is order-isomorphic to a specific string s' . Thus, we can partition all positions y such that $i \in [y, y + 2^{k-1} - 1]$ into groups corresponding to order-isomorphic fragments $w[y..y + 2^{k-1} - 1]$, and then store a pointer to the appropriate list (possibly null, if there is no y). The total number of positions y , over all i and $k \leq k_i$, is $\mathcal{O}(\sigma n)$ by Proposition 19, which bounds the total length of all the lists.

Generating candidates. It remains to describe how to efficiently calculate the results. This requires partitioning all fragments $w[y..y + 2^{k-1} - 1]$ such that $i \in [y, y + 2^{k-1} - 1]$ and $k \leq k_i$ into order-isomorphic groups, and finding for every $i, k \leq k_i, i' \in [\text{begin}_{i,k}, \text{end}_{i,k}]$ a pointer to the list of fragments $w[y..y + 2^{k-1} - 1]$ with $i \in [y, y + 2^{k-1} - 1]$ that are order-isomorphic to $w[i'..i' + 2^{k-1} - 1]$. Both steps can be implemented with the order-preserving suffix tree that is preprocessed in $\mathcal{O}(n)$ time and space for computing a (deterministic) fingerprint of any code($w[x..x + 2^\ell - 1]$) in constant time. Here, a fingerprint is meant as an integer consisting of $\mathcal{O}(\log n)$ bits, denoted $\text{fingerprint}_\ell(x)$, such that $\text{fingerprint}_\ell(x) = \text{fingerprint}_\ell(x')$ iff $\text{code}(w[x..x + 2^\ell - 1]) = \text{code}(w[x'..x' + 2^\ell - 1])$ (or equivalently $w[x..x + 2^\ell - 1] \approx w[x'..x' + 2^\ell - 1]$). We first describe such a mechanism and then provide a more detailed description of how to apply it. The proof of the following lemma directly follows from prior work [11, 12] and is described in the appendix.

► **Lemma 20.** *A compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time, so that for any leaf u and integer k we can query in constant time for a $\mathcal{O}(\log n)$ -bit fingerprint of the ancestor of u at string depth 2^k .*

We apply Lemma 20 on the order-preserving suffix tree. This allows us to calculate any $\text{fingerprint}_\ell(x)$ with the required properties in constant time. Now consider any i and $k \leq k_i$. We first compute $\text{fingerprint}_{k-1}(y)$ for every y such that $i \in [y, y + 2^{k-1} - 1]$. This takes $\mathcal{O}(2^k)$ time. Next, we compute $\text{fingerprint}_{k-1}(i')$ for every $i' \in [\text{begin}_{i,k}, \text{end}_{i,k}]$, also in $\mathcal{O}(2^k)$ time because $|\text{begin}_{i,k}, \text{end}_{i,k}]| \leq 2^{k-1}$. We sort all fingerprints and partition them into groups corresponding to order-isomorphic fragments. We need to implement this step in $\mathcal{O}(2^k)$ time as well. To this end, we observe that we need to sort $\mathcal{O}(2^k)$ integers consisting of $\mathcal{O}(\log n)$ bits, which can be done with radix sort in $\mathcal{O}(2^k + n)$ time. To avoid paying $\mathcal{O}(n)$ for each i and $k \leq k_i$, we observe that this is an offline problem, and all sets corresponding to different i and $k \leq k_i$ can be sorted together. In more detail, we sort tuples of the form $(i, k_i, \text{fingerprint}_\ell(y), y)$ and $(i, k_i, \text{fingerprint}_\ell(i'), i')$. The total number of all tuples is $\mathcal{O}(\sigma n)$ by Lemma 19 and, as each of them can be treated as an integer consisting of $\mathcal{O}(\log n)$ bits, they can be sorted in $\mathcal{O}(\sigma n + n) = \mathcal{O}(\sigma n)$ time. Then, we extract the results for each i and $k \leq k_i$ from the output. For each i and $k \leq k_i$, we consider every group of equal fingerprints. From each group, we first create a list containing all positions y corresponding to $\text{fingerprint}_{k-1}(y)$ belonging to the group. Then, for every $\text{fingerprint}_{k-1}(i')$ belonging to the group we store a pointer to this list. Overall, this takes $\mathcal{O}(\sigma n)$ time and allows us to compute all the results in the same time complexity.

6 Open Problems

An interesting follow-up to our results is first bounding the number of order-preserving squares that are not order-isomorphic, and then designing an algorithm that reports all such squares. In addition, investigating the bounds for parameterized squares is of interest. Moreover, we are not aware of an algorithm reporting parameterized squares in a string, hence, designing such an algorithm is desired.

References

- 1 S. Brlek and S. Li. On the number of squares in a finite word. *arXiv*, 2022. [arXiv:2204.10204](https://arxiv.org/abs/2204.10204).
- 2 R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *STOC*, pages 407–415, 2000.
- 3 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Wal e n. Order-preserving incomplete suffix trees and order-preserving indexes. *SPIRE*, 8214:84–95, 2013.
- 4 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Wal e n. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016.
- 5 M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13:405–425, 1995.
- 6 L. J. Cummings and W. F. Smyth. Weak repetitions in strings. *J. Combinatorial Math. Combinatorial Comput.*, 24:33–48, 1997.
- 7 A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015.
- 8 P. Erd os. Some unsolved problems. *Magy. Tud. Akad. Mat. Kut. Int ez. K ozl.*, 6:221–254, 1961.
- 9 A. A. Evdokimov. Strongly asymmetric sequences generated by a finite number of symbols. *Dokl. Akad. Nauk SSSR*, 179(6):1268–1271, 1968.
- 10 A. S. Fraenkel and J. Simpson. How many squares can a string contain? *Combinatorial Theory, Series A*, 82(1):112–120, 1998.
- 11 P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. *ESA*, 6942:421–432, 2011.

- 12 P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. *arXive*, 2011. [arXiv:1104.4203](https://arxiv.org/abs/1104.4203).
- 13 G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. Shur, and T. Waleń. String periods in the order-preserving model. *Information and Computation*, 270:104463, 2020.
- 14 D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Computer and System Sciences*, 69(4):525–546, 2004.
- 15 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- 16 M. Huova, J. Karhumäki, and A. Saarela. Problems in between words and abelian words: k -abelian avoidability. *Theoretical Computer Science*, 454:172–177, 2012.
- 17 L. Ilie. A simple proof that a word of length n has at most $2n$ distinct squares. *Journal of Combinatorial Theory, Series A*, 112(1):163–164, 2005.
- 18 L. Ilie. A note on the number of squares in a word. *Theoretical Computer Science*, 380(3):373–376, 2007.
- 19 V. Keränen. Abelian squares are avoidable on 4 letters. *Automata, Languages and Programming*, pages 41–52, 1992.
- 20 T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Maximum number of distinct and nonequivalent nonstandard squares in a word. *Theoretical Computer Science*, 648(C):84–95, 2016.
- 21 T. Kociumaka, J. Radoszewski, and B. Wiśniewski. Subquadratic-time algorithms for abelian stringology problems. *Mathematical Aspects of Computer and Information Sciences*, pages 320–334, 2016.
- 22 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
- 23 N. H. Lam. On the number of squares in a string. *AdvOL-Report 2*, 2013.
- 24 F. Manea and S. Seki. Square-density increasing mappings. In *10th WORDS*, 9304:160–169, 2015.
- 25 Y. Matsuoka, T. Aoki, S. Inenaga, H. Bannai, and M. Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theoretical Computer Science*, 656:225–233, 2016.
- 26 P. A. B. Pleasants. Non-repetitive sequences. *Mathematical Proceedings of the Cambridge Philosophical Society*, 68(2):267–274, 1970.
- 27 A. Thierry. A proof that a word of length n has less than $1.5n$ distinct squares. *arXiv*, 2020. [arXiv:2001.02996](https://arxiv.org/abs/2001.02996).
- 28 A. Thue. Über unendliche Zeichenreihen. *Norske Vid Selsk. Skr. I Mat-Nat Kl.(Christiana)*, 7:1–22, 1906.

A Missing Proofs

► **Lemma 16.** *Given a string $w[1..n]$ over an alphabet of size σ , we can construct its order-preserving suffix tree in $\mathcal{O}(\sigma n)$ time and space.*

Proof. As explained in [3], the order-preserving suffix tree of $w[1..n]$ can be constructed using the general framework of Cole and Hariharan [2] for constructing a suffix tree for a quasi-suffix collection of strings w_1, w_2, \dots, w_n . The running time of their algorithm is $\mathcal{O}(n)$ with almost inverse exponential failure probability, assuming that one can access the j -th character of any w_i in constant time. The mechanism for accessing the j -th character of w_i is called the *character oracle*. In this particular application, the string $w_i = \phi(w[i..n])$. We will first describe how to implement a constant-time character oracle for such strings, and then explain why randomization is not needed in our setting.

13:18 Order-Preserving Squares in Strings

We need to implement a new *character oracle* that returns $\phi(w[i..n], j)$, for any i, j , in constant time after $\mathcal{O}(\sigma n)$ time and space preprocessing. This requires being able to calculate $\text{prev}_{<}(w[i..n], j)$ and $\text{prev}_{=}(w[i..n], j)$ in constant time. To this end, we define a two-dimensional array $\text{cnt}[i, x] = |\{k : k < i, w[k] < x\}|$, for $i = 0, 1, \dots, n$ and $x = 0, 1, \dots, \sigma$. All entries in this array can be computed in $\mathcal{O}(\sigma n)$ total time and space. Then, we can calculate any $\text{prev}_{<}(w[i..n], j)$ and $\text{prev}_{=}(w[i..n], j)$ as follows:

$$\begin{aligned} \text{prev}_{<}(w[i..n], j) &= \text{cnt}[i + j - 2, w[i + j - 1]] - \text{cnt}[i - 1, w[i + j - 1]] \\ \text{prev}_{=}(w[i..n], j) &= (\text{cnt}[i + j - 2, w[i + j - 1]] - \text{cnt}[i + j - 2, w[i + j - 1] - 1]) \\ &\quad - (\text{cnt}[i - 1, w[i + j - 1]] - \text{cnt}[i - 1, w[i + j - 1] - 1]). \end{aligned}$$

To remove randomization, we observe that its only source in the algorithm of Cole and Hariharan is the need to maintain, for each explicit node of the current tree, a dictionary indexed by the next character on an outgoing edge. If we could show that there are at most $\mathcal{O}(\sigma)$ such edges, then the dictionary could be implemented as a simple list, increasing the construction time to $\mathcal{O}(\sigma n)$, which is within our claimed bound.

Consider a non-leaf node v of the current tree. It corresponds to a proper prefix of some $\text{code}(w[i..n])\$, which by the definition of $\text{code}(\cdot)$ is equal to $\text{code}(w[i..j])$, for some j . Let $c_1 < c_2 < \dots < c_k$ be the distinct characters of $w[i..j]$, and denote by occ_x the number of occurrences of c_x in $w[i..j]$. Now consider an edge outgoing from v , and let $\text{code}(w[i'..j' + 1])$ correspond to the first node (implicit or explicit) after v there. We know that $\text{code}(w[i'..j']) = \text{code}(w[i..j])$, so the distinct characters of $w[i'..j']$ are $c'_1 < c'_2 < \dots < c'_k$ with occ_x being the number of occurrences of c'_x in $w[i'..j']$. Then, we analyze the possible values of $(\text{prev}_{<}(w[i'..j' + 1], j' - i' + 2), \text{prev}_{=}(w[i'..j' + 1], j' - i' + 2))$, that is, the first character on the considered edge. The first number is always equal to $\sum_{y=1}^{x-1} c_y$, for some $x \in [1, k + 1]$. Then, the second number is either 0 or occ_x . Thus, overall we have only $2k \leq 2\sigma$ possible first characters, which bounds the degree of any v by $\mathcal{O}(\sigma)$. ◀$

► **Lemma 17.** *Consider an op-square $s[1..2\ell]$. Then there exists $q < j$ such that the leftmost occurrences x_q and x_j satisfy $x_j - x_q = \ell$, $x_j \in [\ell + 1, 2\ell]$ and x_j is either the smallest or the largest element of its group.*

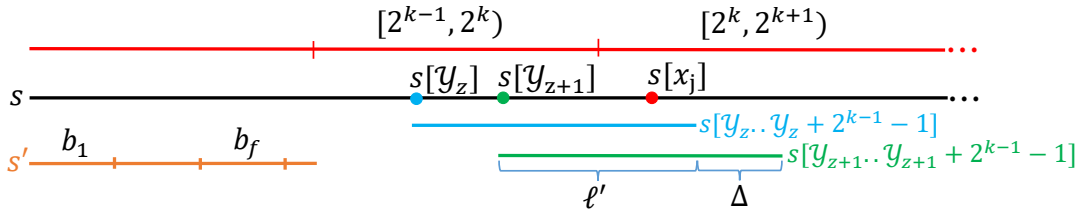
Proof. By Proposition 5, we know that there is a leftmost character in $s[\ell + 1..2\ell]$. Choose the largest k such that $2^k \leq \ell$ (so $2^{k+1} > \ell$). Consider two ranges $[2^k, 2^{k+1})$ and $[2^{k+1}, 2^{k+2})$ corresponding to groups L_k and L_{k+1} , respectively. Because $2^k \leq \ell$ and $2^{k+1} > \ell$, we have $2^k < \ell + 1$, $2^{k+1} \in [\ell + 1, 2\ell]$ and $2\ell < 2^{k+2}$. Consequently, the fragment $s[\ell + 1..2\ell]$ can be represented as the concatenation of a suffix of $s[2^k..2^{k+1})$ and a prefix of $s[2^{k+1}..2^{k+2})$. The leftmost occurrence that falls within $s[\ell + 1..2\ell]$ belongs to the suffix or the prefix. See Figure 10. If it falls within the suffix, the largest element of L_k belongs to $[\ell + 1, 2\ell]$. If it falls within the prefix, the smallest element of L_{k+1} belongs to $[\ell + 1, 2\ell]$. Let $x_j \in [\ell + 1, 2\ell]$ be the corresponding leftmost occurrence. To complete the proof we need to establish that there exists $q < j$ such that $x_j - x_q = \ell$. The character $s[x_j]$ is distinct from all $s[1], s[2], \dots, s[x_j - 1]$, and by $s[1..2\ell] \approx s[\ell + 1..2\ell]$ we obtain that $s[x_j - \ell]$ is distinct from all $s[1], s[2], \dots, s[x_j - \ell - 1]$. Thus, the position $x_j - \ell$ is a leftmost occurrence, hence $x_j - \ell = x_q$ for some $q < j$ as required. ◀

► **Lemma 18.** *The number of fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ is upper bounded by $\mathcal{O}(1 + |L_{k-2}|)$.*

Proof. Consider all such fragments $s[y_1..y_1 + 2^{k-1} - 1], s[y_2..y_2 + 2^{k-1} - 1], \dots, s[y_t..y_t + 2^{k-1} - 1]$. Because $x_j \in [y_z, y_z + 2^{k-1} - 1]$ for every $z = 1, 2, \dots, t$, either $t = 1$ or by the pigeonhole principle there exists z such that $y_{z+1} - y_z < 2^{k-1}/(t - 1)$. If $t = 1$ then we are done. Otherwise, let $\ell' = |s[y_{z+1}..y_z + 2^{k-1} - 1]|$. By assumption, $s' \approx s[y_z..y_z + 2^{k-1} - 1]$ and $s' \approx s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$, so by the transitivity of \approx also $s[y_z..y_z + 2^{k-1} - 1] \approx s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$. We conclude that $s'[1..\ell'] \approx s'[y_{z+1} - y_z + 1..2^{k-1}]$, or in other words ℓ' is an op-border of s' . Let $b_1, b_2, \dots, b_f, b_{f+1}$ be the blocks defined by ℓ' in $s' = s[1..|s'|] = w[i..i + |s'| - 1]$, where each block is of length $\Delta = 2^{k-1} - \ell'$. See Figure 12. Recall that x_j is a leftmost occurrence in $s = w[i..n]$, and by the definition of y_z and y_{z+1} we have $x_j \in [y_{z+1}, y_z + 2^{k-1} - 1]$. Then, by $s'[1..\ell'] \approx s'[y_{z+1} - y_z + 1..2^{k-1}]$ we obtain that $x_j - y_z + 1 \in [y_{z+1} - y_z + 1, 2^{k-1}]$ is also a leftmost occurrence in $s = w[i..n]$. Hence, we have a leftmost occurrence in block b_j , for some $j \geq 2$. This allows us to apply Lemma 10 and then Lemma 11 to conclude that there is a leftmost occurrence in every block b_1, b_2, \dots, b_f . We calculate a lower bound on how many of these leftmost occurrences fall within the range $[2^{k-2}, 2^{k-1}]$:

$$\begin{aligned} \left\lfloor \frac{2^{k-2}}{\Delta} \right\rfloor - 1 &> \frac{2^{k-2}}{2^{k-1} - \ell'} - 2 \\ &= \frac{2^{k-2}}{2^{k-1} - (y_z + 2^{k-1} - y_{z+1})} - 2 = \frac{2^{k-2}}{y_{z+1} - y_z} - 2 \\ &> \frac{2^{k-2}}{2^{k-1}/(t-1)} - 2 = (t-5)/2. \end{aligned}$$

For $t < 6$, we are done as the number of fragments is $\mathcal{O}(1)$. Otherwise, we obtain that $|L_{k-2}| \geq (t-5)/2 \geq t/12$, thus $t = \mathcal{O}(1 + |L_{k-2}|)$ always holds as claimed. ◀



■ **Figure 12** $s, s', s[y_z..y_z + 2^{k-1} - 1]$, and $s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$ are colored in black, orange, blue, and green, respectively. The red line illustrates the ranges.

► **Lemma 20.** *A compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time, so that for any leaf u and integer k we can query in constant time for a $\mathcal{O}(\log n)$ -bit fingerprint of the ancestor of u at string depth 2^k .*

Proof. This follows by applying the method used to solve the substring fingerprint problem mentioned in [11, Lemma 14]. Following the description in the full version [12, Lemma 12], a compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time so that we can locate the (implicit or explicit) node corresponding to the ancestor at string depth 2^k of a given leaf in constant time. If the sought node is implicit (and does not explicitly exist in the compacted trie) we retrieve the edge that contains it. Next, if the node is explicit then we return its identifier. If the node is implicit then we return the identifier of the edge that contains it. Thus, the required range of identifiers is $[2n]$. ◀

MUL-Tree Pruning for Consistency and Compatibility

Christopher Hampson ✉ 

Department of Informatics, King's College London, UK

Daniel J. Harvey ✉

Graduate School of Informatics, Kyoto University, Japan

Costas S. Iliopoulos ✉ 


Department of Informatics, King's College London, UK

Jesper Jansson ✉ 

Graduate School of Informatics, Kyoto University, Japan

Zara Lim ✉ 

Department of Informatics, King's College London, UK

Wing-Kin Sung ✉ 

Department of Chemical Pathology, The Chinese University of Hong Kong, China

Hong Kong Genome Institute, Hong Kong Science Park, Shatin, China

Laboratory of Computational Genomics, Li Ka Shing Institute of Health Sciences,

The Chinese University of Hong Kong, China

Abstract

A multi-labelled tree (or MUL-tree) is a rooted tree leaf-labelled by a set of labels, where each label may appear more than once in the tree. We consider the MUL-tree Set Pruning for Consistency problem (MULSETPC), which takes as input a set of MUL-trees and asks whether there exists a perfect pruning of each MUL-tree that results in a consistent set of single-labelled trees. MULSETPC was proven to be NP-complete by Gascon et al. when the MUL-trees are binary, each leaf label is used at most three times, and the number of MUL-trees is unbounded. To determine the computational complexity of the problem when the number of MUL-trees is constant was left as an open problem.

Here, we resolve this question by proving a much stronger result, namely that MULSETPC is NP-complete even when there are only two MUL-trees, every leaf label is used at most twice, and every MUL-tree is either binary or has constant height. Furthermore, we introduce an extension of MULSETPC that we call MULSETPComp, which replaces the notion of consistency with compatibility, and prove that MULSETPComp is NP-complete even when there are only two MUL-trees, every leaf label is used at most thrice, and every MUL-tree has constant height. Finally, we present a polynomial-time algorithm for instances of MULSETPC with a constant number of binary MUL-trees, in the special case where every leaf label occurs exactly once in at least one MUL-tree.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases multi-labelled tree, phylogenetic tree, consistent, compatible, pruning, algorithm, NP-complete

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.14

Funding *Daniel J. Harvey*: Partially funded by JSPS KAKENHI grant 22H03550.

Jesper Jansson: Partially funded by JSPS KAKENHI grant 22H03550.

1 Introduction

In evolutionary biology, leaf-labelled (phylogenetic) trees are commonly employed to describe the evolution of species using leaf labels to represent different species [11]. Comparisons of these structures are used particularly in phylogenetic inferences – similarities may indicate



© Christopher Hampson, Daniel J. Harvey, Costas S. Iliopoulos, Jesper Jansson, Zara Lim, and Wing-Kin Sung;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 14; pp. 14:1–14:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

evolutionary patterns, whereas differences may highlight genetic mutations. The measure of similarity between phylogenetic trees has been defined by multiple alternate metrics, such as the Robinson-Foulds distance [29], subtree pruning and regraft (SPR) distances [5, 34], and maximum agreement subtrees [2, 7, 12]. Other problems related to phylogenetic trees include constructing supertrees [1, 3, 4, 33] or consensus trees [6, 11, 21] which can determine relations or interactions between smaller phylogenetic trees.

Phylogenetic trees are classically described as single-labelled trees, where no label appears on the leaves of the tree more than once. Typically, construction or comparison algorithms of such phylogenetic trees make use of this property to reduce computational costs. Multi-labelled trees (or MUL-trees) are a generalisation of single-labelled trees in which multiple leaves may be labelled by the same label. MUL-trees can be useful to depict genome duplication, lineage sorting, or lateral gene transfer [23]. Other applications include the construction of phylogenetic networks by folding operations [17, 18, 19], biogeography [13, 24, 25], the study of host-parasite cospeciation [26], and gene evolution studies [23, 27, 30].

MUL-trees have been far less investigated than their single-labelled counterparts and many computational problems become NP-hard when extended to MUL-trees. For example, the majority rule consensus tree for a set of k single-labelled trees with n leaf labels each can be computed in $O(nk)$ time [21], but is NP-hard to compute for MUL-trees [8]. Other approaches convert MUL-trees into single-labelled trees which can be input to existing algorithms [20, 30]. A few polynomial-time algorithms do exist for MUL-trees – Cui et al.[8] presented a $O(n^2k + nk^2)$ -time algorithm for building a majority rule consensus MUL-tree in which each leaf label occurs at most twice, based on a reduction to the Perfect Phylogeny Haplotyping problem [10]. Furthermore, the maximum agreement subtree (MAST) distance between two MUL-trees can be computed in quadratic time, though it also becomes NP-complete when generalised to more than two MUL-trees [13, 22].

This paper investigates MUL-trees by considering the *MUL-tree Set Pruning for Consistency* problem (MULSETPC), which takes as input a set of MUL-trees, and outputs whether or not there exists a pruning of the MUL-trees which gives a consistent set of single-labelled trees (see Section 2 for formal definitions). Gascon et al. showed that, in general, MULSETPC is NP-complete via a polynomial reduction from 3-SAT [15, 16]. However, their reduction from an instance of 3-SAT with m variables and z clauses gives an instance of MULSETPC containing $m + z + 1$ MUL-trees; moreover these MUL-trees may have labels occurring three times. Here we prove that MULSETPC is still NP-complete, even when restricted to instances involving only two MUL-trees, or in which every leaf label appears at most twice within a MUL-tree. This totally resolves the open question of Gascon et al. [15, 16] regarding the parameterised complexity of MULSETPC when the parameter is the number of input trees. Also, we identify tractable fragments of MULSETPC which can be solved in polynomial time, in short, instances in which each label appears exactly once in at least one MUL-tree.

We also present a generalisation of MULSETPC called MULSETPComp, which asks for a *compatible* set of trees instead of a *consistent* set of trees. Tree compatibility is a generalisation of tree consistency where we allow the supertree displaying the set to instead display a refinement of each tree rather than the tree itself—again, a more rigorous definition is given later. Tree compatibility is relevant when determining the existence of a supertree for a given set of phylogenetic trees [1, 31]. This is because experimental data often contains uncertainty, which can be expressed by non-binary nodes in the tree; this necessitates the use of compatibility. Compatibility is also relevant to other questions such as the incomplete directed perfect phylogeny problem [28]. Our interest in tree compatibility was partially motivated by recent improvements in compatibility testing [9]. Here, we prove that MULSETPComp is NP-complete, even when restricted to instances involving only two MUL-trees, or in which every leaf label appears at most thrice within a MUL-tree.

The rest of the paper is organized as follows. In Section 2, we introduce the preliminary notation and definitions. In Section 3 we present the improved NP-completeness proof for MULSETPC by reduction from the Boolean 3-SAT problem. In Section 4 we give a NP-completeness proof for MULSETPComp by reduction from the Exact 3-cover with Multiplicity 3 problem. Section 5 contains our polynomial time results for tractable instances of MULSETPC. Finally, in Section 6 we present our conclusions and a few open problems.

2 Preliminaries

We shall use the following standard definitions on trees.

► **Definition 1** (Basic tree definitions). *All trees we consider are rooted and unordered. If x, y are nodes in a tree T , then y is an ancestor of x (and x a descendant of y) if y lies on the unique path from x to the root of T . We denote this by $x \leq y$. Additionally, if $y \neq x$ then y is a proper ancestor of x , which we denote by $x < y$. If $x < y$ and y is adjacent to x then y is the parent of x and x a child of y . If x and x' are both children of y then x and x' are siblings. The lowest common ancestor of nodes x and y , denoted $\text{lca}_T(x, y)$, is the node z such that $x \leq z$, $y \leq z$, and no proper descendant of z also satisfies these properties. The empty tree, denoted by T_\emptyset , is the unique tree which contains no nodes.*

We use the following definition of leaf-labelled trees, which takes the definitions of Gascon et. al. [15] and generalises them to the case where the tree may not be binary.

► **Definition 2** (Leaf-labelled trees). *A leaf-labelled tree (T, \mathcal{X}) is a (rooted, unordered) tree T where no node has exactly one child and where each leaf has been assigned a label from a set of labels \mathcal{X} . (We will sometimes refer to T as a leaf-labelled tree on \mathcal{X} , and omit \mathcal{X} if it is clear from context.) A leaf-labelled tree is a single-labelled tree if every label in \mathcal{X} is used at most once. Alternatively, a multi-labelled tree or MUL-tree is a leaf-labelled tree where we allow each label in \mathcal{X} to label multiple leaves. We say a MUL-tree has multiplicity k if each leaf label appears at most k times. Let $L(T) \subseteq \mathcal{X}$ denote the set of leaf labels appearing in T and let $D(T) \subseteq L(T)$ denote the set of leaf labels appearing only once in T .*

Note that in a single-labelled tree we sometimes abuse notation and identify the leaf and the leaf label. We do the same in a MUL-tree only if the context is clear and there is no possibility of confusion.

If u is a node of T then T^u denotes the subtree rooted at u containing u and all its descendants, maintaining the same leaf-labelling as T on the remaining leaves. Let $D^u := D(T^u)$.

► **Definition 3** (Pruning and Perfect Pruning). *Given a leaf-labelled tree T , let y denote a leaf node of T and x the parent of y . We prune the leaf y in the following manner:*

- Delete the leaf y .
- If x still has at least two children, do nothing else.
- Alternatively, if x now has only one child and is not the root, suppress the vertex x .
- Finally, if x has only one child z and is the root, delete x and make z the new root.

A perfect pruning of T is a single-labelled tree T' such that $L(T') = L(T)$, created by (possibly repeated) prunings of T . That is, for every label that appears more than once in T , we prune away all but exactly one copy of the label to obtain a single-labelled tree. If T is a single-labelled tree, then its only perfect pruning is itself.

14:4 MUL-Tree Pruning for Consistency and Compatibility

Given a leaf-labelled tree T and a set of leaf labels $L' \subseteq L(T)$, let $T \upharpoonright_{L'}$ denote the leaf-labelled tree constructed by pruning from T every leaf labelled by a label from $L(T) - L'$. (That is, only leaves labelled by L' remain.) Two leaf-labelled trees T_1 and T_2 are *leaf-label isomorphic* if there is an isomorphism between T_1 and T_2 which preserves the labelling of the leaves. We say that a leaf-labelled tree T on L *displays* a single-labelled tree T' on $L' \subseteq L$ if there exists a perfect pruning T^* of T such that $T^* \upharpoonright_{L'}$ is leaf-label isomorphic to T' .

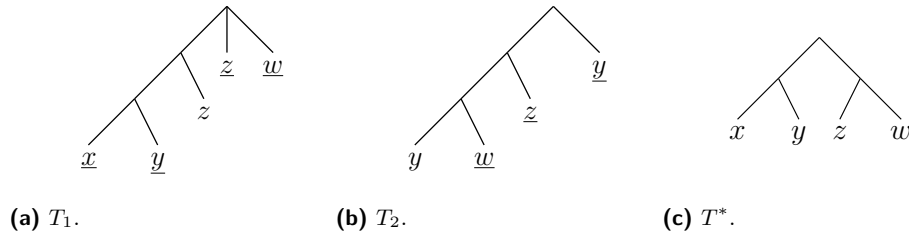
► **Definition 4 (Refinement).** *Given single-labelled trees T and T^* , we say T^* is a refinement of T if T can be obtained from T^* by (possibly repeated) contractions of non-leaf edges, where we treat a contraction as merging the child node into the parent node. We write $T \leq T^*$.*

What follows is the definition of a *consistent set*, and the very similar definition of a *compatible set*.

► **Definition 5 (Consistent Set).** *Consider a set of single-labelled trees T_1, \dots, T_k with corresponding label sets L_1, \dots, L_k . We say this set is consistent if there exists a single-labelled tree T on label set $L = \bigcup_{i=1}^k L_i$ such that for every $i = 1, \dots, k$, T displays T_i .*

Note in the above definition that if $L_1 = \dots = L_k$ then $L = L_1$ and so the set T_1, \dots, T_k is consistent if and only if the trees are pairwise leaf-label isomorphic.

► **Definition 6 (Compatible Set).** *Consider a set of single-labelled trees T_1, \dots, T_k with corresponding label sets L_1, \dots, L_k . We say this set is compatible if there exists a single-labelled tree T on label set $L = \bigcup_{i=1}^k L_i$ such that for every $i = 1, \dots, k$, $T \upharpoonright_{L_i}$ is a refinement of T_i .*



■ **Figure 1** Let T_1 , T_2 , and T^* be the three trees with $\mathcal{X} = \{x, y, z, w\}$ shown above. If we prune away the non-underlined labels in T_1 and T_2 then $T^* \upharpoonright_{L(T_i)}$ is a refinement of the pruned T_i for $i \in \{1, 2\}$, which shows that there exists a perfect pruning of $\{T_1, T_2\}$ giving a compatible set of trees. In contrast, there is no perfect pruning of $\{T_1, T_2\}$ giving a consistent set of trees because neither of the two single-labelled subtrees with leaf labels y, z , and w displayed by T_1 is also displayed by T_2 . Note, however, that if the label w in T_1 is changed to x then $\{T_1, T_2\}$ becomes consistent since this label can be pruned along with one leaf labelled by z to obtain a perfect pruning of T_1 which is displayed by T^* .

Note that every consistent set of trees is also a compatible set, but the converse does not hold in general.

The MUL-set pruning for consistency problem can then be defined as follows:

MUL-tree Set Pruning for Consistency (MULSETPC) Problem:

Input: $(\mathcal{M}, \mathcal{X})$ where \mathcal{M} is a set of MUL-trees on \mathcal{X} .

Output: $\exists?$ a perfect pruning of each tree of \mathcal{M} resulting in a consistent set of trees.

We introduce the following problem, which substitutes *compatible* for *consistent* sets:

MUL-tree Set Pruning for Compatibility (MULSETPComp) Problem:

Input: $(\mathcal{M}, \mathcal{X})$ where \mathcal{M} is a set of MUL-trees on \mathcal{X} .

Output: $\exists?$ a perfect pruning of each tree of \mathcal{M} resulting in a compatible set of trees.

See Figure 1 for an example that illustrates the difference between MULSETPC and MULSETPComp.

3 NP-completeness for MULSETPC instances with two MUL-trees and multiplicity 2

In this section we consider the MULSETPC problem, and show that it is NP-complete even when considering a heavily restricted set of instances. Specifically, we consider instances with at most two MUL-trees, where the multiplicity is 2, and where the MUL-trees are binary. The core of our proof will be a reduction from 3-SAT [14].

3-satisfiability (3-SAT) Problem:

Input: A Boolean set of clauses $C = (C_1 \wedge C_2 \wedge \dots \wedge C_z)$ on a finite set of literals $\{l_1, l_2, \dots, l_m\}$ where each clause is in conjunctive normal and contains 3 literals.

Output: $\exists?$ a satisfying valuation \mathcal{V} of C .

Our first goal is to construct, given an instance of 3-SAT, two MUL-trees T_1 and T_2 which we will use in our corresponding instance of MULSETPC. Our set of leaf labels \mathcal{X} consists of the following:

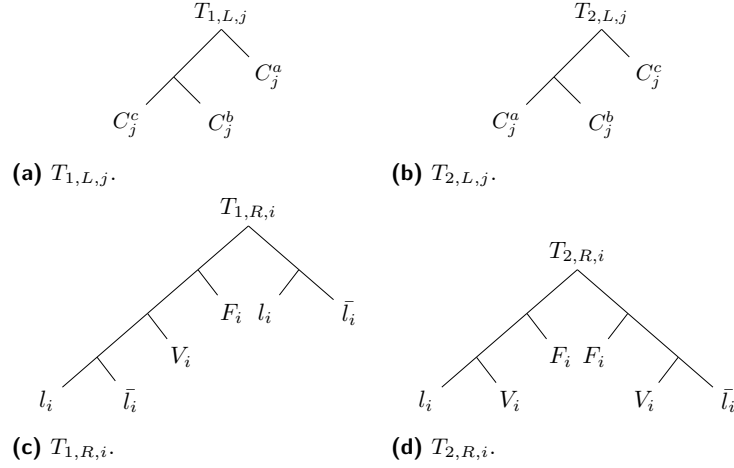
- $\{l_i, \bar{l}_i \mid i = 1, \dots, m\}$, the set of literals,
- $\{F_i, V_i \mid i = 1, \dots, m\}$, a pair of “dummy” labels for each variable and
- $\mathcal{P} := \{C_j^1, C_j^2, C_j^3 \mid j = 1, \dots, z\}$, a triple of “position” labels representing the three places of each clause of C .

Define $h := \mathcal{P} \rightarrow \{l_i, \bar{l}_i \mid i = 1, \dots, m\}$ as the function which maps a position label C_j^x to the literal found in that position. For example, if $C_1 = (l_1 \vee l_4 \vee \bar{l}_6)$ then $h(C_1^1) = l_1$, $h(C_1^2) = l_4$ and $h(C_1^3) = \bar{l}_6$. We treat \mathcal{P} as being ordered first by the index of the clause and then by the position. Let $H(l_i) := \{C_j^x \mid h(C_j^x) = l_i\}$, and define $H(\bar{l}_i)$ similarly.

Our trees T_1, T_2 will be constructed from four types of subtrees $T_{1,L,j}, T_{2,L,j}, T_{1,R,i}^*$ and $T_{2,R,i}^*$, where $i = 1, \dots, m$ and $j = 1, \dots, z$. See Figure 2 for the subtrees $T_{1,L,j}, T_{2,L,j}, T_{1,R,i}$ and $T_{2,R,i}$; we now explain how to construct $T_{1,R,i}^*$ and $T_{2,R,i}^*$ from $T_{1,R,i}$ and $T_{2,R,i}$.

Let v_i denote the leaf labelled V_i in $T_{1,R,i}$ and let u_i denote the parent of v_i . Let u_i^+ and u_i^- denote the parents of leaves labelled l_i and \bar{l}_i respectively in $T_{2,R,i}$. Let v_i^+ denote the child of u_i^+ labelled by V_i , and v_i^- denote the child of u_i^- labelled by V_i .

14:6 MUL-Tree Pruning for Consistency and Compatibility



■ **Figure 2** Subtrees $T_{1,L,j}, T_{2,L,j}, T_{1,R,i}, T_{2,R,i}$ for MULSETPC.

Initialise $T_{1,R,i}^*$ as a copy of $T_{1,R,i}$, then do the following:

- If $H(l_i) \cup H(\bar{l}_i) = \emptyset$, make no further changes.
- Otherwise, subdivide the edge $u_i v_i$ $|H(l_i) \cup H(\bar{l}_i)|$ times, and add to each new node an adjacent leaf. Label these leaves with $H(l_i) \cup H(\bar{l}_i)$, respecting the ordering such that the first label is closest to u_i .

Initialise $T_{2,R,i}^*$ as a copy of $T_{2,R,i}$, and then:

- If $H(l_i) \neq \emptyset$, then subdivide $u_i^+ v_i^+$ $|H(l_i)|$ times and add an leaf adjacent to each new node. Label these leaves with $H(l_i)$, respecting the ordering so that the first label is closest to u_i^+ .
- If $H(\bar{l}_i) \neq \emptyset$, repeat the previous step, substituting $H(\bar{l}_i)$ for $H(l_i)$ and $u_i^- v_i^-$ for $u_i^+ v_i^+$.

Construct $T_{1,L}$ by taking a complete binary tree on z leaves (recall z is the number of clauses), suppressing any nodes with exactly one child, and then identifying the root of each $T_{1,L,j}$ (ordered by j) with exactly one of the leaves (ordered left-to-right). Construct $T_{2,L}$ in the same fashion, substituting $T_{2,L,j}$ for $T_{1,L,j}$. Construct $T_{1,R}$ by taking a complete binary tree on m leaves, suppressing any nodes with exactly one child, and then identifying the root of each $T_{1,R,i}^*$ (ordered by i) with exactly one of the leaves (ordered left-to-right). Again, construct $T_{2,R}$ in the same fashion, substituting $T_{2,R,i}^*$ for $T_{1,R,i}^*$. Finally, construct T_1 by taking a root node $r(T_1)$ and adding an edge to the roots of $T_{1,L}$ and $T_{1,R}$; construct T_2 in the obvious equivalent fashion.

Given an instance C of 3-SAT, we create our instance of MULSETPC, $(\{T_1, T_2\}, \mathcal{X})$. Note that T_1 and T_2 have multiplicity 2; each leaf label $C_j^x \in \mathcal{P}$ appears once in $T_{1,L,j}$ and $T_{2,L,j}$ and once in $T_{1,R,i}^*$ and $T_{2,R,i}^*$ for the single value of i such that $C_j^x \in H(l_i) \cup H(\bar{l}_i)$. By inspection, the labels of $\mathcal{X} - \mathcal{P}$ also appear at most twice. Hence the instance $(\{T_1, T_2\}, \mathcal{X})$ contains two binary MUL-trees with multiplicity 2. It suffices to now show the reduction, in two parts.

► **Lemma 7.** *If C is a satisfied instance of 3-SAT then the corresponding instance $(\{T_1, T_2\}, \mathcal{X})$ of MULSETPC admits a perfect pruning giving a consistent set of trees.*

Proof. Suppose that C is satisfiable. Then there exists a valuation of every variable which satisfies every clause of C ; fix one such valuation and label it \mathcal{V} . For each clause C_j , mark one of the position labels C_j^x for $x \in \{1, 2, 3\}$ such that $h(C_j^x)$ is valued true by \mathcal{V} . Since \mathcal{V} satisfies every clause, we will always be able to choose a label to mark; if there are multiple legitimate choices choose arbitrarily. Refer to any label C_j^x we have not marked as *unmarked*.

By our construction of T_1 and T_2 , if we show that after pruning each $T_{1,L,j}$ and $T_{1,R,i}^*$ is leaf-label isomorphic to $T_{2,L,j}$ and $T_{2,R,i}^*$ respectively, then T_1 is leaf-label isomorphic to T_2 .

Consider first the labels of \mathcal{P} . Prune from each $T_{1,L,j}$ and $T_{2,L,j}$ the one marked label C_j^x , and leave the two unmarked labels C_j^x . We must keep the other copy of the marked C_j^x and prune away the other copies of the unmarked C_j^x in whichever $T_{1,R,i}^*$ and $T_{2,R,i}^*$ they appear. Consider the sets $H(l_i)$ and $H(\bar{l}_i)$. If C_j^x is marked, then $h(C_j^x)$ is true, and so at most one of $H(l_i), H(\bar{l}_i)$ contains a marked label. Keeping this information in mind, we can now simply look at the subtrees themselves.

- After this pruning each $T_{1,L,j}$ will be leaf-label isomorphic to the corresponding $T_{2,L,j}$, by inspection.
- Consider $T_{1,R,i}^*$ and $T_{2,R,i}^*$. We have already pruned away unmarked labels of \mathcal{P} . If l_i is true, prune the copy of l_i in $T_{1,R,i}^*$ closest to the root of $T_{1,R,i}^*$ and the copy of \bar{l}_i furthest from the root; if \bar{l}_i is true do the opposite. In $T_{2,R,i}^*$ prune the copies of F_i, V_i closer to the literal l_i or \bar{l}_i which evaluates as false. Hence we have pruned away the extra copies of each leaf label. It is clear, mostly by inspection, that $T_{1,R,i}^*$ is leaf-label isomorphic to $T_{2,R,i}^*$; the most important point is that since at most one of $H(l_i)$ and $H(\bar{l}_i)$ contains a marked label, at least one of these sets has been pruned away entirely in $T_{2,R,i}^*$ (specifically the set closer to the false literal).

Hence, after pruning, T_1 and T_2 are leaf-label isomorphic, and thus $\{T_1, T_2\}$ is consistent. ◀

See Figure 3 for an illustration of the reduction in Lemma 7.

► **Lemma 8.** *If C is not a satisfied instance of 3-SAT then the corresponding instance $(\{T_1, T_2\}, \mathcal{X})$ of MULSETPC do not admit perfect prunings giving a consistent set of trees.*

We omit the proof of Lemma 8 for reasons of space. We now prove our main result.

► **Theorem 9.** *The MULSETPC problem is NP-complete, even restricted to instances containing at most two binary MUL-trees with multiplicity 2.*

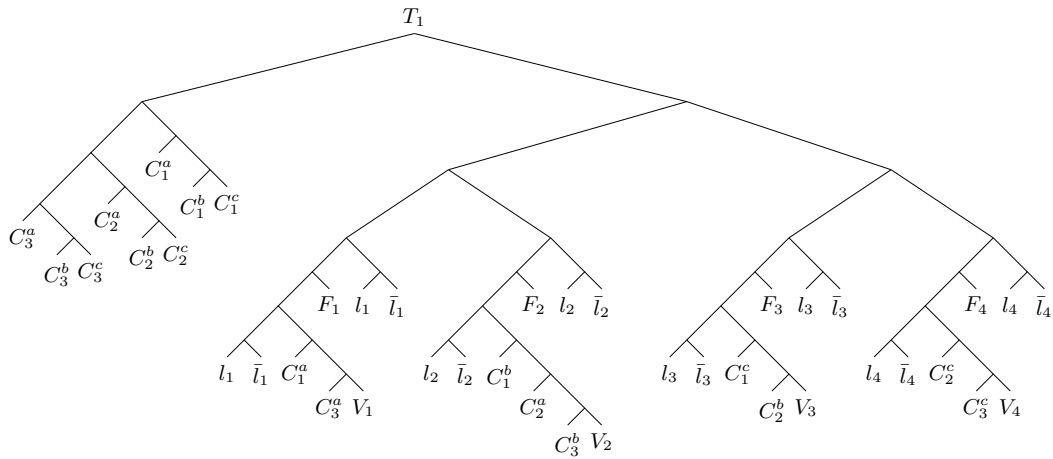
Proof. Note first that MULSETPC is in NP, since given a set of pruned leaves, a perfect pruning can be constructed in polynomial time, and the consistency of the set of trees determined in polynomial time using the BUILD algorithm [1, 15].

The result then follows directly from Lemma 7 and Lemma 8. ◀

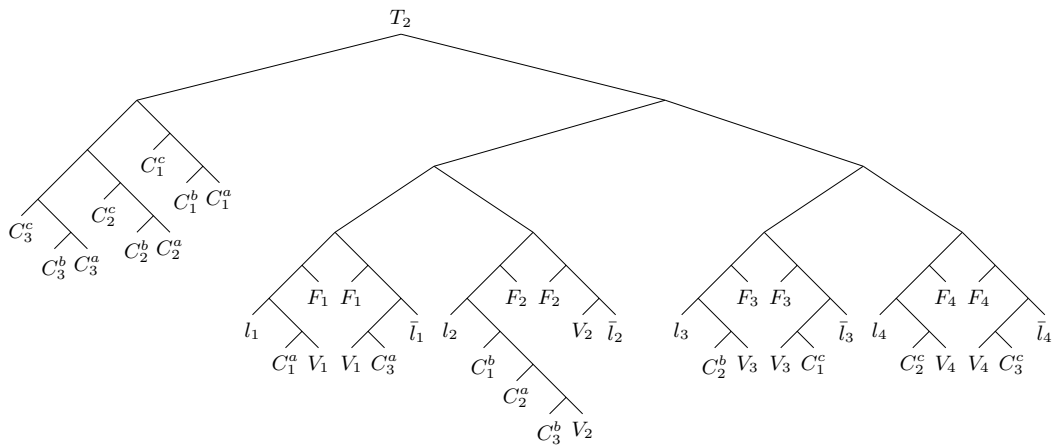
It is also possible to swap our requirement that the MUL-trees are binary for an alternative requirement that the MUL-trees have height at most 5. Proving this result is very similar to the binary case, but we omit it here on grounds of space; the proof will appear in the journal version of this article. Thus we get the following result.

► **Theorem 10.** *The MULSETPC problem is NP-complete, even restricted to instances containing at most two MUL-trees with multiplicity 2 and height at most 5.*

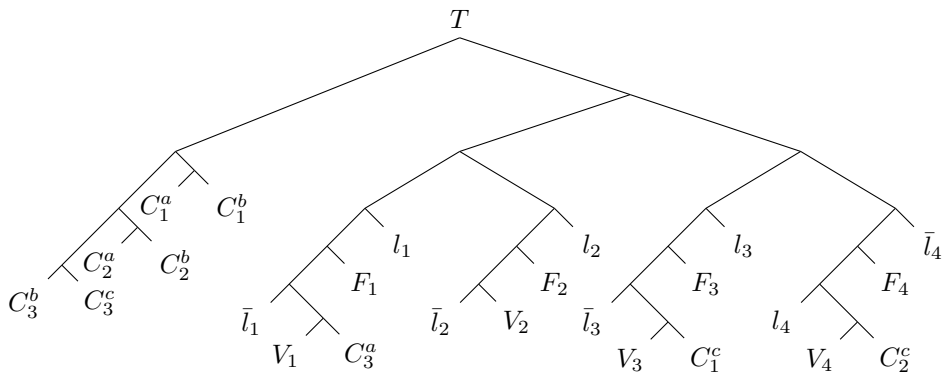
14:8 MUL-Tree Pruning for Consistency and Compatibility



(a) The MUL-tree T_1 .



(b) The MUL-tree T_2 .



(c) Tree T displays the two MUL-trees T_1 and T_2 after they have been perfectly pruned.

■ **Figure 3** Illustrating the reduction from 3-SAT in Lemma 7. The two MUL-trees T_1 and T_2 are constructed from $C = (C_1 \wedge C_2 \wedge C_3)$ where $C_1 = (l_1 \vee l_2 \vee \bar{l}_3)$, $C_2 = (l_2 \vee l_3 \vee l_4)$, and $C_3 = (\bar{l}_1 \vee l_2 \vee \bar{l}_4)$. Figure 3c shows the corresponding tree T which displays the pruned T_1 and T_2 corresponding to a satisfiable assignment $\bar{l}_1 = \bar{l}_2 = \bar{l}_3 = l_4 = true$ with marked labels C_1^c , C_2^c and C_3^a .

4 NP-completeness for MULSETPComp

In this section, we extend our previous results regarding the problem MULSETPC to the similar problem MULSETPComp. We present the following two theorems.

► **Theorem 11.** *MULSETPComp is NP-complete, even when restricted to instances containing at most two MUL-trees with multiplicity at most 3 and where the MUL-trees have height at most 4.*

► **Theorem 12.** *MULSETPComp is NP-complete, even when restricted to instances containing at most two MUL-trees with height at most 3 and where one MUL-tree is a single-labelled tree containing all leaf labels.*

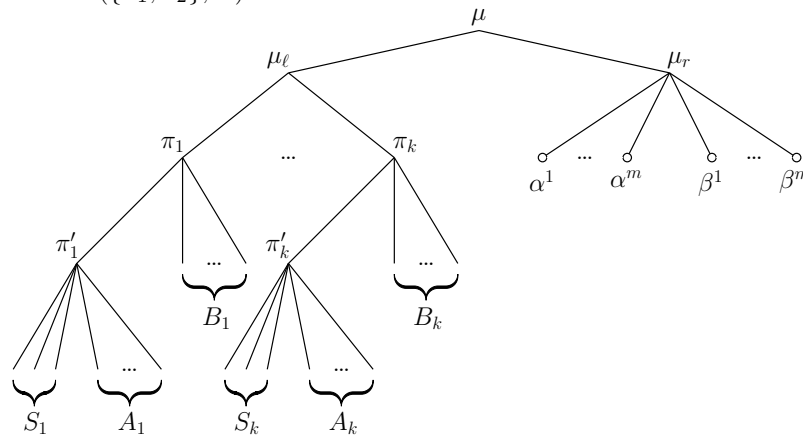
As was the case in Section 3, these two results have very similar proofs. We shall prove Theorem 11, but omit the proof of Theorem 12 for space reasons. Here, we reduce from X3C3, also known to be NP-complete [14].

Exact 3-cover with multiplicity 3 (X3C3) Problem:
Input: A set $X = \{x_1, \dots, x_{3q}\}$ and a collection $C = \{S_1, \dots, S_k\}$ of 3-element subsets of X , such that any element of X appears in at most three sets in C .
Output: $\exists?$ an exact cover for X .

As before, our first goal is to construct, given an instance (X, C) of X3C3, a set of two MUL-trees we shall use to construct our corresponding instance of MULSETPComp.

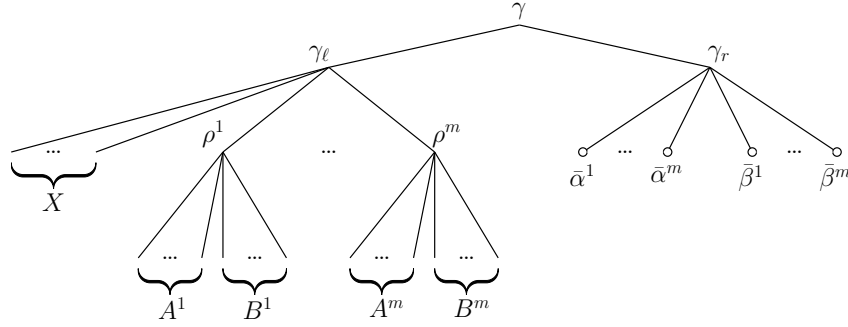
Recall that $|X| = 3q$ and that $k := |C|$. Let $m := k - q$, the number of sets of C not chosen to be part of our exact 3 cover. Define $A := \{a_j^i | i = 1, \dots, m, j = 1, \dots, k\}$ and $B = \{b_j^i | i = 1, \dots, m, j = 1, \dots, k\}$. Let $A^i = \{a_j^i | j = 1 \dots, k\}$ and $A_j = \{a_j^i | i = 1, \dots, m\}$, and define B^i, B_j similarly. The labels in $A \cup B$ are another set of “dummy” labels we use for technical reasons. Let $Y = X \cup A \cup B$. We may assume that $q \geq 3$. We may also assume that k is even; if not, add to X three additional elements $\{x_{3q+1}, x_{3q+2}, x_{3q+3}\}$ (which increases q by 1) and add to C a additional set $S' = \{x_{3q+1}, x_{3q+2}, x_{3q+3}\}$ (which increases $k = |C|$ by 1). It is clear this modified instance contains an exact 3-cover if and only if the original instance did.

We denote our two trees T_1 and T_2 . Our corresponding instance of MULSETPComp for Theorem 11 will be $(\{T_1, T_2\}, Y)$.

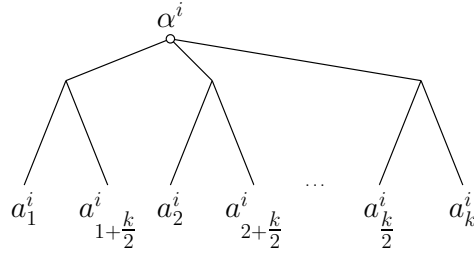


■ **Figure 4** The tree T_1 for MULSETPComp.

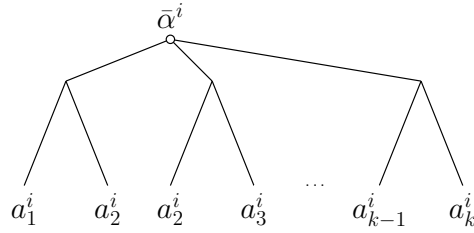
14:10 MUL-Tree Pruning for Consistency and Compatibility



■ **Figure 5** The tree T_2 for MULSETPComp.



■ **Figure 6** The subtree of T_1 rooted by α^i .



■ **Figure 7** The subtree of T_2 rooted by $\bar{\alpha}^i$.

See Figures 4 and 6 for the construction of T_1 , and Figures 5 and 7 for the construction of T_2 . Note that a node labelled α^i in Figure 4 is the root of the appropriate subtree from Figure 6, not a leaf labelled by α^i . An equivalent statement holds for $\bar{\alpha}^i, \beta^i$ and $\bar{\beta}^i$, where the subtree rooted at β^i is found by taking the subtree of Figure 6 and replacing each a_j^i with b_j^i . (An equivalent statement holds for $\bar{\beta}^i$ and Figure 7). Note the following other useful facts about our construction:

- Trees T_1 and T_2 have the same set of leaf labels. Hence a perfect pruning of T_1 and T_2 is a compatible set of trees if and only if there exists a tree T^* on the same leaf label set which is a refinement of both perfect prunings.
- In T_1 any label of X may appear at most three times, since any x_i may appear in at most three sets of C . Every label of A and B appears once in $T_1^{\mu_\ell}$ and once in $T_1^{\mu_r}$. Hence T_1 has multiplicity 3. In T_2 any label of X appears only once, and the labels of A and B appear at most thrice, once in $T_2^{\gamma_\ell}$ and once or twice in $T_2^{\gamma_r}$. Hence T_2 has multiplicity 3.
- The heights of both MUL-trees can be determined by inspection.

We will need the following two technical lemmas – as the proofs are straightforward we omit them.

► **Lemma 13.** Consider a set of elements $X = \{x_1, \dots, x_k\}$ together with a subset $X' \subset X$ such that $|X'| = k - 1$ and a collection of sets $\mathcal{X} = \{x_i, x_{i+i}\}_{i \in [k-1]}$. Then it is possible to construct a set equal to X' by choosing one element from each set in \mathcal{X} .

► **Lemma 14.** *Let T, T^* be single-labelled trees such that $T \leq T^*$, and let u, x, y be leaf nodes in T (and hence also in T^*). If $\text{lca}_T(u, x) < \text{lca}_T(u, y)$ then $\text{lca}_{T^*}(u, x) < \text{lca}_{T^*}(u, y)$.*

The following two lemmas form the core of our main result.

► **Lemma 15.** *If (X, C) is an instance of X3C3 that allows an exact 3-cover C' then the corresponding instance $(\{T_1, T_2\}, Y)$ of MULSETPComp admits a perfect pruning giving a compatible set of trees.*

Proof. Let $\mathcal{I} \subset [k]$ denote the set of m indices i of those S_i we did not choose as part of our exact 3-cover, that is the sets $S_i \in C - C'$. Let $\psi : \mathcal{I} \rightarrow [m]$ be an arbitrary isomorphism. Prune the tree T_1 as follows:

- For each S_i (or equivalently, each subtree rooted at π_i):
 - If $S_i \in C'$ then keep the labels of S_i as the children of π'_i but prune away all other leaf labels in the subtree $T_1^{\pi_i}$. After pruning there are three leaves (with labels corresponding to the elements of S_i) as children of π'_i , which is itself a child of μ_ℓ .
 - If $S_i \notin C'$ then prune away all leaf labels in $T_1^{\pi_i}$ except $a_i^{\psi(i)}$ and $b_i^{\psi(i)}$. After pruning, the remaining leaves will be children of π_i .
- In each $T_1^{\alpha^j}$, prune away a_i^j if the leaf label appears in $T_1^{\mu_\ell}$. Since m sets of C are not in C' , there are m leaf labels of the form a_i^j appearing in $T_1^{\mu_\ell}$, specifically $a_i^{\psi(i)}$ for the m values of i for which $\psi(i)$ is defined, or equivalently for all $\psi(i) = 1, \dots, m$. Hence we must prune away one leaf label in each $T_1^{\alpha^j}$.
- Repeat the previous step for each $T_1^{\beta^j}$ – the argument is identical.

Denote the pruned version of T_1 by T'_1 . Every leaf label of X appears once; this follows directly from C' being an exact 3 cover. All other leaf labels appear only once by inspection; hence this is a perfect pruning.

We now prune T_2 as follows:

- For each $T_2^{\rho^j}$, prune away all leaf labels except $a_{\psi^{-1}(j)}^j$ and $b_{\psi^{-1}(j)}^j$; after pruning ρ^j has two children, both leaves.
- For each $T_2^{\bar{\alpha}^j}$, we wish to prune this subtree to create a $(k-1)$ leaf star with all labels a_i^j for our fixed j except $a_{\psi^{-1}(j)}^j$. This is possible due to Lemma 13; from each pair of leaf labels rooted by a child of $\bar{\alpha}^j$ we pick one leaf to keep and one to prune away such that we keep one copy of everything except $a_{\psi^{-1}(j)}^j$.
- Repeat the previous step for each $T_2^{\bar{\beta}^j}$ – as before the argument is identical.

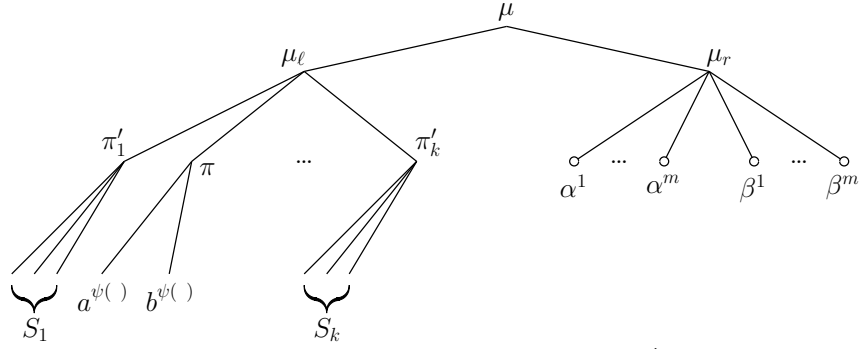
There is one copy of each leaf label of X in T_2 , which we do not prune. We prune the leaf labels of $A \cup B$ in T_2 so that the leaf labels of $A \cup B$ in $T_2^{\gamma_\ell}$ are exactly those that do not appear in $T_2^{\gamma_\ell}$. Hence this is a perfect pruning, which we denote T'_2 .

We now show that T'_2 can be constructed from T'_1 by repeated non-leaf edge contractions, which will show $\{T'_1, T'_2\}$ form a compatible set (with $T^* := T'_1$).

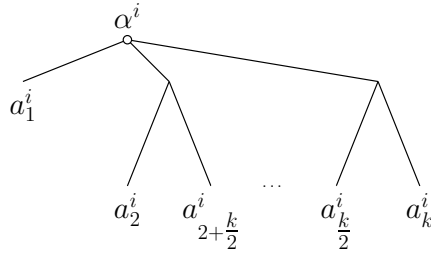
In T'_1 contract every remaining π'_i (one for each of the q sets $S_i \in C'$) into its parent μ_ℓ . Furthermore in each $T_1^{\alpha^i}$ and $T_1^{\beta^i}$, contract every non-leaf edge to create a star rooted at α_i or β_i . By inspection, we can see $T'_2 \leq T'_1$, giving our required compatible set of trees. ◀

See Figures 8, 9, 10, and 11 for an example of a pruning as in Lemma 15.

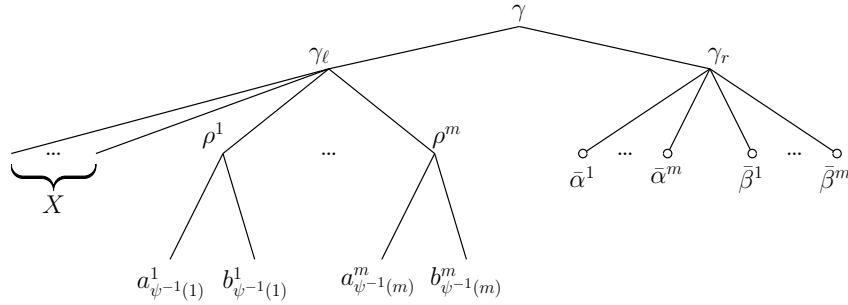
14:12 MUL-Tree Pruning for Consistency and Compatibility



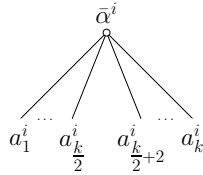
■ **Figure 8** As an illustrated example, a possible perfect pruning T'_1 as in Lemma 15. In this example, $S_1, S_k \in C'$, but $S_2 \notin C'$.



■ **Figure 9** A subtree of T'_1 in the pruning from Figure 8. In this example, $\psi(1 + \frac{k}{2}) = i$.



■ **Figure 10** A possible perfect pruning T'_2 , corresponding to the perfect pruning of Figure 8.



■ **Figure 11** A subtree of T'_2 in the pruning from Figure 10. Again, here $\psi^{-1}(i) = \frac{k}{2} + 1$.

► **Lemma 16.** *If (X, C) is an instance of X3C3 that does not allow an exact 3-cover C' then the corresponding instance $(\{T_1, T_2\}, Y)$ of MULSETPCOMP does not admit a perfect pruning giving a compatible set of trees.*

We omit the proof of Lemma 16 due to space concerns. We now prove Theorem 11.

Proof of Theorem 11. Note first that MULSETPComp is in NP, since given a set of pruned leaves, a perfect pruning can be constructed in polynomial time. The compatibility of this set of trees can be determined in polynomial time using the BUILDST algorithm [9]. The result then follows directly from Lemma 15 and Lemma 16. ◀

We close this section with the following related result.

► **Remark 17.** MULSETPComp is NP-complete when restricted to instances with at most two MUL-trees with multiplicity 2, as long as all trees are binary.

Recall that in general any consistent set of trees is also a compatible set; in the case where all trees are binary the inverse also holds, as a binary tree cannot be refined further. This proves Remark 17.

5 An algorithm for MULSETPC instances with k binary MUL-trees where every label is unique in at least one tree

In this section, we consider the instances of MULSETPC in which we are given k binary MUL-trees T_1, \dots, T_k such that every label appears uniquely in at least one tree, that is, $\bigcup_{i=1}^k D(T_i) = \mathcal{X}$, where $\mathcal{X} = \bigcup_{i=1}^k \mathcal{X}(T_i)$.

We adapt a technique using dynamic programming over k -tuples of nodes previously used for the MAST problem [13, 22, 32]. For all k -tuples of nodes $(a_1, \dots, a_k) \in \prod_{i=1}^k V(T_i)$, let $S(a_1, \dots, a_k) = \bigcup_{i=1}^k D(T_i^{a_i})$ denote the set of unique leaf labels that occur in the subtrees $T_1^{a_1}, \dots, T_k^{a_k}$, rooted at a_1, \dots, a_k , respectively.

We aim to find a binary tree T that is leaf-labelled by $S(a_1, \dots, a_k)$ such that each of $T_i^{a_i}$ for $i = 1, \dots, k$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$. Lemma 18, below, shows that the necessary condition of the existence of such a tree is that $S(a_1, \dots, a_k) \cap \mathcal{X}(T_i) \subseteq \mathcal{X}(T_i^{a_i})$ for $i = 1, \dots, k$.

► **Lemma 18.** *Let T be a binary tree leaf-labelled by $S(a_1, \dots, a_k)$. If $S(a_1, \dots, a_k) \cap \mathcal{X}(T_i) \not\subseteq \mathcal{X}(T_i^{a_i})$ for some i , $T_i^{a_i}$ does not display $T \upharpoonright_{\mathcal{X}(T_i)}$.*

Proof. Suppose that $S(a_1, \dots, a_k) \cap \mathcal{X}(T_i) \not\subseteq \mathcal{X}(T_i^{a_i})$, then there exists $x \in S(a_1, \dots, a_k)$ such that $x \notin \mathcal{X}(T_i^{a_i})$. Since $x \in T \upharpoonright_{\mathcal{X}(T_i)}$, we have that $T_i^{a_i}$ cannot display $T \upharpoonright_{\mathcal{X}(T_i)}$. ◀

Next, for each $a_i \in V(T_i)$, let $P(a_i) = \{\epsilon, a_i, a_i^l, a_i^r\}$, for each $i = 1, \dots, k$, where a_i^l and a_i^r denotes the two (unordered) children of vertex a_i , and with $a_i^l = \epsilon$ and $a_i^r = a_i$ in the case that $a_i \in L$ is a leaf vertex w.l.o.g. Let $c_i : P(a_i) \rightarrow P(a_i)$ be the involution given by

$$c_i(\epsilon) = a_i, \quad c_i(a_i) = \epsilon, \quad c_i(a_i^r) = a_i^l, \quad \text{and} \quad c_i(a_i^l) = a_i^r,$$

associating each $x \in P(a_i)$ with a complement. Let $\Pi(a_1, \dots, a_k) = \prod_{i=1}^k P(a_i) - \{(\epsilon, \dots, \epsilon), (a_1, \dots, a_k)\}$, and note that $|\Pi(a_1, \dots, a_k)| \leq 4^k$, for all $a_i \in V(T_i)$, for $i = 1, \dots, k$.

We define a function $W : \prod_{i=1}^k V(T_i) \rightarrow \{\text{true}, \text{false}\}$ recursively, as follows:

- If $|S(a_1, \dots, a_k)| \leq 1$, $W(a_1, \dots, a_k) = \text{true}$.
- If $S(a_1, \dots, a_k) \cap \mathcal{X}(T_i) \not\subseteq \mathcal{X}(T_i^{a_i})$ for some i , $W(a_1, \dots, a_k) = \text{false}$.
- Otherwise,

$$W(\vec{a}) = \bigvee_{\vec{x} \in \Pi(\vec{a})} \left(W(\vec{x}) \wedge W(\vec{c}(\vec{x})) \wedge Q(\vec{a}, \vec{x}) \wedge Q(\vec{a}, \vec{c}(\vec{x})) \right)$$

14:14 MUL-Tree Pruning for Consistency and Compatibility

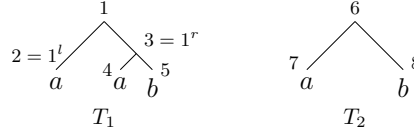
where $\vec{x} = (x_1, \dots, x_k)$, $\vec{c}(\vec{x}) = (c_1(x_1), \dots, c_k(x_k))$ and

$$\begin{aligned} Q(\vec{a}, \vec{x}) &= \bigwedge_{i \neq j} \left(x_i = a_i \wedge x_j = \epsilon \rightarrow L(a_i) \cap L(a_j^\ell) = \emptyset \vee L(a_i) \cap L(a_j^r) = \emptyset \right) \\ &\wedge \bigwedge_{i \neq j} \left(x_i = a_i \wedge x_j = a_j^\ell \rightarrow L(a_i) \cap L(a_j^r) = \emptyset \right) \\ &\wedge \bigwedge_{i \neq j} \left(x_i = a_i \wedge x_j = a_j^r \rightarrow L(a_i) \cap L(a_j^\ell) = \emptyset \right). \end{aligned}$$

We may define a partial ordering \triangleleft on $\prod_{i=1}^k (V(T_i) \cup \{\epsilon\})$ by taking

$$(a_1, \dots, a_k) \triangleleft (a'_1, \dots, a'_k) \iff a_i = \epsilon \text{ or } a_i \prec_i a'_i, \text{ for all } i = 1, \dots, k$$

where \prec_i is the successor relation in T_i , with the unique \triangleleft -minimum element $(\epsilon, \dots, \epsilon)$. An example computation of the function W is described in Figure 12.



■ **Figure 12** Given the trees T_1 and T_2 , $W(1, 6) = \mathbf{true}$ since $W(2, 7) \wedge W(3, 8) \leftarrow W(2, 7) \wedge W(4, \epsilon) \wedge W(5, 8) = \mathbf{true}$. Note that $W(1, 6)$ would also compute $W(2, 8) \wedge W(3, 7)$, $W(2, 6) \wedge W(3, \epsilon)$, $W(2, \epsilon) \wedge W(3, 6)$, $W(1, 7) \wedge W(\epsilon, 8)$, and $W(1, 8) \wedge W(\epsilon, 7)$.

► **Lemma 19.** *Let T_1, T_2, \dots, T_k be a collection of k binary MUL-trees such that $\bigcup_i \mathcal{X}(T_i) = \bigcup_i D(T_i)$. Then $W(a_1, \dots, a_k) = \mathbf{true}$ if and only if there exists a single-labelled tree T leaf-labelled by $S(a_1, \dots, a_k)$ such that $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$, under a mapping that maps $r_i(T) \mapsto x_i$, for all $i = 1, \dots, k$.*

Proof. We prove this by induction on k . For the base case, suppose that each of $(a_1, \dots, a_k) = (\epsilon, \dots, \epsilon)$ is the \prec -minimum, so that $T_i^{a_i} = T_i^\epsilon = T_\emptyset$ is the empty tree, for $i = 1, \dots, k$. In which case $S(\epsilon, \dots, \epsilon) = \emptyset$, and hence $W(a_1, \dots, a_k) = \mathbf{true}$ by definition, while (trivially) the empty tree $T = T_\emptyset$ is such that $T_i^{a_i} = T_\emptyset$ displays $T_\emptyset \upharpoonright_{\mathcal{X}(T_i)} = T_\emptyset$. Next, suppose that the result holds for all $(u_1, \dots, u_k) \triangleleft (a_1, \dots, a_k)$ for some tuple (a_1, \dots, a_k) . We claim that the result holds too for (a_1, \dots, a_k) .

(\Leftarrow) Suppose that T is as described. Then for each $i = 1, \dots, k$ there is some subtree $S_i \subseteq T^{a_i}$ and some label-preserving isomorphism $f_i : V(S_i) \rightarrow V(T \upharpoonright_{\mathcal{X}(T_i)})$.

Let T^ℓ and T^r denote the left and right subtrees attached at $r(T)$. As T is a single-labelled tree, it follows that $L(T^\ell) \cap L(T^r) = \emptyset$, as each label occurs exactly once.

We can partition each $V(S_i)$ into three parts $L_i = \{v \in V(S_i) : f_i(v) \in T^\ell\}$ and $R_i = \{v \in V(S_i) : f_i(v) \in T^r\}$ and $C_i = \{v \in V(S_i) : f_i(v) = r(T)\}$. Note that, since f_i is an isomorphism, if $u \in X$ and $u \prec_i v$ then $v \in X$, for $X \in \{L_i, R_i\}$.

We have three cases depending on which of these three sets lies the root node a_i of the subtree $T_i^{a_i}$:

- If $a_i \in C_i$ then it follows that either $a_i^\ell \in L_i$ and $a_i^r \in R_i$ or $a_i^r \in L_i$ and $a_i^\ell \in R_i$.
 - If $a_i^\ell \in L_i$ and $a_i^r \in R_i$ then $T_i^{a_i^\ell}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$ and $T_i^{a_i^r}$ displays $T^r \upharpoonright_{\mathcal{X}(a_i)}$.
 - If $a_i^r \in L_i$ and $a_i^\ell \in R_i$ then $T_i^{a_i^r}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$ and $T_i^{a_i^\ell}$ displays $T^r \upharpoonright_{\mathcal{X}(a_i)}$.

In each case, there is some $x_i \in \{a_i^\ell, a_i^r\}$ such that $T_i^{x_i}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$ and $T_i^{c_i(x_i)}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$.

- If $a_i \in L_i$ then it follows that $L_i = V(S_i)$ and $C_i = R_i = \emptyset$. Hence we have that $T_i^{a_i}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$, while $T_i^\epsilon = T_\emptyset$ (trivially) displays the empty tree $T^r \upharpoonright_{\mathcal{X}(a_i)} = T_\emptyset$.
- Symmetrically, if $a_i \in R_i$ then $T_i^{a_i}$ displays $T^r \upharpoonright_{\mathcal{X}(a_i)}$ and T_i^ϵ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$.

In all cases there is some $x_i \in P(a_i)$ such that $T_i^{x_i}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$ and $T_i^{c_i(x_i)}$ displays $T^\ell \upharpoonright_{\mathcal{X}(a_i)}$, for all $i = 1, \dots, k$. Since both $(x_1, \dots, x_k), (c_1(x_1), \dots, c_k(x_k)) \triangleleft (a_1, \dots, a_k)$, it follows from the induction hypothesis that $W(\vec{x}) \wedge W(\vec{c}(\vec{x})) = \mathbf{true}$.

For all $i \neq j$, if $x_i = a_i$ and $x_j = \epsilon$ then by definition $a_i \in L_i \subseteq L(T^\ell)$ while $a_j \in C_j$. If $a_j^\ell \in R_i$ then $L(a_i) \cap L(a_j^\ell) = \emptyset$, otherwise $a_j^r \in R_i$ and so $L(a_i) \cap L(a_j^r) = \emptyset$, since $R_i \subseteq L(T^r)$ and $L(T^\ell) \cap L(T^r) = \emptyset$. If $x_i = a_i$ and $x_j = a_j^l$ then $a_i \in L_i \subseteq L(T^l)$ while $a_j^\ell \in L_j$ and $a_j^r \in R_j \subseteq L(T^r)$. From which it follows that $L(a_i) \cap L(a_j^r) = \emptyset$.

Similarly, if $x_i = a_i$ and $x_j = a_j^r$ then it follows $L(a_i) \cap L(a_j^l) = \emptyset$. This is to say that $Q(\vec{a}, \vec{x}) = \mathbf{true}$, and by the same argument, so too that $Q(\vec{a}, \vec{c}(\vec{x})) = \mathbf{true}$.

Hence, by definition, $W(a_1, \dots, a_k) = \mathbf{true}$, as required.

(\Rightarrow) Suppose that $W(a_1, \dots, a_k) = \mathbf{true}$, then there are two possible cases: (i) $|S(a_1, \dots, a_k)| \leq 1$; (ii) there is some $(x_1, \dots, x_k) \in \Pi(a_1, \dots, a_k)$ such that $W(\vec{x}) \wedge W(\vec{c}(\vec{x})) \wedge Q(\vec{x}) = \mathbf{true}$:

- (i) If $S(a_1, \dots, a_k) = \emptyset$ then we may take $T_i^{a_i}$ to display $T = T_\emptyset$ as the empty tree for all $i = 1, \dots, k$. Otherwise, if $S(a_1, \dots, a_k) = \{x\}$ is a singleton then we may take T to be the tree with a single leaf-labelled by x , where it is straightforward to check that $T_i^{a_i}$ can display $T \upharpoonright_{\mathcal{X}(a_i)}$ for all $i = 1, \dots, k$.
- (ii) It follows from the induction hypothesis that there exist single-labelled trees T_x and T_y , leaf-labelled by $S(x_1, \dots, x_k)$ and $S(y_1, \dots, y_k)$, respectively, such that $T_i^{x_i}$ displays $T_x \upharpoonright_{\mathcal{X}(T_i)}$ and $T_i^{y_i}$ under a mapping that maps $r_i(T_x) \mapsto x_i$, and displays $T_y \upharpoonright_{\mathcal{X}(T_i)}$ under a mapping that maps $r_i(T_y) \mapsto y_i$, for all $i = 1, \dots, k$, where $y_i = c(x_i)$. If $L(T_x) \cap L(T_y) = \emptyset$ then we construct a new tree T by connecting the roots $r(T_x)$ and $r(T_y)$ of T_x and T_y to a common (new) root node r .

Otherwise since $Q(\vec{a}, \vec{x}) \wedge Q(\vec{a}, \vec{c}(\vec{x})) = \mathbf{true}$, it follows that either $L(T_x) \cap L(T_y^r) = \emptyset$ or $L(T_x) \cap L(T_y^\ell) = \emptyset$. In the first case we can construct a new tree T by merging T_x with the left sub-tree of T_y , while in the latter case we can construct T by merging T_x with the left sub-tree of T_y .

In all cases, we have that T is a single-labelled tree, as required, as it remains to show that $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$, for each $i = 1, \dots, k$:

- If $x_i = a_i^r$ then $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$, mapping $r(T^x) \mapsto a_i^r$, $r(T^y) \mapsto a_i^\ell$, and $r \mapsto a_i$.
- If $x_i = a_i^\ell$ then $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$, mapping $r(T^x) \mapsto a_i^\ell$, $r(T^y) \mapsto a_i^r$, and $r \mapsto a_i$.
- If $x_i = a_i$ then $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$ under the mapping that maps $r(T^x) \mapsto a_i$.
- If $x_i = \epsilon$ then $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$ under the mapping that maps $r(T^y) \mapsto a_i$.

Hence, it follows from induction that $W(a_1, \dots, a_k) = \mathbf{true}$ if and only if there is some tree T leaf-labelled by $S(a_1, \dots, a_k)$ such that $T_i^{a_i}$ displays $T \upharpoonright_{\mathcal{X}(T_i)}$ under a mapping that maps $r_i(T) \mapsto x_i$, for each $i = 1, \dots, k$, as required. \blacktriangleleft

Lemma 19 provides us with a criterion for deciding the MULSETPC problem for a given collection of binary MUL-trees, that can be computed in polynomial-time in the size of the trees, for any fixed number of trees, and scales exponentially with the number of trees.

► **Theorem 20.** *Let T_1, T_2, \dots, T_k be a collection of k binary MUL-trees such that $\bigcup_i \mathcal{X}(T_i) = \bigcup_i D(T_i)$. Then MULSETPC for this instance can be solved in $O(k^2 \cdot 4^k \prod_{i=1}^k (|T_i| + 1)) = O(4^k \prod_{i=1}^k |T_i|)$ time.*

Proof. Based on Lemma 19, it is sufficient to compute $W(r(T_1), \dots, r(T_k))$, since $T_i^{r(T_i)} = T_i$, by definition. We can compute W via dynamic programming as outlined in Algorithm 1, which will return **true** if T_1, \dots, T_k display a single labelled tree.

■ **Algorithm 1** Recursive dynamic programming algorithm for $W(a_1, \dots, a_k)$.

```

1: let  $S = S(a_1, \dots, a_k)$ 
2: if  $|S| \leq 1$  then return true
3: else if  $S \cap \mathcal{X}(T_i) \not\subseteq \mathcal{X}(T_i^{a_i})$  for some  $i = 1, \dots, k$  then return false
4: else
5:   for  $(x_1, \dots, x_k) \in \Pi(a_1, \dots, a_k)$  do
6:     if  $W(\vec{x}) \wedge W(\vec{c}(\vec{x})) \wedge Q(\vec{a}, \vec{x}) \wedge Q(\vec{a}, \vec{c}(\vec{x}))$  then return true
7:   return false

```

For the time complexity, we can use memoization to store the values of W in a table with at most $O(\prod_{i=1}^k |V(T_i) \cup \{\epsilon\}|) = O(\prod_{i=1}^k (|T_i| + 1))$ entries. Furthermore, we require at most $O(k^2 \cdot 4^k)$ time to compute the value of each entry $W(a_1, \dots, a_k)$, since $|\Pi(a_1, \dots, a_k)| = |P(a_i)| \times \dots \times |P(a_k)| \leq 4^k$, while $Q(\vec{a}, \vec{x})$ and $Q(\vec{a}, \vec{c}(\vec{x}))$ can each be computed in quadratic time. Hence, the running time is $O(k^2 \cdot 4^k \prod_{i=1}^k (|T_i| + 1)) = O(4^k \prod_{i=1}^k |T_i|)$, as required. ◀

6 Conclusions

The above results resolve an open problem posed in [15, 16] as to whether the MULSETPC problem remains NP-complete when the number of MUL-trees is constant. According to Theorems 9 and 10, two MUL-trees are sufficient for NP-completeness, even with each label appearing at most twice in any tree and either the height or the degree constant. Theorems 11 and 12 extend this result and show that the more general MULSETPComp problem also remains NP-complete even when the number of MUL-trees is constant. Theorem 9 is tight in the sense that, if we restrict our attention to MUL-trees in which each label appears uniquely in at least one tree, we obtain a polynomial-upper bound for a fixed number of trees (Theorem 20). However, Theorem 12 suggests the algorithm presented in Theorem 20 for MULSETPC cannot be directly generalised to solve equivalent MULSETPComp instances in polynomial time, unless $P = NP$.

The above results also suggest two new open problems. Firstly, is it possible to improve Theorem 11 to show that MULSETPComp is still NP-complete when restricted to MUL-trees with multiplicity 2? Secondly, what can be said about the complexity of MULSETPC and MULSETPComp for instances in which the multiplicity is not restricted but the number of leaf labels that may appear more than once is restricted? That is, for each MUL-tree, k leaf labels may appear an unbounded number of times in the tree, whereas all other labels appear at most once. For which values of k are these subproblems NP-complete? This question is interesting because of its connection to the instances investigated in Section 5.

References

- 1 Alfred V. Aho, Yehoshua Sagiv, Thomas G. Szymanski, and Jeffrey D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.
- 2 Amihood Amir and Dmitry Kesselman. Maximum Agreement Subtree in a Set of Evolutionary Trees: Metrics and Efficient Algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997.

- 3 Mukul S Bansal. Linear-time algorithms for some phylogenetic tree completion problems under Robinson-Foulds distance. In *RECOMB International conference on Comparative Genomics*, pages 209–226. Springer, 2018.
- 4 Mukul S Bansal, J Gordon Burleigh, Oliver Eulenstein, and David Fernández-Baca. Robinson-Foulds supertrees. *Algorithms for molecular biology*, 5(1):1–12, 2010.
- 5 Magnus Bordewich and Charles Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of combinatorics*, 8(4):409–423, 2005.
- 6 David Bryant. A classification of consensus methods for phylogenetics. In M. F. Janowitz, F.-J. Lapointe, F. R. McMorris, B. Mirkin, and F. S. Roberts, editors, *Bioconsensus*, volume 61 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 163–184. American Mathematical Society, 2003.
- 7 Richard Cole, Martin Farach-Colton, Ramesh Hariharan, Teresa Przytycka, and Mikkel Thorup. An $O(n \log n)$ Algorithm for the Maximum Agreement Subtree Problem for Binary Trees. *SIAM Journal on Computing*, 30(5):1385–1404, 2000.
- 8 Yun Cui, Jesper Jansson, and Wing-Kin Sung. Polynomial-time Algorithms for Building a Consensus MUL-Tree. *Journal of Computational Biology*, 19(9):1073–1088, 2012.
- 9 Yun Deng and David Fernández-Baca. Fast compatibility testing for rooted phylogenetic trees. *Algorithmica*, 80(8):2453–2477, 2018.
- 10 Zhihong Ding, Vladimir Filkov, and Dan Gusfield. A linear-time algorithm for the perfect phylogeny haplotyping (PPH) problem. *Journal of Computational Biology*, 13(2):522–553, 2006.
- 11 Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, Massachusetts, 2004.
- 12 CR Finden and AD Gordon. Obtaining common pruned trees. *Journal of Classification*, 2(1):255–276, 1985.
- 13 Ganeshkumar Ganapathy, Barbara Goodson, Robert Jansen, Hai-son Le, Vijaya Ramachandran, and Tandy Warnow. Pattern identification in biogeography. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):334–346, 2006.
- 14 Michael R Garey and David S Johnson. *Computers and intractability*. A Series of Books in the Mathematical Sciences. W. H. Freeman and Co., San Francisco, Calif., 1979. A guide to the theory of NP-completeness.
- 15 Mathieu Gascon, Riccardo Dondi, and Nadia El-Mabrouk. Complexity and Algorithms for MUL-Tree Pruning. In Paola Flocchini and Lucia Moura, editors, *Combinatorial Algorithms*, pages 324–339, Cham, 2021. Springer International Publishing.
- 16 Mathieu Gascon, Riccardo Dondi, and Nadia El-Mabrouk. MUL-tree pruning for consistency and optimal reconciliation - complexity and algorithms. *Theoret. Comput. Sci.*, 937:22–38, 2022.
- 17 Katharina T Huber and Vincent Moulton. Phylogenetic networks from multi-labelled trees. *Journal of Mathematical Biology*, 52(5):613–632, 2006.
- 18 Katharina T Huber, Vincent Moulton, Mike Steel, and Taoyang Wu. Folding and unfolding phylogenetic trees and networks. *Journal of Mathematical Biology*, 73(6):1761–1780, 2016.
- 19 Katharina T Huber, Bengt Oxelman, Martin Lott, and Vincent Moulton. Reconstructing the evolutionary history of polyploids from multilabeled trees. *Molecular Biology and Evolution*, 23(9):1784–1791, 2006.
- 20 Leo van Iersel, Steven Kelk, Nela Lekić, and Celine Scornavacca. A practical approximation algorithm for solving massive instances of hybridization number for binary and nonbinary trees. *BMC bioinformatics*, 15(1):1–12, 2014.
- 21 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM*, 63(3), 2016. Article 28.
- 22 Manuel Lafond, Nadia El-Mabrouk, Katharina T Huber, and Vincent Moulton. The complexity of comparing multiply-labelled trees by extending phylogenetic-tree metrics. *Theoretical Computer Science*, 760:15–34, 2019.

14:18 MUL-Tree Pruning for Consistency and Compatibility

- 23 Martin Lott, Andreas Spillner, Katharina T Huber, Anna Petri, Bengt Oxelman, and Vincent Moulton. Inferring polyploid phylogenies from multiply-labeled gene trees. *BMC Evolutionary Biology*, 9(1):1–11, 2009.
- 24 Nobuhiro Minaka. Cladograms and reticulated graphs: A proposal for graphic representation of cladistic structures. *Bulletin of the Biogeographical Society of Japan*, 45(1):1–10, 1990.
- 25 Gordon L Nelson and Norman I Platnick. *Systematics and Biogeography: Cladistics and Vicariance*. Columbia University Press, 1981.
- 26 Roderic D M Page. Parasites, phylogeny and cospeciation. *International Journal for Parasitology*, 23(4):499–506, 1993.
- 27 Roderic D M Page. Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas. *Systematic Biology*, 43(1):58–77, 1994.
- 28 Itsik Pe’er, Tal Pupko, Ron Shamir, and Roded Sharan. Incomplete directed perfect phylogeny. *SIAM J. Comput.*, 33(3):590–607, 2004.
- 29 David F Robinson and Leslie R Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981.
- 30 Celine Scornavacca, Vincent Berry, and Vincent Ranwez. Building species trees from larger parts of phylogenomic databases. *Information and Computation*, 209(3):590–605, 2011.
- 31 Mike Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classification*, 9(1):91–116, 1992.
- 32 Mike Steel and Tandy Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree. *Information Processing Letters*, 48:77–82, 1993.
- 33 Christopher Whidden, Norbert Zeh, and Robert G Beiko. Supertrees Based on the Subtree Prune-and-Regraft Distance. *Systematic biology*, 63(4):566–581, 2014.
- 34 Yufeng Wu. A practical method for exact computation of subtree prune and regraft distance. *Bioinformatics*, 25(2):190–196, 2009.


Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

Costas S. Iliopoulos  

Department of Informatics, King's College London, London, UK

Tomasz Kociumaka  

Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

Jakub Radoszewski  



Institute of Informatics, University of Warsaw, Poland

Wojciech Rytter  

Institute of Informatics, University of Warsaw, Poland

Tomasz Waleń  

Institute of Informatics, University of Warsaw, Poland

Wiktor Zuba  

CWI, Amsterdam, The Netherlands

Abstract

Cyclic versions of covers and roots of a string are considered in this paper. A prefix V of a string S is a *cyclic root* of S if S is a concatenation of cyclic rotations of V . A prefix V of S is a *cyclic cover* of S if the occurrences of the cyclic rotations of V cover all positions of S . We present $\mathcal{O}(n)$ -time algorithms computing all cyclic roots (using number-theoretic tools) and all cyclic covers (using tools related to seeds) of a length- n string over an integer alphabet. Our results improve upon $\mathcal{O}(n \log \log n)$ and $\mathcal{O}(n \log n)$ time complexities of recent algorithms of Grossi et al. (WALCOM 2023) for the respective problems and provide novel approaches to the problems. As a by-product, we obtain an optimal data structure for Internal Circular Pattern Matching queries that generalize Internal Pattern Matching and Cyclic Equivalence queries of Kociumaka et al. (SODA 2015).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases cyclic cover, cyclic root, circular pattern matching, internal pattern matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.15

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Tomasz Waleń: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Wiktor Zuba: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.

1 Introduction

Cyclic strings have many real-world applications, such as in bioinformatics [2, 3, 17, 19] and image processing [1, 27, 28, 29]. In particular, they are used for detecting DNA viruses with circular structures [30, 31]. In particular, cyclic strings were studied in the context of circular pattern matching [10, 14, 23, 24].

In this paper, we investigate the complexity of two problems related to cyclic strings. The first one is a cyclic variant of the problem of computing the *roots* of a string S , i.e., strings U such that $S = U^k$ for some integer k . The second one is a cyclic variant of the problem of computing the *covers* of a string S , i.e., strings C whose occurrences cover the



© Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 15; pp. 15:1–15:15



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

15:2 Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

whole string S . The standard roots of a string can be easily computed in linear time using a folklore algorithm. Moore and Smyth [25, 26] gave a linear-time algorithm computing all standard covers of a string. However, the cyclic versions of these problems are more difficult.

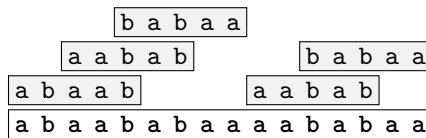
We say that a string V is a (*cyclic*) *rotation* of a string U if there exist strings X and Y such that $U = XY$ and $V = YX$. A string U has a *circular occurrence* in a string T at position i if a rotation of U has a (standard) occurrence in T at position i .¹ By $CircOcc(U, T)$ we denote the set of circular occurrences of U in T . Moreover, we denote

$$Covered(U, T) = \bigcup_{i \in CircOcc(U, T)} [i..i + |U|).$$

► **Definition 1.** A string U is a cyclic cover of a string S if $Covered(U, S) = [0..|S|)$. A string U is a cyclic root of a string S if $S = U_1 \cdots U_k$, where each U_i is a cyclic shift of U .

Note that if U is a cyclic root (cyclic cover) of S , then the prefix $S[0..|U|)$ is also a cyclic root (cyclic cover, respectively) of S .

► **Example 2.** The lengths of the cyclic roots of the Thue–Morse word `abbabaabbaabba` are 2, 4, 8, 16.



■ **Figure 1** The string `abaab` is a cyclic cover of the string `abaababaaaababaa`.

► **Example 3.** The string `ab` is a cyclic cover of each Fibonacci string of length at least 2, e.g., of the string $Fib_5 = \text{abaababa}$. (See [13] for a definition of Fibonacci strings and their properties.) However, it is not a cyclic root of any Fibonacci string longer than 2. Another example of a cyclic cover of a string is illustrated in Figure 1.

We consider the following problems.

CYCLICROOTS

Input: A string S of length n .

Output: The lengths of all cyclic roots of S .

CYCLICCOVERS

Input: A string S of length n .

Output: The lengths of all cyclic covers of S .

Our results

We show linear-time algorithms for both problems. We assume the word-RAM model of computation and that the string S is over an integer alphabet $\{0, \dots, n^{O(1)}\}$.

¹ We assume that the positions of a string T are numbered 0 through $|T| - 1$, where $|T|$ is the length of T .

Previous results

Recently, Grossi et al. [16] presented an $\mathcal{O}(n \log \log n)$ -time algorithm for CYCLICROOTS (named cyclic factorization there) and an $\mathcal{O}(n \log n)$ -time algorithm for CYCLICCOVERS.

► **Remark 4.** A completely different problem of covering a *cyclic string with a standard string cover* was considered in [11, 12]. Another different problem also known under the name “cyclic covers”, related to the shortest superstring problem, was considered in [4, 5, 6].

Our approach

In the case of cyclic covers, we use a recursive algorithm whose general structure partially resembles the structure of the linear-time algorithm for computing seeds from [21]. For this, we need to explore combinatorics of circular occurrences, which is different from that of standard occurrences. The “working horse” of the algorithm (non-recursive parts) is the computation of long cyclic covers, which is based on an efficient implementation of internal circular pattern matching queries. Such queries require to find all circular occurrences of one substring of a text in another substring; the set of occurrences can be represented compactly if the ratio of lengths of the two strings is constant. An auxiliary contribution of our paper is an optimal implementation of these queries (constant-time after linear-time preprocessing).

Also in the case of cyclic roots we use internal queries on strings. Our algorithm is based on number-theoretic tools and fast internal queries for cyclic equivalence, which ask if two substrings of a given string are rotations of each other [20, 22].

Notations

For a string S , by $S[0], \dots, S[|S| - 1]$ we denote its respective letters. By $S[i..j]$ we denote a substring $S[i] \cdots S[j - 1]$; similarly, we define substrings $S[i..j]$, $S(i..j)$ and $S(i..j)$. We say that p is a period of a string S if $S[i] = S[i + p]$ holds for all $i \in [0..|S| - p]$. By $\text{per}(S)$, we denote the smallest period of S , called *the period* of S . For a string S and integer $x \in [0..|S|)$, by $\text{rot}_x(S)$ we denote the rotation $S[x..|S|) \cdot S[0..x)$ of S obtained from S by moving the prefix of S of length x to the end.

A length- m string P has an *occurrence* in a string T at position i if $T[i..i + m) = P$. By $\text{Occ}(P, T)$ we denote the set of starting positions of occurrences of P in T .

2 Internal Circular Pattern Matching and CyclicCovers in $\mathcal{O}(n \log n)$ Time

We introduce the following generalization of Internal Pattern Matching queries from [22].

SIMPLE INTERNAL CIRCULAR PATTERN MATCHING QUERIES (SIMPLE INTERNALCPM)

Input: A string S of length n .

Queries: Given two substrings P and T of S such that $|T| \leq 2|P|$, report the leftmost and the rightmost circular occurrence of P in T .

► **Remark 5.** If we know how to compute the leftmost circular occurrence, then the rightmost circular occurrence can be computed analogously (it suffices to reverse the strings).

The theorem below can be obtained using the methods from [7, 8]. We give its proof in Section 6.

► **Theorem 6.** *The SIMPLE INTERNALCPM queries can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing.*

15:4 Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

► **Remark 7.** In Section 6, we obtain a version of the queries in which a constant-sized representation of *all* circular occurrences is computed in constant time (still if $|T| \leq 2|P|$).

Below, we apply SIMPLE INTERNALCPM queries to a version of the CYCLICCOVERS problem that is used in our $\mathcal{O}(n)$ -time algorithm for CYCLICCOVERS. The following lemma generalizes [16, Lemma 13]; in this section, it will be applied in a simpler setting.

► **Lemma 8.** *After $\mathcal{O}(n)$ -time preprocessing of a string S of length n , for any substrings C and W of S , we can test if C is a cyclic cover of W in $\mathcal{O}(|W|/|C|)$ time.*

Proof. Let $p = |C|$ and $m = |W|$. Consider substrings V_i equal to $W[ip..(i+2)p]$ for $i \in [0.. \lfloor m/p \rfloor - 1]$ and $W[ip..m]$ for $i = \lfloor m/p \rfloor - 1$. For each substring V_i , we use a SIMPLE INTERNALCPM query to compute the leftmost and the rightmost circular occurrence of C . Then, we check whether these occurrences, interpreted as occurrences in W , collectively cover all positions in W . The time complexity is $\mathcal{O}(m/p)$ after the preprocessing of Theorem 6. ◀

Lemma 8 implies a simple $\mathcal{O}(n \log n)$ -time algorithm for the CYCLICCOVERS problem.

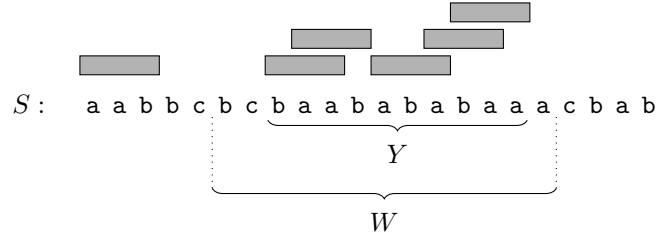
► **Corollary 9.** *The CYCLICCOVERS problem can be solved in $\mathcal{O}(n \log n)$ time.*

Proof. We apply Lemma 8 for $W = S$ iterating with C over all non-empty prefixes of S . The total time complexity is $\mathcal{O}(n + \sum_{i=1}^n \frac{n}{i}) = \mathcal{O}(n \log n)$. ◀

3 Quasi-Covers

We reduce our problem to the computation of the substrings called *quasi-covers*; see Figure 2.

► **Definition 10.** *A string V is a quasi-cover of a substring W of the string S if V is a prefix of S and a cyclic cover of a substring $Y = W[i..j]$ such that $i < |V|$ and $j > |W| - |V|$.*



■ **Figure 2** Example of a quasi-cover **aab** of the substring W of S . By definition, V is a prefix of the whole string S . Observe that **aab** is not a cyclic cover of W .

Henceforth, we fix the string S and consider quasi-covers of its substrings. Let W be a substring and I be an interval, $I \subseteq [1..|W|]$. We denote by $\text{Q-COVERS}_I(W)$ the set of all lengths of quasi-covers of substring W with lengths in I . Furthermore, for $k \in [1..|W|]$ we denote

$$\begin{aligned} \text{Q-COVERS}(W) &= \text{Q-COVERS}_{[1..|W|]}(W), & \text{Q-COVERS}_{\leq k}(W) &= \text{Q-COVERS}_{[1..k]}(W), \\ \text{Q-COVERS}_{>k}(W) &= \text{Q-COVERS}_{(k..|W|]}(W), & \text{Q-COVERS}_k(W) &= \text{Q-COVERS}_{[k..k]}(W). \end{aligned}$$

A prefix of S is its cyclic cover if and only if it is a quasi-cover of S and a rotation of a suffix of S . We use the following queries (generalized by SIMPLE INTERNALCPM queries):

CYCLIC EQUIVALENCE QUERIES (CYCEQ)

Input: A string S of length n .**Queries:** Given two substrings U and V of S , check if V is a rotation of U .

► **Theorem 11** ([20, 22]). *The CYCEQ queries can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing.*

Using CYCEQ queries, we can compute in $\mathcal{O}(n)$ time all prefixes which are rotations of the corresponding suffixes of the string. This yields the following observation.

► **Observation 12.** *The CYCLICCOVERS problem for a string S reduces in linear time to the computation of Q-COVERS(S).*

We will later show how the set Q-COVERS(S) can be computed based on recursive calls to Q-COVERS(W) for substrings W of S .

3.1 Quasi-Covers and Substring Complexity

The substring complexity of a length- m string W is a function that maps each length $k \in [1..m]$ to the number $|\text{SUB}_k(W)|$ of distinct length- k substrings of W . We further define $\beta_k(W) = |\text{SUB}_k(W)| + k - 1$. The term $k - 1$ is added because the sequence $(|\text{SUB}_k(W)|)_{k=1}^m$ does not need to be monotone in general; the resulting sequence $(\beta_k(W))_{k=1}^m$ is now non-decreasing and its monotonicity will be useful later. For a string family \mathcal{S} , let us denote by $\|\mathcal{S}\|$ the sum of lengths of strings in \mathcal{S} .

► **Observation 13.** *Let V be a quasi-cover of a substring W of S . If a substring W' of W satisfies $|W'| \geq 2|V| - 1$, then V is a quasi-cover of W' .*

► **Lemma 14.** *Given a length- m substring W and an integer $k \in [1..m]$, we can compute in $\mathcal{O}(m)$ time a family $\mathbf{G}_k(W)$ of substrings of W such that $\|\mathbf{G}_k(W)\| \leq \beta_k(W)$ and*

$$\text{Q-COVERS}_{\leq \lceil k/4 \rceil}(W) = \bigcap_{W' \in \mathbf{G}_k(W)} \text{Q-COVERS}_{\leq \lceil k/4 \rceil}(W').$$

Proof. It was shown in [21] that one can construct in linear time a string family, denoted in [21] as COMPR_t , such that $\|\text{COMPR}_t\| \leq \beta_{2t-1}(W)$ and the strings in COMPR_t contain all length- t substrings of W . First, we reformulate the corresponding fact from [21] taking $\mathbf{G}_k(W) = \text{COMPR}_{\lceil k/2 \rceil}$.

▷ **Claim 15** ([21, Lemma 5.4, proof of Lemma 5.3 and proof of Theorem 9 (“Computing S ”)]). Given $k \in [1..m]$, we can compute in $\mathcal{O}(m)$ time a string family $\mathbf{G}_k(W)$ such that

$$\|\mathbf{G}_k(W)\| \leq \beta_k(W) \quad \text{and} \quad \text{SUB}_{\lceil k/2 \rceil}(W) = \bigcup_{W' \in \mathbf{G}_k(W)} \text{SUB}_{\lceil k/2 \rceil}(W').$$

The next claim turns out to be similar to [21, Lemma 2.2].

▷ **Claim 16.** If $t \in [1..m]$, then

$$\text{Q-COVERS}_{\leq \lceil t/2 \rceil}(W) = \bigcap_{W' \in \text{SUB}_t(W)} \text{Q-COVERS}_{\leq \lceil t/2 \rceil}(W').$$

15:6 Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

Proof. We prove two inclusions separately.

(\subseteq) If V is a quasi-cover of W of length at most $\lceil t/2 \rceil$ and $W' \in \text{SUB}_t(W)$, then Observation 13 implies that V is a quasi-cover of W' .

(\supseteq) Assume that V is a quasi-cover of all $W' \in \text{SUB}_t(W)$ and $|V| = \ell \leq \lceil t/2 \rceil$. Consider a position $i \in [\ell - 1 .. m - \ell]$ and a substring $W' = W(i - \ell .. i + \ell)$. Note that V is a quasi-cover of W' (this follows from Observation 13 because W' is a substring of some length- t substring of W). Thus, there is a circular occurrence of V in W that covers the middle position of W' . Interpreted as a circular occurrence of V in W , it covers position i of W . \triangleleft

The thesis follows directly from the two claims above, taking $t = \lceil k/2 \rceil$ in the second claim. \blacktriangleleft

► **Lemma 17.** *If a string W has a quasi-cover V of length $|V| \geq 2k$, then*

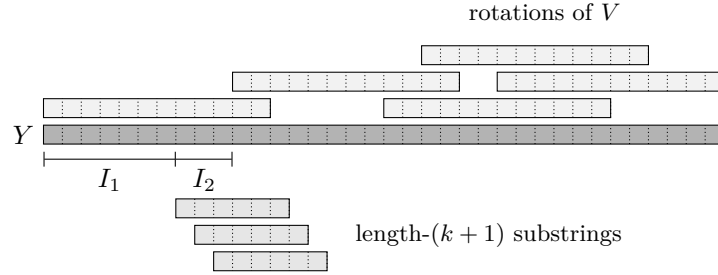
$$|\text{SUB}_{k+1}(W)| \leq \frac{1}{2}|W| + \frac{3}{2}|V|.$$

Proof. First, we show the following claim (cf. Figure 3).

► **Claim 18.** *If a string Y has a cyclic cover V of length $|V| \geq 2k$, then $|\text{SUB}_{k+1}(Y)| \leq \frac{1}{2}(|Y| + |V|)$.*

Proof. We denote by $\text{CSUB}_{k+1}(V)$ the set of distinct length- $(k+1)$ substrings of all rotations of V ; note that $|\text{CSUB}_{k+1}(V)| \leq |V|$.

For each $i \in \text{CircOcc}(V, Y)$, let us mark positions $j \in [i .. i + |V| - k]$; observe that if a position j is marked, then $Y[j .. j + k] \in \text{CSUB}_{k+1}(V)$.



■ **Figure 3** A string V is a cyclic cover of a string Y and $|V| \geq 2k$. Each length- $(k+1)$ substring starting in I_1 belongs to $\text{CSUB}_{k+1}(V)$, but length- $(k+1)$ substrings starting in I_2 do not need to belong to $\text{CSUB}_{k+1}(V)$. The interval I_1 of marked positions is of size at least k , whereas the interval $|I_2|$ of unmarked positions is of size at most k .

Let Y' be the prefix of Y of length $|Y| - |V|$. We partition Y' into inclusion-wise maximal intervals of marked positions and inclusion-wise maximal intervals of unmarked positions. Each interval I_2 of unmarked positions is preceded by an interval I_1 of marked positions, where $|I_1| \geq |V| - k \geq k \geq |I_2|$ (otherwise, V would not be a cyclic cover of Y).

Hence, at most half of positions of Y' are unmarked, which is $(|Y| - |V|)/2$. Each length- $(k+1)$ substring starting at marked position belongs to $\text{CSUB}_{k+1}(V)$. Hence, $|\text{SUB}_{k+1}(Y)| \leq (|Y| - |V|)/2 + |\text{CSUB}_{k+1}(V)| \leq (|Y| + |V|)/2$. \triangleleft

If V is a quasi-cover of W then V is a cyclic cover of $Y = W[i .. j]$ with $i < |V|$ and $j > m - |V|$. We have, due to inequality $\frac{1}{2}(|W| - |Y|) < |V|$,

$$|\text{SUB}_{k+1}(W)| \leq |\text{SUB}_{k+1}(Y)| + |W| - |Y| < |\text{SUB}_{k+1}(Y)| + \frac{1}{2}(|W| - |Y|) + |V|.$$

Now, Claim 18 implies $|\text{SUB}_{k+1}(Y)| \leq \frac{1}{2}(|Y| + |V|)$ and thus $|\text{SUB}_{k+1}(W)| \leq \frac{1}{2}|W| + \frac{3}{2}|V|$. \blacktriangleleft

► **Example 19.** Let $\{a_1, a_2, \dots, a_{2k-1}\}$, $\{b_1, b_2, \dots, b_{2k}\}$, $\{c_1, c_2, \dots, c_{2k-1}\}$ be disjoint sets, and

$$X = a_1 a_2 \dots a_{2k-1}, \quad V_1 = b_1 b_2 \dots b_k, \quad V_2 = b_{k+1} b_{k+2} \dots b_{2k}, \quad Y = c_1 c_2 \dots c_{2k-1}.$$

Then $V = V_1 V_2$ is a quasi-cover of $W = X V_1 V_2 V_2 V_1 Y$, with $|V| = 2k$ and $|W| = 8k - 2$. All length- $(k+1)$ substrings of W are different. Hence:

$$|\text{SUB}_{k+1}(W)| = |W| - k = \frac{1}{2}|W| + \frac{3}{2}|V| - o(|W|).$$

We use the following crucial property of quasi-covers.

► **Lemma 20 (Work-Reduction Lemma).** *For a length- m substring W and $k \in [0..m)$, if $\beta_{k+1}(W) > \frac{5}{6}m$, then*

$$\text{Q-COVERS}_{[2k.. \lfloor m/6 \rfloor]}(W) = \emptyset.$$

Proof. The proof is by contradiction. Suppose that $V \in \text{Q-COVERS}_{[2k.. \lfloor m/6 \rfloor]}(W)$ and V is a cyclic cover of $Y = W[i..j]$ with $i < |V|$ and $j > m - |V|$. Due to Lemma 17 and inequality $k \leq |V|/2$,

$$|\text{SUB}_{k+1}(W)| + k \leq \frac{1}{2}|W| + \frac{3}{2}|V| + \frac{1}{2}|V| = \frac{1}{2}|W| + 2|V| \leq \frac{1}{2}m + 2 \cdot \frac{m}{6} = \frac{5}{6}m.$$

This contradicts our assumption that $\beta_{k+1}(W) = |\text{SUB}_{k+1}(W)| + k > \frac{5}{6}m$. ◀

4 Solution to CyclicCovers Problem

Our algorithm is recursive; its non-recursive parts correspond to (simple) fast computation of length-limited cyclic covers. We say that an interval $I = [a..b]$ of positive integers is *balanced* if $b = \mathcal{O}(a)$.

► **Lemma 21.** *After $\mathcal{O}(n)$ -time preprocessing, for a balanced interval $I = [a..b]$ and a length- m substring W , the set $\text{Q-COVERS}_I(W)$ can be computed in $\mathcal{O}(m)$ time.*

Proof. We consider each length $\ell \in I$ separately. Let $C = S[0..\ell)$. We use two SIMPLE INTERNALCPM queries to check if C has a circular occurrence starting within the first ℓ positions of W and a circular occurrence ending within the last ℓ positions of W . If any of these two conditions does not hold, C is not a quasi-cover of W . Otherwise, we use Lemma 8 to check if C is a cyclic cover of the substring of W spanned by the first and the last circular occurrence of C in W that were discovered in the previous step. The total time complexity is $\mathcal{O}(\sum_{i \in I} \frac{m}{i}) = \mathcal{O}(\sum_{i \in I} \frac{m}{a}) = \mathcal{O}(m \cdot |I|/a) = \mathcal{O}(m \cdot b/a) = \mathcal{O}(m)$, after $\mathcal{O}(n)$ -time preprocessing in Theorem 6 and Lemma 8. ◀

Our solution is based on Lemmas 14, 20, and 21. We use a recursive approach that was initially developed for seeds computation; see [21].

► **Theorem 22.** *The CYCLICCOVERS problem can be solved in $\mathcal{O}(n)$ time.*

Proof. We run the recursive function `ComputeQuasiCovers` (Algorithm 1) initially for $W = S$.

Correctness. In the base case, where $\beta_1(W) > \frac{5}{6}m$, there are more than $\frac{5}{6}m$ different letters in W , and then Lemma 20 implies $\text{Q-COVERS}_{\leq \lfloor m/6 \rfloor}(W) = \emptyset$.

In the recursive step, we reduce the computation of quasi-covers to the ones with lengths in two balanced intervals, $J_1 = (\lceil k/4 \rceil .. 2k)$ and $J_2 = (\lfloor m/6 \rfloor .. m)$, and the ones (the set Q) with sufficiently small lengths (at most $\lceil k/4 \rceil$). By Lemmas 14 and 20, the algorithm returns precisely the set $\text{Q-COVERS}(W)$.

■ **Algorithm 1** ComputeQuasiCovers(W).

Input: A substring W of length m .
Output: The set Q-COVERS(W) of lengths of quasi-covers of W .
 Compute $\beta_\ell(W)$ for all $\ell \in [1..m]$
if $\beta_1(W) > \frac{5}{6}m$ **then** ▶ see Lemma 20
 return Q-COVERS $_{\lfloor m/6 \rfloor .. m}(W)$ ▶ $\lfloor m/6 \rfloor .. m$ is a balanced interval
 $k := \max \{ \ell \in [1..m] : \beta_\ell(W) \leq \frac{5}{6}m \}$
 Let $\mathbf{G}_k(W)$ be the set of fragments as in Lemma 14
foreach string $W' \in \mathbf{G}_k(W)$ **do**
 $Q_{W'} := \text{ComputeQuasiCovers}(W')$ ▶ Recursive call
 $Q := \bigcap_{W' \in \mathbf{G}_k(W)} Q_{W'} \cap [1.. \lceil k/4 \rceil]$ ▶ $Q = \text{Q-COVERS}_{\leq \lceil k/4 \rceil}(W)$ (Lemma 14)
 $J_1 := (\lceil \frac{k}{4} \rceil .. 2k), J_2 := (\lfloor \frac{m}{6} \rfloor .. m)$ ▶ J_1, J_2 are balanced intervals
 return $Q \cup \text{Q-COVERS}_{J_1}(W) \cup \text{Q-COVERS}_{J_2}(W)$ ▶ $\text{Q-COVERS}_{[2k.. \lfloor m/6 \rfloor]}(W) = \emptyset$

Complexity. To bound the running time, denote by $T(m)$ the maximum number of operations performed by the algorithm for a substring W of length m . The sequence $\beta_i(W)$ for a length- m substring W of S can be computed in $\mathcal{O}(m)$ time [21, Lemma 5.1]. Due to Lemmas 14 and 21,

$$T(m) = \mathcal{O}(m) + \sum_i T(m_i), \quad \text{where } \sum_i m_i \leq \frac{5}{6}m.$$

This recurrence yields $T(m) = \mathcal{O}(m)$.

Due to Observation 12, the CYCLICCOVERS problem can be reduced in linear time to the computation of all quasi-covers. Finally, Lemma 21 requires $\mathcal{O}(n)$ -time preprocessing of S . This completes the proof. ◀

5 Solution to CyclicRoots Problem

We denote by $\sigma_0(n) = \sum_{p|n} 1$ the number of divisors of n and by $\sigma_1(n) = \sum_{p|n} p$ the sum of divisors of n . We use the following known estimations: $\sigma_0(n) = 2^{\mathcal{O}(\log n / \log \log n)}$ [18, §18.1] and $\sigma_1(n) = \mathcal{O}(n \log \log n)$ [18, §22.9]. They directly imply the following fact.

▶ **Fact 23.**

- $\sigma_0(n) = o(\sqrt{n} / \log n)$ and $\log \sigma_0(n) = \mathcal{O}(\log n / \log \log n)$
- $\sigma_1(n) = \sum_{p|n} \frac{n}{p} = \mathcal{O}(n \log \log n)$

Using CYCEQ queries (Theorem 11), we derive the following subroutine:

▶ **Observation 24.** After linear-time preprocessing of a string S , we can test if $S[0..p]$ is a cyclic root of a substring W of S in $\mathcal{O}(|W|/p)$ time.

In particular, in [16] the CYCLICROOTS problem was solved in $\mathcal{O}(\sigma_1(n)) = \mathcal{O}(n \log \log n)$ time (cf. Fact 23) by using $\frac{n}{p}$ CYCEQ queries for each divisor p of n .

Let us now develop an $\mathcal{O}(n)$ -time solution. We reduce testing if $S[0..p]$ is a cyclic root of the whole text to testing if $S[0..p]$ is a cyclic root of each substring F in a suitably chosen family \mathcal{F} of substrings.

The intuition behind this improvement is as follows. It turns out that the asymptotic upper bound on $\sigma_1(n)$ significantly depends on a few largest divisors. In the $\mathcal{O}(n \log \log n)$ -time algorithm, this corresponds to the smallest lengths p of the candidate cyclic root. Hence, for small p , we will adopt a different approach.

The factorization of S into length- q substrings, for $q \mid n$, will be called the q -factorization.

► **Observation 25.** *If S has a cyclic root of length p , then its $(k \cdot p)$ -factorization \mathcal{F} contains at most p^k distinct substrings, consequently the number of distinct factors in \mathcal{F} is at most $\min\left(p^k, \frac{n}{k \cdot p}\right)$.*

Thus, if the number of different factors in the $(k \cdot p)$ -factorization is greater than p^k , then we know that S does not have a cyclic root of length p .

Otherwise, if k is small enough, the number of different substrings in the $(k \cdot p)$ -factorization will be smaller than $n/(k \cdot p)$, and we can check each of them using CYCEQ queries in $\mathcal{O}(k)$ time. On the other hand, if k is large enough, then the $\mathcal{O}(n/(k \cdot p))$ work spent on computing the factorization will be much less than $\mathcal{O}(n/p)$.

■ **Algorithm 2** `CyclicRoot(S, p)`: Does S have a cyclic root of length $p \mid n$?

```

 $k := \max(\lfloor \frac{1}{2} \log_p n \rfloor, 1)$ 
Let  $S = S_1 S_2$ , where  $|S_2| = n \bmod (k \cdot p)$ 
 $\mathcal{F} :=$  all distinct factors in the  $(k \cdot p)$ -factorization of  $S_1$     ►  $\mathcal{O}(n/(k \cdot p))$  time [15]
if  $|\mathcal{F}| > p^k$  then return NO    ► Observation 25
if  $|S_2| > 0$  then  $\mathcal{F} := \mathcal{F} \cup \{S_2\}$     ►  $|\mathcal{F}| = \mathcal{O}\left(\min\left(p^k, \frac{n}{k \cdot p}\right)\right)$ 
foreach  $F \in \mathcal{F}$  do    ►  $\mathcal{O}(k \cdot |\mathcal{F}|) = \mathcal{O}(\sqrt{n} \log n)$  time
    if  $S[0..p]$  is not a cyclic root of  $F$  then return NO    ► Observation 24
return YES
```

► **Theorem 26.** *The CYCLICROOTS problem can be solved in $\mathcal{O}(n)$ time.*

Proof. We use Algorithm 2. After $\mathcal{O}(n)$ -time preprocessing, all different substrings in \mathcal{F} can be found in $\mathcal{O}(n/(k \cdot p))$ time using deterministic substring hashing [15]. By Observation 24, after $\mathcal{O}(n)$ -time preprocessing, we can test in $\mathcal{O}(k)$ time for each $F \in \mathcal{F}$ if $S[0..p]$ is its cyclic root; this sums up to $\mathcal{O}(k \cdot \min(\frac{n}{k \cdot p}, p^k))$ time. For $k = \max(\lfloor \frac{1}{2} \log_p n \rfloor, 1)$, we have

$$\frac{n}{k \cdot p} = \mathcal{O}\left(\frac{n \log p}{p \log n}\right) \quad \text{and} \quad k \cdot \min\left(p^k, \frac{n}{k \cdot p}\right) = \mathcal{O}\left(p^{\frac{1}{2} \log_p n} \cdot \log_p n + \min\left(p, \frac{n}{p}\right)\right) = \mathcal{O}(\sqrt{n} \log n).$$

Thus, after $\mathcal{O}(n)$ -time preprocessing, all calls to the algorithm `CyclicRoot(S, p)` for all divisors p of n work in total time

$$\mathcal{O}(A(n) + B(n)), \quad \text{where} \quad A(n) = \sum_{p \mid n} \frac{n \log p}{p \log n} \quad \text{and} \quad B(n) = \sum_{p \mid n} \sqrt{n} \cdot \log n.$$

Estimating $B(n)$. By Fact 23, we have

$$B(n) = \sqrt{n} \log n \cdot \sum_{p \mid n} 1 = \sqrt{n} \log n \cdot \sigma_0(n) = o(n).$$

15:10 Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

Estimating $A(n)$. We partition the underlying sum into elements that do not exceed $\sigma_0(n)$ and the remaining elements. The former is bounded from above, due to Fact 23, as:

$$\sum_{\substack{p|n \\ p \leq \sigma_0(n)}} \frac{n \log p}{p \log n} \leq \sum_{p|n} \frac{n \log \sigma_0(n)}{p \log n} = \frac{\log \sigma_0(n)}{\log n} \sum_{p|n} \frac{n}{p} = \mathcal{O}\left(\frac{1}{\log \log n} \cdot \sigma_1(n)\right) = \mathcal{O}(n).$$

The latter sum is bounded from above by:

$$\sum_{\substack{p|n \\ p > \sigma_0(n)}} \frac{n \log p}{p \log n} \leq \sum_{p|n} \frac{n}{\sigma_0(n)} = \frac{n}{\sigma_0(n)} \sum_{p|n} 1 = \frac{n}{\sigma_0(n)} \cdot \sigma_0(n) = n.$$

This concludes the complexity analysis of the algorithm. ◀

6 InternalCPM via PILLAR Model

6.1 PILLAR Model

We use the so-called PILLAR model that was introduced in [9]. In this model, we assume that the following primitive queries can be performed efficiently, where the argument strings are represented as substrings of strings in a given collection \mathcal{X} :

- **Extract**(U, ℓ, r): Retrieve the substring $U[\ell..r]$.
- **LCP**(U, V), **LCP_R**(U, V): Find the length of the longest common prefix/suffix of U and V .
- **IPM**(U, V): Assuming that $|V| < 2|U|$, compute the starting positions of all exact occurrences of U in V , expressed as an arithmetic sequence. If the sequence has at least three terms, its difference equals $\text{per}(U)$.
- **Access**(U, i): Retrieve the letter $U[i]$.
- **Length**(U): Compute the length $|U|$ of the string U .

The runtime of algorithms in this model can be expressed in terms of the number of primitive PILLAR operations. The following result combines several known techniques to obtain constant-time implementations of all PILLAR operations in the standard setting. Efficient implementations of the PILLAR operations in other settings, including a dynamic and a compressed setting, are also known; cf. [9].

► **Theorem 27** ([9, Theorem 7.2]). *After an $\mathcal{O}(n)$ -time preprocessing of a collection of strings of total length n over an integer alphabet, each PILLAR operation can be performed in $\mathcal{O}(1)$ time.*

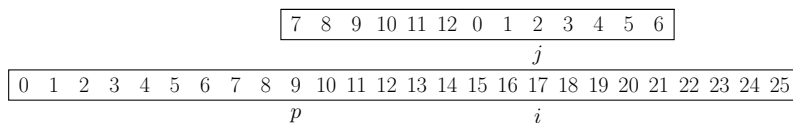
6.2 Interval Chains and PairMatch problem

For an integer set A and an integer r , let $A \oplus r = \{a + r : a \in A\}$. An *interval chain* is a set of the form $I \cup (I \oplus q) \cup (I \oplus 2q) \cup \dots \cup (I \oplus aq)$ for an interval I and non-negative integers a and q . In particular, a single interval is an interval chain (with $a = 0$).

First, we introduce an auxiliary operation **PAIRMATCH**. Denote by $\text{PAIRMATCH}(T, P, i, j)$ the set of all circular occurrences of P in T such that position i in T is aligned with position j in P (see also Figure 4):

$$\text{PAIRMATCH}(T, P, i, j) = \{p \in (i - m..i] : T[p..p+m) = \text{rot}_x(P), i - p = (j - x) \bmod m\}.$$

In particular $\text{PAIRMATCH}(T, P, i, 0)$ is the set of circular occurrences of P such that the leftmost position of P is aligned with position i in T .



■ **Figure 4** Let us put original numbers in positions of the pattern P ; they are moved after rotation of P . Assume that $p = 9$ is a position of a circular occurrence of P in T such that $p \in \text{Occ}(\text{rot}_7(P), T)$. Then, in particular, $p \in \text{PAIRMATCH}(T, P, 17, 2)$. In this case, $x = 7$ and $i - p = 8 = (j - x) \bmod 13$. We also have $p \in \text{PAIRMATCH}(T, P, 15, 0)$.

The following lemma is a consequence of [7, Lemma 10], where the PILLAR model was not used explicitly. A similar fact was shown in [16, Lemmas 5 and 6]. We include its proof for completeness.

► **Lemma 28.** *For any given i, j , the set $\text{PAIRMATCH}(T, P, i, j)$, represented as a union of at most two intervals, can be computed in $\mathcal{O}(1)$ time in the PILLAR model.*

Proof. First we explain how to compute $\text{PAIRMATCH}(T, P, i, 0)$. Let $p(i) = \text{LCP}(T[i..], P)$ and $s(i) = \text{LCP}_R(T[.i], P)$. If $p(i) + s(i) \geq m$, an interval $[i - s(i) .. i + p(i) - m]$ of starting positions of circular occurrences of P in T is reported; otherwise the answer is an empty set.

In general $\text{PAIRMATCH}(T, P, i, j)$ can be computed using (at most) two queries of the type $\text{PAIRMATCH}(T, P, i', 0)$, for $i' = i - j$ and $i' = i - j + m$. A respective query is asked only if $i' \in [0..n - m]$. The resulting intervals need to be intersected with $(i - m..i]$ to ensure that the circular occurrence contains position i . ◀

6.3 Internal CPM

The circular pattern matching problem is formally defined as follows.

CIRCULAR PATTERN MATCHING (CPM)
Input: A text T of length n and a pattern P of length m .
Output: All positions in T where circular occurrences of P start.

We will show an efficient solution in the PILLAR model of CPM in the case when the lengths of the pattern and of the texts are similar. The algorithm below applies the results of [7, 8]. These results considered the approximate CPM problem with $k \geq 1$ mismatches or edits. In the proof of the following theorem, we show that they can be adapted to the case of the exact CPM problem, obtaining an even simpler algorithm. The main idea of the algorithm is illustrated in Figure 7.

► **Theorem 29.** *If $n \leq 2m$, the answer to the CPM problem, represented as a union of $\mathcal{O}(1)$ interval chains, can be computed in $\mathcal{O}(1)$ time in the PILLAR model.*

Proof. Let $P = P_1P_2$, where $|P_1| = \lfloor m/2 \rfloor$. Each circular occurrence of P in T implies a standard occurrence of at least one of P_1 and P_2 in T . Henceforth, we assume that it implies an occurrence of P_1 ; the remaining case can be treated symmetrically.

Let $A = \text{Occ}(P_1, T)$. As $|T| \leq 4|P_1| + 3$, a representation of A consisting of $\mathcal{O}(1)$ arithmetic sequences can be computed using $\mathcal{O}(1)$ IPM queries by the so-called standard trick. We consider each of the arithmetic sequences B separately.

15:12 Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String

Nonperiodic case. If an arithmetic sequence B contains at most two occurrences, then we ask a query $\text{PAIRMATCH}(T, P, i, 0)$ for each $i \in B$. The resulting intervals contain positions of all circular occurrences of P in T that imply an occurrence of P_1 in T at a position $i \in B$, and possibly some other circular occurrences of P in T (that imply an occurrence of P_2).

Periodic case. Assume now that an arithmetic sequence B contains at least three elements. As already mentioned, its difference is $q := \text{per}(P_1)$.

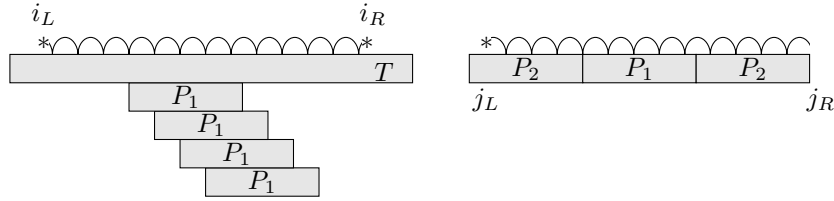
Let i be any element of B . We compute the largest index $i_L < i$ and the smallest index $i_R > i$ such that

$$T[i_L] \neq T[i_L + q] \text{ (or } i_L = -1), \quad T[i_R] \neq T[i_R - q] \text{ (or } i_R = |T|).$$

Let $Q = P_2P_1P_2$ and $j = |P_2|$. Similarly (see Figure 5), we compute the largest index $j_L < j$ and the smallest index $j_R > j$ such that

$$Q[j_L] \neq Q[j_L + q] \text{ (or } j_L = -1), \quad Q[j_R] \neq Q[j_R - q] \text{ (or } j_R = |Q|).$$

The indices i_L, i_R, j_L, j_R , which can be called *misperiods*, can be computed using a constant number of LCP and LCP_R queries on T and P .



■ **Figure 5** Misperiods i_L, i_R, j_L ; in this case, there is no misperiod j_R .

We consider two cases:

Case (1). The cyclic occurrence is an occurrence of a rotation of P that is a length- m substring of $Q(j_L \dots j_R)$; the occurrence is contained within a substring $T(i_L \dots i_R)$ in the text. Both strings in scope are periodic with period q ; it only matters if the periods are synchronized. Let

$$X = (j_L \dots j_R - m] \text{ and } Z = (i_L \dots i_R - m].$$

The set X consists of the positions in Q where a rotation of P contained in $Q(j_L \dots j_R)$ starts. The set $Z' := \{z \in Z : \exists x \in X \ z \equiv x \pmod{q}\}$ consists of the starting positions of circular occurrences of P contained in $T(i_L \dots i_R)$. By the following claim, the set Z' can be computed in $\mathcal{O}(1)$ time.

▷ **Claim 30** ([7, Lemma 7]). Let X and Z be intervals and q be a positive integer. The set $Z' := \{z \in Z : \exists x \in X \ z \equiv x \pmod{q}\}$, represented as a disjoint union of at most three interval chains, can be computed in $\mathcal{O}(1)$ time.

Case (2). In this case, two misperiods, one in T and one in P , need to be synchronized. It suffices to take the union of results of a $\text{PAIRMATCH}(T, P, i_L, |P_1| + j_L)$ query if neither of i_L, j_L equals -1 and of a $\text{PAIRMATCH}(T, P, i_R, j_R - |P_2|)$ query if $i_R \neq |T|$ and $j_R \neq |P|$.

Overall, the result is a union of $\mathcal{O}(1)$ intervals and interval chains and can be computed in $\mathcal{O}(1)$ time in the PILLAR model using Lemma 28 and Claim 30. ◀

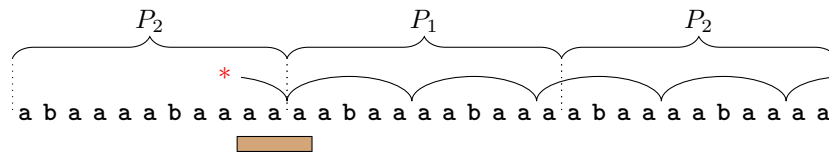


Figure 6 The interval X (shaded box) represents the starting positions of the rotations of $P = (\text{aabaa})^4\text{aa}$ contained in $Q(j_L..j_R) = Q(8..33)$. Five copies of X (two of them partial) constitute the output set Z' (the shaded boxes in Figure 7).

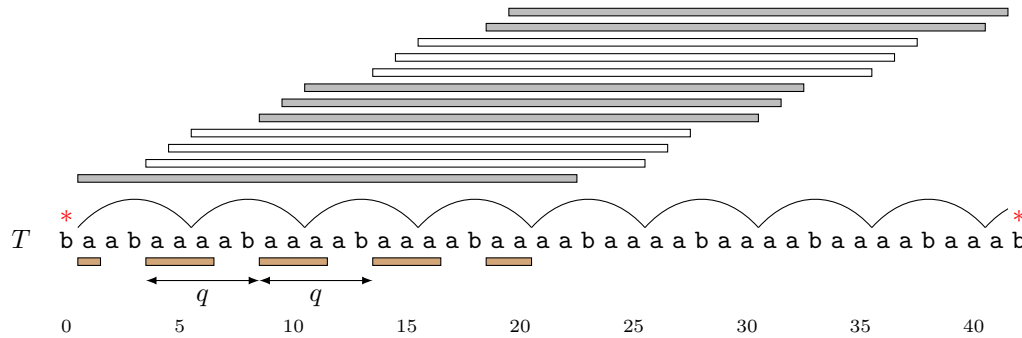


Figure 7 The starting positions of the circular occurrences of the pattern $P = (\text{aabaa})^4\text{aa}$ in the text T form two intervals $([1..1]$ and $[19..20])$ and one interval chain $I, I \oplus q, I \oplus 2q$, where $I = [4..6]$ and $q = 5$.

We introduce the following generalization of IPM queries.

INTERNAL CIRCULAR PATTERN MATCHING QUERIES (INTERNALCPM)
Input: A string S of length n .
Queries: Given two substrings P and T of S such that $|T| \leq 2|P|$, report all the starting positions of all circular occurrences of P in T .

Combining Theorems 27 and 29, we obtain the following result, which generalizes Theorem 6.

► **Theorem 31.** *The answer to an INTERNALCPM query, represented as a union of $\mathcal{O}(1)$ interval chains, can be computed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing.*

7 Final Remarks

We took a recursive approach proposed in the computation of seeds and adjusted it to the case of cyclic covers. Despite the similarity, several major changes were necessary due to circularity. We hope that such a recursive approach can be used in other problems on strings.

We also demonstrated the importance of a new tool in computations on cyclic strings: internal circular pattern matching queries. Hopefully, they could be used for other problems related to cyclic substrings.

References

- 1 Lorraine A. K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. doi:10.1016/j.patrec.2017.01.018.
- 2 Lorraine A. K. Ayad and Solon P. Pissis. MARS: improving multiple circular sequence alignment using refined sequences. *BMC Genomics*, 18(1):86, 2017. doi:10.1186/s12864-016-3477-5.

- 3 Carl Barton, Costas S. Iliopoulos, Ritu Kundu, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. Accurate and efficient methods to improve multiple circular sequence alignment. In Evripidis Bampis, editor, *Experimental Algorithms, SEA 2015*, volume 9125 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2015. doi:10.1007/978-3-319-20086-6_19.
- 4 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*, pages 536–545. IEEE, 2016. doi:10.1109/DCC.2016.79.
- 5 Bastien Cazaux and Eric 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.
- 6 Bastien Cazaux and Eric Rivals. The power of greedy algorithms for approximating Max-ATSP, Cyclic Cover, and superstrings. *Discrete Applied Mathematics*, 212:48–60, 2016. doi:10.1016/j.dam.2015.06.003.
- 7 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszłyński, Tomasz Waleń, and Wiktor Zuba. Circular pattern matching with k mismatches. *Journal of Computer and System Sciences*, 115:73–85, 2021. doi:10.1016/j.jcss.2020.07.003.
- 8 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022*, volume 244 of *LIPICs*, pages 35:1–35:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ESA.2022.35.
- 9 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.
- 10 Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-parallel algorithms for exact circular string matching. *The Computer Journal*, 57(5):731–743, March 2013. doi:10.1093/comjnl/bxt023.
- 11 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszłyński, Tomasz Waleń, and Wiktor Zuba. Shortest covers of all cyclic shifts of a string. *Theoretical Computer Science*, 866:70–81, 2021. doi:10.1016/j.tcs.2021.03.011.
- 12 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszłyński, Tomasz Waleń, and Wiktor Zuba. Linear-time computation of shortest covers of all rotations of a string. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022*, volume 223 of *LIPICs*, pages 22:1–22:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.22.
- 13 Maxime Crochemore, Thierry Lecroq, and Wojciech Rytter. *125 Problems in Text Algorithms: With Solutions*. Cambridge University Press, 2021.
- 14 Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009. doi:10.1016/j.jda.2008.09.001.
- 15 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium*, volume 6942 of *Lecture Notes in Computer Science*, pages 421–432. Springer, 2011. doi:10.1007/978-3-642-23719-5_36.
- 16 Roberto Grossi, Costas S. Iliopoulos, Jesper Jansson, Zara Lim, Wing-Kin Sung, and Wiktor Zuba. Finding the cyclic covers of a string. In Chun-Cheng Lin, Bertrand M. T. Lin, and Giuseppe Liotta, editors, *Algorithms and Computation - 17th International Conference and Workshops, WALCOM 2023*, volume 13973 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2023. doi:10.1007/978-3-031-27051-2_13.

- 17 Roberto Grossi, Costas S. Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11:12, 2016. doi:10.1186/s13015-016-0076-6.
- 18 Godfrey H. Hardy and Edward M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 1960.
- 19 Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and indexing circular patterns. In Mourad Elloumi, editor, *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.
- 20 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, October 2018. URL: <https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 21 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.
- 22 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *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.
- 23 M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, 2005. URL: http://www.cambridge.org/gb/knowledge/isbn/item1172552/?site_locale=en_GB.
- 24 Andrés Marzal, Ramón Mollineda, Guillermo Penis, and Enrique Vidal. Cyclic string matching: Efficient exact and approximate algorithms. In Dechang Chen and Xiuzhen Cheng, editors, *Pattern Recognition and String Matching*, pages 477–497. Springer US, Boston, MA, 2002. doi:10.1007/978-1-4613-0231-5_19.
- 25 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.
- 26 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.
- 27 Vicente Palazón-González and Andrés Marzal. On the dynamic time warping of cyclic sequences for shape retrieval. *Image Vision Computing*, 30(12):978–990, 2012. doi:10.1016/j.imavis.2012.08.012.
- 28 Vicente Palazón-González and Andrés Marzal. Speeding up the cyclic edit distance using LAESA with early abandon. *Pattern Recognition Letters*, 62:1–7, 2015. doi:10.1016/j.patrec.2015.04.013.
- 29 Vicente Palazón-González, Andrés Marzal, and Juan Miguel Vilar. On hidden Markov models and cyclic strings for shape recognition. *Pattern Recognition*, 47(7):2490–2504, 2014. doi:10.1016/j.patcog.2014.01.018.
- 30 Michael J. Tisza, Diana V. Pastrana, Nicole L. Welch, Brittany Stewart, Alberto Peretti, Gabriel J. Starrett, Yuk-Ying S. Pang, Siddharth R. Krishnamurthy, Patricia A. Pesavento, David H. McDermott, et al. Discovery of several thousand highly diverse circular DNA viruses. *eLife*, 9:e51971, 2020. doi:10.7554/eLife.51971.
- 31 Edward K. Wagner, Martinez J. Hewlett, David C. Bloom, and David Camerini. *Basic Virology*, volume 3. Blackwell Science Malden, MA, 1999.

Faster Prefix-Sorting Algorithms for Deterministic Finite Automata

Sung-Hwan Kim ✉ 

DAIS, Ca' Foscari University of Venice, Italy

Francisco Olivares ✉ 

CeBiB – Centre for Biotechnology and Bioengineering, Santiago, Chile
Department of Computer Science, University of Chile, Santiago, Chile

Nicola Prezza ✉ 

DAIS, Ca' Foscari University of Venice, Italy

Abstract

Sorting is a fundamental algorithmic pre-processing technique which often allows to represent data more compactly and, at the same time, speeds up search queries on it. In this paper, we focus on the well-studied problem of sorting and indexing string sets. Since the introduction of suffix trees in 1973, dozens of suffix sorting algorithms have been described in the literature. In 2017, these techniques were extended to sets of strings described by means of finite automata: the theory of Wheeler graphs [Gagie et al., TCS'17] introduced automata whose states can be *totally*-sorted according to the co-lexicographic (co-lex in the following) order of the prefixes of words accepted by the automaton. More recently, in [Cotumaccio, Prezza, SODA'21] it was shown how to extend these ideas to arbitrary automata by means of *partial* co-lex orders. This work showed that a co-lex order of minimum width (thus optimizing search query times) on deterministic finite automata (DFAs) can be computed in $O(m^2 + n^{5/2})$ time, m being the number of transitions and n the number of states of the input DFA.

In this paper, we exhibit new combinatorial properties of the minimum-width co-lex order of DFAs and exploit them to design faster prefix sorting algorithms. In particular, we describe two algorithms sorting arbitrary DFAs in $O(mn)$ and $O(n^2 \log n)$ time, respectively, and an algorithm sorting acyclic DFAs in $O(m \log n)$ time. Within these running times, all algorithms compute also a smallest chain partition of the partial order (required to index the DFA). We present an experiment result to show that an optimized implementation of the $O(n^2 \log n)$ -time algorithm exhibits a nearly-linear behaviour on large deterministic pan-genomic graphs and is thus also of practical interest.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases String Matching, Deterministic Finite Automata, Graph Indexing, Co-lexicographical Sorting

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.16

Supplementary Material *Software:* <https://github.com/regindex/DFA-suffix-doubling>

Funding *Sung-Hwan Kim:* Funded by the European Union (ERC, REGINDEX, 101039208).

Francisco Olivares: Funded by Ph.D Scholarship 21210579, ANID, Chile.

Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.



© Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 16; pp. 16:1–16:16



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In this paper, we study the problem of indexing string sets for *pattern matching* queries: pre-process a set $\mathcal{L} \subseteq \Sigma^*$ of strings from a finite alphabet Σ so that later we can efficiently answer queries of the form “is a given query pattern $P \in \Sigma^*$ substring of some string in \mathcal{L} ?”.

Clearly, an algorithmic solution to this problem requires the set \mathcal{L} to be representable in *finite* space (even though \mathcal{L} itself could contain an *infinite* number of strings); in this paper, we focus on string sets described by finite state automata, that is, on regular languages. Our results build on a successful line of previous research based on the following idea: after sorting all prefixes $\text{Pref}(\mathcal{L})$ of the strings in \mathcal{L} in colexicographic (co-lex for brevity) order¹, pattern matching queries translate to finding the strings in $\text{Pref}(\mathcal{L})$ that are suffixed by pattern string P . Being $\text{Pref}(\mathcal{L})$ co-lex sorted, those strings form a range in co-lex order; notice that, if the sorted $\text{Pref}(\mathcal{L})$ is explicitly stored, such a range can be easily found by binary search. Recall, however, that (due to limited available working space) we work with a particular representation of \mathcal{L} : a finite state automaton \mathcal{A} . This requires re-formulating the pattern matching problem on \mathcal{A} . It is easy to see that pattern matching queries on \mathcal{L} translate to finding paths of \mathcal{A} whose labels, when concatenated, form P . When using \mathcal{A} to index \mathcal{L} , the main question becomes therefore “how does the total co-lex order on $\text{Pref}(\mathcal{L})$ map onto the states of \mathcal{A} ?”. In particular cases, such a mapping yields a total order among \mathcal{A} ’s states. This happens, for example, when \mathcal{A} is a path (i.e. a string; corresponding data structures include the suffix tree [22], the suffix array [18, 13], and the FM-index [10]), a finite set of disjoint paths (eBWT [19]), or a labeled arborescence (XBWT [9]). A total order on the states of \mathcal{A} is obtained even in particular cases where \mathcal{A} may accept an infinite language: this is the case, for example, of de Bruijn graphs (BOSS [2]) and Wheeler graphs [11] (the latter generalize all the above classes of totally-sortable labeled graphs).

More recently, in [6, 5] it was shown that in the general case (arbitrary NFAs) the total co-lex order on $\text{Pref}(\mathcal{L})$ maps very naturally onto a family of *partial co-lex orders* among the states of \mathcal{A} . Such a family contains only one order for any given DFA, while NFAs may admit multiple admissible co-lex orders. Letting p be the *width* of a smallest-width partial order $<_{\mathcal{A}}$, it was shown that pattern matching queries on \mathcal{L} can be solved in time $\tilde{O}(p^2)$ per query character². Note that this generalizes the total order case $p = 1$, where indeed queries take $\tilde{O}(1)$ time using the aforementioned solutions (e.g. indexes on strings and labeled trees). Building the index of [6, 5] requires the computation of a smallest *chain partition* for the co-lex order $<_{\mathcal{A}}$, i.e. a minimum-size partition C_1, \dots, C_p of \mathcal{A} ’s states such that $(C_i, <_{\mathcal{A}})$ is a total order for each $i = 1, \dots, p$ (note that the index does not require the order $<_{\mathcal{A}}$ itself, just a chain partition). Letting n and m be the number of states and transitions of \mathcal{A} , respectively, [6] showed how to build such a chain partition in $O(n^{5/2} + m^2)$ time in the case where \mathcal{A} is a deterministic finite automaton (DFA). The work [5] presented a solution running in $\tilde{O}(m^2)$ time w.h.p. In the general nondeterministic (NFA) case, the problem is known to be NP-complete³, even though polynomial algorithms do exist for co-lex *pre-orders* [4] (which still allow indexing and whose width is never larger than that of co-lex orders).

¹ Historically, the lexicographic order of suffixes was used first; however, with finite state automata the symmetric co-lex order of $\text{Pref}(\mathcal{L})$ turns out to be more natural.

² The notation \tilde{O} hides factors polylogarithmic in the size of \mathcal{A} .

³ Hardness follows from hardness of the $p = 1$ case [12], while membership in NP follows from the fact that the properties defining a co-lex order can be checked in polynomial time, given a candidate order.

1.1 Our results

In this work, we focus on the problem of computing the smallest-width partial co-lex order $<_{\mathcal{A}}$ when the input is a DFA. On DFAs, $<_{\mathcal{A}}$ has a very intuitive definition: letting u, v be states of \mathcal{A} , we have $u <_{\mathcal{A}} v$ if and only if $\alpha < \beta$ for every $\alpha \in I_u$ and $\beta \in I_v$, where $<$ denotes the co-lex order among strings and I_u denotes the set of strings (in fact, a regular language) labeling all paths from the source of \mathcal{A} to u . We first observe that $<_{\mathcal{A}}$ is completely specified by pairs $(\inf I_u, \sup I_u)$ over the co-lex sorted $\text{Pref}(\mathcal{L})$: in fact, we prove that $u <_{\mathcal{A}} v$ holds if and only if $\sup I_u \leq \inf I_v$. This allows finding a smallest chain decomposition of $<_{\mathcal{A}}$ in $O(n)$ time through a solution of the *interval partitioning problem*, given that the co-lex ranks of strings $\inf I_u$ and $\sup I_u$ are known for each state u . Observing that these strings can be easily encoded with two pruned versions of the DFA \mathcal{A} , this leaves the problem of computing and sorting them – ideally, in $\tilde{O}(m)$ time. We give three different solutions for this problem, which could be of independent interest. The first two solutions work on arbitrary DFAs and run in time $O(mn)$ and $O(n^2 \log n)$, respectively. The latter of these two solutions is based on *suffix doubling*, the technique at the core of the first suffix array construction algorithm [18], and is close to optimal on dense graphs. We show that an optimized implementation of this algorithm exhibits a sub-quadratic behaviour on large deterministic pan-genomic graphs (in fact, we experimentally observe a linearithmic running time). The third solution works on acyclic DFAs, runs in $O(m \log n)$ time, and generalizes a well-known algorithm for building the Burrows-Wheeler transform in an online fashion; in our case, we process the automaton's states in any topological order and, for each processed state u , compute $\inf I_u$ and $\sup I_u$ using the results computed on the already-processed states.

2 Preliminaries

Notation $[i, j]$, where $i, j \in \mathbb{N}$, denotes the integer set $\{i, i+1, \dots, j\}$ (if $i > j$, then $[i, j] = \emptyset$). Let Σ be a finite alphabet. A *finite string* $\alpha \in \Sigma^*$ (or *string of finite length*) is a finite concatenation of characters from Σ . The notation $|\alpha|$ indicates the length of the string α . The symbol ϵ denotes the empty string. The notation $\alpha[i]$ denotes the i -th character from the beginning of α ; indices start from 1, so $\alpha[1]$ is the first character of α . Letting $\alpha, \beta \in \Sigma^*$, $\alpha \cdot \beta$ (or simply $\alpha\beta$) denotes the concatenation of strings. The notation $\alpha[i..j]$ denotes $\alpha[i] \cdot \alpha[i+1] \cdot \dots \cdot \alpha[j]$; if $i > j$, then $\alpha[i..j]$ is the empty string ϵ . The notation $\alpha \sqsubseteq \beta$, where $\alpha, \beta \in \Sigma^*$, indicates that α is a prefix of β , i.e. $\alpha = \beta[1..i]$ for some $i \leq |\beta|$. An ω -string $\beta \in \Sigma^\omega$ (or *infinite string / string of infinite length*) is an infinite numerable concatenation of characters from Σ . In this paper, we work with *left-infinite* ω -strings, meaning that $\beta \in \Sigma^\omega$ is constructed from the empty string ϵ by prepending an infinite number of characters to it. In particular, the operation of appending a character $a \in \Sigma$ at the end of a ω -string $\alpha \in \Sigma^\omega$ is well-defined and yields the ω -string αa . The notation α^ω , where $\alpha \in \Sigma^*$, denotes the concatenation of an infinite (numerable) number of copies of string α .

► **Definition 1.** A *Deterministic Finite-State Automaton (DFA)* is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ where Q is the finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $s \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states.

As is customary, we extend the transition function to words $\alpha \in \Sigma^*$ as follows: for $a \in \Sigma$, $\alpha \in \Sigma^*$, and $q \in Q$: $\delta(q, a \cdot \alpha) = \delta(\delta(q, a), \alpha)$ and $\delta(q, \epsilon) = q$. By $\delta^{-1}(u)$, we denote the set of states from which there exists a transition to u : i.e. $\delta^{-1}(u) = \{v \in Q : (\exists a \in \Sigma)(\delta(v, a) = u)\}$.

In the rest of the paper, $n = |Q|$ denotes the number of states and $m = |\delta| = |\{(u, v, a) \in Q \times Q \times \Sigma : \delta(u, a) = v\}|$ the number of transitions of the DFA under consideration.

Following [1], we use the following notation for the set of words reaching a given state:

► **Definition 2.** Let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA. If $q \in Q$, let I_q be the set of words reaching q from the initial state:

$$I_q = \{\alpha \in \Sigma^* : q = \delta(s, \alpha)\};$$

I_q is also called the regular language recognized by q .

The language $\mathcal{L}(\mathcal{A})$ recognized by \mathcal{A} is defined as $\mathcal{L}(\mathcal{A}) = \cup_{q \in F} I_q$.

The co-lexicographic (or co-lex) order of two strings $\alpha, \beta \in \Sigma^* \cup \Sigma^\omega$ is defined as follows. (i) $\epsilon < \alpha$ for every $\alpha \in \Sigma^+ \cup \Sigma^\omega$, and (ii) if $\alpha = \alpha'a$ and $\beta = \beta'b$ (with $a, b \in \Sigma$ and $\alpha', \beta' \in \Sigma^* \cup \Sigma^\omega$), $\alpha < \beta$ holds if and only if $(a < b) \vee (a = b \wedge \alpha' < \beta')$. In this paper, the symbols $<$ and \leq will be used to denote the total order between the alphabet's characters, the co-lexicographic order between strings/ ω -strings, and the co-lex partial order among the states of an automaton (Definition 3). The meaning of symbols $<$ and \leq will always be clear from the context. In all cases, the symbol \leq has the following meaning: $x \leq y$ if and only if $x < y$ or $x = y$ (i.e. $x < y$ or x and y are the same state, the same character, or the same string, depending on the context).

Let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA. We assume that s has no incoming edges; any automaton can always be transformed into an equivalent automaton with this property. We also assume that every state is reachable from the source: for every $v \in Q$, there exists $\alpha \in \Sigma^*$ such that $\delta(s, \alpha) = v$. Moreover, we assume *input consistency*: for every $u, v, v' \in Q$ and $c, c' \in \Sigma$, if $\delta(v, c) = \delta(v', c') = u$, then $c = c'$. We denote with $\lambda(v)$ such a uniquely-defined character and take $\lambda(s) = \#$ for the source s , where $\# \notin \Sigma$ is such that $\# < c$ for every $c \in \Sigma$. Note that input consistency is equivalent to working with state-labeled automata. Also this assumption is not too restrictive, since any automaton can be converted into an equivalent input-consistent automaton by just multiplying its size by a factor of $|\Sigma|$.

The following concepts can be defined more in general for NFAs (see [6]), but for the purposes of this article it will be sufficient to introduce them just on DFAs:

► **Definition 3.** Let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA. A co-lex order on \mathcal{A} is a partial order \leq on Q that satisfies the following two axioms:

1. (Axiom 1) For every $u, v \in Q$, if $u < v$, then $\lambda(u) \leq \lambda(v)$;
2. (Axiom 2) For every $a \in \Sigma$ and $u, v, u', v' \in Q$, if $u = \delta(u', a)$, $v = \delta(v', a)$ and $u < v$, then $u' \leq v'$.

The *width* of a partial order is the size of its largest antichain or, equivalently by Dilworth's theorem [7], the size of a smallest chain partition of the order.

► **Definition 4.** The co-lex width of a DFA \mathcal{A} is the minimum width of a co-lex order on \mathcal{A} :

$$\text{width}(\mathcal{A}) = \min\{\text{width}(\leq) : \leq \text{ is a co-lex order on } \mathcal{A}\}$$

On DFAs, the following co-lex order is of particular interest:

► **Definition 5.** Let \mathcal{A} be a DFA. The relation $<_{\mathcal{A}}$ over Q is defined by:

$$u <_{\mathcal{A}} v \text{ if and only if } (\forall \alpha \in I_u)(\forall \beta \in I_v) (\alpha < \beta).$$

In fact, by [5, Lem. 1] the following holds:

► **Lemma 6.** If \mathcal{A} is a DFA, then $<_{\mathcal{A}}$ is a co-lex order on \mathcal{A} and $\text{width}(<_{\mathcal{A}}) = \text{width}(\mathcal{A})$. The order $<_{\mathcal{A}}$ is called the maximum co-lex order on \mathcal{A} .

Computing the smallest-width co-lex order is of interest because, as shown in [5, 6], there exists a linear-space index over any DFA \mathcal{A} answering *subpath queries* (find all the states of \mathcal{A} reached by a path labeled with a given query string P) in time proportional to $\text{width}(\mathcal{A})^2$ time per query character. In fact, the index is even compressed and uses $\log(\text{width}(\mathcal{A})) + \log|\Sigma| + O(1)$ bits per transition of \mathcal{A} . Building such an index requires computing a smallest-size chain partition of $<_{\mathcal{A}}$. State-of-the art algorithms for this problem run in time $O(m^2 + n^{5/2})$ [6] and $\tilde{O}(m^2)$ w.h.p. [5]. The goal of our paper is to improve these bounds by exploiting a new characterization for $<_{\mathcal{A}}$, introduced in the next section.

3 A new characterization of the maximum co-lex order of a DFA

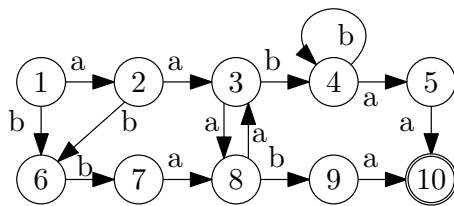
In this section, we give a new interval-based characterization of the maximum co-lex order $<_{\mathcal{A}}$ of a DFA. We show that this yields an $O(n)$ -space representation for $<_{\mathcal{A}}$ (observe that, in general, a partial order requires $O(n^2)$ space to be represented) and that, given this representation, one can compute a smallest chain partition of $<_{\mathcal{A}}$ in linear $O(n)$ time.

3.1 Infimum and supremum strings

Let u be a state of a DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$. For the set I_u of strings recognized by $u \in Q$, consider a (possibly infinite) string β such that β is a lower bound of I_u ; i.e. $\beta \leq \alpha$ for every $\alpha \in I_u$. Consider the co-lex largest string γ among such lower bounds of I_u . We call such a string the *infimum string* of u , and denote it by $\text{inf } I_u$. Similarly, we define the supremum string $\text{sup } I_u$ of u as the least upper bound of I_u ; see Figure 1 for an example.

► **Definition 7** (Infimum and supremum strings). *Let $u \in Q$ be a state of a DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$. The infimum string $\text{inf } I_u$ and the supremum string $\text{sup } I_u$ are defined as:*

$$\begin{aligned} \text{inf } I_u &= \gamma \in \Sigma^* \cup \Sigma^\omega \text{ s.t. } (\forall \beta \in \Sigma^* \cup \Sigma^\omega \text{ s.t. } (\forall \alpha \in I_u \beta \leq \alpha) \beta \leq \gamma) \\ \text{sup } I_u &= \gamma \in \Sigma^* \cup \Sigma^\omega \text{ s.t. } (\forall \beta \in \Sigma^* \cup \Sigma^\omega \text{ s.t. } (\forall \alpha \in I_u \alpha \leq \beta) \gamma \leq \beta) \end{aligned}$$



i	$\text{inf } I_{v_i}$	$\text{sup } I_{v_i}$	i	$\text{inf } I_{v_i}$	$\text{sup } I_{v_i}$
1	ϵ	ϵ	6	b	ab
2	a	a	7	bb	abb
3	aa	abbaa	8	aaa	abba
4	aab	b^ω	9	aaab	abbab
5	aaba	$b^\omega a$	10	aabaa	abbaba

■ **Figure 1** Example DFA with its infimum/supremum strings.

As a warm up, we make several observations on I_u , infimum, and supremum strings.

► **Observation 8.** *Let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA. For any $u \in Q$, the following hold:*

1. For every $\alpha \in I_u$, α is finite.
2. For any $v (\neq u) \in Q$, $I_u \cap I_v$ is the empty set.
3. For any finite suffix $\alpha \in \Sigma^*$ of $\text{inf } I_u$ (or $\text{sup } I_u$), there exists $\beta \in \Sigma^*$ such that $\beta\alpha \in I_u$.
4. $\text{inf } I_u \in I_u$ if and only if $\text{inf } I_u$ is a finite string; similar for $\text{sup } I_u$.
5. I_u is a singleton if and only if $\text{inf } I_u = \text{sup } I_u$. In such a case, $I_u = \{\text{inf } I_u (= \text{sup } I_u)\}$ and $\text{inf } I_u = \text{sup } I_u \in I_u$ is a finite string.
6. For $v (\neq u) \in Q$, if $\text{inf } I_u = \text{inf } I_v$ or $\text{inf } I_u = \text{sup } I_v$ then $\text{inf } I_u$ has infinite length; similar for $\text{sup } I_u$.

Proof.

1. By definition of I_u .
2. By definition of DFA, for any string α , there exists only one state u such that $\delta(s, \alpha) = u$.
3. Let α be any finite suffix of $\inf I_u$; the $\sup I_u$ case is analogous. We claim that there must exist $v \in Q$ such that $\delta(v, \alpha) = u$. This will prove our main claim, since for any string $\beta \in I_u$, $\delta(s, \beta \cdot \alpha) = \delta(\delta(s, \beta), \alpha) = \delta(v, \alpha) = u$ by definition, so $\beta \cdot \alpha \in I_u$.
 To prove the claim, assume by contradiction that there is no such $v \in Q$. Let $\alpha' \in \Sigma^*$ be the longest suffix of α such that there exists $v' \in Q$ such that $\delta(v', \alpha') = u$. Let $\alpha'' \in \Sigma^* \cup \Sigma^\omega$ and $a \in \Sigma$ be a string and a symbol such that $\inf I_u = \alpha'' \cdot a \cdot \alpha'$. Let $b \in \Sigma$ be the smallest alphabet symbol that is greater than a . Note that such b must exist; if not, every v' must have an incoming transition labeled by a symbol $a' (\in \Sigma) \leq a$; since $a' \neq a$ (otherwise α' would be longer), there exists a string in I_u suffixed by $a' \alpha'$ which is co-lex smaller than $\inf I_u$, which causes a contradiction. Then, for every such v' and every $v'' \in Q$ and $c \in \Sigma$ such that $\delta(v'', c) = v'$, we have $a < b \leq c$. Note that $\inf I_u < b \cdot \alpha' \leq \gamma$ for every $\gamma \in I_u$, so $\inf I_u$ is not the greatest lower bound of I_u . This is a contradiction with the definition of $\inf I_u$.
4. (\Rightarrow) By definition of I_u .
 (\Leftarrow) Let $\inf I_u$ be finite. Let us assume, by contradiction, that $\inf I_u \notin I_u$. Let $\alpha = a \cdot \inf I_u$ where $a \in \Sigma$ is the smallest symbol of the alphabet. We claim that $(\inf I_u <) \alpha \leq \beta$ for every $\beta \in I_u$, which contradicts the definition of $\inf I_u$. Consider a $\beta \in I_u$. Let k be the length of the longest common suffix of β and $\inf I_u$. If $k < |\inf I_u|$, then obviously $\alpha < \beta$ because prepending a symbol to $\inf I_u$ does not affect the relative co-lex order of $\inf I_u$ and β . If $k = |\inf I_u|$, then $\inf I_u$ is a suffix of β and $|\inf I_u| + 1 \leq |\beta|$ because $\inf I_u \notin I_u$. Therefore after prepending the smallest symbol a to $\inf I_u$, we still have $a \cdot \inf I_u \leq \beta$.
 To prove the other case for $\sup I_u$, let $\sup I_u$ be finite and let us assume for a contradiction that $\sup I_u \notin I_u$. From (3), there exists $\beta \in \Sigma^*$ such that $\beta \alpha \in I_u$ where $\alpha = \sup I_u$; note that a string is a suffix of itself. Because $\delta(s, \alpha) \neq u$, it holds $\beta \neq \epsilon$. However, then we have $\sup I_u = \alpha < \beta \alpha \in I_u$, which contradicts with the definition of $\sup I_u$ being an upper bound of I_u .
5. (\Rightarrow) if $I_u = \{\alpha\}$, then clearly $\inf I_u = \alpha$ and $\sup I_u = \alpha$, so $\inf I_u = \sup I_u$.
 (\Leftarrow) If $\inf I_u = \sup I_u$, then they are the same *finite* string. To see this, assume by contradiction that $\inf I_u = \sup I_u$ have infinite length. Then, for every $\alpha \in I_u$, $\inf I_u < \alpha < \sup I_u$. Since $\inf I_u = \sup I_u$, no such α can exist thus $I_u = \emptyset$. This is a contradiction, because it must hold $I_u \neq \emptyset$ by the assumption that there always exists $\alpha \in \Sigma^*$ such that $\delta(s, \alpha) = u$ (and, for the source s , $I_s = \{\epsilon\}$). Since $\inf I_u (= \sup I_u)$ is a finite string, $\inf I_u \in I_u$ by (4). In addition, $\inf I_u$ is the unique string in I_u because for every $\alpha \in I_u$, $\inf I_u \leq \alpha \leq \sup I_u$ and $\inf I_u = \sup I_u$, therefore $\inf I_u = \alpha = \sup I_u$.
6. Immediate from Observations (2) and (4). ◀

To conclude the section, we prove a lemma showing that infimum and supremum strings can always be expressed as a (possibly, infinite) concatenation of a constant number of distinct strings whose length does not exceed the number of states. This lemma will be useful later to bound the sorting depth of our algorithms computing $<_{\mathcal{A}}$.

► **Lemma 9.** *For a DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ and a state $u \in Q$, let $\gamma \in \{\inf I_u, \sup I_u\}$ be either the infimum or the supremum string of I_u . Then,*

1. *If γ is finite, then $|\gamma| < |Q|$.*
2. *If γ has infinite length, then $\gamma = \beta^\omega \alpha$ for some $\alpha, \beta \in \Sigma^*$ such that $|\alpha| + |\beta| < |Q|$.*

Proof. Suppose $|\gamma| \geq |Q|$. Let $\gamma = \gamma'\gamma''$, where γ'' is the length- $|Q|$ suffix of γ . Consider a sequence of states $v_1, v_2, \dots, v_{|Q|+1}$ such that $v_{|Q|+1} = u$ and $\delta(v_k, \gamma''[k]) = v_{k+1}$ for $1 \leq k \leq |Q|$. Note that $v_i \neq s$ for every $2 \leq i \leq |Q| + 1$ because the start state s does not have an incoming transition. Then, there are at most $|Q| - 1$ distinct states among the $|Q|$ states $v_2, \dots, v_{|Q|+1}$ so by the pigeonhole principle there must be $2 \leq i < j \leq |Q| + 1$ such that $v_i = v_j$. Let $\beta' = \gamma'\gamma''[1..i-1]$, $\beta = \gamma''[i..j-1]$, and $\alpha = \gamma''[j..|Q|]$. Note that $\beta\alpha$ is a proper suffix of γ'' (*proper* because $i \geq 2$), therefore $|\alpha| + |\beta| = |\beta\alpha| < |\gamma''| = |Q|$. Note also that (by definition of β' , β , and α) $\gamma = \beta'\beta\alpha$.

Let us assume $\gamma = \inf I_u$; the other case $\gamma = \sup I_u$ is analogous. Note that $\gamma = \beta'\beta\alpha \leq \beta'\beta^k\alpha$ for every $k \geq 0$. To see this, observe that if γ is a finite string, then $\gamma = \beta'\beta\alpha \in I_u$ by Observation 8.4. Since $\delta(s, \beta') = v_i$, $\delta(v_i, \beta) = v_j = v_i$, and $\delta(v_j (= v_i), \alpha) = v_{|Q|+1} = u$, we have $\beta'\beta^k\alpha \in I_u$ for every $k \geq 0$. By definition of $\inf I_u$, $\gamma = \inf I_u \leq \beta'\beta^k\alpha$ for every $k \geq 0$. On the other hand, if γ has infinite length, assume for a contradiction that there exists $k' \geq 0$ such that $\beta'\beta^{k'}\alpha < \beta'\beta\alpha = \gamma$. Consider the length- $(l+1)$ suffix $\beta''\alpha$ of $\beta'\beta^{k'}\alpha$ where l is the length of the longest common suffix between $\beta'\beta^{k'}\alpha$ and γ ; clearly, such a l is finite and $l \geq |\alpha|$ since α suffixes both strings. Then by Observation 8.3, there exists $\beta''' \in \Sigma^*$ such that $\beta'''\beta''\alpha \in I_u$. However we have $\beta'''\beta''\alpha < \beta'\beta\alpha = \gamma = \inf I_u$, which contradicts the definition of $\inf I_u$.

By plugging $k = 0$ into the inequality $\beta'\beta\alpha \leq \beta'\beta^k\alpha$ above, we obtain $\beta'\beta\alpha \leq \beta'\alpha$. Equivalently (by removing the common suffix α) it holds $\beta'\beta \leq \beta'$; but then, we can plug again a common suffix $\beta^k\alpha$ for any $k \geq 0$ and obtain that $\beta'\beta^{k+1}\alpha \leq \beta'\beta^k\alpha$ for any $k \geq 0$. In particular, this implies that $\beta'\beta^k\alpha \leq \beta'\beta\alpha = \gamma$ for any $k \geq 1$.

Since in the previous two paragraphs we proved that $\gamma \leq \beta'\beta^k\alpha$ and $\beta'\beta^k\alpha \leq \gamma$ for any $k \geq 1$, we conclude that $\gamma = \beta'\beta^k\alpha$ for any $k \geq 1$, i.e. γ must be an ω -string of the form $\gamma = \beta^\omega\alpha$. This proves claim (2). Claim (1) also follows since the assumption that γ is finite and $|\gamma| \geq |Q|$ leads to $\gamma = \beta^\omega\alpha$ (a contradiction to the finiteness of γ), hence its negation (i.e. claim 1) must hold. \blacktriangleleft

3.2 $O(n)$ -space representation of $<_{\mathcal{A}}$

Let $\mathcal{K}(\mathcal{A}) = \{\inf I_u : u \in Q\} \cup \{\sup I_u : u \in Q\} \subseteq \Sigma^* \cup \Sigma^\omega$ be the set of all infimum and supremum strings of \mathcal{A} . Let $\text{rank}(\alpha)$, for $\alpha \in \mathcal{K}(\mathcal{A})$, denote the position of α in the total order $(\mathcal{K}(\mathcal{A}), <)$ (e.g. $\text{rank}(\alpha) = 1$ for the co-lex smallest string $\alpha \in \mathcal{K}(\mathcal{A})$, and so on).

Our new representation of $<_{\mathcal{A}}$ is the set of n integer pairs $\{(\text{rank}(\inf I_u), \text{rank}(\sup I_u)) : u \in Q\} \subseteq [1, 2n] \times [1, 2n]$ (note that $|\mathcal{K}(\mathcal{A})| \leq 2n$). With the next theorem, we show that this set is indeed sufficient to reconstruct $<_{\mathcal{A}}$.

► Theorem 10. *Let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA. Then, for any $u, v (\neq u) \in Q$, $u <_{\mathcal{A}} v$ if and only if $\sup I_u \leq \inf I_v$.*

Proof. (\Rightarrow) To prove $u <_{\mathcal{A}} v \Rightarrow \sup I_u \leq \inf I_v$ for all $u, v \in Q$, assume by contradiction that there exist $u, v \in Q$ such that $u <_{\mathcal{A}} v$ and $\inf I_v < \sup I_u$. We claim that, in this case, there must exist $\alpha \in I_u, \beta \in I_v$ such that $\beta < \alpha$. By Definition 5, this contradicts $u <_{\mathcal{A}} v$. First, note that there must exist $\alpha \in I_u$ such that $\inf I_v < \alpha$, otherwise it would be $\sup I_u \leq \inf I_v$. We divide the proof by contradiction in the two cases (i) $\inf I_v$ is a finite string and (ii) $\inf I_v$ has infinite length.

- (i) If $\inf I_v$ is finite, then $\inf I_v \in I_v$ by Observation 8.4. Choosing $\beta = \inf I_v$, we have $\beta = \inf I_v (\in I_v) < \alpha$. This contradicts $u <_{\mathcal{A}} v$.
- (ii) If $\inf I_v$ has infinite length, then by Lemma 9 we can write it as $\inf I_v = \gamma_2^\omega \gamma_1$ for some strings $\gamma_1, \gamma_2 \in \Sigma^*$. Note that, for every $k \geq 0$, there exists a string $\gamma_3 \in \Sigma^*$

such that $\gamma_3\gamma_2^k\gamma_1 \in I_v$ (by Observation 8.3 because $\gamma_2^k\gamma_1$ is a suffix of $\inf I_u$). Choose any integer k' such that $|\gamma_2^{k'}\gamma_1| > |\alpha|$ (such an integer exists since α is finite). Since $\inf I_v = \gamma_2^{\omega}\gamma_1 < \alpha$, we also have $\gamma_3\gamma_2^{k'}\gamma_1 (= \beta \in I_v) < \alpha$. Again, this contradicts $u <_{\mathcal{A}} v$.

(\Leftarrow) Let $\sup I_u \leq \inf I_v$, and choose any $\alpha \in I_u$ and $\beta \in I_v$. We need to prove that $\alpha <_{\mathcal{A}} \beta$. By definition of $\sup I_u$ and $\inf I_v$, we have $\alpha \leq \sup I_u \leq \inf I_v \leq \beta$. If $\sup I_u < \inf I_v$, then $\alpha < \beta$. If, on the other hand, $\sup I_u = \inf I_v$ then both $\sup I_u$ and $\inf I_v$ must be infinite strings by Observation 8.6. Since α and β are both finite, it must be the case that $\alpha \neq \sup I_u$ and $\beta \neq \inf I_v$, therefore $\alpha < \sup I_u = \inf I_v < \beta$. Since this holds for any $\alpha \in I_u$ and $\beta \in I_v$, by definition of $<_{\mathcal{A}}$ it holds $\alpha <_{\mathcal{A}} \beta$. \blacktriangleleft

Equivalently, Theorem 10 shows that $<_{\mathcal{A}}$ can be interpreted as a set of intervals on the co-lex sorted $\text{Pref}(\mathcal{L}(\mathcal{A}))$. This characterization of $<_{\mathcal{A}}$ will allow us to compute this order faster than the state-of-the-art by (i) co-lex sorting the infimum and supremum strings (Section 4), and (ii) computing a smallest chain partition for $<_{\mathcal{A}}$ in linear time (Section 3.3).

3.3 Linear-time chain partitioning algorithm

In general, a partial order over n elements requires $O(n^2)$ space to be represented. Moreover, the fastest general-purpose algorithms for computing the smallest chain partition of a partial order run either in worst-case time $O(n^{5/2})$ (see, for example, [6, Lem. 6.1]) or in $\tilde{O}(n^2)$ time w.h.p. [16]. In this section we show that given the $O(n)$ -space representation $S = \{\text{rank}(\inf I_u), \text{rank}(\sup I_u) : u \in Q\}$ of $<_{\mathcal{A}}$, from which the order can be represented using intervals, we can compute a smallest chain partition of this order in optimal $O(n)$ time. It is known that the optimal solution of a smallest chain partition of interval orders can be computed with a greedy method (see [14, Sec. 6.8]). Moreover, given the sorted intervals, one can compute it in linear time [3]; for completeness here we give the details.

Based on Theorem 10, we now show a simple linear-time reduction from the smallest chain partition problem (where the input order is represented as described in Section 3.2) to the following problem:

► **Definition 11** (Interval partitioning problem, cf. [15, Sec. 4.1]). *Let $\{[s_1, f_1], \dots, [s_n, f_n]\}$ be a set of n activities that must be served (each) by a device. One device can handle at most one activity at the same time. $[s_i, f_i]$ is an interval, where s_i and f_i are the starting and finishing time of activity i , respectively. Determine the minimum number of devices to serve all the activities.*

Let $S = \{a_1 = (s_1, f_1), \dots, a_n = (s_n, f_n)\}$ be an instance of the *smallest chain partition* problem for $<_{\mathcal{A}}$ (that is, a particular instance of $<_{\mathcal{A}}$). Our reduction from this instance to an instance of the *interval partitioning problem* works as follows:

1. For each pair $a_i = (s_i, f_i)$, with $i \in [1..n]$, let $s'_i = 2s_i + 1$ and $f'_i = 2f_i$.
2. Return the set of intervals $S'' = \{a''_i\}_{i=1}^n$, where $a''_i = [s''_i = \min(s'_i, f'_i), f''_i = \max(s'_i, f'_i)]$

The following Lemma shows that our reduction is correct:

► **Lemma 12.** *Let $(s_i, f_i), (s_j, f_j)$ be two input pairs, with $s_i \leq s_j$ without loss of generality. Let moreover $[s''_i, f''_i], [s''_j, f''_j]$ be the intervals into which the two pairs get transformed by the above reduction. Then, $f_i \leq s_j$ if and only if $f''_i < s''_j$ (i.e. $[s''_i, f''_i]$ and $[s''_j, f''_j]$ do not overlap).*

Proof. We divide the proof into two cases: (Case 1) at least one of $s_i = f_i$ or $s_j = f_j$ holds, and (Case 2) both $s_i < f_i$ and $s_j < f_j$ hold.

(Case 1). First, we show that $f_i \neq s_j$. Assume that $s_i = f_i$ (the other case $s_j = f_j$ is analogous). Let u be the state associated with the pair (s_i, f_i) , and v be the state associated with the pair (s_j, f_j) . By Observation 8.5, $s_i = f_i$ implies that $\inf I_u = \sup I_u$ is a finite string, and $\inf I_u = \sup I_u \in I_u$. If $\inf I_v$ is an infinite string, then clearly $\sup I_u \neq \inf I_v$ (being $\sup I_u$ a finite string), i.e. $f_i \neq s_j$. If, on the other hand, $\inf I_v$ is a finite string, then by Observation 8.4 we have $\inf I_v \in I_v$; since by Observation 8.2, we have $I_u \cap I_v = \emptyset$, also in this case we derive that $\sup I_u \neq \inf I_v$, i.e. $f_i \neq s_j$.

Knowing $f_i \neq s_j$, we obtain that $f_i \leq s_j \Leftrightarrow f_i < s_j \Leftrightarrow f_i + 1 \leq s_j$. Note that, since $s_j'' = 2s_j + 1 > 2s_j$ (if $s_j \neq f_j$) or $s_j'' = 2f_j = 2s_j$ (if $s_j = f_j$), we have $2s_j \leq s_j''$. Similarly, $f_i'' \leq 2f_i + 1$. Hence $2s_i \leq s_i'' < f_i'' \leq 2f_i + 1$ (note that $s_i'' < f_i''$ always holds for any interval in our reduction). Therefore, we have $f_i \leq s_j \Rightarrow f_i + 1 \leq s_j \Rightarrow f_i'' \leq 2f_i + 1 < 2(f_i + 1) \leq 2s_j \leq s_j''$. For the other direction, note that $2f_i \leq f_i''$ and $s_j'' \leq 2s_j + 1$. Then, using these inequalities we obtain: $f_i'' < s_j'' \Rightarrow 2f_i \leq f_i'' < s_j'' \leq 2s_j + 1 \Rightarrow 2f_i < 2s_j + 1 \Rightarrow f_i \leq s_j$.

(Case 2). In this case, we have $f_i \leq s_j \Rightarrow f_i'' = 2f_i < 2s_j + 1 = s_j'' \Rightarrow f_i'' < s_j''$. For the other direction, note that $f_i'' < s_j'' \Rightarrow 2f_i = f_i'' < s_j'' = 2s_j + 1 \Rightarrow f_i \leq s_j$. ◀

By Lemma 12, we can now solve *smallest chain partition problem* for the particular order $<_{\mathcal{A}}$ by reducing it to an instance of the *interval partitioning problem*. Moreover, it is easy to see that the reduction works in linear time so the linearity of our strategy relies on the cost of the algorithm we use to solve the latter problem. We can use a greedy method (cf. [3, 8]) to optimally solve the interval partitioning problem (namely, using the smallest possible number of devices). The algorithm processes the intervals in non-decreasing order of starting times, breaking ties arbitrarily. For each interval, we choose any idle device among the available ones. We can keep track of the list of the available devices if the starting and finishing times of the intervals are already sorted. If all devices are busy, we add a new device.

The above-sketch algorithm spends amortized constant time on every activity, plus the time required to sort the input set of intervals. As said earlier, the elements of our input pairs (i.e. before the reduction) are integer values in the range $[1, 2n]$. After the reduction, this range gets expanded to $[2, 4n + 1]$. This allows us to radix-sort the intervals in $O(n)$ time. As a result, in our scenario we can solve the *interval partition problem* in $O(n)$ time and, in particular, find the *smallest chain partition* of $<_{\mathcal{A}}$ given its ranked-pair representation in linear time.

4 Co-lex sorting infimum/supremum strings

In this section, we present three algorithms to compute and sort the set containing all infimum and supremum strings of a DFA. The first two algorithms sort the strings in such a way that for every iteration the strings are co-lex sorted with respect to a longer suffix; we present one simple solution that increases the suffix length by 1 at each iteration, and one that doubles the suffix length at each iteration. The third algorithm is a generalization of online BWT construction and is based on the online algorithm for sorting Wheeler DFA presented in [1, Sec. 3.2]. This algorithm works only on acyclic DFAs but has a lower time complexity than the former two solutions.

For ease of explanation, we consider only infimum strings since the supremum string case is analogous. Indeed, one can easily compute and sort both infimum and supremum strings at the same time by creating two copies of the input DFA and then running our algorithms on the union of the two DFAs, extracting the infimum strings on one DFA and the supremum strings on the other DFA while at the same time sorting the union of these two string sets.

4.1 Simple $O(mn)$ -time algorithm

Let us establish some notations before describing the algorithm. For a (possibly infinite) string α and an integer $k \geq 0$, we denote by $\text{suf}_k(\alpha)$ the length- k suffix of α . When $|\alpha| < k$, we pad $\text{suf}_k(\alpha)$ by prepending $k - |\alpha|$ copies of a special symbol $\# \notin \Sigma$, with $\# < c$ for all $c \in \Sigma$; in this way, we guarantee that $\text{suf}_k(\alpha)$ is always a string of length k and we do not affect the co-lex order of such suffixes (which remains the same before and after the padding).

For state $u \in Q$ and integer $k \geq 0$, we denote by $\text{rank}_k(u) \in \mathbb{N}$ the intermediate rank at iteration k of u in the total order we are computing; this integer indicates the co-lex rank of $\text{suf}_k(\inf I_u)$ among $\{\text{suf}_k(\inf I_v) : v \in Q\}$. More formally, for $u \in Q$ and $k \geq 0$,

$$\begin{aligned} \text{rank}_0(u) &= 1 \\ \text{rank}_k(u) &= |\{\text{suf}_k(\inf I_v) : v \in Q \wedge \text{suf}_k(\inf I_v) \leq \text{suf}_k(\inf I_u)\}| \text{ for } k > 0 \end{aligned}$$

Observe that two states are assigned the same rank if their corresponding length- k suffixes are equal. The algorithm works by *pruning* transitions of the input automaton, i.e. by removing, for every state u , transitions coming from a state with a non-minimum rank among the predecessors of u . We denote by δ_k the (pruned) transition function at iteration k .

The algorithm works as follows. At iteration $k \geq 0$, we perform the following operations:

1. **Compute** rank_{k+1} . Sort the states $\{u \in Q\}$ by their label $\lambda(u)$ with ties broken by $\text{rank}_k(v)$ for any $v \in \delta_k^{-1}(u)$ (the step below will guarantee that all predecessors v of u have the same $\text{rank}_k(v)$).
2. **Compute** δ_{k+1} . For each $u \in Q$, keep only the transitions from the min-rank predecessors: for $v \in \delta_k^{-1}(u)$, $v \in \delta_{k+1}^{-1}(u)$ iff $\text{rank}_{k+1}(v) = \min\{\text{rank}_{k+1}(u') : u' \in \delta_k^{-1}(u)\}$.

As far as the running time of each iteration is concerned, computing rank_{k+1} can be done in $O(n)$ time by 2-pass radix sorting (that is, by incoming label and breaking ties by any predecessor's rank rank_k). Computing δ_{k+1} takes $O(|\delta_k|) = O(|\delta|) = O(m)$ time. Hence, each iteration takes $O(m)$ time.

Since $\forall k \geq 0$, $\forall u \in Q$, and $\forall v \in \delta_k^{-1}(u)$ we have $\text{suf}_{k+1}(\inf I_u) = \text{suf}_k(\inf I_v) \cdot \lambda(u)$, it is easy to see that the following invariant always holds at the beginning of iteration k : the infimum strings are sorted with respect to the co-lex order of their length- k suffixes. This invariant shows that the number of iterations we have to perform is exactly the length of the suffixes that need to be sorted to obtain the correct co-lex order of the infimum strings. We are left to find an upper bound to this length; observe that this is not a trivial problem, since infimum strings may have infinite length.

Consider any two infimum strings $\alpha, \beta \in \{\inf I_u : u \in Q\}$. The upper bound above can be computed by upper-bounding the length of the longest common suffix between α and β . If any of the two strings is finite, then by Lemma 9 their longest common suffix does not exceed length n . If both strings are infinite, then by Lemma 9 we can write them as $\alpha = \alpha_2^\omega \alpha_1$ and $\beta = \beta_2^\omega \beta_1$ and we can use the following:

► **Lemma 13** (cf. [19]). *For two infinite strings $\alpha = \alpha_2^\omega \alpha_1$ and $\beta = \beta_2^\omega \beta_1$, where $\alpha_1, \beta_1 \in \Sigma^*$ and $\alpha_2, \beta_2 \in \Sigma^+$, let α' and β' be their suffixes of length $k = |\alpha_2| + |\beta_2| + \max\{|\alpha_1|, |\beta_1|\}$. Then, $\alpha' < \beta'$ if and only if $\alpha < \beta$.*

Proof. Without loss of generality, let us assume $|\alpha_1| \leq |\beta_1|$. Moreover, note that without loss of generality we can also assume that $|\alpha_2| + |\alpha_1| > |\beta_1|$; if this does not hold, then re-write $\alpha_1 \leftarrow \alpha_2^k \alpha_1$ for the only integer $k > 0$ such that $|\alpha_1| \leq |\beta_1| < |\alpha_2| + |\alpha_1|$ holds; after the transformation, α can still be written as $\alpha = \alpha_2^\omega \alpha_1$.

If α_1 is not a suffix of β_1 , then clearly the longest common suffix between α and β is at most $|\alpha_1|$, so the claim holds. Let us assume therefore that α_1 is a suffix of β_1 , i.e. $\beta_1 = \beta'_1\alpha_1$ for some $\beta'_1 \in \Sigma^*$. Since by assumption $|\alpha_2| + |\alpha_1| > |\beta_1|$, note that $|\alpha_2| > |\beta'_1|$. Similarly as above, if β'_1 does not suffix α_2 (i.e. $\beta_1 = \beta'_1\alpha_1$ does not suffix $\alpha = \alpha_2^\omega\alpha_1$) then the longest common suffix between α and β is at most $|\beta_1|$ and the claim holds, so let us assume that β'_1 is a suffix of α_2 , i.e. $\alpha_2 = \alpha'_2\beta'_1$ for some $\alpha'_2 \in \Sigma^*$. Since $\alpha_2^\omega = (\alpha'_2\beta'_1)^\omega = (\beta'_1\alpha'_2)^\omega\beta'_1$, we conclude that comparing co-lexicographically $\alpha = (\beta'_1\alpha'_2)^\omega\beta'_1\alpha_1$ and $\beta = \beta_2^\omega\beta'_1\alpha_1$ reduces to comparing $(\beta'_1\alpha'_2)^\omega$ and β_2^ω . According to [19, Proposition 5], given any $\gamma_1, \gamma_2 \in \Sigma^+$ it is sufficient to compare the length- k' suffixes of γ_1^ω and γ_2^ω to determine their co-lex order, where $k' = |\gamma_1| + |\gamma_2| - \gcd(|\gamma_1|, |\gamma_2|)$. Our claim easily follows since $|\beta'_1\alpha'_2| = |\alpha'_2\beta'_1| = |\alpha_2|$. ◀

► **Corollary 14.** *The co-lex order of the infimum and supremum strings of a DFA is the same as the co-lex order of their length- $(2n)$ suffixes.*

Proof. By Lemma 9, we can represent two infimum/supremum strings α, β as $\alpha = \alpha_2^\omega\alpha_1$ and $\beta = \beta_2^\omega\beta_1$. By the same lemma, each of $|\alpha_1|, |\alpha_2|, |\beta_1|$ and $|\beta_2|$, as well as $|\alpha_1| + |\alpha_2|$ and $|\beta_1| + |\beta_2|$, are bounded by n . From Lemma 13, it is sufficient to compare the suffixes of length at most $|\alpha_2| + |\beta_2| + \max\{|\alpha_1|, |\beta_1|\} = \max\{(|\alpha_1| + |\alpha_2|) + |\beta_2|, |\alpha_2| + (|\beta_1| + |\beta_2|)\}$ of α and β in order to discover their co-lex order. Therefore, $2n$ is a sufficient suffix length for sorting all the infimum strings correctly. ◀

Putting everything together, we conclude:

► **Lemma 15.** *The infimum and supremum strings of an input-consistent DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ can be computed and sorted in $O(mn)$ time, where $n = |Q|$ is the number of states and $m = |\delta|$ is the number of transitions.*

Equivalently, the above lemma shows that the representation of $<_{\mathcal{A}}$ of Section 3.2 can be computed in $O(mn)$ time. Plugging the linear-time chain partition algorithm of Section 3.3, we obtain:

► **Theorem 16.** *Given an input-consistent DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, we can compute a minimum-size chain partition of $<_{\mathcal{A}}$ in $O(mn)$ time, where $n = |Q|$ is the number of states and $m = |\delta|$ is the number of transitions.*

4.2 $O(n^2 \log n)$ -time suffix doubling algorithm

Instead of increasing the length of the sorted suffixes only by 1 at every iteration, we can double it via a generalization of the prefix doubling algorithm [18], the first suffix array construction algorithm that appeared in the literature. Again, for simplicity we describe the algorithm just for infimum strings; it is easy to modify it so that it computes and sorts the union of all infimum and supremum strings.

Algorithm 1 describes our sorting procedure, which we explain in detail in the rest of the section. At every iteration $k \geq 0$, Algorithm 1 keeps the infimum strings sorted by their length- 2^k suffixes. Suppose we already sorted the infimum strings with respect to their length- 2^k suffixes. To enable the doubling procedure, we need to show how to compute the length- 2^{k+1} suffix of each $\inf I_u$ given as input the length- 2^k suffixes of each $\inf I_u$, for all $u \in Q$. Given that the infimum strings are sorted by their length- 2^k suffixes, for each $u \in Q$ we can achieve this goal by finding a state $v \in Q$ such that

$$suf_{2^{k+1}}(\inf I_u) = suf_{2^k}(\inf I_v) \cdot suf_{2^k}(\inf I_u).$$

■ **Algorithm 1** Suffix doubling algorithm for sorting the infimum strings of a DFA.

Input: An input-consistent DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
Output: $rank_{2^k}(u)$ for each $u \in Q$, with $2n \leq 2^k < 4n$.

```

1  $k \leftarrow 0$ ;
2 for  $u \in Q$  do
3    $rank_{2^k}(u) \leftarrow \lambda(u)$ ;
4    $P_k(u) \leftarrow \{v \in \delta^{-1}(u) : (\forall v' \in \delta^{-1}(u))(\lambda(v) \leq \lambda(v'))\}$ ;
5 while  $2^k < 2n$  do
6   for  $u \in Q$  do
7      $a_u \leftarrow rank_{2^k}(u)$ ;
8     if  $P_k(u) = \emptyset$  then
9        $b_u \leftarrow -\infty$ ;
10    else
11      Pick any  $v \in P_k(u)$ ;
12       $b_u \leftarrow rank_{2^k}(v)$ ;
13    Compute  $rank_{2^{k+1}}(\cdot)$  by radix-sorting pairs  $(a_u, b_u)$ ;
14    for  $u \in Q$  do
15       $\hat{P}_{k+1}(u) \leftarrow \bigcup_{v \in P_k(u)} P_k(v)$ ;
16       $P_{k+1}(u) \leftarrow \{v \in \hat{P}_{k+1}(u) : (\forall v' \in \hat{P}_{k+1}(u))(rank_{2^{k+1}}(v) \leq rank_{2^{k+1}}(v'))\}$ ;
17     $k \leftarrow k + 1$ ;
18 return  $rank_{2^k}(\cdot)$ 

```

We call such v an *extender of u (at distance 2^k)*. More formally, the set $P_k(u)$ of all extenders of u at distance 2^k is defined as:

$$P_0(u) = \{v \in \delta^{-1}(u) : (\forall v' \in \delta^{-1}(u))(\lambda(v) \leq \lambda(v'))\}$$

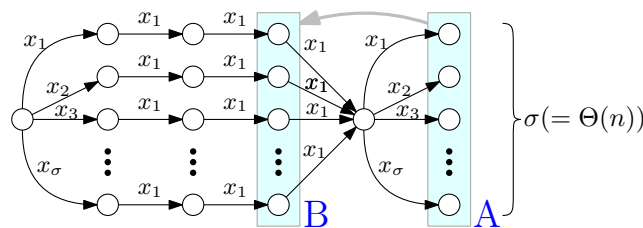
$$P_k(u) = \{v \in Q : \delta(v, suf_{2^k}(\inf I_u)) = u \wedge suf_{2^k}(\inf I_v) \sqsubseteq suf_{2^{k+1}}(\inf I_u)\} \text{ for } k > 0$$

For $u \in Q$, let $rank_{2^k}(u)$ be the co-lex rank of $suf_{2^k}(\inf I_u)$, as defined in the previous section. Observe that, by definition, for every $u \in Q$ and $v_1, v_2 \in P_k(u)$, $rank_{2^k}(v_1) = rank_{2^k}(v_2)$.

We implement a suffix doubling step as follows. Assume $P_k(u)$ and $rank_{2^k}(u)$ have been computed for all $u \in Q$. We associate with u the pair (a_u, b_u) where $a_u = rank_{2^k}(u)$ and b_u is chosen as follows. If $P_k(u) \neq \emptyset$, $b_u = rank_{2^k}(v)$ with any $v \in P_k(u)$; otherwise, $b_u = -\infty$ is chosen⁴. Finally, we compute $rank_{2^{k+1}}(\cdot)$ by radix-sorting pairs (a_u, b_u) in $O(n)$ time.

After computing $rank_{2^{k+1}}(\cdot)$, we need to compute $P_{k+1}(\cdot)$ for the next doubling step. For a state $u \in Q$, let $\hat{P}_{k+1}(u) = \bigcup_{u' \in P_k(u)} P_k(u')$ be the union of the extender sets of u 's extenders at distance 2^k . Then, we claim that we can compute $P_{k+1}(u)$ by removing all non-minimum-rank states (i.e. non-minimum $rank_{2^{k+1}}(\cdot)$) from $\hat{P}_{k+1}(u)$. The correctness of this procedure follows from the fact that $P_{k+1}(u)$ can also be defined as the largest subset of $\hat{P}_{k+1}(u)$ such that, for every $v \in P_{k+1}(u)$ and $\hat{v} \in \hat{P}_{k+1}(u)$, $rank_{2^{k+1}}(v) \leq rank_{2^{k+1}}(\hat{v})$. To see this, first observe that $P_{k+1}(u) \subseteq \hat{P}_{k+1}(u)$ because (i) $v \in \hat{P}_{k+1}(u)$

⁴ In this case, there are no more characters to be prepended to $\inf I_u$ (i.e. $|\inf I_u| < 2^k$). Since we radix-sort pairs (a_u, b_u) , this choice is consistent with the fact that $suf_{2^k}(\inf I_u)$ is left-padded with copies of symbol $\#$ in order to reach length 2^k , with $\# < c$ for all $c \in \Sigma$ (see definition of suf_{2^k} in the previous section).



■ **Figure 2** The DFA that has a quadratic number of extenders: the $\sigma = \Theta(n)$ states in the rightmost column (indicated with A) have $\sigma = \Theta(n)$ extenders each (indicated with B) at distance 2^k , where $k = 1$.

if and only if $\delta(v, \text{suffix}_{2^{k+1}}(\text{inf } I_u)) = u$, and (ii) $v \in P_{k+1}(u) \Rightarrow \delta(v, \text{suffix}_{2^{k+1}}(\text{inf } I_u)) = u$. Also, by the definition of P_{k+1} , $\text{suffix}_{2^{k+1}}(\text{inf } I_v)$ for $v \in P_{k+1}(u)$ must not be greater than $\text{suffix}_{2^{k+1}}(\text{inf } I_{\hat{v}})$ for any $\hat{v} \in \hat{P}_{k+1}(u)$, which is equivalent to $\text{rank}_{2^{k+1}}(v) \leq \text{rank}_{2^{k+1}}(\hat{v})$; otherwise, $\text{suffix}_{2^{k+1}}(\text{inf } I_{\hat{v}}) \cdot \text{suffix}_{2^{k+1}}(\text{inf } I_u) < \text{suffix}_{2^{k+1}}(\text{inf } I_v) \cdot \text{suffix}_{2^{k+1}}(\text{inf } I_u) = \text{suffix}_{2^{k+2}}(\text{inf } I_u)$, which contradicts the definition of $\text{inf } I_u$.

Since $\text{rank}_{2^{k+1}}(v)$ has already been computed for all $v \in Q$ and can thus be evaluated in constant time, from the above characterization of $P_{k+1}(u)$ we obtain that the time required to compute this set is proportional to the time we spend to compute the union $\hat{P}_{k+1}(u) = \bigcup_{u' \in P_k(u)} P_k(u')$. Observe that, if there were repeated states among the sets $P_k(u')$, for $u' \in P_k(u)$, then computing such a union could take time $O(n^2)$ (for every $u \in Q$), leading to a cubic algorithm. Luckily, with the next lemma we show that this is not the case: being the input automaton deterministic, those sets are pairwise disjoint and their union can thus be computed by just concatenating them.

► **Lemma 17.** *Let $u \in Q$ be a state of a DFA, and let $v_1, v_2 (\neq v_1) \in P_k(u)$ be extenders of u at distance 2^k . Then $P_k(v_1) \cap P_k(v_2) = \emptyset$.*

Proof. Let $v_1, v_2 (\neq v_1) \in P_k(u)$ be extenders of the same state $u \in Q$ at distance 2^k . Let $\alpha_1 = \text{suffix}_{2^k}(\text{inf } I_{v_1})$ and $\alpha_2 = \text{suffix}_{2^k}(\text{inf } I_{v_2})$. Assume, for a contradiction, that there exists $v' \in P_k(v_1) \cap P_k(v_2)$. By definition of P_k , since $v' \in P_k(v_1)$, it holds $\delta(v', \alpha_1) = v_1$. Similarly, it also holds $\delta(v', \alpha_2) = v_2$. Since $v_1, v_2 \in P_k(u)$ are extenders of the same state $u \in Q$ at distance 2^k , both α_1 and α_2 are equal to the length- 2^k prefix of $\text{suffix}_{2^{k+1}}(\text{inf } I_u)$, therefore $\alpha_1 = \alpha_2$. Consequently, we have $v_1 = \delta(v', \alpha_1) = \delta(v', \alpha_2) = v_2$, i.e. reading a string $\alpha_1 = \alpha_2$ from a state v' we reach two distinct states $v_1 \neq v_2$. This is a contradiction with the fact that the automaton is deterministic, so the claim $P_k(v_1) \cap P_k(v_2) = \emptyset$ must be true. ◀

From Lemma 17, we can compute $\hat{P}_{k+1}(u)$ in time proportional to its cardinality $|\hat{P}_{k+1}(u)| \leq n$. Since finding the minimum-rank states can be done in linear $O(|\hat{P}_{k+1}(u)|)$ time as well, the computation of $P_{k+1}(u)$ takes time $O(n)$ for each $u \in Q$. We conclude that each iteration of the suffix doubling algorithm takes $O(n^2)$ time.

It is worth noting that, since we keep track of each set $P_k(u)$, the running time of an iteration is lower-bounded by the total number of extenders therein. In the worst case, however, the total number of extenders at a single iteration could be truly quadratic even on acyclic DFAs: see Figure 2; in this example, there are $n = 4\sigma + 2$ states and $\sigma^2 + 2\sigma + 1 = \Theta(n^2)$ extenders at distance 2^k for $k = 1$.

Putting all together, we have the following result for the suffix doubling algorithm described in this section.

► **Lemma 18.** *The infimum and supremum strings of an input-consistent DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ can be computed and sorted in $O(n^2 \log n)$ time, where $n = |Q|$ is the number of states.*

Equivalently, the above lemma shows that the representation of $<_{\mathcal{A}}$ of Section 3.2 can be computed in $O(n^2 \log n)$ time. Plugging the linear-time chain partition algorithm of Section 3.3, we obtain:

► **Theorem 19.** *Given an input-consistent DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, we can compute a minimum-size chain partition of $<_{\mathcal{A}}$ in $O(n^2 \log n)$ time, where $n = |Q|$ is the number of states.*

The suffix doubling algorithm in practice. Although every iteration of the suffix doubling algorithm needs to keep track of $O(n^2)$ extenders per iteration in the worst case, we conjecture that it is not likely to have a quadratic number of extenders on realistic datasets. To demonstrate this, we conducted a brief experiment using a pan-genomic graph, which is considered to be one of the most important real-world applications of our problem. We downloaded the Chromosome 22 sequence of the GRCh38 human reference genome and its variation data from 1000 Genome project [17]. This variation dataset contains a set of substitutions, insertions and deletions appearing on the reference human genome sequence collected from 2,548 samples. Using this dataset, we constructed a pan-genomic graph using VG [21], then converted it into a DFA using the classical powerset construction algorithm [20]. We ran an implementation of our suffix doubling algorithm to sort the infimum and supremum strings and measured the number of extenders at each iteration. The largest $\hat{P}_k(u)$ and $P_k(u)$ (extenders at distance 2^k before/after filtering non-minimum-rank states) during the procedure had cardinality 60 and 37, respectively, which might be considered not negligible but quite small when compared to the DFA's size ($n=51,904,782$, $m=53,049,316$). In addition, the sum $\sum_{u \in Q} |\hat{P}_k(u)|$ of the number of extender candidates (the union of extenders before filtering non-minimum-rank states) at any fixed distance 2^k was at most two times the number of edges, suggesting that in practice our algorithm exhibits a linearithmic complexity on pan-genomic graphs. C++ source code is available at: <https://github.com/regindex/DFA-suffix-doubling>.

4.3 $O(m \log n)$ -time algorithm for acyclic DFAs

If the input DFA is acyclic, then we can sort the infimum strings more efficiently using the algorithm described in [1, Sec. 3.2]. This algorithm processes the states of any acyclic *Wheeler DFA* \mathcal{A} (that is, $\text{width}(\mathcal{A}) = 1$) and their incoming edges in any topological order u_1, \dots, u_n while updating $<_{\mathcal{A}}$ in an online fashion; more precisely, as soon as step $1 \leq i \leq n$ has finished, the algorithm has computed the total order $<_{\mathcal{A}}$ of the set $\{u_1, \dots, u_i\}$. The basic idea is to process the states in any topological order while maintaining a dynamic data structure that stores the relative co-lex order of the states according to any representative of I_u (in fact, [1] proves that on Wheeler DFAs, any string in I_u can be chosen as a representative of the whole I_u to sort the automaton's states). This is possible because, when $u_i \in Q$ is being processed, the structure is able to check if $\text{rank}(v) \leq \text{rank}(u_j)$ for any $v \in \delta^{-1}(u_i)$ and $j < i$, where $\text{rank}(v)$ denotes the position of v in the total order $<_{\mathcal{A}}$ of the already-processed states u_1, \dots, u_{i-1} (and, by definition of topological order, $\text{rank}(v)$ and $\text{rank}(u_j)$ have already been computed in the previous steps). This information is sufficient to compute $\text{rank}(u_i)$ among the sorted u_1, \dots, u_i .

In our case (arbitrary acyclic DFAs), we use the above data structure as follows: after topologically sorting \mathcal{A} (in linear time) we process states in this order. When processing state u_i , assume that states u_1, \dots, u_{i-1} have already been co-lex sorted according to their strings $\text{inf } I_{u_1}, \dots, \text{inf } I_{u_{i-1}}$ using the data structure of [1, Section 3.2]. By scanning the predecessors of u_i , we find the min-rank state $v^* = \arg \min_{v \in \delta^{-1}(u_i)} \text{rank}(v)$ among them. At this point, we insert state u_i , as well as transition (labeled edge) $(v^*, u_i, \lambda(u_i))$, in the data structure. Note that, since only $(v^*, u_i, \lambda(u_i))$ is inserted, the data structure of [1, Section 3.2] maintains a spanning tree of \mathcal{A} rooted at the start state s . By construction, it is easy to see that the unique path connecting s to u_i in this spanning tree is labeled with string $\text{inf } I_{u_i}$: the spanning tree encodes the infimum strings. As a result, our sorting problem is equivalent to sorting this spanning tree. It is known that a labeled tree is a special case of Wheeler graphs (see [11]), so the computed co-lex node order of this spanning tree is precisely the co-lex order of all infimum strings.

Since it is immediate to extend the idea to the union of all infimum and supremum strings, taking into account the cost of each update of the data structure [1, Sec. 3.2], we obtain:

► **Lemma 20.** *The infimum and supremum strings of an input-consistent acyclic DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ can be computed and sorted in $O(m \log n)$ time where $n = |Q|$ is the number of states and $m = |\delta|$ is the number of transitions.*

Proof. First of all, the data structure [1, Sec. 3.2] supports the following two operations in $O(\log n)$ time⁵: (i) computing the relative rank of a state among those that are already processed, and (ii) inserting a new edge (a state is inserted into the structure after all its incoming edges have been inserted). As a result, finding $v^* = \arg \min_{v \in \delta^{-1}(u_i)} \text{rank}(v)$ takes time $O(|\delta^{-1}(u_i)| \cdot \log n)$. After v^* has been found, inserting the labeled edge $(v^*, u_i, \lambda(u_i))$, as well as state u_i , into the structure takes time $O(\log n)$. Overall, after all states have been processed the cost of the above operations amounts to $O(m \log n)$ time. Since a topological order of \mathcal{A} can be computed in $O(m)$ time, the total running time is $O(m \log n)$. ◀

Equivalently, the above lemma shows that the representation of $<_{\mathcal{A}}$ of Section 3.2 can be computed in $O(m \log n)$ time when \mathcal{A} is acyclic. Plugging the linear-time chain partition algorithm of Section 3.3, we obtain:

► **Theorem 21.** *Given an input-consistent acyclic DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$, we can compute a minimum-size chain partition of $<_{\mathcal{A}}$ in $O(m \log n)$ time, where $n = |Q|$ is the number of states and $m = |\delta|$ is the number of transitions.*

References

- 1 Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of 31st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 911–930, 2020. doi:10.1137/1.9781611975994.55.
- 2 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012. doi:10.1007/978-3-642-33122-0_18.
- 3 Martin C. Carlisle and Errol L. Lloyd. On the k -coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995. doi:10.1016/0166-218X(95)80003-M.

⁵ In fact, the running time per operation is $O(\log m')$, where m' is the number of edges. However, in our case, the number of transitions inserted into the data structure is only $m' = n - 1$ since we insert one transition per state (except s). Therefore, in our case, the time taken per operation is $O(\log n)$.

- 4 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281, 2022. doi:10.1109/DCC52660.2022.00035.
- 5 Nicola Cotumaccio, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages. Part I, 2022. doi:10.48550/ARXIV.2208.04931.
- 6 Nicola Cotumaccio and Nicola Prezza. On Indexing and Compressing Finite Automata. In *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2585–2599, 2021. doi:10.1137/1.9781611976465.153.
- 7 Robert P. Dilworth. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics*, 51(1):161–166, 1950. doi:10.2307/1969503.
- 8 Ulrich Faigle and Willem M. Nawijn. Note on Scheduling Intervals on-line. *Discrete Applied Mathematics*, 58(1):13–17, 1995. doi:10.1016/0166-218X(95)00112-5.
- 9 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005. doi:10.1109/SFCS.2005.69.
- 10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
- 11 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. doi:10.1016/j.tcs.2017.06.016.
- 12 Daniel Gibney and Sharma V. Thankachan. On the Hardness and Inapproximability of Recognizing Wheeler Graphs. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA)*, pages 51:1–51:16, 2019. doi:10.4230/LIPIcs.ESA.2019.51.
- 13 Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New Indices for Text: Pat Trees and Pat Arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice Hall, 1992.
- 14 Mitchel T. Keller and William T. Trotter. *Applied Combinatorics*. CreateSpace Independent Publishing Platform, 2017. URL: <https://www.appliedcombinatorics.org/book/app-comb.html>.
- 15 Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson/Addison-Wesley, 2006.
- 16 Shimon Kogan and Merav Parter. Beating Matrix Multiplication for $n^{1/3}$ -Directed Shortcuts. In *Proceedings of 49th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 82:1–82:20, 2022. doi:10.4230/LIPIcs.ICALP.2022.82.
- 17 Ernesto Lowy-Gallego, Susan Fairley, Xiangqun Zheng-Bradley, Magali Ruffier, Laura Clarke, Paul Flicek, and The 1000 Genomes Project Consortium. Variant calling on the GRCh38 assembly with the data from phase three of the 1000 Genomes Project. *Wellcome Open Research*, 4(50), 2020. doi:10.12688/wellcomeopenres.15126.2.
- 18 Udi Manber and Gene Myers. Suffix Arrays: A New Method for on-Line String Searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327, 1990. doi:10.1137/0222058.
- 19 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.
- 20 M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM Journal on Research and Development*, 3(2):114–125, 1959. doi:10.1147/rd.32.0114.
- 21 Jouni Sirén, Erik Garrison, Adam M. Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2019. doi:10.1093/bioinformatics/btz575.
- 22 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

Encoding Hard String Problems with Answer Set Programming

Dominik Köppl   

Department of Computer Science, Universität Münster, Germany

Abstract

Despite the simple, one-dimensional nature of strings, several computationally hard problems on strings are known. Tackling hard problems beyond sizes of toy instances with straight-forward solutions is infeasible. To solve these problems on datasets of even small sizes, effort has to be put into the conception of algorithms leveraging profound characteristics of the input. Finding these characteristics can be eased by rapidly creating and evaluating prototypes of new concepts in how to tackle hard problems. Such a rapid-prototyping method for hard problems is answer set programming (ASP). In this light, we study the application of ASP on five NP-hard optimization problems in the field of strings. We provide MAX-SAT and ASP encodings, and empirically reason about the merits and flaws when working with ASP solvers.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Artificial intelligence; Theory of computation → Discrete optimization; Hardware → Theorem proving and SAT solving

Keywords and phrases optimization problems, answer set programming, MAX-SAT encoding, NP-hard string problems

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.17

Supplementary Material *Software (Source Code)*: <https://github.com/koeppl/aspstring>
archived at [sw:h1:dir:fccb46f95c8f782e61c7e63a9a154a6e2fd8dad9](https://sw.h1.dir:fccb46f95c8f782e61c7e63a9a154a6e2fd8dad9)

Funding Supported by JSPS KAKENHI Grant Numbers JP21K17701 and JP23H04378.

Acknowledgements We thank Mutsunori Banbara for drawing our attention to the ASP language.

1 Introduction

Despite the fact that most string problems found in literature are solvable in polynomial time or even close to linear time or beyond, there are several problems that are known to be NP-hard. Among those, we focus on five problems that are well-perceived regarding the number of publications studying these problems: CLOSEST STRING (CSP)¹, CLOSEST SUBSTRING (CSS), LONGEST COMMON SUBSEQUENCE (LCS), MINIMUM COMMON STRING PARTITION (MCSP), and SHORTEST COMMON SUPERSTRING (SCS). These problems have been studied under various viewpoints. With respect to fixed-parameter tractability (FPT), Bulteau et al. [9] gave a comprehensive survey on various NP-hard problems related to strings; this survey comprises the problems studied in this paper. Also, Basavaraju et al. [2] studied the kernelization of a majority of our problems. We address other related work in the individual sections of each problem, but omit references to approximation algorithms due to their amount, and because we put focus on the *exact* solution of the aforementioned problems formulated as optimization problems.

¹ We stick to the commonly used abbreviation CSP in literature despite that CS would fit better with the abbreviations of the other problems.



© Dominik Köppl;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 17; pp. 17:1–17:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A major problem in tackling these problems in practice is that naive solutions quickly become impractical with respect to the time complexity. Tailored algorithms² are hard to implement, and thus a burden on the algorithm engineering side. Our contribution is to advertise *answer set programming* (ASP) as a rapid-prototype programming tool for solving NP-hard string problems on small instances. ASP is a declarative programming language geared towards solving hard problems [40, 12]. ASP has been successfully applied in robotics [3], or for computing the n -queens and the knight’s tour problem [18]. There is also a competition on ASP solvers on various classic problems addressing mainly problems on graphs [28]. See [19, 20] and the references therein for an overview of other use cases.

Although well-devised algorithms can outperform ASP-based approaches, the programming effort for writing in an expressive, declarative programming language such as ASP is considerably small. In this paper, we devise MAX-SAT encodings for the above addressed problems, and subsequently translate these encodings into the ASP language. With respect to tackling hard string problems via MAX-SAT encodings we are aware of the work of Bannai et al. [1] who studied MAX-SAT encodings for repetitiveness measures that are also known to be NP-hard.

2 Preliminaries

Common to all problems treated in this paper is the input of a set of m strings $\mathcal{S} = \{S_1, \dots, S_m\}$. For simplicity, we assume that all strings have the same length n , and that all characters are drawn from an alphabet Σ of size $\sigma = |\Sigma|$. Hence, $|S_x| = n$ denotes the length of each input string and $S_x[i] \in \Sigma$ for all $i \in [1..n]$ and $x \in [1..m]$. Except for MCSP, the output is a string T that is object to an optimization argument with respect to the input strings (and, additionally for CSS, with respect to an integer parameter specifying the length of T).

Encoding Annotations. Beginning with the next section, we state rules and constraints with numbered equations, and add to each equation, in square brackets, the number of generated clauses and the size of each such clause. For instance, the equation

$$[\mathcal{O}(n), \mathcal{O}(1)] \quad \forall i \in [1..n] : p_i \implies p_{i+1} \quad (1)$$

defines n clauses, each of the form $(\neg p_i \vee p_{i+1})$, so its complexity is $[\mathcal{O}(n), \mathcal{O}(1)]$.

Experiments. We implemented our MAX-SAT-formulations in the ASP language, and used the solver `clingo` [26, 27]³ for evaluation. We compare the results with brute-force approaches written in the python language on randomly generated data. Our filenames are formatted like `s03m04n005i1` to denote that the alphabet size is $\sigma = 3$, the number of strings is $m = 4$, the length of each string is $n = 5$, and this file is the $i = 1$ -st sample of a batch of files with the same characteristics (σ, m and n). For MCSP, we have file formats like `2s02n008i2.txt` where the prefix 2 denotes that $m = 2$ is fixed. For the MCSP files, we assume that the two strings given have the same Parikh vector. Our implementations and datasets are available at <https://github.com/koepp1/aspstring>. For the evaluation, all experiments ran single-threaded on a machine with Intel Core i3-9100 CPU and Debian 11.

² Meaning that such algorithms usually are based on theoretical results that can be put hardly into practice.

³ <https://github.com/potassco/clingo>

	1	2	3	4	5	6	7	8	9	10	11	12	13		1	2	3	4	5	6	7	8	9	10	11	12	13
$S_1 =$	l	n	e	e	p	l	e	s	s	n	e	l	s	$S_1 =$	l	n	e	e	p	l	e	s	s	n	e	l	s
$S_2 =$	s	l	e	e	p	s	l	s	s	n	e	s	n	$S_2 =$	s	l	e	e	p	s	l	s	s	n	e	s	n
$S_3 =$	n	l	e	l	p	l	e	s	s	n	s	s	s	$S_3 =$	n	l	e	l	p	l	e	s	s	n	s	s	s
$S_4 =$	s	n	e	e	p	l	e	l	s	n	s	s	s	$S_4 =$	s	n	e	e	p	l	e	l	s	n	s	s	s
$S_5 =$	s	l	l	e	e	l	e	s	s	n	s	s	s	$S_5 =$	s	l	l	e	e	l	e	s	s	n	s	s	s
														$T =$	s	l	e	e	p	l	e	s	s	n	e	s	s

■ **Figure 1** Example for CSP (Sect. 3) with $n = 13$. The input set $\mathcal{S} = \{S_1, \dots, S_5\}$ is shown on the left figure. The right figure shows that the solution $T = \text{sleeplessness}$ has three mismatches with each of the input strings in the Hamming distance. Mismatching characters are highlighted by surrounding boxes.

3 Closest String Problem (CSP)

The CLOSEST STRING PROBLEM (CSP)⁴ asks for a string T such that $\max_{x \in [1..m]} \text{dist}_{\text{ham}}(S_x, T)$ is minimal, where the *Hamming distance* dist_{ham} is given by $\text{dist}_{\text{ham}}(S_x, T) := |\{i \in [1..n] : S_x[i] \neq T[i]\}|$. An example is shown in Fig. 1. Here, and in the following examples we stick to the alphabet $\Sigma := \{e, l, p, n, s\}$ with size $\sigma = 5$.

Related Work. Frances and Litman [24] and Lanctôt et al. [39] proved that CSP and its generalization, the CLOSEST SUBSTRING PROBLEM (CSS), are NP-hard for any alphabet with $\sigma \geq 2$ in n and m . The parameterized complexities have been surveyed in [48, Section 5.1] and [57], with focus also on CSS. For the decision problem with a Hamming distance of d , Gramm et al. [32] showed that CSP can be solved in $\mathcal{O}(mn + d^d)$ time or $2^{2^{\mathcal{O}(m \log m)}} \mathcal{O}(\log n)$ time. Regarding integer linear programming (ILP), Chimani et al. [13] gave ILP formulations, also for CSS. There is a line of research on further practical ILP formulations [16, 43, 54]. Finally, Knop et al. [38] gave also an ILP formulation and an exact algorithm running in $m^{\mathcal{O}(m^2)} \mathcal{O}(\log n)$ time.

With respect to different kinds of optimization approaches, Kelsey and Kotthoff [37] studied CSP as a constraint satisfaction problem, Huan et al. [35] provided an ant colony optimization algorithm, and Vilca and de Freitas [55] gave a specialized algorithm for fixed $m = 3$.

3.1 MAX-SAT encoding

We use the known fact that we have to select, for the i -th character of the output T , a character appearing at the i -th position of one of the input strings.

► **Lemma 1** ([37, Lemma 2]). *For each $i \in [1..n]$, $T[i] = S_x[i]$ for an $x \in [1..m]$.*

Let us define $\Sigma_i := \{S_1[i], \dots, S_m[i]\}$ to be the set of characters appearing at text position i of all input strings. Then $\sigma_i := |\Sigma_i| \leq \min(m, \sigma)$, and σ_i can be much less than m or σ if the number of distinct characters is small. We can express the alphabets per position Σ_i by a Boolean matrix $M[1..n][1..\sigma]$ with $M[i][c] = 1$ if $c \in \Sigma_i$.

⁴ Alternative names are, among others, MINIMUM RADIUS, CENTER STRING or CONSENSUS STRING problem.

17:4 Encoding Hard String Problems with Answer Set Programming

Further, we define the variables $T_{i,c} = 1$ to encode that $T[i] = c$, for $i \in [1..n], c \in \{S_1[i], \dots, S_m[i]\}$. To state that $T[i] = S_x[i]$, we want that, for a fixed position $i \in [1..n]$, only one $T_{i,c}$ is set:

$$[\mathcal{O}(n), \mathcal{O}(\min(m, \sigma))] \quad \forall i \in [1..n] : \sum_{c \in \Sigma_i} T_{i,c} = 1 \quad (\text{CSP1})$$

Next, we define the cost variables $C_{i,x}$ for all $i \in [1..n]$ and $x \in [1..m]$ with $C_{i,x}$ being set if $T[i] \neq S_x[i]$. Thus the Hamming distance between T and S_x is $\text{dist}_{\text{ham}}(T, S_x) = \sum_{i \in [1..n]} C_{i,x}$. Therefore:

$$[\mathcal{O}(nm\sigma), \mathcal{O}(1)] \quad \forall i \in [1..n], c \in \Sigma_i, x \in [1..m] : T_{i,c} \wedge S_x[i] \neq c \implies C_{i,x} \quad (\text{CSP2})$$

A statement for setting $C_{i,x}$ to false is not needed as the optimizer will try to do so if it does not violate (CSP2). This is achieved by the following objective:

$$[\mathcal{O}(1), \mathcal{O}(mn)] \quad \text{minimize } \max_{x \in [1..m]} \sum_{i \in [1..n]} C_{i,x} \quad (\text{CSP3})$$

Complexities. We have $\mathcal{O}(n\sigma)$ selectable variables ($T_{i,c}$), $\mathcal{O}(nm)$ helper variables ($C_{i,x}$), $\mathcal{O}(nm\sigma)$ clauses (CSP2). The largest clause contains $\mathcal{O}(mn)$ variables (CSP3).

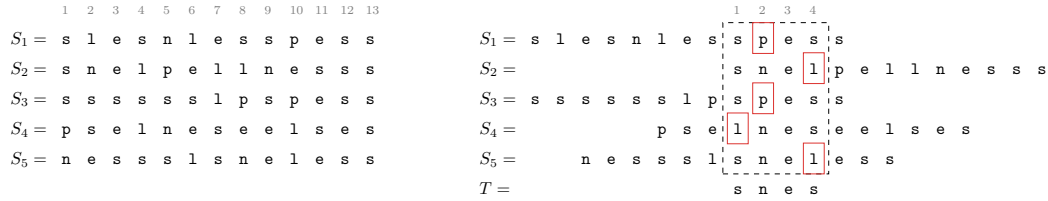
Implementation. Our implementation in ASP is given in Listing 1. In all listings, the percent sign % introduces a comment until the end of the line, which we use to refer to the MAX-SAT equation that is represented by the respective line of code. Red curly arrows symbolize line breaks. If not otherwise stated, in all code listings onwards, we assume that the input is of the form $\mathbf{s}(X, I, C)$, denoting that $S_x[I] = C \in \Sigma$. We use the helper variables $\mathbf{mat}(X, I)$ to denote the existence of $S_x[I]$. For encoding (CSP3) in ASP, we additionally define the helper variables \mathbf{cost} and \mathbf{mcost} encoding $\sum_{i \in [1..n]} C_{i,x}$ and $\max_{x \in [1..m]} \mathbf{cost}(x)$, respectively. The `#show` directives at the end define the variables the solver has to output. The evaluation for our implementation is deferred until we have introduced the CSS problem, which we conjointly evaluate in Sect. 4.2.

■ **Listing 1** ASP for CSP (Sect. 3).

```
mat(X,I) :- s(X,I,_).
1 {t(I,C) : s(_,I,C)} 1 :- mat(_,I). %(CSP1)
c(X,I) :- t(I,C), s(X,I,A), C != A. %(CSP2)
cost(X,C) :- C = #sum {1,I : c(X,I)}, mat(X,_). %(CSP3)
mcost(M) :- M = #max {C : cost(_,C)}.
#minimize {M : mcost(M)}.
#show t/2. #show mcost/1. #show cost/2.
```

4 Closest Substring (CSS)

For the CSS problem, we additionally require a parameter λ as input to specify the length of the output string T . CSS asks for the string T with $|T| = \lambda$ such that $\max_{x \in [1..m]} \text{dist}_\lambda(S_x, T)$ is minimal, where $\text{dist}_\lambda(S_x, T) := \min_{i \in [1..n-\lambda+1]} \text{dist}_{\text{ham}}(S_x[i..i+\lambda-1], T)$ is the number of mismatches we need to be able to detect T via approximate pattern matching in S_x with $\text{dist}_\lambda(S_x, T)$ mismatches. An example is shown in Fig. 2.



■ **Figure 2** Example for CSS (Sect. 4) with $n = 13$ and query length $\lambda = 4$. The input is shown on the left figure. We can observe in the right figure that $T = \text{snes}$ is the CSS having one mismatch with each of the input strings in the Hamming distance by horizontally shifting the input strings.

Related Work. The decision problem for δ mismatches is also called δ -MISMATCH problem. Gramm et al. [32, Theorem 2] solved the decision problem in $\mathcal{O}(m\lambda + (n - \lambda)m\delta^{\delta+1})$ time. Marx [46] showed that CSS can be solved in $\mathcal{O}(\sigma^{\delta(\lg \delta + 2)}(nm)^{\mathcal{O}(\lg \delta)})$ or $\mathcal{O}((\sigma\delta)^{\mathcal{O}(m\delta)}(nm)^{\mathcal{O}(\log \log m)})$ time. A survey on further results can be found in [31]. With respect to other optimization approaches, we are aware of a genetic algorithm [47].

4.1 MAX-SAT encoding

Following [32, Section 3.3], we reduce CSS to CSP by selecting shifts $d_x \in [0..n - \lambda]$ of each input string S_x such that the CSP of $\{S_1[1 + d_1.. \lambda + d_1], \dots, S_m[1 + d_m.. \lambda + d_m]\}$ is a solution of CSS if we take the minimum distance over all shifts d_x .

In what follows, we represent the shifts by a matrix of selectable Boolean variables of size $\mathcal{O}(m(n - \lambda))$. We redefine the alphabet for the i -th character to be $\Sigma_i := \{S_1[i + d_1], \dots, S_m[i + d_m]\}$. We define the variables $T_{i,c}$ and $C_{i,x}$ as before. We copy (CSP1) as it is since it only states from which string S_x we select the i -th character of T , except that we have $\mathcal{O}(\lambda)$ instead of $\mathcal{O}(n)$ clauses since $|T| = \lambda$. The major difference is that for checking equality, we must add the offsets and obtain the following modification of (CSP2):

$$[\mathcal{O}(\lambda nm\sigma), \mathcal{O}(1)] \quad \forall i \in [1..\lambda], c \in \Sigma_i, x \in [1..m] : T_{i,c} \wedge S_x[i + d_x] \neq c \implies C_{i,x} \quad (\text{CSS2})$$

The additional n -term in the complexity stems from the fact that the offsets d_x are represented as a two-dimensional binary array. The other equations as well as the objective are kept in the same way.

Complexities. We have $\mathcal{O}(\lambda\sigma + m(n - \lambda))$ selectable variables ($T_{i,c}$ and d_x), $\mathcal{O}(\lambda m)$ helper variables ($C_{i,x}$), $\mathcal{O}(\lambda mn\sigma)$ clauses. The largest clause has size $\mathcal{O}(\lambda m)$. Our implementation in ASP is given in Listing 2, where we expect an additional input of the form `#const lambda= λ .` for the requested substring length λ .

■ **Listing 2** ASP for CSS (Sect. 4).

```

mat(X,I) :- s(X,I,_).
1 {d(X,D) : D = 0..n-lambda} 1 :- mat(X,0).
sigma(I,C) :- s(X,J,C), d(X,D), J-D >= 0, I = J-D.
1 {t(I,C) : sigma(I,C)} 1 :- mat(_,I), I < lambda. %(CSP1)
c(X,I) :- t(I,C), s(X,J,A), d(X,D), I+D == J, I < lambda, A != C. %(CSS2)
cost(X,C) :- C = #sum {1,I : c(X,I)}, mat(X,_). %(CSP3)
mcost(M) :- M = #max {C : cost(_,C)}.
#minimize {M : mcost(M)}.
#show t/2. #show mcost/1. #show cost/2.

```

■ **Table 1** Evaluation for the CLOSEST STRING PROBLEM (CSP) for $\lambda = 0$ and CLOSEST SUBSTRING PROBLEM (CSS) for $\lambda > 0$. The column *dist* shows the maximum Hamming distance of the reported string to all input strings. The column *rules* is the number of created SAT rules, *vars* is the number of variables, and *choices* is the number of choices or configurations the solver or brute-force algorithm tries. Reported times are in seconds ([s]).

file	λ	dist	ASP				brute-force	
			rules	vars	choices	time [s]	choices	time [s]
s05m09n009i0	0	6	1288	321	725	0.01	640 000	5.47
s05m09n009i0	7	4	1932	1122	1663	0.02	78 125	1.96
s05m09n009i0	8	5	1764	969	3666	0.05	390 625	7.29
s05m09n009i0	9	6	1427	330	676	0.01	1 953 125	21.59
s06m07n009i1	0	7	1078	268	1767	0.02	768 000	5.12
s06m07n009i1	7	4	1765	1069	3235	0.04	279 936	5.49
s06m07n009i1	8	5	1550	868	1314	0.02	1 679 616	24.45
s06m07n009i1	9	7	1194	275	2058	0.02	10 077 696	87.80
s06m08n009i0	0	6	1191	295	1074	0.01	750 000	5.67
s06m08n009i0	7	5	1907	1147	4266	0.05	279 936	6.23
s06m08n009i0	8	6	1698	954	4021	0.05	1 679 616	27.90
s06m08n009i0	9	6	1319	303	1273	0.01	10 077 696	100.23
s06m08n009i1	0	7	1248	299	2378	0.02	1 800 000	13.63
s06m08n009i1	7	5	1971	1203	4834	0.07	279 936	6.27
s06m08n009i1	8	6	1770	1012	5093	0.08	1 679 616	27.77
s06m08n009i1	9	7	1380	307	2163	0.02	10 077 696	99.98
s06m08n009i2	0	7	1248	299	2128	0.02	1 800 000	13.61
s06m08n009i2	7	5	1907	1147	5556	0.07	279 936	6.28
s06m08n009i2	8	6	1698	955	5552	0.08	1 679 616	27.91
s06m08n009i2	9	7	1380	307	2210	0.02	10 077 696	99.84
s06m09n009i0	0	7	1303	322	1837	0.02	800 000	6.81
s06m09n009i0	7	4	2142	1301	4331	0.05	279 936	7.02
s06m09n009i0	8	5	1920	1093	5334	0.08	1 679 616	31.38
s06m09n009i0	9	7	1443	331	1962	0.02	10 077 696	111.16
s06m09n009i1	0	7	1396	328	1849	0.02	2 700 000	22.97
s06m09n009i1	7	5	2177	1334	5341	0.07	279 936	7.04
s06m09n009i1	8	6	1920	1100	5693	0.10	1 679 616	31.20
s06m09n009i1	9	7	1542	337	1746	0.02	10 077 696	110.05
s06m09n009i2	0	6	1336	324	1874	0.02	1 080 000	9.07
s06m09n009i2	7	4	2177	1333	3706	0.05	279 936	6.92
s06m09n009i2	8	5	1946	1114	4565	0.06	1 679 616	30.52
s06m09n009i2	9	6	1478	333	1920	0.02	10 077 696	107.62

4.2 Evaluation of csp and css

Although there are efficient heuristics like choosing a majority string [8], we compared our ASP encoding for CSP to a basic brute-force algorithm that enumerates all possible assignments for the characters of the closest substring. The number of possible configurations for T is $c_S := \prod_{i=1}^n \sigma_i \in \mathcal{O}(\min(\sigma^n), m^n)$ dependent on the shape of the strings in \mathcal{S} . A brute-force algorithm trying each configuration spends $\mathcal{O}(c_S n m)$ time on computing the Hamming distances of the resulting string T with all strings of \mathcal{S} .

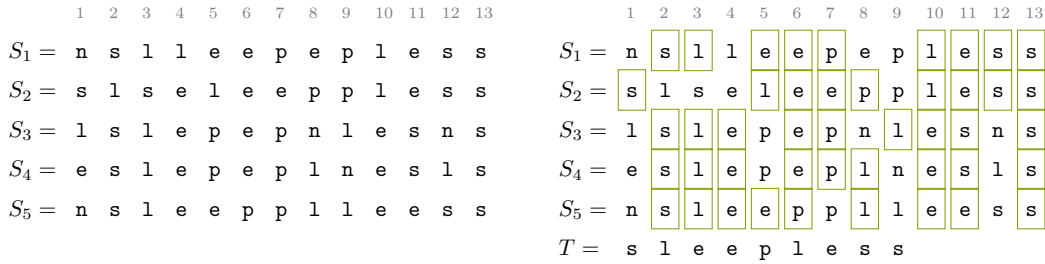
This algorithm can be easily adopted for CSS. For that, we consider all possible offsets of the input strings like in the ASP encoding. Hence, the number of configurations is the number of configurations for the CSP instance, multiplied by $(n - \lambda)^m$ for each possible offset value. If λ is small, then it suffices to compute all configurations of T , which are σ^λ many, and compute the Hamming distances in $\mathcal{O}(\lambda m)$ time for each such configuration. We implemented the former brute-force approach, whose time complexity grows exponentially with all parameters σ , n , and m , for randomly generated strings. We can observe this case in Table 1, where the ASP implementation outperforms the brute-force approach.

■ **Table 2** Evaluation of the CLOSEST STRING problem (SCP) on datasets provided by Torres and Hoshino [54]. The column *distance* is the maximal Hamming distance of the output to any of the input strings.

file	distance	rules	vars	choices	time [s]
rand-4-150-150-5-2	2	31 329	12 942	19	0.06
rand-4-50-50-5-2	2	10 529	4342	21	0.015
rand-4-100-100-5-2	2	20 929	8642	24	0.031
rand0-2-10-10-20-5	4	4286	842	43	0.011
rand0-2-10-10-20-4	4	4286	842	60	0.011
rand0-4-10-10-20-5	4	4887	1179	65	0.012
rand0-2-10-10-20-3	5	4474	848	72	0.012
rand-20-50-50-5-2	2	17 323	4549	78	0.018
rand-20-150-150-5-2	2	55 819	13 197	100	0.082
rand0-20-10-10-20-5	4	5573	1359	121	0.013
rand-20-100-100-5-2	2	37 213	8894	129	0.041
rand-4-150-150-5-1	5	31 329	12 942	189	0.117
rand-4-50-50-5-1	5	10 529	4342	199	0.021
rand0-2-10-10-20-2	6	4474	922	202	0.014
rand-4-100-100-5-1	5	20 929	8642	248	0.056
rand0-2-10-10-20-1	7	4474	922	265	0.015
rand-4-50-50-10-2	5	12 279	3082	494	0.035
rand-20-100-100-5-1	5	37 213	8894	501	0.068
rand-20-150-150-5-1	5	55 819	13 197	525	0.131
rand-20-50-50-5-1	5	18 595	4585	548	0.029
rand0-4-10-10-20-4	5	5008	1264	555	0.018
rand0-4-10-10-20-3	5	4869	1241	627	0.019
rand0-20-10-10-20-4	5	5800	1384	998	0.027
rand-20-50-50-10-2	5	26 397	3511	1057	0.053
rand0-20-10-10-20-3	6	6520	1477	2369	0.058
rand-20-50-50-15-2	7	40 426	5288	3512	0.235
rand-4-50-50-15-2	7	18 454	4622	4192	0.251
rand-4-50-50-10-1	8	12 279	3082	7320	0.343
rand0-4-10-10-20-2	8	5255	1334	18 095	0.373
rand-20-50-50-10-1	9	28 623	3574	23 622	1.255
rand0-20-10-10-20-2	9	6964	1540	48 538	1.265
rand-4-50-50-20-2	10	24 654	6162	98 610	12.379
rand-4-50-50-15-1	11	18 454	4622	119 367	8.76
rand-20-50-50-20-2	10	53 844	7047	168 793	28.348
rand0-4-10-10-20-1	11	5404	1360	770 565	19.168
rand-20-50-50-15-1	12	42 864	5357	2 716 507	358.345
rand-4-50-50-20-1	15	24 654	6162	39 265 111	7009.909

In Table 2, we depict the results of a larger evaluation on the datasets provided in [54]⁵, which are also used in [16, 43]. We kept their file naming, which is the format `rand- σ - $\frac{m}{2}$ - $\frac{m}{2}$ - n - i` , where i is an iteration counter to have multiple files with the same characteristics (m , n , and σ). The prefix `rand` can be followed by a zero. We observe that larger distances correlate with the number of choices, affecting the overall running time. Even for large inputs with short distances like the dataset `rand-4-150-150-5-1`, the running time is short.

⁵ <https://github.com/jeanpttorres/dssp>



■ **Figure 3** Example for LCS (Sect. 5) with $n = 13$. The input is shown on the left figure. In the right figure, we highlighted the subsequences matching $T = \text{sleepless}$ by surrounding the respective characters with boxes in each input string. Here, $T = \text{sleepless}$ is the LCS of all input strings.

5 Longest Common Subsequence (LCS)

The LCS problem asks for the longest string T such that T is a subsequence of S_x for every $x \in [1..m]$. See Fig. 3 for an example.

Existence. A solution exists if all strings share at least one common character in the alphabet.

Related Work. Maier [45] showed that LCS is NP-hard for $\sigma \geq 2$, and the same holds for SCS with $\sigma \geq 5$. Later, Blin et al. [5] gave a proof that LCS stays NP-hard even if the input strings are well-compressible with the run-length encoding. For exact algorithms, we can extend the classic dynamic programming (DP) algorithm of Wagner and Fischer [56] to m strings, which then takes $\mathcal{O}(n^m)$ time. Irving and Fraser [36] gave two algorithms running in $\mathcal{O}(mn(n-\ell)^{m-1})$ or $\mathcal{O}(m\ell(n-\ell)^{m-1} + m\sigma n)$ time, where ℓ is the length of the output. This result implies that LCS is FPT in m and $n - \ell$. Bulteau et al. [10] improved the result of [36] with an algorithm running in $\mathcal{O}((n - \ell + 1)^{n-\ell+1}mn)$ time, which is an FPT in the number of deletions $n - \ell$. Finally, there is a genetic algorithm [34] and an ant colony optimization algorithm [50].

5.1 MAX-SAT encoding

Our idea is to select a subsequence T_x for each input string S_x and maximize the length of T_x under the constraint that all T_x 's have to be equal. The subsequence T_x of S_x is given by a sequence of indices $i_1 < \dots < i_{|T_x|}$ such that $S_x[i_1] \dots S_x[i_{|T_x|}] = T_x$. We can encode the subsequences T_x by the selectable variables $C_{x,\ell,i}$ encoding whether $T_x[\ell] = S_x[i]$, for each $x \in [1..m], \ell \in [1..n]$. We make use of $C_{x,\ell,i}$ as follows. First, for each $T_x[\ell]$, we define the range for the selectable variables $C_{x,\ell,i}$.⁶

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : \sum_{i \in [\ell..n]} C_{x,\ell,i} \geq 0 \tag{LCS1}$$

⁶ Logically, we would expect in (LCS1) a “ ≤ 1 ” instead of a “ ≥ 0 ”. However, the former suffices together with the following constraints and is cheaper than “ ≤ 1 ”.

If we have selected $T_x[\ell]$ to be $S_x[i]$, then $T_x[\ell-1]$ must be a character chosen in $S_x[1..i-1]$:

$$[\mathcal{O}(n^2m), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [2..n], i \in [\ell..n] : \\ C_{x,\ell,i} \implies \sum_{j \in [1..i-1]} C_{x,\ell-1,j} = 1 \quad (\text{LCS2})$$

Next, we define the helper variables $V_{x,\ell}$ encoding whether T_x has a length of at least ℓ , for each $x \in [1..m], \ell \in [1..n]$. If we have selected a character for $T_x[\ell]$ via $C_{x,\ell,i}$, then we set $V_{x,\ell}$ to true to specify that T_x has a length of at least ℓ .

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : \bigvee_{i \in [1..n]} C_{x,\ell,i} \implies V_{x,\ell} \quad (\text{LCS3})$$

We now restrict all T_x 's to be of equal length, which we do in a Round-Robin encoding:

$$[\mathcal{O}(nm), \mathcal{O}(1)] \quad \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies V_{(x+1) \bmod n, \ell} \quad (\text{LCS4})$$

Here, $\bmod n : \{1, 2, \dots\} \rightarrow [1..n]$ is the modulo operation with $n \bmod n = n$ and $(n+1) \bmod n = 1$. To achieve that all T_x store the same characters, we use the following constraint.

$$[\mathcal{O}(n^3m), \mathcal{O}(1)] \quad \forall x \in [1..m], \ell \in [1..n], i, j \in [1..n] : \\ C_{x,\ell,i} \wedge C_{(x+1) \bmod n, \ell, j} \implies S_x[i] = S_{(x+1) \bmod n}[j] \quad (\text{LCS5})$$

Finally, we enforce that we need to select a position for $T_x[\ell]$ if $V_{x,\ell}$ is set:

$$[\mathcal{O}(nm), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n] : V_{x,\ell} \implies \bigvee_{i \in [\ell..n]} C_{x,\ell,i} \quad (\text{LCS6})$$

Alternatively to (LCS5) and (LCS6), we can state that the next subsequence must select one of the text positions j for $T_{x+1}[\ell]$ with $S_{x+1}[j] = S_x[i]$.

$$[\mathcal{O}(n^2m), \mathcal{O}(n)] \quad \forall x \in [1..m], \ell \in [1..n], i \in [1..n] : \\ C_{x,\ell,i} \implies \sum_{j: S_x[i] = S_{(x+1) \bmod n}[j]} C_{(x+1) \bmod n, \ell, j} = 1 \quad (\text{LCS5'})$$

Finally, we formulate our optimization problem as

$$[\mathcal{O}(1), \mathcal{O}(n)] \quad \text{maximize} \quad \sum_{\ell \in [1..n]} V_{1,\ell} \quad (\text{LCS7})$$

Complexities. Our implementation in ASP is given in Listing 3. We have $\mathcal{O}(mn^2)$ selectable variables ($C_{x,\ell,i}$), $\mathcal{O}(mn)$ helper variables ($V_{x,\ell}$), and $\mathcal{O}(n^2m)$ clauses (LCS5'). The largest clause has $\mathcal{O}(n)$ variables. An improvement for short LCS solutions could be to encode the existence problem for a fixed length λ in ASP such that we have $\mathcal{O}(m\lambda)$ selectable variables for encoding T_x , and call this encoding while varying λ to find the largest value for λ admitting a solution.

■ **Table 3** Evaluation of the LONGEST COMMON SUBSEQUENCE problem (LCS).

file	length	ASP				brute-force	
		rules	vars	choices	time [s]	choices	time [s]
s02m11n023i1	10	166 538	21 494	23 617	1.00	8 388 608	47.29
s02m10n023i2	10	151 627	19 540	34 146	1.02	8 388 608	43.35
s02m09n023i1	11	137 112	17 586	10 964	0.61	8 388 608	39.73
s03m08n023i1	8	138 002	15 632	4831	0.40	8 388 608	39.20
s04m09n023i1	6	162 617	17 586	3927	0.39	8 388 608	39.07
s03m11n023i2	8	188 366	21 494	20 672	1.18	8 388 608	38.99
s03m08n023i2	7	136 795	15 632	11 046	0.63	8 388 608	38.54
s03m07n023i2	9	119 551	13 678	5945	0.40	8 388 608	37.59
s04m11n023i1	6	197 886	21 494	5767	0.58	8 388 608	37.06
s03m08n023i0	8	136 968	15 632	6301	0.45	8 388 608	37.05
s03m08n022i0	8	120 880	14 256	5467	0.37	4 194 304	17.87
s03m08n022i1	7	120 416	14 256	3970	0.32	4 194 304	17.69
s02m11n022i1	11	146 880	19 602	11 779	0.53	4 194 304	17.61
s03m07n022i2	9	105 785	12 474	2763	0.24	4 194 304	17.34
s03m11n022i2	7	165 908	19 602	7974	0.63	4 194 304	17.31
s04m11n022i1	6	175 570	19 602	8045	0.58	4 194 304	17.02
s02m09n022i1	12	121 186	16 038	6522	0.27	4 194 304	16.85
s03m08n022i2	8	120 313	14 256	4442	0.34	4 194 304	16.80
s04m09n022i1	6	143 324	16 038	5791	0.45	4 194 304	16.72
s02m10n022i2	10	135 128	17 820	9640	0.47	4 194 304	15.94

■ **Listing 3** ASP for LCS (Sect. 5).

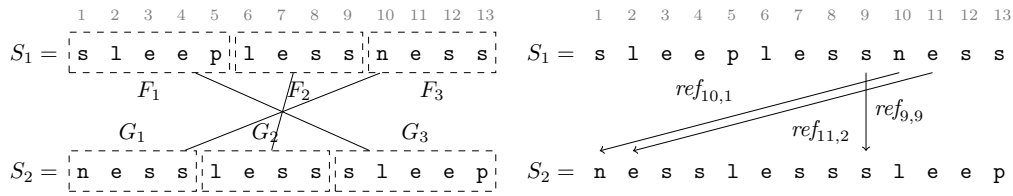
```

mat(X,I) :- s(X,I,_).
0 {c(X,L,I) : mat(X,I), I >= L} :- mat(X,L). %(LCS1)
1 {c(X,L,J) : J < I, mat(X,J)} 1 :- c(X,L+1,I), mat(X,L), mat(X,L+1). %(
  ↪ LCS2)
v(X,L) :- c(X,L,I), mat(X,I), mat(X,L). %(LCS3)
v(X+1,L) :- v(X,L), mat(X,L), mat(X+1,L). %(LCS4)
v(0,L) :- v(m-1,L).
:- c(X+1,L,J), c(X,L,I), s(X,I,D), not s(X+1,J,D). %(LCS5)
:- c(0,L,J), c(m-1,L,I), s(m-1,I,D), not s(0,J,D).
1 {c(X,L,I) : mat(X,I), I >= L} :- v(X,L). %(LCS6)
#maximize {1,L : v(0,L)}. %(LCS7)
#show c/3.

```

5.2 Evaluation

A DP approach would need $\mathcal{O}(n^m)$ time (cf. [15, Chapter IV, Section 15.4] for a textbook reference). Here, we stick to a trivial approach that tries all distinct subsequences of the first string S_1 , and for each such subsequence we check whether it is a subsequence of all other input strings. The number of these subsequences is at most $2^n - 1$. If we select these subsequences with respect to their lengths, starting with the longest possible one, we can terminate whenever the selected subsequence is found in all other strings. In the worst



■ **Figure 4** Example for MCSP (Sect. 6) with $n = 13$. We can factorize $S_1 = F_1F_2F_3$ into three factors, with $F_1 = G_3$, $F_2 = G_2$ and $F_3 = G_1$ such that $S_2 = G_1G_2G_3$. Hence, the solution for this example is a partition of length three. On the right is a partial assignment of the variable ref based on this partition, where ref induces a factor starting at position 10 in S_1 .

case, the time complexity of this approach grows exponentially in n , but only linearly in m , independent of the alphabet size. We therefore restrict our evaluation in Table 3 to scaling n while keeping the other parameters unchanged. Like in Sect. 4.2, the ASP implementation outperforms the brute-force approach. However, a DP implementation might outperform the ASP implementation by re-using memoized results.

6 Minimum Common String Partition (MCSP)

For the special case of $m = 2$ input strings S_1 and S_2 , the MCSP problem, introduced by Goldstein et al. [29] and Swenson et al. [52], asks, for a given $z \in [1..n]$, a factorization of S_1 into $S_x = F_1 \cdots F_z$ and a permutation π of $[1..z]$ such that $F_{\pi(1)} \cdots F_{\pi(z)} = S_2$. The optimization problem is to find the smallest z for which a solution exists. We give an example in Fig. 4.

Existence. A sufficient condition for whether a solution for any $z \in [1..n]$ exists is to check that the Parikh vectors of S_1 and S_2 are the same, such that at least a permutation on $[1..n]$ exist to rearrange the characters of S_1 to form S_2 .

Related Work. While introducing MCSP, Goldstein et al. [29] also showed that it is NP-hard. Bulteau and Komusiewicz [11] showed that MCSP is FPT in z . For constant alphabets ($\sigma = \mathcal{O}(1)$), Cygan et al. [17] presented an exact algorithm running in $2^{\mathcal{O}(n \lg \lg n / \lg n)}$ time. Recently, Chromý and Sinnl [14] studied a DP algorithm. It is known that MCSP can be tackled by probabilistic tree searches [7], ILP formulations [6, 23], and an ant colony optimization algorithm [22].

6.1 MAX-SAT encoding

We adapt the MAX-SAT encoding of Bannai et al. [1] for the shortest bidirectional macro scheme problem [51]. To this end, we define the sets $\mathcal{M}_i := \{j \in [1..n] \mid S_1[i] = S_2[j]\} \subset [1..n]$ specifying the positions in S_2 that match with $S_1[i]$. In what follows, we make use of the following selectable Boolean variables:

- p_i or q_i encode if $S_1[i]$ or $S_2[i]$ is the start of a factor, respectively, for $i \in [1..n]$.
- $ref_{i \rightarrow j}$ encodes whether position i of S_1 references position j of S_2 , and vice versa, for $i \in [1..n]$ and $j \in \mathcal{M}_i$.

17:12 Encoding Hard String Problems with Answer Set Programming

We have $\mathcal{O}(n^2)$ Boolean variables, which we use as follows. On the one hand, each position in S_1 has exactly one reference:

$$[\mathcal{O}(n), \mathcal{O}(n)] \quad \forall i \in [1..n] : \sum_{j \in \mathcal{M}_i} \text{ref}_{i \rightarrow j} = 1 \quad (\text{MCSP1})$$

On the other hand, each position in S_2 has exactly one reference:

$$[\mathcal{O}(n), \mathcal{O}(n)] \quad \forall j \in [1..n] : \sum_{i \in [1..n]} \text{ref}_{i \rightarrow j} = 1 \quad (\text{MCSP2})$$

In what follows, we add implications for the factor starting positions that are due to how we set the references. First, a factor starts always at the first text position, so p_1 and q_1 are always true. If $S_1[i]$ references $S_2[j]$ and i is a factor starting position of S_1 , so is j for S_2 .

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [1..n], j \in \mathcal{M}_i : p_i \wedge \text{ref}_{i \rightarrow j} \implies q_j \quad (\text{MCSP3})$$

Next, if $S_1[i]$ references $S_2[j]$ and j is a factor starting position of S_2 , so is i for S_1 . We only have to check that condition for q_1 since all other constraints set p_i and constraint (MCSP3) then implies that q_j has to be set.

$$[\mathcal{O}(n), \mathcal{O}(1)] \quad \forall i \in [1..n] : q_1 \wedge \text{ref}_{i \rightarrow 1} \implies p_i \quad (\text{MCSP4})$$

Another condition is that if the previous text positions have mismatching characters, we cannot extend the factor to the left.

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [1..n], j \in \mathcal{M}_i \text{ with } S_1[i-1] \neq S_2[j-1] : \text{ref}_{i \rightarrow j} \implies p_i \quad (\text{MCSP5})$$

Even if the previous characters match, when the reference of the previous text positions is different, we need to make a factor starting position:

$$[\mathcal{O}(n^2), \mathcal{O}(1)] \quad \forall i \in [2..n], \forall j \in \mathcal{M}_i \text{ such that } j > 1 \text{ and } S_2[i-1] = S_2[j-1] : \\ \neg \text{ref}_{i-1 \rightarrow j-1} \wedge \text{ref}_{i \rightarrow j} \implies p_i \quad (\text{MCSP6})$$

$$[\mathcal{O}(1), \mathcal{O}(n)] \quad \text{Finally, we minimize } \sum_{i \in [1..n]} p_i \quad (\text{MCSP7})$$

Complexities. We have $\mathcal{O}(n^2)$ selectable variables, and $\mathcal{O}(n^2)$ clauses (MCSP3). The largest clause has $\mathcal{O}(n)$ variables (MCSP2). Our implementation in ASP is given in Listing 4. Note that we start counting at zero, so $p(0)$ is equivalent to setting p_1 to true. Instead of `mat` we use the helper variables `spos` and `tpos` denoting the existence of $S_1[i]$ and $S_2[i]$, respectively.

■ **Listing 4** ASP for MCSP (Sect. 6).

```
spos(I) :- s(0,I,_).
tpos(J) :- s(1,J,_).
p(0). q(0).
arc(I,J) :- s(0,I,C), s(1,J,C).
1 {ref(I,J) : arc(I,J)} 1 :- spos(I). %(MCSP1)
1 {ref(I,J) : arc(I,J)} 1 :- tpos(J). %(MCSP2)
q(J) :- p(I), ref(I,J). %(MCSP3)
p(I) :- q(1), ref(I,1). %(MCSP4)
p(I) :- ref(I,J), s(0,I-1,C), s(1,J-1,D), C != D. %(MCSP5)
p(I) :- not ref(I-1,J-1), ref(I,J). %(MCSP6)
#minimize {1,X : p(X)}. %(MCSP7)
#show ref/2. #show p/1. #show q/1.
```


■ **Table 4** Evaluation of the MINIMUM COMMON STRING PARTITION problem (MCSP). Note that the time for the ASP solution is in milliseconds. The column z denotes the number of factors of the returned partition.

file	z	ASP				brute-force	
		rules	vars	choices	time [ms]	choices	time [s]
2s03n009i2	4	443	124	25	1.0	986 409	6.24
2s02n009i0	4	586	165	61	2.0	986 409	6.31
2s02n009i1	4	586	165	59	2.0	986 409	6.43
2s03n009i0	6	426	124	52	1.0	986 409	6.44
2s03n009i1	2	367	116	30	1.0	986 409	6.49
2s02n009i2	6	521	149	39	1.0	986 409	6.95
2s03n010i1	4	604	162	67	2.0	9 864 100	68.81
2s02n010i0	4	510	213	108	2.0	9 864 100	70.92
2s03n010i0	6	484	147	37	1.0	9 864 100	71.04
2s03n010i2	4	584	164	47	2.0	9 864 100	71.38
2s02n010i2	4	637	189	77	2.0	9 864 100	73.78
2s02n010i1	3	639	187	103	2.0	9 864 100	74.28

■ **Table 5** Evaluation of the MINIMUM COMMON STRING PARTITION problem (MCSP) on prefixes of the SARS-CoV-2 dataset.

length	z	rules	vars	choices	time [s]
10	4	447	146	34	0.001
20	12	1273	445	269	0.003
30	14	2282	911	1951	0.017
40	16	3720	1685	4683	0.047
50	21	5468	2442	2 050 092	18.609
60	24	7451	3422	6 866 999	80.256

6.2 Evaluation

Without leveraging the actual contents of the characters like in our SAT formulation, a naive way is to factorize both strings S_1 and S_2 with factors of the same lengths, and check whether there exists a permutation such that we can match factors of S_1 with factors of S_2 . To this end, we iterate over the size z of the partition from 1 to n . For each $z \in [1..n]$, we partition S_1 into z factors $S_1 = F_1 \cdots F_z$. There are $\binom{n}{z}$ such ways to partition S_1 . For each permutation π_z on $[1..z]$, we define the factorization $G_1 \cdots G_z = S_2$ with $|G_x| = |F_{\pi(x)}|$ for all $x \in [1..z]$. If $G_x = F_{\pi(x)}$ for all $x \in [1..z]$, then we have found a solution, and terminate. The number of configurations is $\sum_{z=1}^n \binom{n}{z} z!$, and each check takes $\mathcal{O}(n)$ time. Like the brute-force approach for LCS (Sect. 5.2), this approach has an exponential dependency on the text length n . In Table 4, we observe that specifying the choices for the references for each position individually (as we do in our ASP encoding) reduces the number of choices significantly when compared to the choices the brute-force algorithm processes.

Since our ASP encoding for MCSP seems quite efficient, we subsequently performed a benchmark on real data. In detail, we conducted an experiment by scaling the prefix length of a given input sequence, and report results in Table 5. For that, we used the SARS-CoV-2

<table style="border-collapse: collapse;"> <tr><td style="padding-right: 5px;">1</td><td style="padding-right: 5px;">2</td><td style="padding-right: 5px;">3</td><td style="padding-right: 5px;">4</td><td style="padding-right: 5px;">5</td></tr> <tr><td>$S_1 =$</td><td>e</td><td>e</td><td>p</td><td>l</td><td>e</td></tr> <tr><td>$S_2 =$</td><td>l</td><td>e</td><td>s</td><td>s</td><td>n</td></tr> <tr><td>$S_3 =$</td><td>p</td><td>l</td><td>e</td><td>s</td><td>s</td></tr> <tr><td>$S_4 =$</td><td>s</td><td>l</td><td>e</td><td>e</td><td>p</td></tr> <tr><td>$S_5 =$</td><td>s</td><td>n</td><td>e</td><td>s</td><td>s</td></tr> </table>	1	2	3	4	5	$S_1 =$	e	e	p	l	e	$S_2 =$	l	e	s	s	n	$S_3 =$	p	l	e	s	s	$S_4 =$	s	l	e	e	p	$S_5 =$	s	n	e	s	s	<table style="border-collapse: collapse;"> <tr><td style="padding-right: 5px;">1</td><td style="padding-right: 5px;">2</td><td style="padding-right: 5px;">3</td><td style="padding-right: 5px;">4</td><td style="padding-right: 5px;">5</td><td style="padding-right: 5px;">6</td><td style="padding-right: 5px;">7</td><td style="padding-right: 5px;">8</td><td style="padding-right: 5px;">9</td><td style="padding-right: 5px;">10</td><td style="padding-right: 5px;">11</td><td style="padding-right: 5px;">12</td><td style="padding-right: 5px;">13</td></tr> <tr><td>$S_1 =$</td><td></td><td></td><td>e</td><td>e</td><td>p</td><td>l</td><td>e</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>$S_2 =$</td><td></td><td></td><td></td><td></td><td></td><td></td><td>l</td><td>e</td><td>s</td><td>s</td><td>n</td><td></td></tr> <tr><td>$S_3 =$</td><td></td><td></td><td></td><td></td><td></td><td>p</td><td>l</td><td>e</td><td>s</td><td>s</td><td></td><td></td></tr> <tr><td>$S_4 =$</td><td>s</td><td>l</td><td>e</td><td>e</td><td>p</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>$S_5 =$</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>s</td><td>n</td><td>e</td><td>s</td><td>s</td></tr> <tr><td>$T =$</td><td>s</td><td>l</td><td>e</td><td>e</td><td>p</td><td>l</td><td>e</td><td>s</td><td>s</td><td>n</td><td>e</td><td>s</td><td>s</td><td></td></tr> </table>	1	2	3	4	5	6	7	8	9	10	11	12	13	$S_1 =$			e	e	p	l	e						$S_2 =$							l	e	s	s	n		$S_3 =$						p	l	e	s	s			$S_4 =$	s	l	e	e	p								$S_5 =$										s	n	e	s	s	$T =$	s	l	e	e	p	l	e	s	s	n	e	s	s	
1	2	3	4	5																																																																																																																															
$S_1 =$	e	e	p	l	e																																																																																																																														
$S_2 =$	l	e	s	s	n																																																																																																																														
$S_3 =$	p	l	e	s	s																																																																																																																														
$S_4 =$	s	l	e	e	p																																																																																																																														
$S_5 =$	s	n	e	s	s																																																																																																																														
1	2	3	4	5	6	7	8	9	10	11	12	13																																																																																																																							
$S_1 =$			e	e	p	l	e																																																																																																																												
$S_2 =$							l	e	s	s	n																																																																																																																								
$S_3 =$						p	l	e	s	s																																																																																																																									
$S_4 =$	s	l	e	e	p																																																																																																																														
$S_5 =$										s	n	e	s	s																																																																																																																					
$T =$	s	l	e	e	p	l	e	s	s	n	e	s	s																																																																																																																						

■ **Figure 5** Example for SCS (Sect. 7) with $n = 5$. The input is shown on the left figure. By the right figure, the SCS is $T = \text{sleeplessness}$, where we shifted the input strings to match their occurrences in T .

reference in FASTA format introduced in the analysis of Farkas et al. [21]⁷, after removing the header line and the newline characters. For each extracted prefix of this FASTA file, we created an instance for MCSP, where the second string is a random permutation of the original prefix. We can observe in Table 5 that the output size z exponentially correlates with the number of choices and the running time.

7 Shortest Common Superstring (SCS)

The SCS problem asks for the shortest string T such that S_x is a substring of T , for all $x \in [1..m]$. Figure 5 shows an example.

Existence. A trivial common superstring is the concatenation $S_1 \cdots S_m$. Permuting the strings and removing overlapping parts lead to the solution [25].

Related Work. Gallant et al. [25] showed that SCS is NP-hard for $n \geq 3$ with respect to the number of strings m and unbounded alphabet size, but can be solved in linear time if $n \leq 2$. For binary alphabet $\sigma = 2$, they showed that the problem is still NP-hard for $n = \Omega(\log(nm))$. It is known that SCS can be solved with neural networks [44] and genetic algorithms [30]. Most research on SCS is devoted to the analysis and improvement of the approximation algorithm presented by Tarhio and Ukkonen [53, Theorem 2.3]. This algorithm builds the so-called *overlap graph* of \mathcal{S} . The authors observed that a Hamiltonian path on the overlap graph [49] maximizing the weights of the selected edges solves SCS.

7.1 Reduction to Hamiltonian Path

We follow the idea of Tarhio and Ukkonen [53] by reducing SCS to the search of the Hamiltonian path maximizing the weights of the selected edges. The ASP encoding of finding a Hamiltonian cycle in an unweighted graph has already been studied in [42, 41]. We build on one of their approaches and extend it by maximizing the weights while omitting the weight of one edge to turn the cycle into a Hamiltonian path⁸. An *overlap graph* (\mathcal{S}, A, w) is a weighted

⁷ <https://github.com/cfarkas/SARS-CoV-2-freebayes>

⁸ We make a distinction between Hamiltonian path and Hamiltonian cycle in the sense that the cycle visits exactly one node twice.

directed graph, having the input strings \mathcal{S} as nodes and the arcs $A := \{(S_x, S_y) : x \neq y\}$. The weights are defined by a weight function $w : A \rightarrow [0..n]$ with $w(S_x, S_y) := \max\{|U| : U \text{ is suffix of } S_x \text{ and prefix of } S_y\}$. Hence, $w(S_x, S_y)$ is the number of overlapping characters, which we can omit if we want to build the superstring of S_x and S_y that starts with S_x . With respect to the overlap graph, a *path* is a sequence of strings, and a *Hamiltonian path* in the overlap graph is a path that visits each node exactly once, i.e., a permutation π of the list $[S_1, \dots, S_m]$. Our goal is to find a permutation that maximizes $\sum_{x=1}^{m-1} w(S_{\pi(x)}, S_{\pi(x+1)})$, i.e., to find the Hamiltonian path whose arcs have maximal weights in sum.

7.2 MAX-SAT encoding

We define the following $\mathcal{O}(m^2)$ Boolean variables:

- $cycle_{x,y}$ encoding whether we have the arc (S_x, S_y) in our Hamiltonian cycle, for $x, y \in [1..m]$;
- $reach_{x,y}$ encoding whether we can reach S_y from S_x by following the transitive closure of $cycle$, for $x, y \in [1..m]$;
- $start_x$ encoding whether our superstring starts with S_x , for $x \in [1..m]$.

First, we select arcs from the overlap graph for $cycle_{x,y}$. To this end, for each string S_x , we select exactly one out-going arc and one in-coming arc:

$$[\mathcal{O}(m), \mathcal{O}(m)] \quad \forall x \in [1..m] : \sum_{y=1}^m cycle_{x,y} = 1 \text{ and } \forall y \in [1..m] : \sum_{x=1}^m cycle_{x,y} = 1 \quad (\text{SCS1})$$

The transitive closure of $cycle$ can be encoded as follows. First we initialize $reach$ by the direct connections due to $cycle$.

$$[\mathcal{O}(m^2), \mathcal{O}(1)] \quad \forall x, y \in [1..m], x \neq y : cycle_{x,y} \implies reach_{x,y} \quad (\text{SCS2})$$

Next, if we can reach y from x , and there is an arc (y, z) , then we can reach z from x :

$$[\mathcal{O}(m^3), \mathcal{O}(1)] \quad \forall x, y, z \in [1..m], x \neq y \neq z : reach_{x,y} \wedge cycle_{y,z} \implies reach_{x,z} \quad (\text{SCS3})$$

To make the path selected by $cycle_{x,y}$ an Hamiltonian path, we want that all strings are connected via $reach$:

$$[\mathcal{O}(m^2), \mathcal{O}(1)] \quad \forall x, y \in [1..m], x \neq y : reach_{x,y} = 1 \quad (\text{SCS4})$$

For the Hamiltonian path it is left to select a designated start string⁹.

$$[\mathcal{O}(1), \mathcal{O}(m)] \quad \sum_{y=1}^m start_y = 1 \quad (\text{SCS5})$$

Finally, our objective is to maximize the weights on the path starting from $start_x$ of length m :

$$[\mathcal{O}(1), \mathcal{O}(m^2)] \quad \text{maximize} \quad \sum_{x,y \in [1..m] : cycle_{x,y} \wedge \neg start_y} w(x,y) \quad (\text{SCS6})$$

⁹ It actually suffices to check in (SCS4) that all strings can be reached from this start string, but doing so had a negative impact on the overall running time in the experiments.

17:16 Encoding Hard String Problems with Answer Set Programming

■ **Table 6** Evaluation of the SHORTEST COMMON SUPERSTRING problem (SCS). $|T|$ is the length of the SCS.

file	$ T $	ASP				brute-force	
		rules	vars	choices	time [s]	choices	time [s]
s02m10n008i0	42	2090	1416	198 756	3.58	10 240	0.02
s02m10n008i1	33	2206	1465	1 854 941	40.73	10 240	0.02
s02m10n008i2	39	2200	1464	1 401 686	29.49	10 240	0.02
s02m11n008i0	49	2639	1825	2 150 681	44.96	22 528	0.03
s02m11n008i1	35	2699	1861	6 652 411	154.48	22 528	0.03
s02m11n008i2	50	2611	1817	6 980 725	136.00	22 528	0.02

Complexities. We have $\mathcal{O}(m^2)$ selectable variables and $\mathcal{O}(m^3)$ clauses (SCS3). The largest clause has $\mathcal{O}(m^2)$ variables (SCS6). Our implementation in ASP is given in Listing 5. We expect an input of the form $w(X,Y,C)$ encoding the weight $w(X,Y) = C$. The helper variables `node` and `gain` define the nodes of the overlap graph and the value of the optimization argument in (SCS6), respectively.

■ **Listing 5** ASP for scs (Sect. 7).

```
node(X) :- w(X,_,_).
1 {cycle(X,Y) : w(X,Y,_)} 1 :- node(X). %(SCS1)
1 {cycle(X,Y) : w(X,Y,_)} 1 :- node(Y).
reach(X,Y) :- cycle(X,Y). %(SCS2)
reach(X,Z) :- reach(X,Y), cycle(Y,Z). %(SCS3)
:- not reach(X,Y), node(X), node(Y). %(SCS4)
1 {start(X) : node(X)} 1. %(SCS5)
gain(D) :- D = #sum {C,X : cycle(X,Y), w(X,Y,C), not start(Y)}. %(SCS6)
#maximize {D : gain(D)}.
#show cycle/2. #show start/1.
```

7.3 Evaluation

The overlap graph can be computed in $\mathcal{O}(nm + m^2)$ time [33]. Given the overlap graph, the easiest approach is to enumerate all $m!$ permutations, and compute the sum of the selected weights in $\Theta(m)$ time. The time bound can be improved by using a DP approach taking $\mathcal{O}(m^2 2^m)$ time¹⁰. In the experiments of Table 6, we use this DP approach as our brute-force solution. We observe that it outperforms our ASP implementation on all instances. That is due to the fact that (a) our ASP encoding does not make use of more information than the DP approach, and that (b) the number of choices in our encoding for the Hamiltonian path is prohibitively large. As a matter of fact, efficient SAT and ASP encodings for Hamiltonian cycles are actively studied, cf. [58] for SAT and [4] for ASP.

¹⁰<https://leetcode.com/problems/find-the-shortest-superstring/solutions/194891/official-solution/>

■ **Table 7** Encoding complexities of the studied problems. Columns *prob.*, *#sel. vars.*, *#h.vars.*, *#clauses* and *max. cl.* denote the problem name, the number of defined selectable variables, the number of helper variables, the number of clauses, and the maximum size a clause can have.

prob.	#sel.vars	#h.vars	#clauses	max cl.
CSP	$\mathcal{O}(n\sigma)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm\sigma)$	$\mathcal{O}(mn)$
CSS	$\mathcal{O}(\lambda\sigma + (n-\lambda)m)$	$\mathcal{O}(\lambda m)$	$\mathcal{O}(nm\sigma\lambda)$	$\mathcal{O}(\lambda m)$
LCS	$\mathcal{O}(n^2m)$	$\mathcal{O}(mn)$	$\mathcal{O}(n^2m)$	$\mathcal{O}(n)$
MCSP	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
SCS	$\mathcal{O}(m^2)$	$\mathcal{O}(1)$	$\mathcal{O}(m^3)$	$\mathcal{O}(m^2)$

8 Conclusion

We provided encodings in ASP for five prominent examples of NP-hard problems in the field of stringology. We summarized the complexities of the encodings in Table 7. We observed that, on the one hand, by leveraging characteristics of the input data such as for MCSP, our solution is far superior than simple brute-force approaches that omit those characteristics. On the other hand, for SCS, we observed that if the problem can be easily reduced to instances of problems like finding a Hamiltonian path, DP approaches are already efficient enough to find the answer faster than an ASP solver. It therefore depends on the nature of the problem we study for whether an application of an ASP solver makes sense. Nevertheless, the programming in ASP is highly expressive as can be seen by the short program codes in Listings 1–5, and therefore can be understood as a tool for rapid prototyping. Other advantages of ASP solvers like clingo are that they can work in parallel, report approximate solutions when reaching a given timeout, and enumerate all solutions, provided that the specified constraints do not exclude one of them. An evaluation of those features is left as future work since it would go beyond the scope of this paper.

References

- 1 Hideo Bannai, Keisuke Goto, Masakazu Ishihata, Shunsuke Kanda, Dominik Köppl, and Takaaki Nishimoto. Computing NP-hard repetitiveness measures via MAX-SAT. In *Proc. ESA*, volume 244 of *LIPICs*, pages 12:1–12:16, 2022. doi:10.4230/LIPICs.ESA.2022.12.
- 2 Manu Basavaraju, Fahad Panolan, Ashutosh Rai, M. S. Ramanujan, and Saket Saurabh. On the kernelization complexity of string problems. *Theor. Comput. Sci.*, 730:21–31, 2018. doi:10.1016/j.tcs.2018.03.024.
- 3 Riccardo Bertolucci, Alessio Capitanelli, Carmine Dodaro, Nicola Leone, Marco Maratea, Fulvio Mastrogiovanni, and Mauro Vallati. An ASP-based framework for the manipulation of articulated objects using dual-arm robots. In *Proc. LPNMR*, volume 11481 of *LNCS*, pages 32–44, 2019. doi:10.1007/978-3-030-20528-7_3.
- 4 Manuel Bichler, Bernhard Bliem, Marius Moldovan, Michael Morak, and Stefan Woltran. Treewidth-preserving modeling in ASP. Technical Report DBAI-TR-2016-97, Technische Universität Wien, 2016.
- 5 Guillaume Blin, Laurent Bulteau, Minghui Jiang, Pedro J. Tejada, and Stéphane Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Proc. CPM*, volume 7354 of *LNCS*, pages 138–148, 2012. doi:10.1007/978-3-642-31265-6_11.
- 6 Christian Blum. ILP-based reduced variable neighborhood search for large-scale minimum common string partition. *Electron. Notes Discret. Math.*, 66:15–22, 2018. doi:10.1016/j.endm.2018.03.003.

- 7 Christian Blum, José Antonio Lozano, and Pedro Pinacho Davidson. Iterative probabilistic tree search for the minimum common string partition problem. In *Proc. HM*, volume 8457 of *LNCS*, pages 145–154, 2014. doi:10.1007/978-3-319-07644-7_11.
- 8 Christina Boucher and Kathleen P. Wilkie. Why large closest string instances are easy to solve in practice. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 106–117, 2010. doi:10.1007/978-3-642-16321-0_10.
- 9 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for np-hard string problems. Technical report, European Association for Theoretical Computer Science, 2014.
- 10 Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau. An FPT-algorithm for longest common subsequence parameterized by the maximum number of deletions. In *Proc. CPM*, volume 223 of *LIPICs*, pages 6:1–6:11, 2022. doi:10.4230/LIPICs.CPM.2022.6.
- 11 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *Proc. SODA*, pages 102–121, 2014. doi:10.1137/1.9781611973402.8.
- 12 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020. doi:10.1017/S1471068419000450.
- 13 Markus Chimani, Matthias Woste, and Sebastian Böcker. A closer look at the closest string and closest substring problem. In *Proc. ALENEX*, pages 13–24, 2011. doi:10.1137/1.9781611972917.2.
- 14 Milos Chromý and Markus Sinnl. On solving the minimum common string partition problem by decision diagrams. In *Proc. ICORES*, pages 177–184, 2022. doi:10.5220/0010830200003117.
- 15 Thomas H Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 2009.
- 16 Federico Della Croce and Fabio Salassa. Improved lp-based algorithms for the closest string problem. *Comput. Oper. Res.*, 39(3):746–749, 2012. doi:10.1016/j.cor.2011.06.010.
- 17 Marek Cygan, Alexander S. Kulikov, Ivan Mihajlin, Maksim Nikolaev, and Grigory Reznikov. Minimum common string partition: Exact algorithms. In *Proc. ESA*, volume 204 of *LIPICs*, pages 35:1–35:16, 2021. doi:10.4230/LIPICs.ESA.2021.35.
- 18 Warley Gramacho da Silva, Tiago da Silva Almeida, Rafael Lima de Carvallho, Edeilson Milhomem da Silva, Ary Henrique de Oliveira, Glenda Michele Botelho, and Glêndara Aparecida de Souza Martins. Two classic chess problems solved by answer set programming. *International Journal of Advanced Engineering Research and Science*, 6(4):1–5, 2019. doi:10.22161/ijaers.6.4.43.
- 19 Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Magazine*, 37(3):53–68, 2016. doi:10.1609/aimag.v37i3.2678.
- 20 Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018. doi:10.1007/s13218-018-0548-6.
- 21 Carlos Farkas, Andy Mella, Maxime Turgeon, and Jody J Haigh. A novel sars-cov-2 viral sequence bioinformatic pipeline has found genetic evidence that the viral 3' untranslated region (utr) is evolving and generating increased viral diversity. *Frontiers in microbiology*, 12(665041):1–14, 2021. doi:10.3389/fmicb.2021.665041.
- 22 S. M. Ferdous and M. Sohel Rahman. Solving the minimum common string partition problem with the help of ants. *Math. Comput. Sci.*, 11(2):233–249, 2017. doi:10.1007/s11786-017-0293-5.
- 23 S. M. Ferdous and Mohammad Sohel Rahman. An integer programming formulation of the minimum common string partition problem. *PLOS ONE*, 10(7):1–16, 2015. doi:10.1371/journal.pone.0130266.

- 24 Moti Frances and Ami Litman. On covering problems of codes. *Theory Comput. Syst.*, 30(2):113–119, 1997. doi:10.1007/s002240000044.
- 25 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.
- 26 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Proc. ICLP*, volume 52 of *OASICS*, pages 2:1–2:15, 2016. doi:10.4230/OASICS.ICLP.2016.2.
- 27 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.
- 28 Martin Gebser, Marco Maratea, and Francesco Ricca. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.*, 20(2):176–204, 2020. doi:10.1017/S1471068419000061.
- 29 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *Electron. J. Comb.*, 12, 2005. doi:10.37236/1947.
- 30 Luis C. González, Heidi J. Romero, and Carlos A. Brizuela. A genetic algorithm for the shortest common superstring problem. In *Proc. GECCO*, volume 3103 of *LNCS*, pages 1305–1306, 2004. doi:10.1007/978-3-540-24855-2_139.
- 31 Jens Gramm. Closest substring. In *Encyclopedia of Algorithms*, pages 324–326. Springer, 2016. doi:10.1007/978-1-4939-2864-4_74.
- 32 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for CLOSEST STRING and related problems. *Algorithmica*, 37(1):25–42, 2003. doi:10.1007/s00453-003-1028-3.
- 33 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. doi:10.1016/0020-0190(92)90176-V.
- 34 Brenda Hinkemeyer and Bryant A. Julstrom. A genetic algorithm for the longest common subsequence problem. In *Proc. GECCO*, pages 609–610, 2006. doi:10.1145/1143997.1144105.
- 35 Hoang Xuan Huan, Dong Do Duc, and Nguyen Manh Ha. An efficient two-phase ant colony optimization algorithm for the closest string problem. In *Proc. SEAL*, volume 7673 of *LNCS*, pages 188–197, 2012. doi:10.1007/978-3-642-34859-4_19.
- 36 Robert W. Irving and Campbell Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *Proc. CPM*, volume 644 of *LNCS*, pages 214–229, 1992. doi:10.1007/3-540-56024-6_18.
- 37 Tom Kelsey and Lars Kotthoff. Exact closest string as a constraint satisfaction problem. In *Proc. ICCS*, volume 4 of *Procedia Computer Science*, pages 1062–1071, 2011. doi:10.1016/j.procs.2011.04.113.
- 38 Dusan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n-fold integer programming and applications. *Math. Program.*, 184(1):1–34, 2020. doi:10.1007/s10107-019-01402-2.
- 39 J. Kevin Lanctôt, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Inf. Comput.*, 185(1):41–55, 2003. doi:10.1016/S0890-5401(03)00057-9.
- 40 Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019. doi:10.1007/978-3-030-24658-7.
- 41 Liu Liu. *The Performance Optimization of ASP Solving Based on Encoding*. PhD thesis, University of Kentucky, 2022.
- 42 Liu Liu and Mirosław Truszczyński. Encoding selection for solving hamiltonian cycle problems with ASP. In *Proc. ICLP*, volume 306 of *EPTCS*, pages 302–308, 2019. doi:10.4204/EPTCS.306.35.
- 43 Xiaolan Liu, Shenghan Liu, Zhifeng Hao, and Holger Mauch. Exact algorithm and heuristic for the closest string problem. *Comput. Oper. Res.*, 38(11):1513–1520, 2011. doi:10.1016/j.cor.2011.01.009.

- 44 Domingo López-Rodríguez and Enrique Mérida Casermeiro. Shortest common superstring problem with discrete neural networks. In *Proc. ICANNGA*, volume 5495 of *LNCS*, pages 62–71, 2009. doi:10.1007/978-3-642-04921-7_7.
- 45 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978. doi:10.1145/322063.322075.
- 46 Dániel Marx. The closest substring problem with small distances. In *Proc. FOCS*, pages 63–72, 2005. doi:10.1109/SFCS.2005.70.
- 47 Holger Mauch. Closest substring problem – results from an evolutionary algorithm. In *Proc. ICONIP*, volume 3316 of *LNCS*, pages 205–211, 2004. doi:10.1007/978-3-540-30499-9_30.
- 48 Rolf Niedermeier. Ubiquitous parameterization – invitation to fixed-parameter algorithms. In *Proc. MFCS*, volume 3153 of *LNCS*, pages 84–103, 2004. doi:10.1007/978-3-540-28629-5_4.
- 49 Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *Proc. IFIP*, pages 59–64, 1983.
- 50 Shyong Jian Shyu and Chun-Yuan Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Comput. Oper. Res.*, 36(1):73–91, 2009. doi:10.1016/j.cor.2007.07.006.
- 51 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.
- 52 Krister M. Swenson, Mark Marron, Joel V. Earnest-DeYoung, and Bernard M. E. Moret. Approximating the true evolutionary distance between two genomes. *ACM J. Exp. Algorithmics*, 12:3.5:1–3.5:17, 2008. doi:10.1145/1227161.1402297.
- 53 Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57:131–145, 1988. doi:10.1016/0304-3975(88)90167-3.
- 54 Jean P. Tremeşchin Torres and Edna Ayako Hoshino. Lp-based heuristics for the distinguishing string and substring selection problems. *Ann. Oper. Res.*, 316(2):1205–1234, 2022. doi:10.1007/s10479-021-04138-5.
- 55 Omar Vilca and Rosiane de Freitas. An efficient algorithm for the closest string problem. In *Anais do I Encontro de Teoria da Computação*, pages 879–882, Porto Alegre, RS, Brasil, 2016. doi:10.5753/etc.2016.9850.
- 56 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.
- 57 Lusheng Wang, Ming Li, and Bin Ma. Closest string and substring problems. In *Encyclopedia of Algorithms*, pages 321–324. Springer, 2016. doi:10.1007/978-1-4939-2864-4_73.
- 58 Neng-Fa Zhou. In pursuit of an efficient SAT encoding for the Hamiltonian cycle problem. In *Proc. CP*, volume 12333 of *LNCS*, pages 585–602, 2020. doi:10.1007/978-3-030-58475-7_34.

A Alternative CSS Encoding

For small values of λ , the offsets can be quite large. Here, we present an alternative encoding without the offsets. The resulting encoding has fewer variables, but has more variables that are subject to the optimization argument. In what follows, we can encode $T[1..\lambda]$ by the Boolean variables $T_{i,c}$ specifying with $T_{i,c} = 1$ that $T[i] = c$:

$$[\mathcal{O}(\lambda), \mathcal{O}(\sigma)] \quad \forall i \in [1..\lambda] : \sum_{c \in \Sigma} T_{i,c} = 1 \quad (\text{CSS1}')$$

We now let the costs encode the offsets by the variables $C_{i,x,o}$ being set if $S_x[o+i] \neq T[i]$.

$$[\mathcal{O}(\lambda n m \sigma), \mathcal{O}(1)] \quad \forall i \in [1..\lambda], c \in \Sigma_i, x \in [1..m], o \in [1..n - \lambda] : \\ T_{i,c} \wedge S_x[i+o] \neq c \implies C_{i,x,o} \quad (\text{CSS2}')$$

■ **Table 8** Used Entities.

entity	meaning
Σ	alphabet
σ	alphabet size, $\sigma = \Sigma $
\mathcal{S}	set of input strings $\{S_1, \dots, S_m\}$
m	size of \mathcal{S} , i.e., $m = \mathcal{S} $
n	length of an input string
S_x	input string
T	string to output
ℓ	length for a subsequence
δ	distance of the output to all S_x
i, j	indices for text positions in an input string
x, y	indices for an input string
c	character in Σ

The objective function becomes

$$[\mathcal{O}(1), \mathcal{O}(mn^2)] \quad \text{minimize} \quad \max_{x \in [1..m]} \min_{o \in [1..n-\lambda]} \sum_{i \in [1..n]} C_{i,x,o} \quad (\text{CSS3}')$$

On the Complexity of Parameterized Local Search for the Maximum Parsimony Problem

Christian Komusiewicz  


Institute of Computer Science, Friedrich Schiller Universität Jena, Germany

Simone Linz  

School of Computer Science, University of Auckland, New Zealand

Nils Morawietz  

Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany

Jannik Schestag  

Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany

Abstract

MAXIMUM PARSIMONY is the problem of computing a most parsimonious phylogenetic tree for a taxa set X from character data for X . A common strategy to attack this notoriously hard problem is to perform a local search over the phylogenetic tree space. Here, one is given a phylogenetic tree T and wants to find a more parsimonious tree in the neighborhood of T . We study the complexity of this problem when the neighborhood contains all trees within distance k for several classic distance functions. For the nearest neighbor interchange (NNI), subtree prune and regraft (SPR), tree bisection and reconnection (TBR), and edge contraction and refinement (ECR) distances, we show that, under the exponential time hypothesis, there are no algorithms with running time $|I|^{o(k)}$ where $|I|$ is the total input size. Hence, brute-force algorithms with running time $|X|^{O(k)} \cdot |I|$ are essentially optimal.

In contrast to the above distances, we observe that for the sECR-distance, where the contracted edges are constrained to form a subtree, a better solution within distance k can be found in $k^{O(k)} \cdot |I|^{O(1)}$ time.

2012 ACM Subject Classification Theory of computation \rightarrow Parameterized complexity and exact algorithms; Applied computing \rightarrow Molecular evolution; Applied computing \rightarrow Computational genomics

Keywords and phrases phylogenetic trees, parameterized complexity, tree distances, NNI, TBR

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.18

Funding CK and SL were supported by Catalyst Seeding funding provided by the New Zealand Ministry of Business, Innovation and Employment and administered by the Royal Society of New Zealand Te Apārangi. NM was supported by the Deutsche Forschungsgemeinschaft (DFG), project OPERAH, KO 3669/5-1.

1 Introduction

MAXIMUM PARSIMONY is one of the most popular methods for inferring phylogenetic (evolutionary) trees from sequences of morphological or molecular characters. Given sequences of characters for n taxa, this method reconstructs a phylogenetic tree T whose n leaves are labeled bijectively by the n taxa and that has the minimum *parsimony score* over all such trees. The parsimony score is the number of character state changes along the tree edges that are necessary when extending the sequences for the leaves of T to all internal vertices of T . Note that for each character, this score is at least $s - 1$, where s denotes the number



© Christian Komusiewicz, Simone Linz, Nils Morawietz, and Jannik Schestag; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 18; pp. 18:1–18:18



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of different character states. A phylogenetic tree is called perfect if it achieves score $s - 1$ for every character. Such a perfect phylogeny does not always exist. For a more comprehensive introduction to MAXIMUM PARSIMONY, we refer the interested reader to [9].

From an algorithmic point of view, the MAXIMUM PARSIMONY problem is notoriously hard: It is NP-complete even for binary characters [12]. Moreover, the current best running time is $\Omega((2n - 3)!!)$, where $(2n - 3)!! = 1 \cdot 3 \cdot \dots \cdot (2n - 5) \cdot (2n - 3)$ [4]. The associated algorithm generates all possible binary phylogenetic trees on n leaves in a bottom-up fashion. Hence, the best known algorithm is essentially a brute-force-method. This running time bound is impractical when $n > 15$. Better running times are possible when the instance has a near-perfect phylogeny and the number of different character states s is small. Here, the running time is measured also in terms of the excess q over the score of a perfect phylogeny. In the general case, MAXIMUM PARSIMONY can be solved in $nm^{\mathcal{O}(q)}2^{\mathcal{O}(q^2s^2)}$ time [10], where m is the length of the character sequences. In 2007, the running time was improved to $\mathcal{O}(21^q + 8^qnm^2)$ for the special case of binary characters and the practical usefulness of the improved algorithm was demonstrated for $q \leq 10$ [30]. In the worst case, however, q can be essentially as large as m . Moreover, MAXIMUM PARSIMONY is NP-hard even for $q = 0$ when the number of different character states is unbounded [3].

Given the hardness of MAXIMUM PARSIMONY, solving this problem exactly is impractical for many real-world datasets due to prohibitive running times. Consequently, heuristic approaches, in particular local search, play an important role in computing good, but not necessarily optimal, solutions [2, 13, 14, 16, 17, 18, 19, 25, 26]. These approaches search the space of all possible phylogenetic trees on n taxa. In the course of such a search, the parsimony score of a subset of the phylogenetic trees in the space is computed. For any given tree, this step takes polynomial time using Fitch's or Sankoff's algorithm [11, 28]. A search through tree space starts by first computing a starting tree T before computing the parsimony score of all neighbors of T . If there is a neighboring tree T' whose parsimony score is smaller than that of T , then the search is continued by computing the parsimony score of all neighbors of T' and so on until a local optimum is found. In each iteration of the search, the neighboring trees are those that can be obtained from the current best tree by one or more rearrangement operations. The most well-known rearrangement operations on trees that are also considered in local search approaches for MAXIMUM PARSIMONY, are nearest neighbor interchange (NNI), subtree prune and regraft (SPR), and tree bisection and reconnection (TBR) [1]. Each of these operations deletes an edge of a tree and then reconnects the resulting two subtrees. Depending on the operation, the reconnection is more or less restrictive, with SPR being a generalization of NNI and TBR being a generalization of SPR. The set of all trees that can be obtained by one operation is called the NNI, SPR, or TBR neighborhood, respectively. More general, we say that a tree T' is in the k -neighborhood with respect to NNI, SPR, or TBR of another tree T , if T' can be obtained from T by at most k NNI, SPR, or TBR operations, respectively.

In addition to NNI, SPR, and TBR, the k -ECR operation has also been considered in the literature (see for example the works by Ganapathy et al. [13, 14]). This latter operation first contracts up to k edges and then refines the resulting tree arbitrarily. Here, the k -ECR neighborhood contains all trees that can be obtained from a starting tree by applying one k -ECR operation. The 1-ECR neighborhood is exactly the NNI neighborhood, but the 2-ECR neighborhood strictly contains the set of trees reachable by two NNI moves [14]. The k -ECR neighborhood appeared earlier implicitly under the term *sectorial search* [19]. The k -sECR neighborhood, a restricted version of the k -ECR neighborhood where the contracted edges must form a subtree was considered by Sankoff et al. [29]. They found that for larger

values of k , the k -sECR neighborhood gives better results than the 1-ECR neighborhood or, equivalently, the NNI neighborhood. Guo et al. [20] found that exploring the k -ECR neighborhood is too costly and thus proposed a restriction of this neighborhood which already leads to very good local optima. Their approach contracts k edges and then refines the resulting tree by using neighbor joining, a fast distance-based method to reconstruct phylogenetic trees. To summarize, local search is an important paradigm for designing heuristics for MAXIMUM PARSIMONY, and it has been noted that larger neighborhoods such as the k -ECR neighborhood give better results at the cost of higher running times. So far, there is however no study of how hard exploring larger neighborhoods actually is.

To analyze the computational complexity of exploring neighborhoods under NNI, SPR, TBR, k -ECR, and k -sECR, we use the framework of *parameterized local search* [8, 15, 23, 24]. Here, one studies local search problems with a neighborhood whose size can be adjusted by a parameter k . In the canonical parameterized local search problem, one is then given some solution for an optimization problem and the question is whether there is a better solution in the k -neighborhood. Local search for any of the aforementioned neighborhoods that are associated with distances between two trees fits exactly into this framework: we are given a phylogenetic tree and want to know whether there is one with a better parsimony score in the k -neighborhood. Typically, the k -neighborhood has a size of $\mathcal{O}(|I|^{f(k)})$, where $|I|$ is the input size. In our case, the input size $|I|$ is in $\mathcal{O}(n^2 \cdot m)$. Thus, using a brute-force algorithm, one can find a better solution in the neighborhood if it exists in $|I|^{f(k)}$ time. The algorithmic question is now whether this can be done much faster. In particular, a running time of $f(k) \cdot |I|^{\mathcal{O}(1)}$ would be desirable since the explosion in the running time would then depend only on k and not on $|I|$. Parameterized algorithmics provides toolkits to design such algorithms or to show that such algorithms are unlikely. The latter can be done by showing W[1]-hardness with respect to k [6, 7] or by giving tight running time bounds based on the exponential time hypothesis (ETH) [21].

Our results are as follows. We show that even when all characters are binary, searching the k -ECR neighborhood is W[1]-hard with respect to k . The reduction that we use to establish this result also shows that, under the ETH, a running time of $|I|^{\Omega(k)}$ is necessary. Moreover, the reduction implies hardness for searching the k -neighborhood with respect to NNI, SPR, and TBR. In a nutshell, our results show that one cannot gain a substantial speed-up over the brute-force algorithm when trying to search these large neighborhoods. We then establish that $n^{\mathcal{O}(k)} \cdot m$ time is sufficient to search the k -neighborhoods with respect to any of NNI, SPR, TBR, and k -ECR, giving tight upper and lower bounds for the running time dependence on k . Finally, we observe that the k -sECR neighborhood of Sankoff [29] can be searched in $k^{\mathcal{O}(k)} \cdot |I|^{\mathcal{O}(1)}$ time, making it possible to consider much larger values of k than for the other neighborhoods. Let us remark that, while we formally study the decision problem that asks for the existence of a better tree in the k -neighborhood, our hardness results and algorithms also apply to the problem of finding an optimal tree in the k -neighborhood.

Proofs of statements marked with (*) are deferred to a full version of the article.

2 Preliminaries

For details about relevant definitions of parameterized complexity such as fixed-parameter tractability, W[1]-hardness, parameterized reductions and ETH, refer to the standard monographs [6, 7].

Graph notation. For a graph $G = (V, E)$ and a vertex set $K \subseteq V$, let $E(K)$ denote the set of edges of G where both endpoints are from K . The *subdivision of an edge* $e \in E$ in G results in the graph G' obtained by removing e from G and adding a new vertex which is adjacent to both endpoints of e . Let v be a vertex of degree 2 in G . The *suppression of v* in G results in the graph G' obtained by removing v from G and joining both neighbors of v by an edge.

Phylogenetic trees. Throughout this paper, X denotes a non-empty finite set of *taxa*.

An *unrooted phylogenetic X -tree* (for short, *X -tree*) T is a tree with leaf-set X and where no vertex has degree 2. If all non-leaf vertices of T have degree three, then T is called *binary*. Furthermore, if an edge e is incident with a leaf of T , then e is called a *pendant edge* and, otherwise, an *internal edge*. For two disjoint sets of taxa A and B , we say that $A|B$ is a *split* of an X -tree T if there is an edge e in T such that the deletion of e results in two subtrees where one has leaf set A and the other has leaf set B . The set of all splits of T is denoted by $\Sigma(T)$. Furthermore, we say that an X -tree T' is a *refinement* of T if $\Sigma(T) \subseteq \Sigma(T')$. Additionally, if T' is binary, then T' is a *binary refinement* of T . We say that two X -trees T and T' are *isomorphic* if $\Sigma(T) = \Sigma(T')$. Equivalently, two X -trees T and T' are *isomorphic* if there is a bijection φ between the vertices of T and the vertices of T' such that $\varphi(x) = x$ for all $x \in X$, and for all distinct vertices u and v of T , $\{u, v\}$ is an edge of T if and only if $\{\varphi(u), \varphi(v)\}$ is an edge of T' .

Now, let T be an X -tree and let V' be a subset of the vertices of T . Then $T(V')$ denotes the minimal subtree of T containing all vertices in V' . Let A be a non-empty and proper subset of X and let T be a binary X -tree. If $A|(X \setminus A)$ is a split of T , then the subtree $T(A)$ is a *pendant A -tree*. Moreover, the *pseudo-root* of $T(A)$ is the unique vertex of degree 2 in $T(A)$ if $|A| > 1$ and the unique vertex of $T(A)$, otherwise.

Maximum parsimony. A *character*¹ c on X is a function $c : X \rightarrow C$. If $|C| = 2$, then c is called a *binary character*. Intuitively, C can be thought of as the underlying alphabet and each element in the alphabet is a *character state*. Let T be an X -tree with vertex set V , and let c be a character on X whose set of character states is C . An *extension* c^* of c to V is a function $c^* : V \rightarrow C$ such that $c^*(x) = c(x)$ for each taxon $x \in X$. Let c^* be an extension of c . A *mutation edge of c^* in T* is an edge $\{u, v\}$ in T such that $c^*(u) \neq c^*(v)$ and we let $\text{score}_{c^*}(T)$ denote the number of mutation edges of c^* in T . Then the *parsimony score* of c on T , denoted by $\text{score}_c(T)$, is obtained by minimizing $\text{score}_{c^*}(T)$ over all possible extensions c^* of c . An extension c^* that minimizes $\text{score}_{c^*}(T)$ is called an *optimal extension of c in T* . Moreover the *maximum parsimony score* of c , denoted by $\text{MP}(c)$, is the parsimony score of c minimized over all binary X -trees.

Now let $S = (c_1, c_2, \dots, c_m)$ be a sequence of characters on X . Then the parsimony score of S on an X -tree T is defined as $\text{score}_S(T) = \sum_{i=1}^m \text{score}_{c_i}(T)$ and, similarly, the maximum parsimony score of S , denoted by $\text{MP}(S)$, is the parsimony score of S minimized over all binary X -trees.

We may abuse notation by writing $c \in S$ if the character c is contained in the sequence S .

SPR and TBR. Let T be a binary X -tree. Let $e = \{u, v\}$ be an edge of T , and let T_1 and T_2 be the two trees obtained from T by deleting e and suppressing u if its degree is 2. Without loss of generality, we may assume that T_2 contains v . If T_1 contains at least one

¹ Characters as defined here are not elements of some alphabet but functions that assign an element of some alphabet to each taxon.

edge, subdivide an edge of T_1 with a new vertex u' ; otherwise, set u' to be the single isolated vertex of T_1 . Finally, obtain a binary X -tree T' by adding the new edge $\{u', v\}$. We say that T' has been obtained from T by a single *subtree prune and regraft (SPR)* operation. We next define a generalization of the SPR operation. Again, let e be an edge of T , and let T_1 and T_2 be the two trees obtained from T by deleting e and suppressing any resulting degree-2 vertices. For each $i \in \{1, 2\}$, if T_i has at least one edge, subdivide an edge in T_i with a new vertex v_i and, otherwise, set v_i to be the single vertex of T_i . Obtain a binary X -tree T' by adding the new edge $\{v_1, v_2\}$. We say that T' has been obtained from T by a single *tree bisection and reconnection (TBR)* operation.

NNI, k -ECR, and k -sECR. Let T be a binary X -tree. Let $e = \{u, v\}$ be an edge of T and let $e' = \{v, w\}$ be an internal edge of T that is adjacent to e . Let T' be a binary X -tree obtained from T by deleting e , suppressing v , subdividing an edge that is incident with w with a new vertex v' , and joining u and v' via a new edge. We say that T' has been obtained from T by a single *nearest neighbor interchange (NNI)* operation. Equivalently, if T' is a binary refinement of the tree obtained from T by contracting e' and T' is non-isomorphic to T , then T' is obtained from T by a single NNI operation.

Now let T be a binary X -tree, and let k be a positive integer. Let T' be a binary refinement of a tree obtained from T by contracting k (distinct) internal edges E' . If T' and T are non-isomorphic, then we say that T' is a single *k -edge contract and refine (k -ECR)* operation [13] apart from T and that E' is a *contraction set* for T and T' . Note that an NNI operation is a 1-ECR operation and vice versa. We denote the restricted version of a k -ECR operation that requires the k contracted edges to form a subtree of T as k -sECR [29].

Distance measures. Let T and T' be binary X -trees. For each $\Theta \in \{\text{NNI}, \text{SPR}, \text{TBR}\}$, the distance $d_\Theta(T, T')$ is defined as the minimum number of Θ operations to transform T into T' [1]. The distance $d_{\text{ECR}}(T, T')$ is defined as the smallest number k such that T and T' are one k -ECR operation apart. Analogously, the distance $d_{\text{sECR}}(T, T')$ is defined as the smallest number k such that T and T' are one k -sECR operation apart.

Considered problems. In this work, we consider the parameterized complexity of the following problem for each distance measure $d \in \{d_{\text{NNI}}, d_{\text{SPR}}, d_{\text{TBR}}, d_{\text{ECR}}, d_{\text{sECR}}\}$.

d -LS MAXIMUM PARSIMONY

Input: A set of taxa X , a binary X -tree T , a sequence of characters S , and an integer k .

Question: Is there a binary X -tree T' with $d(T, T') \leq k$ and $\text{score}_S(T') < \text{score}_S(T)$?

3 Properties of the Considered Distance Measures

In this section, we analyze the relation of the different distance measures.

► **Observation 3.1** ([1, 27]). *The distance measures d_{NNI} , d_{SPR} , and d_{TBR} are metrics.*

► **Lemma 3.2** (*). *The distance measure d_{ECR} is a metric.*

► **Observation 3.3.** *Let T and T' be distinct binary X -trees and let $k > 0$ be an integer. If $d_{\text{ECR}}(T, T') = k$, then there is a binary X -tree \tilde{T} with $d_{\text{sECR}}(\tilde{T}, T') > 0$ such that $d_{\text{ECR}}(T, T') = d_{\text{ECR}}(T, \tilde{T}) + d_{\text{sECR}}(\tilde{T}, T')$.*

The idea behind Observation 3.3 is to consider the connected components of T induced by the contraction set S between T and T' . If S forms a subtree of T , then S is connected and $d_{\text{sECR}}(T, T') = d_{\text{ECR}}(T, T')$. Hence, the statement holds for $\tilde{T} = T'$. Otherwise, let \tilde{S} be an inclusion-maximal subset of S , such that \tilde{S} forms a subtree of T . Since \tilde{S} is inclusion-maximal, we can obtain T' from T in two steps: First, we can obtain an intermediate X -tree \tilde{T} from T by an sECR operation with contraction set \tilde{S} . Second, we can obtain T' from \tilde{T} by an ECR operation with contraction set $S \setminus \tilde{S}$.

► **Lemma 3.4 (*)**. *Let T and T' be binary X -trees. Then, $d_{\text{NNI}}(T, T') \geq d_{\text{ECR}}(T, T')$.*

► **Lemma 3.5**. *Let T and T' be binary X -trees. Then, $d_{\text{sECR}}(T, T') \geq d_{\text{SPR}}(T, T')$.*

Proof. Let $k = d_{\text{sECR}}(T, T')$. Hence, there is a set S of k internal edges in T such that T' can be obtained by an sECR operation with contraction set S . Let V' be the vertices of T incident with some edge of S and let V^* be the neighbors of V' in T that are not incident with any edge of S . Recall that by definition of sECR operations, the edges of S induce a subtree of T . Hence, $T(V^*)$ is a binary V^* -tree having the set S as internal edges. For each vertex v of V^* , let T_v denote the pendant subtree of T with pseudo-root v obtained by removing the edge between v and the unique neighbor of v in V' . Since T' can be obtained by an sECR operation with contraction set S , T' contains a subtree T'_v isomorphic to T_v for each vertex v of V^* . Hence, $d_{\text{SPR}}(T, T') = d_{\text{SPR}}(T_S, T'_S)$, where T_S is obtained from T by replacing T_v by the auxiliary taxa v for each vertex v of V^* and where T'_S is obtained from T' by replacing T'_v by the auxiliary taxa v for each vertex v of V^* [1]. Note that $T_S = T(V^*)$.

Hence, it remains to show that $d_{\text{SPR}}(T_S, T'_S) \leq k$. Since T is binary and the edges of S induce a subtree of T , $|V^*| = |S| + 3$. Moreover, since for each set of taxa X' and each two binary X' -trees \tilde{T} and \hat{T} , $d_{\text{SPR}}(\tilde{T}, \hat{T}) \leq |X'| - 3$ [1], we conclude $d_{\text{SPR}}(T_S, T'_S) \leq |V^*| - 3 = |S| = k$. Consequently, $d_{\text{SPR}}(T, T') \leq k = d_{\text{sECR}}(T, T')$. ◀

► **Lemma 3.6 (*)**. *Let T and T' be binary X -trees. Then, $d_{\text{ECR}}(T, T') \geq d_{\text{SPR}}(T, T')$.*

4 Hardness of d -LS Maximum Parsimony

In this section, we establish our main theorem.

► **Theorem 4.1**. *For each distance measure $d \in \{d_{\text{NNI}}, d_{\text{ECR}}, d_{\text{SPR}}, d_{\text{TBR}}\}$ and even if each character is binary, d -LS MAXIMUM PARSIMONY*

- *is NP-complete, W[1]-hard when parameterized by k , and*
- *cannot be solved in $f(k) \cdot |I|^{\alpha(k)}$ time for any computable function f , unless the ETH fails.*

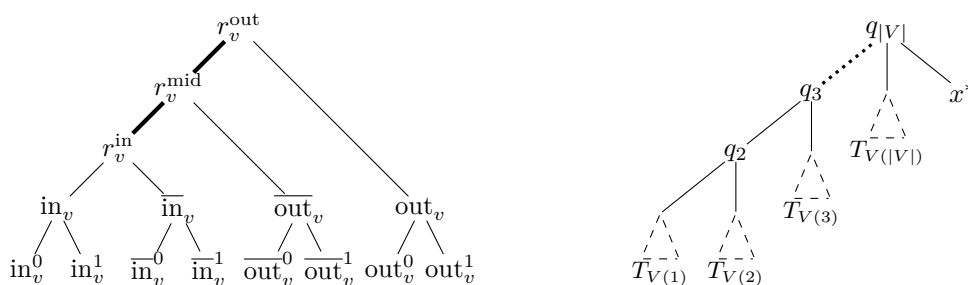
We reduce from CLIQUE which is NP-hard [22], W[1]-hard when parameterized by k [7], and cannot be solved in $f(k) \cdot |I|^{\alpha(k)}$ time for any computable function f , unless the ETH fails [5, 6].

CLIQUE

Input: An undirected graph $G = (V, E)$ and an integer k .

Question: Is there a *clique* of size k in G , that is, a set of vertices K of size k , such that $|E(K)| = \binom{k}{2}$?

Let $I = (G = (V, E), k)$ be an instance of CLIQUE and let $d \in \{d_{\text{NNI}}, d_{\text{ECR}}, d_{\text{SPR}}, d_{\text{TBR}}\}$ be a distance measure. We describe how to construct an equivalent instance $I' = (X, T = (V', E'), S, k')$ of d -LS MAXIMUM PARSIMONY in polynomial time where $k' := k$ if $d \in \{d_{\text{SPR}}, d_{\text{TBR}}\}$ and $k' := 2k$ if $d \in \{d_{\text{NNI}}, d_{\text{ECR}}\}$.



(a) For a vertex $v \in V$, the pendant X_v -tree T_v . The bold edges are the only edges of T_v that are not in R .

(b) The subtree of T connecting the pendant trees T_v for each vertex $v \in V$.

■ **Figure 1** The construction of the X -tree T .

Definition of X and T . We start with an empty taxa set X and add for each vertex $v \in V$, a set X_v consisting of the eight taxa

$$\text{in}_v^0, \text{in}_v^1, \overline{\text{in}}_v^0, \overline{\text{in}}_v^1, \overline{\text{out}}_v^0, \overline{\text{out}}_v^1, \text{out}_v^0, \text{and } \text{out}_v^1$$

to X . Additionally, we add a taxon x^* to X . This completes the definition of X .

Next, we define the binary X -tree $T = (V', E')$. Since X contains $8 \cdot |V| + 1$ taxa and each internal vertex of T has three neighbors, T' has $16 \cdot |V|$ vertices and $2 \cdot |X| - 3 = 16 \cdot |V| - 1$ edges. By definition, V' is a superset of X . Additionally, for each vertex $v \in V$, the set V' contains the seven vertices

$$\text{in}_v, \overline{\text{in}}_v, \overline{\text{out}}_v, \text{out}_v, r_v^{\text{in}}, r_v^{\text{mid}}, \text{and } r_v^{\text{out}}.$$

The subtree $T_v := T(X_v)$ is depicted in Figure 1a.

Moreover, V' contains $|V| - 1$ additional vertices q_i with $i \in [2, |V|]$. Fix some arbitrary ordering of the vertices of V and let $V(i)$ denote the i th vertex of V according to that ordering. The vertex q_2 is adjacent to $r_{V(1)}^{\text{out}}, r_{V(2)}^{\text{out}}$, and q_3 . For each $i \in [3, |V| - 1]$, the vertex q_i is adjacent to q_{i-1}, q_{i+1} , and $r_{V(i)}^{\text{out}}$. Finally, $q_{|V|}$ is adjacent to $q_{|V|-1}, r_{V(|V|)}^{\text{out}}$, and x^* . See Figure 1b for an illustration. This completes the definition of T .

Intuition. The idea of the reduction is as follows: Some of the characters that we define in the following will ensure that each binary X -tree T' that improves over T contains a pendant subtree $T'(X_v)$ for each vertex $v \in V$. Further characters will ensure that there are only two non-isomorphic trees for $T'(X_v)$ which are depicted in Figure 1a and Figure 2. Intuitively, these two choices then function as a selection gadget for selecting vertex v as a vertex of the sought clique K . The budget k' bounds how many such vertices can be selected. Finally, further characters will ensure that T' improves over T only if $E(K)$ contains at least $\binom{k}{2}$ edges.

Definition of the characters of S . Next, we define the characters of S which are all binary characters whose character states are 0 and 1. We obtain S by concatenating two sequences of characters, S_G and S_R , which we describe in the following.

First, we describe the characters of S_G . An overview of the characters is given in Table 1. We initialize S_G as the empty sequence.

For each edge $e \in E$, we add a character c_e to S_G . Let e be an edge of E . We set $c_e(x^*) := 1$. Let v be a vertex of V . If v is an endpoint of e , we set $c_e(x) := 1$ for each taxon $x \in \{\text{in}_v^0, \text{in}_v^1, \overline{\text{in}}_v^0, \overline{\text{in}}_v^1\}$ and we set $c_e(x) := 0$ for each taxon $x \in \{\overline{\text{out}}_v^0, \overline{\text{out}}_v^1, \text{out}_v^0, \text{out}_v^1\}$. Otherwise, if v is not an endpoint of e , we set $c_e(x) := 1$ for each taxon $x \in \{\text{in}_v^1, \overline{\text{in}}_v^1, \overline{\text{out}}_v^1, \text{out}_v^1\}$ and we set $c_e(x) := 0$ for each taxon $x \in \{\text{in}_v^0, \overline{\text{in}}_v^0, \overline{\text{out}}_v^0, \text{out}_v^0\}$. Let S_E denote the sequence of characters c_e for each edge $e \in E$.

Next, we define a character c_{mal} . We set $c_{\text{mal}}(x^*) := 1$. For each vertex $v \in V$, we set $c_{\text{mal}}(\text{out}_v^0) = c_{\text{mal}}(\text{out}_v^1) := 1$ and we set $c_{\text{mal}}(x) := 0$ for each taxon $x \in X_v \setminus \{\text{out}_v^0, \text{out}_v^1\}$. We add a sequence S_{mal} of $\binom{k}{2} - 1$ copies of c_{mal} to S_G . Intuitively, in a binary X -tree T' , if both endpoints of an edge $e \in E$ are in the selected set K , then the parsimony score of c_e in T' is exactly the parsimony score of c_e in T minus one. Moreover, if T' is non-isomorphic to T , then the parsimony score of S_{mal} in T' is exactly the parsimony score of S_{mal} in T plus $|S_{\text{mal}}|$. Hence, the characters of S_{mal} act as a hurdle to ensure that $E(K)$ contains at least $|S_{\text{mal}}| + 1 = \binom{k}{2}$ edges.

Finally, for each vertex $v \in V$, we define four characters $c_{v,\text{in}}$, $c_{v,\text{out}}$, $c_{v,\text{ri}}$, and $c_{v,\text{ro}}$. For each taxon x of $X \setminus X_v$, we set $c_{v,\text{in}}(x) := c_{v,\text{out}}(x) := c_{v,\text{ri}}(x) := c_{v,\text{ro}}(x) := 1$. Now, let x be a taxon of X_v .

- If x is in $\{\text{in}_v^0, \text{in}_v^1\}$, we set $c_{v,\text{in}}(x) := 1$, $c_{v,\text{out}}(x) := 0$, $c_{v,\text{ri}}(x) := 1$, and $c_{v,\text{ro}}(x) := 0$.
- If x is in $\{\overline{\text{in}}_v^0, \overline{\text{in}}_v^1\}$, we set $c_{v,\text{in}}(x) := 1$, $c_{v,\text{out}}(x) := 0$, $c_{v,\text{ri}}(x) := 0$, and $c_{v,\text{ro}}(x) := 0$.
- If x is in $\{\overline{\text{out}}_v^0, \overline{\text{out}}_v^1\}$, we set $c_{v,\text{in}}(x) := 0$, $c_{v,\text{out}}(x) := 1$, $c_{v,\text{ri}}(x) := 0$, and $c_{v,\text{ro}}(x) := 0$.
- If x is in $\{\text{out}_v^0, \text{out}_v^1\}$, we set $c_{v,\text{in}}(x) := 0$, $c_{v,\text{out}}(x) := 1$, $c_{v,\text{ri}}(x) := 0$, and $c_{v,\text{ro}}(x) := 1$.

Let $\alpha := 2|X| \cdot (|E| + \binom{k}{2})$. Note that α is larger than $\text{score}_{S_E}(T') + \text{score}_{S_{\text{mal}}}(T')$ of any binary X -tree T' , since such a tree T' contains less than $2|X|$ edges and $|S_E| + |S_{\text{mal}}| = |E| + \binom{k}{2} - 1$. For each vertex $v \in V$, we extend S_G by

- a sequence $S_{v,\text{in}}$ of α copies of $c_{v,\text{in}}$,
- a sequence $S_{v,\text{out}}$ of α copies of $c_{v,\text{out}}$,
- a sequence $S_{v,\text{ri}}$ of 2α copies of $c_{v,\text{ri}}$, and
- a sequence $S_{v,\text{ro}}$ of 2α copies of $c_{v,\text{ro}}$.

Let S_v denote the combined sequences of $S_{v,\text{in}}$, $S_{v,\text{out}}$, $S_{v,\text{ri}}$, and $S_{v,\text{ro}}$. Intuitively, for each binary X -tree T' that improves over T and contains $T'(X_v)$ as a pendant subtree, the characters of S_v ensure that $T'(X_v)$ is isomorphic to either the pendant tree depicted in Figure 1a or the pendant tree depicted in Figure 2. These two choices then function as a selection gadget for the vertices of the sought clique in G . This completes the construction of S_G . Note that $|S_G| = |E| + \binom{k}{2} - 1 + 6\alpha \cdot |V|$.

Next, we describe the sequence of characters S_R . Let $\beta := 2|X| \cdot |S_G|$. Note that β is larger than $\text{score}_{S_G}(\tilde{T})$ of any binary X -tree \tilde{T} , since such a tree \tilde{T} contains less than $2|X|$ edges. Let $R := E' \setminus \{\{r_v^{\text{in}}, r_v^{\text{mid}}\}, \{r_v^{\text{mid}}, r_v^{\text{out}}\} \mid v \in V\}$. For each edge e of R , we define a character c_R^e . Let $A|B$ be the split of T induced by e . For each taxon $x \in A$, we set $c_R^e(x) := 0$ and for each taxon $x \in B$, we set $c_R^e(x) := 1$. We add as sequence S_R^e of β copies of c_R^e to S_R . Intuitively, the characters of S_R ensure that each binary X -tree T' that improves over T , shares the split that is induced by e in T for each edge e of R . This implies that $T'(X_v)$ is a pendant subtree of T' for each vertex $v \in V$.

Properties of binary X -trees. Before we show the correctness of the reduction, we first make some observations about binary X -trees with the characters of the construction.

Note that for each binary X -tree T' and each edge e of R , $\text{score}_{c_R^e}(T') \geq 1$.

■ **Table 1** An overview of the characters of S_G .

	$c_e \in S_E$ $v \in e$	$c_e \in S_E$ $v \notin e$	c_{mal}	$c_{v,\text{in}}$	$c_{v,\text{out}}$	$c_{v,\text{ri}}$	$c_{v,\text{ro}}$	$c \in S_w$ $w \neq v$
x^*	1	1	1	1	1	1	1	1
in_v^0	1	0	0	1	0	1	0	1
in_v^1	1	1	0	1	0	1	0	1
$\overline{\text{in}}_v^0$	1	0	0	1	0	0	0	1
$\overline{\text{in}}_v^1$	1	1	0	1	0	0	0	1
$\overline{\text{out}}_v^0$	0	0	0	0	1	0	0	1
$\overline{\text{out}}_v^1$	0	1	0	0	1	0	0	1
out_v^0	0	0	1	0	1	0	1	1
out_v^1	0	1	1	0	1	0	1	1

► **Definition 4.2.** Let T' be a binary X -tree. We say that T' is split-consistent for T and R if for each edge e of R , the split of T induced by e is also a split of T' .

In preparation for the next observation, note that if a binary X -tree T' is not split-consistent for T and R , then there is some edge e of R such that $\text{score}_{c_e^e}(T') \geq 2$ and thus $\text{score}_{S_R^e}(T') \geq 2 \cdot \beta$. Hence, $\text{score}_S(T') \geq \text{score}_{S_R}(T') \geq \beta \cdot (|R| + 1)$. Since $\beta > \text{score}_{S_G}(T)$, this implies $\text{score}_S(T') > \text{score}_S(T)$. Hence, we conclude the following.

► **Observation 4.3.** Let T' be a binary X -tree. a) If $\text{score}_S(T') \leq \text{score}_S(T)$, then T' is split-consistent for T and R . b) If T' is split-consistent for T and R , then $\text{score}_{S_R}(T') = \beta \cdot |R|$.

To determine whether I' is a yes-instance of d -LS MAXIMUM PARSIMONY, we analyze the structure of binary X -trees T' with $\text{score}_S(T') \leq \text{score}_S(T)$. Due to Observation 4.3, we only need to consider binary X -trees that are split-consistent for T and R in the following.

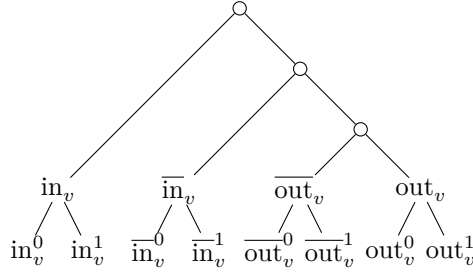
Let v be a vertex of V and let T' be a binary X -tree which is split-consistent for T and R . Since there is an edge e_v in T such that e_v induces the split $X_v|(X \setminus X_v)$ in T and e_v is contained in R , $X_v|(X \setminus X_v)$ is a split in T' . Hence, $T'(X_v)$ is a pendant tree. Moreover, since all edges incident with in_v are in R , we can assume that in_v is the common neighbor of in_v^0 and in_v^1 in T' . Similarly, we may assume that $\overline{\text{in}}_v$ is the common neighbor of $\overline{\text{in}}_v^0$ and $\overline{\text{in}}_v^1$ in T' , $\overline{\text{out}}_v$ is the common neighbor of $\overline{\text{out}}_v^0$ and $\overline{\text{out}}_v^1$ in T' , and out_v is the common neighbor of out_v^0 and out_v^1 in T' .

► **Definition 4.4.** Let T' be a binary X -tree which is split-consistent for T and R , let v be a vertex of V , and let r be the pseudo-root of the pendant tree $T'(X_v)$. We say that $T'(X_v)$ is an in-rooting of T_v if in_v is adjacent to r , $\overline{\text{in}}_v$ has distance 2 to r , and both $\overline{\text{out}}_v$ and out_v have distance 3 to r . Similarly, we say that $T'(X_v)$ is an out-rooting of T_v if out_v is adjacent to r , $\overline{\text{out}}_v$ has distance 2 to r , and both $\overline{\text{in}}_v$ and in_v have distance 3 to r .

Figure 1a shows an out-rooting of T_v and Figure 2 shows an in-rooting of T_v .

Note that for each vertex v of V , there is a unique in-rooting of T_v with respect to isomorphism. Similarly, there is a unique out-rooting of T_v with respect to isomorphism. Note that for each vertex $v \in V$, T_v is an out-rooting of T_v . We call a binary X -tree T' well-rooted if T' is split-consistent for T and R and if for each vertex $v \in V$, $T'(X_v)$ is either an in-rooting or an out-rooting of T_v . Note that T is well-rooted.

► **Lemma 4.5 (*)**. Let T' be a binary X -tree which is split-consistent for T and R and let v be a vertex of V . If $T'(X_v)$ is an in-rooting of T_v or an out-rooting of T_v , then $\text{score}_{S_v}(T') = 9\alpha$. Otherwise, $\text{score}_{S_v}(T') \geq 10\alpha$.



■ **Figure 2** An in-rooting of T_v .

Next, we describe for a given well-rooted binary X -tree T' the maximum parsimony scores of T' with respect to the characters of S_E and S_{mal} . The idea is that in a well-rooted binary X -tree T' , for each edge $e = \{u, v\} \in E$ where $T'(X_u)$ is an in-rooting of T_u and where $T'(X_v)$ is an in-rooting of T_v , the parsimony score of the character c_e in T' is exactly the parsimony score of the character c_e in T minus one. Moreover, if $T'(X_v)$ is an in-rooting of T_v for at least one vertex $v \in V$, then the parsimony score of the characters of S_{mal} in T' is exactly the parsimony score of the characters of S_{mal} in T plus $\binom{k}{2} - 1$.

- **Lemma 4.6.** *Let T' be a well-rooted binary X -tree. Let $e = \{u, v\}$ be an edge of E .*
- If $T'(X_u)$ is an in-rooting of T_u and $T'(X_v)$ is an in-rooting of T_v , then $\text{score}_{c_e}(T') = 4(|V| - 2) + 2$. Otherwise, $\text{score}_{c_e}(T') = 4(|V| - 2) + 3$.*
 - If there is a vertex $w \in V$ such that $T'(X_w)$ is an in-rooting of T_w , then $\text{score}_{c_{\text{mal}}}(T') = |V| + 1$. Otherwise, that is, if T' is isomorphic to T , $\text{score}_{c_{\text{mal}}}(T') = |V|$.*

Proof. For each vertex w of V , let $T'_w := T'(X_w)$. Let V_{in} be those vertices w of V , where T'_w is an in-rooting of T_w and let $V_{\text{out}} = V \setminus V_{\text{in}}$ be those vertices w of V , where T'_w is an out-rooting of T_w . For each vertex $w \in V_{\text{in}}$, let r_w^{in} be the name of the pseudo-root of T'_w , let r_w^{mid} and in_w be the neighbors of r_w^{in} , and let r_w^{out} and $\overline{\text{in}}_w$ be the neighbors of r_w^{mid} . Analogously, for each vertex $w \in V_{\text{out}}$, let r_w^{out} be the name of the pseudo-root of T'_w , let r_w^{mid} and out_w be the neighbors of r_w^{out} , and let r_w^{in} and $\overline{\text{out}}_w$ be the neighbors of r_w^{mid} . Recall that since T' is well-rooted, for each vertex $w \in V$, in_w is adjacent to both in_w^0 and in_w^1 , $\overline{\text{in}}_w$ is adjacent to both $\overline{\text{in}}_w^0$ and $\overline{\text{in}}_w^1$, out_w is adjacent to both out_w^0 and out_w^1 , and $\overline{\text{out}}_w$ is adjacent to both $\overline{\text{out}}_w^0$ and $\overline{\text{out}}_w^1$.

First, we show statement a). Let c_e be a character for some edge $e = \{u, v\}$ of E . For each vertex $w \in V \setminus \{u, v\}$,

- let P_{in_w} be the unique path between in_w^0 and in_w^1 in T' ,
- let $P_{\overline{\text{in}}_w}$ be the unique path between $\overline{\text{in}}_w^0$ and $\overline{\text{in}}_w^1$ in T' ,
- let $P_{\overline{\text{out}}_w}$ be the unique path between $\overline{\text{out}}_w^0$ and $\overline{\text{out}}_w^1$ in T' , and
- let P_{out_w} be the unique path between out_w^0 and out_w^1 in T' .

Note that each of these four paths only contains two edges and that these four paths are pairwise edge-disjoint. Let $\mathcal{P}_w := \{P_{\text{in}_w}, P_{\overline{\text{in}}_w}, P_{\overline{\text{out}}_w}, P_{\text{out}_w}\}$. Let P' be a path in \mathcal{P}_w and let w^0 and w^1 be the terminals of P' . Since by definition $c_e(w^0) \neq c_e(w^1)$, for each extension c_e^* of c_e in T' at least one edge of P' is a mutation edge of c_e^* . Note that each path in \mathcal{P}_w is edge-disjoint with each path in $\mathcal{P}_{w'}$ for distinct vertices w and w' of $V \setminus \{u, v\}$. Moreover, let P_u be the path between $\overline{\text{in}}_u^0$ and $\overline{\text{out}}_u^1$ in T' and let P_v be the path between $\overline{\text{in}}_v^0$ and $\overline{\text{out}}_v^1$ in T' . Note that P_u and P_v are edge-disjoint and that both are edge-disjoint with each path $P_w \in \mathcal{P}_w$ for each vertex $w \in V \setminus \{u, v\}$. Since $c_e(\overline{\text{in}}_u^0) = 0$ and $c_e(\overline{\text{out}}_u^1) = 1$, for

each extension c_e^* of c_e in T' , at least one edge of P_u is a mutation edge of c_e^* . Similarly, since $c_e(\overline{\text{in}}_v^0) = 0$ and $c_e(\overline{\text{out}}_v^0) = 1$, for each extension c_e^* of c_e in T' , at least one edge of P_v is a mutation edge of c_e^* . Hence, $\text{score}_{c_e}(T') \geq 4(|V| - 2) + 2$.

Case 1: T'_u is an in-rooting of T_u and T'_v is an in-rooting of T_v . We define an extension c_e^* of c_e in T' , such that $\text{score}_{c_e^*}(T') = 4(|V| - 2) + 2$. We set $c_e^*(\text{out}_u) := c_e^*(\overline{\text{out}}_u) := c_e^*(r_u^{\text{out}}) := 0$ and $c_e^*(\text{out}_v) := c_e^*(\overline{\text{out}}_v) := c_e^*(r_v^{\text{out}}) := 0$. For each remaining internal vertex v' of T' , we set $c_e^*(v') := 1$. Hence, the edge set

$$\begin{aligned} & \{\{r_u^{\text{out}}, r_u^{\text{mid}}\}, \{r_v^{\text{out}}, r_v^{\text{mid}}\}\} \\ & \cup \{\{\text{in}_w^0, \text{in}_w\}, \{\overline{\text{in}}_w^0, \overline{\text{in}}_w\}, \{\overline{\text{out}}_w^0, \overline{\text{out}}_w\}, \{\text{out}_w^0, \text{out}_w\} \mid w \in V \setminus \{u, v\}\} \end{aligned}$$

contains the mutation edges of c_e^* in T' . Consequently, $\text{score}_{c_e^*}(T') = 4(|V| - 2) + 2$ which implies $\text{score}_{c_e}(T') = 4(|V| - 2) + 2$.

Case 2: T'_u is an out-rooting of T_u or T'_v is an out-rooting of T_v . Assume without loss of generality that T'_v is an out-rooting of T_v . Let P_x^* be the unique path between out_v^0 and x^* in T' . Since $c_e(\text{out}_v^0) = 0$ and $c_e(x^*) = 1$, for each extension c_e^* of c_e in T' , at least one edge of P_x^* is a mutation edge of c_e^* . Note that P_x^* is edge-disjoint with P_u and edge-disjoint with each path $P_w \in \mathcal{P}_w$ for each vertex $w \in V \setminus \{u, v\}$. Moreover, since T'_v is an out-rooting of T_v , P_x^* is also edge-disjoint with P_v . Hence, $\text{score}_{c_e}(T') \geq 4(|V| - 2) + 3$. We define an extension c_e^* of c_e in T' , such that $\text{score}_{c_e^*}(T') = 4(|V| - 2) + 3$. To this end, we distinguish whether T'_u is an in-rooting of T_u or an out-rooting of T_u .

Case 2.1: T'_u is an in-rooting of T_u . We set $c_e^*(\text{out}_u) := c_e^*(\overline{\text{out}}_u) := c_e^*(r_u^{\text{out}}) := 0$ and $c_e^*(\text{out}_v) := c_e^*(\overline{\text{out}}_v) := 0$. For each remaining internal vertex v' of T' , we set $c_e^*(v') := 1$. Hence, the edge set

$$\begin{aligned} & \{\{r_u^{\text{out}}, r_u^{\text{mid}}\}, \{r_v^{\text{mid}}, \overline{\text{out}}_v\}, \{r_v^{\text{out}}, \text{out}_v\}\} \\ & \cup \{\{\text{in}_w^0, \text{in}_w\}, \{\overline{\text{in}}_w^0, \overline{\text{in}}_w\}, \{\overline{\text{out}}_w^0, \overline{\text{out}}_w\}, \{\text{out}_w^0, \text{out}_w\} \mid w \in V \setminus \{u, v\}\} \end{aligned}$$

contains the mutation edges of c_e^* in T' .

Case 2.2: T'_u is an out-rooting of T_u . We set $c_e^*(\text{in}_u) := c_e^*(\overline{\text{in}}_u) := c_e^*(r_u^{\text{in}}) := 1$ and $c_e^*(\text{in}_v) := c_e^*(\overline{\text{in}}_v) := c_e^*(r_v^{\text{in}}) := 1$. For each remaining internal vertex v' of T' , we set $c_e^*(v') := 0$. Hence, the edge set

$$\begin{aligned} & \{\{r_u^{\text{in}}, r_u^{\text{mid}}\}, \{r_v^{\text{in}}, r_v^{\text{mid}}\}, \{x^*, q_n\}\} \\ & \cup \{\{\text{in}_w^1, \text{in}_w\}, \{\overline{\text{in}}_w^1, \overline{\text{in}}_w\}, \{\overline{\text{out}}_w^1, \overline{\text{out}}_w\}, \{\text{out}_w^1, \text{out}_w\} \mid w \in V \setminus \{u, v\}\} \end{aligned}$$

contains the mutation edges of c_e^* in T' .

Consequently, in both cases $\text{score}_{c_e^*}(T') = 4(|V| - 2) + 3$ which implies $\text{score}_{c_e}(T') = 4(|V| - 2) + 3$.

Next, we show statement b). Consider the character c_{mal} . For each vertex $v \in V$, let P_v be the unique path between $\overline{\text{out}}_v^0$ and out_v^0 in T' . Since $c_{\text{mal}}(\overline{\text{out}}_v^0) = 0$ and $c_{\text{mal}}(\text{out}_v^0) = 1$, for each extension c_{mal}^* of c_{mal} in T' at least one edge of P_v is a mutation edge of c_{mal}^* . Note that the paths P_v and P_w are edge-disjoint for distinct vertices v and w of V . Hence, $\text{score}_{c_{\text{mal}}}(T') \geq |V|$.

Case 1: There is some vertex $v \in V$ such that T'_v is an in-rooting of T_v . Let P_x^* be the unique path between in_v^0 and x^* in T' . Since $c_{\text{mal}}(\text{in}_v^0) = 0$ and $c_{\text{mal}}(x^*) = 1$, for each extension c_{mal}^* of c_{mal} in T' , at least one edge of P_x^* is a mutation edge of c_{mal}^* . Note that P_x^* is edge-disjoint with P_w for each vertex $w \in V$ distinct from v . Moreover, since T'_v is an in-rooting of T_v , P_x^* is also edge-disjoint with P_v . Hence, $\text{score}_{c_{\text{mal}}}(T') \geq |V| + 1$. We define an extension c_{mal}^* of c_{mal} in T' , such that $\text{score}_{c_{\text{mal}}^*}(T') = |V| + 1$. We set $c_{\text{mal}}^*(\text{out}_w) := 1$, for each vertex $w \in V$. For each remaining internal vertex v' of T' , we set $c_{\text{mal}}^*(v') := 0$. Hence, the edge set $\{\{q_n, x^*\}\} \cup \{\{\text{out}_v, r_v^{\text{out}}\} \mid v \in V\}$ contains the mutation edges of c_{mal}^* in T' . Consequently, $\text{score}_{c_{\text{mal}}^*}(T') = |V| + 1$ which implies $\text{score}_{c_{\text{mal}}}(T') = |V| + 1$.

Case 2: For each vertex $v \in V$, T'_v is an out-rooting of T_v . Hence, T' is isomorphic to T . We define an extension c_{mal}^* of c_{mal} in T' , such that $\text{score}_{c_{\text{mal}}^*}(T') = |V|$. We set $c_{\text{mal}}^*(\text{in}_v) := c_{\text{mal}}^*(\overline{\text{in}}_v) := c_{\text{mal}}^*(\text{out}_v) := c_{\text{mal}}^*(r_v^{\text{in}}) := c_{\text{mal}}^*(r_v^{\text{mid}}) := 0$, for each vertex $v \in V$. For each remaining internal vertex v' of T' , we set $c_{\text{mal}}^*(v') = 1$. Hence, the edge set $\{\{r_v^{\text{mid}}, r_v^{\text{out}}\} \mid v \in V\}$ contains the mutation edges of c_{mal}^* in T' . Consequently, $\text{score}_{c_{\text{mal}}^*}(T') = |V|$ which implies that $\text{score}_{c_{\text{mal}}}(T') = |V|$. \blacktriangleleft

The score of improving X -trees with respect to S . Since T is well-rooted, and for each vertex $v \in V$, T_v is an out-rooting of T_v , Observation 4.3, Lemma 4.5, and Lemma 4.6 imply the following.

► **Corollary 4.7.** $\text{score}_S(T) = |E| \cdot (4(|V| - 2) + 3) + \binom{k}{2} \cdot |V| + |V| \cdot 9\alpha + |R| \cdot \beta$.

Note that by definition, $\alpha = 2(8|V| + 1) \cdot (|E| + \binom{k}{2}) > |E| \cdot (4(|V| - 2) + 3) + \binom{k}{2} \cdot |V|$. Hence, $\text{score}_S(T) < \alpha \cdot (9|V| + 1) + |R| \cdot \beta$.

► **Corollary 4.8.** *Let T' be a binary X -tree with $\text{score}_S(T') < \text{score}_S(T)$. Then, T' is well-rooted.*

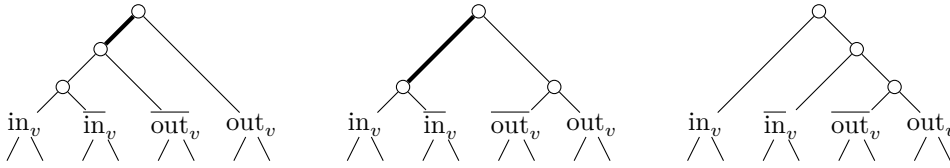
Proof. Due to Observation 4.3, T' is split-consistent for T and R and $\text{score}_{S_R}(T') = |R| \cdot \beta$. Assume towards a contradiction that there is a vertex $v \in V$ such that $T'(X_v)$ is neither an in-rooting of T_v nor an out-rooting of T_v . Hence, Lemma 4.5 implies $\text{score}_{S_v}(T') \geq 10\alpha$ and $\text{score}_{S_w}(T') \geq 9\alpha$ for each vertex $w \in V \setminus \{v\}$. Consequently, $\text{score}_S(T') \geq 10\alpha + (|V| - 1) \cdot 9\alpha + |R| \cdot \beta = \alpha \cdot (9|V| + 1) + |R| \cdot \beta > \text{score}_S(T)$, a contradiction. \blacktriangleleft

Distances between well-rooted binary X -trees. Next, we describe for each distance measure $d \in \{d_{\text{NNI}}, d_{\text{ECR}}, d_{\text{SPR}}, d_{\text{TBR}}\}$ the distance between T and any other well-rooted binary X -tree T' .

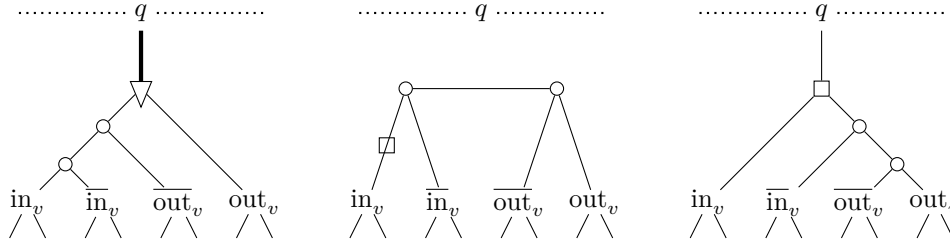
► **Lemma 4.9.** *Let T' be a binary and well-rooted X -tree. Moreover, let K be the set of vertices of V such that $T'(X_v)$ is an in-rooting of T_v for each vertex $v \in K$ and $T'(X_w)$ is an out-rooting of T_w for each vertex $w \in V \setminus K$. Then, $d_{\text{NNI}}(T, T') = d_{\text{ECR}}(T, T') = 2 \cdot |K|$ and $d_{\text{SPR}}(T, T') = d_{\text{TBR}}(T, T') = |K|$.*

Proof. First, we show that $d_{\text{NNI}}(T, T') = d_{\text{ECR}}(T, T') = 2 \cdot |K|$. To this end, we show that $d_{\text{NNI}}(T, T') \leq 2 \cdot |K|$ and that $d_{\text{ECR}}(T, T') \geq 2 \cdot |K|$. Since $d_{\text{NNI}}(T, T') \geq d_{\text{ECR}}(T, T')$ due to Lemma 3.4, this then implies $d_{\text{NNI}}(T, T') = d_{\text{ECR}}(T, T') = 2 \cdot |K|$.

To show that $d_{\text{NNI}}(T, T') \leq 2 \cdot |K|$, we prove the following: Let \tilde{T} be a well-rooted binary X -tree and let v be a vertex such that $\tilde{T}(X_v)$ is an out-rooting of T_v . Then, $d_{\text{NNI}}(\tilde{T}, \hat{T}) \leq 2$, where \hat{T} is a well-rooted binary X -tree with $\tilde{T}(X \setminus X_v) = \hat{T}(X \setminus X_v)$ and where $\hat{T}(X_v)$



■ **Figure 3** The two consecutive NNI operation transforming an out-rooting into an in-rooting.



■ **Figure 4** Transforming an out-rooting into an in-rooting by an SPR operation. First, the bold edge is removed and the triangular vertex is suppressed. Second, the unique internal edge incident with in_v is subdivided by the rectangular vertex. Finally, the rectangular vertex is joined with q by a new edge.

is an in-rooting of T_v . To show the claim, we describe two consecutive NNI operations transforming \tilde{T} into \hat{T} . See Figure 3 for an illustration of these NNI operations. Let r_v^{out} be name of the pseudo-root of the pendant tree $\tilde{T}(X_v)$, let r_v^{out} be the name of the common neighbor of r_v^{mid} and out_v in \tilde{T} , and let r_v^{mid} be the name of the common neighbor of r_v^{in} and $\overline{\text{out}}_v$ in \tilde{T} . Moreover, let q be the unique neighbor of r_v^{out} outside of $\tilde{T}(X_v)$ in \tilde{T} . We obtain the well-rooted binary X -tree \hat{T} from \tilde{T} by

- firstly removing the edges $\{q, r_v^{\text{out}}\}$ and $\{\overline{\text{out}}_v, r_v^{\text{mid}}\}$ and adding the edges $\{\overline{\text{out}}_v, r_v^{\text{out}}\}$ and $\{q, r_v^{\text{mid}}\}$, and
- secondly removing the edges $\{q, r_v^{\text{mid}}\}$ and $\{\overline{\text{in}}_v, r_v^{\text{in}}\}$ and adding the edges $\{\overline{\text{in}}_v, r_v^{\text{mid}}\}$ and $\{q, r_v^{\text{in}}\}$.

Since this can be done by two consecutive NNI operations and $\tilde{T}(X \setminus X_v) = \hat{T}(X \setminus X_v)$, we conclude $d_{\text{NNI}}(\tilde{T}, \hat{T}) \leq 2$. Since d_{NNI} is a metric one can then show via induction over any arbitrary ordering of the vertices of K , that $d_{\text{NNI}}(T, T') \leq 2 \cdot |K|$.

It remains to show that $d_{\text{ECR}}(T, T') \geq 2 \cdot |K|$. Let \tilde{E} be a subset of the internal edges of T , such that T' can be obtained from T by an ECR operation with contraction set \tilde{E} . We show that $|\tilde{E}| \geq 2 \cdot |K|$. Let v be a vertex of K . Recall that T_v is an out-rooting of T_v and that T'_v is an in-rooting of T_v . Hence, the edge $\{r_v^{\text{out}}, r_v^{\text{mid}}\}$ induces the split $A|B$ in T with $A := \{\text{in}_v^0, \text{in}_v^1, \overline{\text{in}}_v^0, \overline{\text{in}}_v^1, \overline{\text{out}}_v^0, \overline{\text{out}}_v^1\}$ and $B := X \setminus A$. Since $A|B$ is not a split of T' , the edge $\{r_v^{\text{out}}, r_v^{\text{mid}}\}$ is contained in \tilde{E} . Similar, since the edge $\{r_v^{\text{mid}}, r_v^{\text{in}}\}$ induces the split $A|B$ in T with $A := \{\text{in}_v^0, \text{in}_v^1, \overline{\text{in}}_v^0, \overline{\text{in}}_v^1\}$ and $B := X \setminus A$. Since $A|B$ is not a split of T' , the edge $\{r_v^{\text{mid}}, r_v^{\text{in}}\}$ is contained in \tilde{E} . Hence, for each vertex v of V , \tilde{E} contains at least two edges of $T(X_v)$. Consequently, $|\tilde{E}| \geq 2 \cdot |K|$ which implies $d_{\text{ECR}}(T, T') \geq 2 \cdot |K|$.

Second, we show that $d_{\text{SPR}}(T, T') = d_{\text{TBR}}(T, T') = |K|$. Similar to the first part of the proof, we show that $d_{\text{SPR}}(T, T') \leq |K|$ and that $d_{\text{TBR}}(T, T') \geq |K|$. Since $d_{\text{SPR}}(T, T') \geq d_{\text{TBR}}(T, T')$ this then implies $d_{\text{SPR}}(T, T') = d_{\text{TBR}}(T, T') = |K|$.

To show that $d_{\text{SPR}}(T, T') \leq |K|$, we prove the following: Let \tilde{T} be a well-rooted binary X -tree and let v be a vertex such that $\tilde{T}(X_v)$ is an out-rooting of T_v . Then, $d_{\text{SPR}}(\tilde{T}, \hat{T}) \leq 1$, where \hat{T} is a well-rooted binary X -tree with $\hat{T}(X \setminus X_v) = \tilde{T}(X \setminus X_v)$ and where $\hat{T}(X_v)$ is an in-rooting of T_v .

To show this claim, we describe an SPR operation transforming \tilde{T} into \hat{T} . See Figure 4 for an illustration of this SPR operation. Let r_v^{out} be the name of the pseudo-root of the pendant tree $\tilde{T}(X_v)$ and let q be the name of the unique neighbor of r_v^{out} outside of $\tilde{T}(X_v)$ in \tilde{T} . Moreover, let r_v^{in} be the name of the common neighbor of in_v and $\overline{\text{in}}_v$ in \tilde{T} . We obtain the well-rooted binary X -tree \hat{T} from \tilde{T} by: removing the edge $\{r_v^{\text{out}}, q\}$, suppressing the vertex r_v^{out} , subdividing the edge $\{\text{in}_v, r_v^{\text{in}}\}$ by a vertex q' , and adding the edge $\{q, q'\}$. Since this can be done by a single SPR operation and $\tilde{T}(X \setminus X_v) = \hat{T}(X \setminus X_v)$, we conclude $d_{\text{SPR}}(\tilde{T}, \hat{T}) \leq 1$. Since d_{SPR} is a metric, one can then show via induction over any arbitrary ordering of the vertices of K , that $d_{\text{SPR}}(T, T') \leq |K|$.

It remains to show that $d_{\text{TBR}}(T, T') \geq |K|$. This proof is deferred to a full version of the article. \blacktriangleleft

Correctness. Finally, we are able to show that I is a yes-instance of CLIQUE if and only if I' is a yes-instance of d -LS MAXIMUM PARSIMONY with appropriate distance bounds.

► **Lemma 4.10.** *The following statements are equivalent:*

1. *There is a clique of size k in G .*
2. *There is a binary X -tree T' with $\text{score}_S(T') < \text{score}_S(T)$ and $d_{\text{SPR}}(T, T') \leq k$.*
3. *There is a binary X -tree T' with $\text{score}_S(T') < \text{score}_S(T)$ and $d_{\text{TBR}}(T, T') \leq k$.*
4. *There is a binary X -tree T' with $\text{score}_S(T') < \text{score}_S(T)$ and $d_{\text{NNI}}(T, T') \leq 2k$.*
5. *There is a binary X -tree T' with $\text{score}_S(T') < \text{score}_S(T)$ and $d_{\text{ECR}}(T, T') \leq 2k$.*

Proof. First, we show that Item 1 implies each of Item 2–5. Let $K \subseteq V$ be a clique of size k in G . Further, let T' be a well-rooted binary X -tree such that for each vertex $v \in K$, $T'(X_v)$ is an in-rooting of T_v , and for each vertex $v \in V \setminus K$, $T'(X_v)$ is an out-rooting of T_v . Due to Lemma 4.9, $d_{\text{SPR}}(T, T') = d_{\text{TBR}}(T, T') = k$ and $d_{\text{NNI}}(T, T') = d_{\text{ECR}}(T, T') = 2k$. It remains to show that $\text{score}_S(T') < \text{score}_S(T)$. Since T' is well-rooted, due to Observation 4.3, $\text{score}_{S_R}(T') = |R| \cdot \beta$ and due to Lemma 4.5, for each vertex $v \in V$, $\text{score}_{S_v}(T') = 9\alpha$. Moreover, since K is non-empty, we obtain by Lemma 4.6, that $\text{score}_{S_{\text{mal}}}(T') = \binom{k}{2} \cdot (|V| + 1)$. Since K is a clique in G , $|E(K)| = \binom{k}{2}$. Finally, by Lemma 4.6, for each edge e of $E(K)$, $\text{score}_{c_e}(T') = 4(|V| - 2) + 2$, and for each edge e of $E \setminus E(K)$, $\text{score}_{c_e}(T') = 4(|V| - 2) + 3$. We conclude

$$\begin{aligned} \text{score}_S(T') &= |E| \cdot (4(|V| - 2) + 3) - \binom{k}{2} + \left(\binom{k}{2} - 1 \right) \cdot (|V| + 1) + |V| \cdot 9\alpha + |R| \cdot \beta \\ &= |E| \cdot (4(|V| - 2) + 3) + \left(\binom{k}{2} - 1 \right) \cdot |V| + |V| \cdot 9\alpha + |R| \cdot \beta - 1 = \text{score}_S(T) - 1, \end{aligned}$$

due to Corollary 4.7. Hence, T' is a binary X -tree with $\text{score}_S(T') < \text{score}_S(T)$, $d_{\text{SPR}}(T, T') = d_{\text{TBR}}(T, T') = k$, and $d_{\text{NNI}}(T, T') = d_{\text{ECR}}(T, T') = 2k$.

Second, we show that each of Item 2–5 implies Item 1. Let T' be a binary X -tree with a) $\text{score}_S(T') < \text{score}_S(T)$ and b) $d_{\text{SPR}}(T, T') \leq k$, $d_{\text{TBR}}(T, T') \leq k$, $d_{\text{NNI}}(T, T') \leq 2k$, or $d_{\text{ECR}}(T, T') \leq 2k$. Since $\text{score}_S(T') < \text{score}_S(T)$, due to Corollary 4.8, T' is well-rooted, that is, for each vertex $v \in V$, $T'_v := T'(X_v)$ is either an in-rooting of T_v or an out-rooting of T_v . Let $K \subseteq V$ be the set of all vertices v of V where T'_v is an in-rooting of T_v . We show that K is a clique of size k in G . Since $d_{\text{SPR}}(T, T') \leq k$, $d_{\text{TBR}}(T, T') \leq k$, $d_{\text{NNI}}(T, T') \leq 2k$, or $d_{\text{ECR}}(T, T') \leq 2k$, Lemma 4.9 implies that K has size at most k . Moreover, since $\text{score}_S(T') < \text{score}_S(T)$, T' is not isomorphic to T , which implies that K is nonempty. Hence due to Lemma 4.6, $\text{score}_{S_{\text{mal}}}(T') = \binom{k}{2} \cdot (|V| + 1)$. Moreover, since T' is well-rooted, due to Observation 4.3, $\text{score}_{S_R}(T') = |R| \cdot \beta$ and due to Lemma 4.5,

for each vertex $v \in V$, $\text{score}_{S_v}(T') = 9\alpha$. Finally, by Lemma 4.6, for each edge $e \in E \setminus E(K)$, $\text{score}_{c_e}(T') = 4(|V|-2)+3$, and for each edge $e \in E(K)$, $\text{score}_{c_e}(T') = 4(|V|-2)+2$. Consequently, $\text{score}_S(T) - \text{score}_S(T') = |E(K)| - \binom{k}{2} - 1$.

Since $\text{score}_S(T') < \text{score}_S(T)$, we have $|E(K)| \geq \binom{k}{2}$. Hence, K is a size- k clique in G . ◀

Since $k' = k$ if $d \in \{d_{\text{SPR}}, d_{\text{TBR}}\}$ and $k' = 2k$ if $d \in \{d_{\text{NNI}}, d_{\text{ECR}}\}$, Lemma 4.10 implies that I is a yes-instance of CLIQUE if and only if I' is a yes-instance of d -LS MAXIMUM PARSIMONY. This completes the proof of Theorem 4.1.

5 Essentially Tight Brute-Force Algorithms

We now show that simple brute-force algorithms for d -LS MAXIMUM PARSIMONY for each distance measure $d \in \{d_{\text{NNI}}, d_{\text{ECR}}, d_{\text{SPR}}, d_{\text{TBR}}\}$ essentially match the lower bounds shown in Theorem 4.1. First, consider a distance measure $d \in \{d_{\text{NNI}}, d_{\text{SPR}}, d_{\text{TBR}}\}$.

► **Observation 5.1.** *Let T be a binary X -tree, let $d \in \{d_{\text{NNI}}, d_{\text{SPR}}, d_{\text{TBR}}\}$ be a distance measure, and let k be an integer. One can enumerate all binary X -trees T' with $d(T, T') \leq k$ in $|X|^{\mathcal{O}(k)}$ time.*

Observation 5.1 can be seen as follows: there are $|X|^{\mathcal{O}(1)}$ many binary X -trees T' such that $d(T, T') = 1$, all these trees can be enumerated in $|X|^{\mathcal{O}(1)}$ time, and for each binary X -tree T' with $d(T, T') > 0$, there is a binary X -tree \hat{T} with $d(\hat{T}, T') = 1$ and $d(T, T') = d(T, \hat{T}) + 1$.

Furthermore, we may enumerate all binary X -trees T' with $d_{\text{sECR}}(T, T') \leq k$ as follows: First, we enumerate all subtrees of T with at most k edges. Second, for each connected subtree T_s of T with at most k edges, we enumerate all binary refinements of T after contracting all edges of T_s . In Lemma 5.2, we show that the first step can be done in $\mathcal{O}(4^k \cdot k^{-0.5} \cdot |X|)$ time. In Lemma 5.3, we show that both steps can be performed in $\mathcal{O}((2k+1)!! \cdot 4^k \cdot k\sqrt{k} \cdot |X|^2)$ time where $(2k+1)!! := 1 \cdot 3 \cdot \dots \cdot (2k+1)$.

► **Lemma 5.2 (*)**. *For every binary X -tree T and every integer k , all connected subtrees of T with at most k edges can be enumerated in $\mathcal{O}(4^k \cdot k^{-0.5} \cdot |X|)$ time.*

► **Lemma 5.3 (*)**. *For a given binary X -tree T and an integer k , there are $\mathcal{O}((2k+1)!! \cdot 4^k \cdot k^{-0.5} \cdot |X|)$ binary X -trees T' with $d_{\text{sECR}}(T, T') \leq k$. Moreover, all these binary X -trees can be enumerated in $\mathcal{O}((2k+1)!! \cdot 4^k \cdot k\sqrt{k} \cdot |X|^2)$ time.*

Hence, we obtain the following due to the fact that the parsimony score of a given X -tree can be computed in $\mathcal{O}(|X| \cdot |S|)$ time [11].

► **Theorem 5.4.** *d_{sECR} -LS MAXIMUM PARSIMONY can be solved in $\mathcal{O}((2k+1)!! \cdot 4^k \cdot k\sqrt{k} \cdot |X|^2 \cdot |S|) = 2^{\mathcal{O}(k \cdot \log k)} \cdot |X|^2 \cdot |S|$ time.*

Finally, we describe how to enumerate all binary X -trees T' with $d_{\text{ECR}}(T, T') \leq k$.

► **Lemma 5.5.** *Let T be a binary X -tree and let k be an integer. One can enumerate all binary X -trees T' with $d_{\text{ECR}}(T, T') \leq k$ in $|X|^{\mathcal{O}(k)}$ time.*

Proof. We show this statement by induction over k .

Base case. Consider $k = 0$. Hence, T is the only binary X -tree T' with $d_{\text{ECR}}(T, T') = 0$ and can be enumerated in $|X|^{\mathcal{O}(1)}$ time.

Inductive step. For the inductive step, suppose that for each binary X -tree \tilde{T} and for each $k' < k$, one can compute all binary X -trees T' with $d_{\text{ECR}}(\tilde{T}, T') \leq k'$ in $|X|^{\mathcal{O}(k')}$ time. Note that this implies that for each $k' < k$ there are $|X|^{\mathcal{O}(k')}$ binary X -trees T' with $d_{\text{ECR}}(\tilde{T}, T') = k'$. For each $i < k$, let \mathcal{T}_i be the collection of all binary X -trees \tilde{T} with $d_{\text{ECR}}(T, \tilde{T}) = i$ and let $\mathcal{T}_{<k}$ be the collection of all binary X -trees \tilde{T} with $d_{\text{ECR}}(T, \tilde{T}) < k$, that is, $\mathcal{T}_{<k} = \cup_{i=0}^{k-1} \mathcal{T}_i$. Moreover, let $\mathcal{T}_{\text{sECR}}$ be the collection of all binary X -trees \tilde{T} with $d_{\text{sECR}}(T, \tilde{T}) = k$. Note that $\mathcal{T}_{<k}$ can be computed in $|X|^{\mathcal{O}(k-1)}$ time and due to Lemma 5.3, $\mathcal{T}_{\text{sECR}}$ can be computed in $k^{\mathcal{O}(k)} \cdot |X|^{\mathcal{O}(1)}$ time. Let

$$\mathcal{T}'_k := \mathcal{T}_{\text{sECR}} \cup \bigcup_{i=1}^{k-1} \bigcup_{\tilde{T} \in \mathcal{T}_i} \{T' \mid d_{\text{ECR}}(\tilde{T}, T') \leq k - i\}.$$

Recall that by the induction hypothesis, for each $i < k$, \mathcal{T}_i has size $|X|^{\mathcal{O}(i)}$ and for each binary X -tree $\tilde{T} \in \mathcal{T}_i$ the collection $\{T' \mid d_{\text{ECR}}(\tilde{T}, T') \leq k - i\}$ can be computed in $|X|^{\mathcal{O}(k-i)}$ time. Hence, \mathcal{T}'_k can be computed in $|X|^{\mathcal{O}(k)}$ time. We set $\mathcal{T} := \mathcal{T}'_k \cup \mathcal{T}_{<k}$ and show that \mathcal{T} contains exactly the binary X -trees T' with $d_{\text{ECR}}(T, T') \leq k$.

Assume towards a contradiction that this is not the case.

Case 1: There is a binary X -tree T' with $d_{\text{ECR}}(T, T') \leq k$ such that T' is not in \mathcal{T} . By definition, $\mathcal{T}_{<k}$ contains all binary X -trees \tilde{T} with $d_{\text{ECR}}(T, \tilde{T}) < k$. Consequently, $d_{\text{ECR}}(T, T') = k$. Hence, due to Observation 3.3, there is a binary X -tree \tilde{T} with $d_{\text{sECR}}(\tilde{T}, T') > 0$ such that $d_{\text{ECR}}(T, T') = d_{\text{ECR}}(T, \tilde{T}) + d_{\text{sECR}}(\tilde{T}, T')$. Let $i := d_{\text{ECR}}(T, \tilde{T})$.

Note that $i \leq k - 1$. If $i = 0$, then T is isomorphic to \tilde{T} and thus $d_{\text{sECR}}(T, T') = d_{\text{sECR}}(\tilde{T}, T') = k$. Hence, T' is contained in $\mathcal{T}_{\text{sECR}}$, a contradiction. Otherwise, if $i > 0$, then \tilde{T} is contained in \mathcal{T}_i . Moreover, since $d_{\text{sECR}}(\tilde{T}, T') = d_{\text{ECR}}(T, T') - d_{\text{ECR}}(T, \tilde{T}) = k - i$ and $d_{\text{sECR}}(\tilde{T}, T') \geq d_{\text{ECR}}(\tilde{T}, T')$, we have $d_{\text{ECR}}(\tilde{T}, T') \leq k - i$ which implies that T' is contained in \mathcal{T} , a contradiction.

Case 2: There is a binary X -tree T' with $d_{\text{ECR}}(T, T') > k$ such that T' is contained in \mathcal{T} . Hence, T' is contained in $\mathcal{T}'_k \setminus \mathcal{T}_{\text{sECR}}$. That is, there is some i with $1 \leq i \leq k$ and a binary X -tree \tilde{T} in \mathcal{T}_i such that $d_{\text{ECR}}(\tilde{T}, T') \leq k - i$. Since d_{ECR} is a metric, due to the triangle inequality, $d_{\text{ECR}}(T, T') \leq d_{\text{ECR}}(T, \tilde{T}) + d_{\text{ECR}}(\tilde{T}, T') \leq k$, a contradiction.

Since \mathcal{T} can be computed in $|X|^{\mathcal{O}(k)}$ time, the statement holds. \blacktriangleleft

We conclude the following.

► Theorem 5.6 (*). *For each distance measure $d \in \{d_{\text{NNI}}, d_{\text{ECR}}, d_{\text{SPR}}, d_{\text{TBR}}\}$, d -LS MAXIMUM PARSIMONY can be solved in $|X|^{\mathcal{O}(k)} \cdot |S|$ time.*

6 Conclusion

A clear goal for future research would be to improve the running time of the algorithm for the k -sECR neighborhood. This seems promising since the current bottleneck is the enumeration of the binary refinements of the tree obtained after contracting k edges. However, an algorithm for d_{sECR} -LS MAXIMUM PARSIMONY running in $2^{o(k \cdot \log k)} \cdot |I|^{\mathcal{O}(1)}$ time would imply an algorithm for MAXIMUM PARSIMONY running in $2^{o(|X| \cdot \log |X|)} \cdot |I|^{\mathcal{O}(1)}$ time: when applying the d_{sECR} -LS MAXIMUM PARSIMONY algorithm with $k := |X| - 3$, locally optimal solution are also globally optimal. Hence, a more immediate question is whether MAXIMUM

PARSIMONY can be solved in $2^{o(|X| \cdot \log |X|)} \cdot |I|^{\mathcal{O}(1)}$ time. A further goal would be to find other neighborhoods for which d -LS MAXIMUM PARSIMONY can be solved in time $f(k) \cdot |I|^{\mathcal{O}(1)}$. Finally, it is open whether better running times are possible when searching the neighborhood not for a better tree but for a perfect phylogeny, that is, for a tree where for each character, the parsimony score is equal to the number of character states minus one.

References

- 1 Benjamin L. Allen and Mike Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Ann. Comb.*, 5(1):1–15, 2001.
- 2 Alexandre A. Andreatta and Celso C. Ribeiro. Heuristics for the phylogeny problem. *J. Heuristics*, 8(4):429–447, 2002.
- 3 Hans L. Bodlaender, Michael R. Fellows, and Tandy J. Warnow. Two strikes against perfect phylogeny. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 273–283. Springer, 1992.
- 4 Amir Carmel, Noa Musa-Lempel, Dekel Tsur, and Michal Ziv-Ukelson. The worst case complexity of maximum parsimony. *J. Comput. Biol.*, 21(11):799–808, 2014.
- 5 Jianer Chen, Benny Chor, Mike Fellows, Xiuzhen Huang, David W. Juedes, Iyad A. Kanj, and Ge Xia. Tight lower bounds for certain parameterized NP-hard problems. *Inf. Comput.*, 201(2):216–231, 2005. doi:10.1016/j.ic.2005.05.001.
- 6 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.
- 7 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. doi:10.1007/978-1-4471-5559-1.
- 8 Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances A. Rosamond, Saket Saurabh, and Yngve Villanger. Local search: Is brute-force avoidable? *J. Comput. Syst. Sci.*, 78(3):707–719, 2012. doi:10.1016/j.jcss.2011.10.003.
- 9 Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates Sunderland, 2004.
- 10 David Fernández-Baca and Jens Lagergren. A polynomial-time algorithm for near-perfect phylogeny. *SIAM J. Comput.*, 32(5):1115–1127, 2003. doi:10.1137/S0097539799350839.
- 11 Walter M. Fitch. Toward defining the course of evolution: minimum change for a specific tree topology. *Systematic Biology*, 20(4):406–416, 1971.
- 12 Les R. Foulds and Ronald L. Graham. The Steiner problem in phylogeny is NP-complete. *Adv. Appl. Math.*, 3(1):43–49, 1982.
- 13 Ganeshkumar Ganapathy, Vijaya Ramachandran, and Tandy Warnow. On contract-and-refine transformations between phylogenetic trees. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, pages 900–909, 2004.
- 14 Ganeshkumar Ganapathy, Vijaya Ramachandran, and Tandy J. Warnow. Better hill-climbing searches for parsimony. In *Proceedings of the 3rd International Workshop on Algorithms in Bioinformatics (WABI '03)*, volume 2812 of *Lecture Notes in Computer Science*, pages 245–258. Springer, 2003. doi:10.1007/978-3-540-39763-2_19.
- 15 Serge Gaspers, Eun Jung Kim, Sebastian Ordyniak, Saket Saurabh, and Stefan Szeider. Don't be strict in local search! In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI '12)*. AAAI Press, 2012.
- 16 Adrien Goëffon, Jean-Michel Richer, and Jin-Kao Hao. Local search for the maximum parsimony problem. In *Proceedings of the First International Conference on Advances in Natural Computation (ICNC '05)*, volume 3612 of *Lecture Notes in Computer Science*, pages 678–683. Springer, 2005.

- 17 Adrien Goëffon, Jean-Michel Richer, and Jin-Kao Hao. Progressive tree neighborhood applied to the maximum parsimony problem. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 5(1):136–145, 2008. doi:10.1109/TCBB.2007.1065.
- 18 Pablo A Goloboff. Character optimization and calculation of tree lengths. *Cladistics*, 9(4):433–436, 1993.
- 19 Pablo A. Goloboff. Analyzing large data sets in reasonable times: Solutions for composite optima. *Cladistics*, 15(4):415–428, 1999. doi:10.1006/clad.1999.0122.
- 20 Maozu Guo, Jian-Fu Li, and Yang Liu. Improving the efficiency of p-ECR moves in evolutionary tree search methods based on maximum likelihood by neighbor joining. In *Proceeding of the Second International Multi-Symposium of Computer and Computational Sciences (IMSCCS '07)*, pages 60–67. IEEE Computer Society, 2007.
- 21 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.
- 22 Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 23 Christian Komusiewicz and Nils Morawietz. Parameterized local search for vertex cover: When only the search radius is crucial. In *Proceedings of the 17th International Symposium on Parameterized and Exact Computation (IPEC '22)*, volume 249 of *LIPICs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- 24 Dániel Marx. Searching the k -change neighborhood for TSP is $W[1]$ -hard. *Oper. Res. Lett.*, 36(1):31–36, 2008.
- 25 Kevin C. Nixon. The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, 15(4):407–414, 1999. doi:10.1006/clad.1999.0121.
- 26 Celso C. Ribeiro and Dalessandro Soares Vianna. A GRASP/VND heuristic for the phylogeny problem using a new neighborhood structure. *Int. Trans. Oper. Res.*, 12(3):325–338, 2005.
- 27 David F. Robinson. Comparison of labeled trees with valency three. *J. Comb. Theory B*, 11(2):105–119, 1971.
- 28 David Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl. Math.*, 28(1):35–42, 1975.
- 29 David Sankoff, Yvon Abel, and Jotun Hein. A tree · a window · a hill; generalization of nearest-neighbor interchange in phylogenetic optimization. *J. Classif.*, 11(2):209–232, 1994.
- 30 Srinath Sridhar, Kedar Dhamdhere, Guy E. Blelloch, Eran Halperin, R. Ravi, and Russell Schwartz. Algorithms for efficient near-perfect phylogenetic tree reconstruction in theory and practice. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, 4(4):561–571, 2007. doi:10.1109/TCBB.2007.1070.

String Factorization via Prefix Free Families

Matan Kraus ✉

Bar-Ilan University, Ramat-Gan, Israel

Moshe Lewenstein ✉

Bar-Ilan University, Ramat-Gan, Israel

Alexandru Popa ✉

Faculty of Mathematics and Computer Science, University of Bucharest, Romania

Ely Porat ✉

Bar-Ilan University, Ramat-Gan, Israel

Yonathan Sadia ✉

Bar-Ilan University, Ramat-Gan, Israel

Abstract

A factorization of a string S is a partition of w into substrings u_1, \dots, u_k such that $S = u_1 u_2 \dots u_k$. Such a partition is called equality-free if no two factors are equal: $u_i \neq u_j, \forall i, j$ with $i \neq j$. The *maximum equality-free factorization problem* is to find for a given string S , the largest integer k for which S admits an equality-free factorization with k factors.

Equality-free factorizations have lately received attention because of their applications in DNA self-assembly. The best approximation algorithm known for the problem is the natural greedy algorithm, that chooses iteratively from left to right the shortest factor that does not appear before. This algorithm has a \sqrt{n} approximation ratio (SOFSEM 2020) and it is an open problem whether there is a better solution.

Our main result is to show that the natural greedy algorithm is a $\Theta(n^{1/4})$ approximation algorithm for the maximum equality-free factorization problem. Thus, we disprove one of the conjectures of Mincu and Popa (SOFSEM 2020) according to which the greedy algorithm is a $\Theta(\sqrt{n})$ approximation.

The most challenging part of the proof is to show that the greedy algorithm is an $O(n^{1/4})$ approximation. We obtain this algorithm via *prefix free* factor families, i.e. a set of non-overlapping factors of the string which are pairwise non-prefixes of each other. In the paper we show the relation between prefix free factor families and the maximum equality-free factorization. Moreover, as a byproduct we present another approximation algorithm that achieves an approximation ratio of $O(n^{1/4})$ that we believe is of independent interest and may lead to improved algorithms. We then show that the natural greedy algorithm has an approximation ratio that is $\Omega(n^{1/4})$ via a clever analysis which shows that the greedy algorithm is $\Theta(n^{1/4})$ for the maximum equality-free factorization problem.

2012 ACM Subject Classification Theory of computation \rightarrow Approximation algorithms analysis

Keywords and phrases string factorization, NP-hard problem, approximation algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.19

Funding This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0253, within PNCDI III. Matan kraus, Ely Porat and Yonathan Sadia were supported by ISF grants no. 1278/16 and 1926/19, by a BSF grant 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).



© Matan Kraus, Moshe Lewenstein, Alexandru Popa, Ely Porat, and Yonathan Sadia; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 19; pp. 19:1–19:10

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A factorization of a string S is a partition of S into substrings u_1, u_2, \dots, u_k such that $S = u_1 u_2 \dots u_k$. Factorizations are central objects of study in stringology, a famous example being the Lempel-Ziv algorithm [14]. String factorizations have many other applications as we show next. For instance, finding an occurrence of a string v in a text T can be formulated as T admitting a factorization $T = uvw$. Then, a string v is a prefix of another string T if $T = vw$ and it is a suffix of T if $T = uv$. Moreover, many string problems can be seen as string factorization problems [9] such as: SHORTEST COMMON SUPERSTRING, LONGEST COMMON SUBSEQUENCE and SHORTEST COMMON SUPERSEQUENCE, to name a few. Another example of a string factorization problem is the MINIMUM COMMON STRING PARTITION [6, 7], a problem concerned with identifying factorizations for two strings such that the sequence of factors for one string is the permutation of the other's.

In this paper we focus on the equality-free factorization, a special case of string factorization in which all factors are distinct. The equality-free factorization is a restricted variant of a more famous problem, termed *generalized function matching* which has a long history starting from 1979 (see, e.g., [12] and the references therein for more details). In the generalized function matching the input consists of a text t over an alphabet Σ_1 and a pattern $p = p_1 p_2 \dots p_m$ over an alphabet Σ_2 . The goal is to find an injective function from $f : \Sigma_2 \rightarrow \Sigma_1^+$ such that $t = f(p_1) f(p_2) \dots f(p_m)$. Thus, the maximum equality free factorization problem is a particular case of the generalized function matching in which all the characters of the pattern p are distinct. In turn, generalized function matching is a particular case of string equations, which is a notoriously difficult problem (see, e.g., the JACM paper [13]). In fact, even the version which restricts character-to-character function matching is extremely difficult, see [1], as opposed to the more restricted parameterized matching [2, 3, 10] which is simpler. Thus, maximum equality free factorization is part of family of fundamental problems in stringology.

The maximum equality free factorization problem is also motivated by applications in DNA synthesis [4]. More specifically, it is possible to produce short DNA fragments that will self-assemble into the wanted DNA structure. However, to obtain the desired structure, it is required that no two fragments are identical. Since the fragments must be short, one approach is to split the target DNA sequence into as many distinct pieces as possible.

Previous work

The equality-free factorization problem was first introduced by Condon, Mañuch and Thachuk [4] where it was presented as the *string partitioning problem*. The string partitioning problem asks for a factorization into distinct factors such that each factor is at most of a certain length. The problem was studied in a more general setting where the measure of collision between two factors is either equality or one is a prefix/suffix of the other. Condon et al. showed that these variants are \mathcal{NP} -complete. More recently, Fernau, Manea, Mercas and Schmid [5] presented a similar problem that imposes a lower bound on the number of factors instead of an upper bound on factor length. Fernau et al. showed that this variant is also \mathcal{NP} -complete. Afterwards, Schmid [9] studied the Fixed-Parameter Tractability of the two problems.

The decision version of the problem, that is, given a string S and an integer k , decide if there exists an equality free factorization of S with at least k factors, is termed MAXEFF-s (this is the notation of Schmid [9] and we decide to use it for the sake of consistency). The optimization version, in which we are given S and the goal is to find an equality free

factorization with as many factors as possible, is termed OPTEFF-s. The acronyms for the two problems were introduced in the previous papers [5, 9, 11] and we will use them in our paper for consistency (OPT stands for *optimization*).

Mincu and Popa [11] study OPTEFF-s and another variant named Maximum Gapped Equality Free Factorization (OPTGEFF-s). In the latter it is *not* required that all the characters of the input strings are part of the factorization. That is, the goal is to find an equality free factorization using a maximum number of factors of a substring of the input string. More formally, a *gapped factorization* of string S over alphabet Σ is a tuple (u_1, u_2, \dots, u_k) such that $S = \alpha_0 u_1 \alpha_1 u_2 \alpha_2 \dots \alpha_{k-1} u_k \alpha_k$, where $u_i \in \Sigma^+$ are the factors and $\alpha_i \in \Sigma^*$ are the gaps. OPTGEFF-s asks, for a given string S , to find the largest integer k such that S admits a gapped equality-free factorization of size k . In [11] a 2-approximation for the OPTGEFF-s and a \sqrt{n} -approximation (where n is the size of the input string) for the OPTEFF-s were shown. Moreover, it was conjectured [11] that the greedy approximation ratio is $\Omega(\sqrt{n})$. Grüttemeier et al. [8] show a randomized algorithm for solving the MAXEFF-s with running time $2^k \cdot k^{O(1)} + O(n)$.

Our results

As mentioned, the best-known approximation algorithm, the greedy algorithm, for the OPTEFF-s has an approximation ratio of \sqrt{n} and it was conjectured that the greedy algorithm has an approximation ratio of $\Theta(\sqrt{n})$. In this paper, we show a better approximation algorithm for OPTEFF-s with ratio $O(n^{1/4})$. We then use this algorithm to show that the greedy algorithm has the same approximation ratio of $O(n^{1/4})$. Hence, this disproves the conjecture from [11] saying that the approximation ratio of the greedy algorithm is $\Omega(\sqrt{n})$.

The challenge is to show that the greedy algorithm has an approximation ratio of $O(n^{1/4})$. To get our approximation ratio we start with an (approximate) prefix free solution for the version with gaps. Then, we use the prefix free property to map the factors of a solution returned by the greedy algorithm to the aforementioned prefix free solution. Moreover, besides the greedy algorithm, we introduce another approximation algorithm for OPTEFF-s with an approximation ratio of $O(n^{1/4})$, that uses some interesting techniques and is of independent interest. We claim that our techniques give some key insights and perhaps open the path for better approximation algorithms for the problem.

Finally, we use a clever analysis to also show that the greedy algorithm cannot have an approximation ratio better than $O(n^{1/4})$ and, hence, the approximation ratio of the greedy algorithm is $\Theta(n^{1/4})$ for the maximum equality-free factorization problem.

2 The prefix-free property

For the OPTGEFF-s problem (the version of factorization with gaps), a 2-approximation algorithm via a reduction from a scheduling problem was shown in [11]. A natural direction for proving an approximation algorithm for OPTEFF-s is to transform a factorization with gaps, obtained from the approximation algorithm of OPTGEFF-s, into a solution without gaps. However, it is difficult to transform a given factorization with gaps into a factorization without gaps with roughly the same number of factors.

In this section we show that it is possible to transform a special case of a factorization with gaps into a factorization without gaps. We introduce the notion of a *prefix-free gapped factorization*, which has an important role in our algorithm and might be of independent interest.

19:4 String Factorization via Prefix Free Families

► **Definition 1.** Let $n, k \in \mathbb{N}$, let S be a string of length n and let $F_k = \{S_1, S_2, \dots, S_k\}$ be a set of non-overlapping factors of S (possibly with gaps). F is a prefix-free gapped factorization of S if for all $i \neq j$, S_i is not a prefix of S_j .

Given a prefix-free gapped factorization F such that $|F| = k$, we prove that there is a transformation of F into a factorization without gaps with the same number of factors k , since each factor S_i can be extended until the next factor S_{i+1} without colliding with another factor S_j .

► **Lemma 2.** Let $n, k \in \mathbb{N}$ and let S be a string of length n with a prefix-free gapped factorization F with $|F| = k$. Then, there is an equality free factorization for S without gaps with at least k factors.

Proof. Denote $S = T_0S_1T_1S_2T_2S_2 \dots T_{k-1}S_kT_k$. Denote with $R_i = S_iT_i$. Note that for each $i \neq j$, R_i and R_j are not prefixes of each other because their prefixes are S_i and S_j , respectively, which are not prefixes of each other.

Now, consider $S = T_0R_1R_2 \dots R_k$. If, for all i , $T_0 \neq R_i$, then we have an equality free $(k+1)$ -factorization. Otherwise, there exists an i such that $T_0 = R_i$. We distinguish two cases.

In the first case, if $i < k$, then we set $Q_i = R_iR_{i+1}$. Thus, Q_i and all other factors R_j are still not prefixes of each other. T_0 , which equals R_i , is not a prefix of any other R_j (because it equaled R_i) and is shorter than Q_i . Hence S has a k -factorization $S = R_iR_1R_2 \dots R_{i-1}Q_iR_{i+2} \dots R_k$.

In the second case, if $i = k$, then we set $Q_k = R_{k-1}R_k$ and using a similar argument as above we obtain a k -factorization $S = R_kR_1R_2 \dots R_{k-2}Q_k$. ◀

3 An $O(n^{1/4})$ -approximation algorithm

In this section we show an algorithm that has an $O(n^{1/4})$ -approximation to OPTEFF-s.

Our algorithm (see Algorithm 1) is composed of two algorithms: a greedy algorithm, called Greedy1 (see Algorithm 3), which always yields an \sqrt{n} -approximation, and a new algorithm (Algorithm 2) which is described next. Algorithm 1 simply selects the better of the two algorithms and returns it.

■ **Algorithm 1** An $O(n^{1/4})$ -approximation algorithm for OPTEFF-s.

Input: String S .

- 1 $F \leftarrow \text{Algorithm 2}(S)$;
- 2 $G \leftarrow \text{Algorithm 3}(S)$;
- 3 **if** $|G| > |F|$ **then**
- 4 **return** G
- 5 **return** F

The basic idea behind Algorithm 2 is to find, for every fixed integer $1 \leq i \leq 2\sqrt{n}$, a greedy equality free gapped factorization of the input string in which every factor has length exactly i . The algorithm chooses from these gapped fixed length factorizations, the factorization with the most factors. Then, due to Lemma 2, we append to each of these factors the following adjacent (possibly empty) gap and we obtain an equality free factorization. See Algorithm 2 for more details.

■ **Algorithm 2** Fixed length greedy factorization.

Input: String S .

- 1 $F \leftarrow \emptyset$;
- 2 **for** $i \leftarrow 1$ **to** $2\sqrt{n}$ **do**
- 3 $j \leftarrow 1, G \leftarrow \emptyset$;
- 4 **while** $j \leq n - i$ **do**
- 5 **if** $S[j..j+i-1] \notin G$ **then**
- 6 $G \leftarrow G \cup \{S[j..j+i-1]\}$;
- 7 $j \leftarrow j+i-1$;
- 8 $j \leftarrow j+1$;
- 9 **if** $|G| > |F|$ **then**
- 10 $F \leftarrow G$;
- 11 Extend each factor $w_i \in F$ until factor w_{i+1} (for the last factor, extend it until the end of S);
- 12 **return** F

► **Lemma 3.** *Algorithm 2 yields an equality-free factorization without gaps.*

Proof. First, in the for loop, at each step, Algorithm 2 adds to G only distinct substrings of S . Then, notice that for every two factors $w_1, w_2 \in G$, it holds that w_1 is not a prefix of w_2 , since both w_1 and w_2 have the same length. Therefore, G is a prefix-free gapped factorization, and due to Lemma 2, the factors are extended as in line 11 to have an equality-free factorization without gaps. ◀

Analysis

Here we prove that when the optimal solution, denoted OPT , has “many” factors, Algorithm 2 returns a good approximation of OPT .

Formally, let F be the factorization returned by Algorithm 2. Let α be $n/|OPT|$. Notice that $|OPT| = n/\alpha$. We claim that $|F| = \Omega(n/\alpha^2)$.

We first give an overview of the proof. First, we prove in Lemma 5 that there are at least $n/2\alpha$ short factors (of length at most 2α) in OPT . Then we prove in Lemma 6 that there are at least $\Omega(n/\alpha^2)$ factors of the exact same length in OPT . Next, we prove in Lemma 9 that the factorization F returned in Algorithm 2 is a 2-approximation of optimal fixed length factorization (see Definition 7). Finally, we prove in Lemma 10 that $|F| = \Omega(n/\alpha^2)$.

► **Definition 4.** *An x -short factor of S is a factor of length $\leq x$. An x -long factor of S is a factor of length $> x$. When x is clear we will simply call them short factors and long factors.*

► **Lemma 5.** *There exist at least $n/2\alpha$ factors in OPT that are 2α -short.*

Proof. Let LF denote the set of 2α -long factors in OPT and SF denote set of the 2α -short factors in OPT . We will use an argument on n , the length of S . Each long factor must be, by definition, of length $\geq 2\alpha + 1$. Hence, by length arguments, $|LF| \cdot (2\alpha + 1) + |SF| \cdot 1 \leq n$ and, hence, $|LF| \leq n/(2\alpha+1) - |SF|/(2\alpha+1)$. On the other hand, since $|OPT| = n/\alpha$, we have that $|SF| = n/\alpha - |LF|$. Putting these two equations together yields that $|SF| = n/\alpha - |LF| \geq n/\alpha - n/(2\alpha + 1) + |SF|/(2\alpha + 1)$ and hence, $|SF| - |SF|/(2\alpha + 1) \geq n/\alpha - n/(2\alpha + 1)$ which in turn yields $2\alpha|SF|/(2\alpha + 1) \geq (n\alpha + n)/\alpha(2\alpha + 1)$. Hence, $2\alpha^2|SF| \geq n\alpha + n$ and $|SF| \geq n/2\alpha + n/2\alpha^2 \geq n/2\alpha$. ◀

19:6 String Factorization via Prefix Free Families

Next, we show that among the short factors, $\Omega(1/\alpha)$ fraction of them actually have exactly the same length.

► **Lemma 6.** *There exists an integer $\ell \leq 2\alpha$ such that there are at least $n/4\alpha^2$ short factors in OPT of length exactly ℓ .*

Proof. By Lemma 5, there are at least $n/2\alpha$ short factors in OPT . The average number of factors of each short length, is at least $\frac{n/2\alpha}{2\alpha} = n/4\alpha^2$. By the pigeonhole principle, there exists an integer $\ell \leq 2\alpha$ such that there are at least $n/4\alpha^2$ short factors in OPT of length exactly ℓ . ◀

Next we prove that Algorithm 2 is a constant approximation algorithm to the problem of gapped factorization with fixed lengths.

► **Definition 7.** *The Fixed-Length Maximum Gapped Equality-Free Factorization Size (FLOptGEFF-s) problem is defined as follows. For a given string S and an integer r , find the largest integer m , such that S admits a gapped equality-free factorization of size m where all factors are of length r .*

In [11], the problem of OPTGEFF-s is reduced to the Job Interval Selection Problem with k intervals (JISPk, see Theorem 8), which has a 2-approximation algorithm.

► **Theorem 8 (restated from [11]).** *Given n jobs containing k time intervals each, find the maximum number of intervals that can be selected such that (i) no two intervals intersect and (ii) at most one time interval is selected per job.*

Analogously to [11], FLOptGEFF-s is reducible as well to JISPk, and here we briefly show the reduction.

► **Lemma 9.** *Algorithm 2 is a 2-approximation for FLOptGEFF-s.*

Sketch Proof. We construct an instance of JISPk with $O(n)$ jobs from a string S with n characters. For each distinct substring of S with length r , we create a job. For each substring s we add $[a, b)$ as a time interval of s for all occurrences $s = S[a, b]$ in S .

Since JISPk has a 2-approximation algorithm, we have that FLOptGEFF-s has a 2-approximation algorithm as well. Moreover, the algorithm that approximates FLOptGEFF-s(S, r) for some string S and integer r is in fact the inner loop of Algorithm 2, on the iteration where $i = r$. ◀

We are ready to prove a lower bound on the number of factors returned by Algorithm 2.

► **Lemma 10.** *Let F be the factorization returned by Algorithm 2. Then, $|F| = \Omega(n/\alpha^2)$.*

Proof. By Lemma 6, there exists ℓ such that there are at least $n/4\alpha^2$ short factors in OPT of length ℓ .

Let S_{ALG}^ℓ be the number of factors of length ℓ produced by Algorithm 2, let S_{GEFF}^ℓ be FLOptGEFF-s(S, ℓ), and let S_{OPT}^ℓ be the number of factors of length ℓ in OPT .

By Lemma 9, there is a polynomial algorithm that is a 2-approximation of the number of occurrences of a factor of length ℓ in OPT . Moreover, the algorithm behind Lemma 9 is in fact the inner loop of Algorithm 2. Notice that $\ell \leq 2\alpha \leq 2n/|OPT| \leq 2\sqrt{n}$, since $|OPT| \geq \sqrt{n}$ and therefore there is an iteration where $i = \ell$. Then,

$$S_{OPT}^\ell/2 \leq S_{GEFF}^\ell/2 \leq S_{ALG}^\ell$$

where the first inequality is due to the definition of FLOptGEFF-s and the second inequality is due to Lemma 9.

Hence, combining Lemma 6 and Lemma 9, for the iteration where $i = \ell$ on line 9, $|G| = S_{ALG}^\ell \geq (n/4\alpha^2)/2$. Since the number of factors returned by the algorithm is at least $|G|$ (i.e. $|F| \geq S_{ALG}^\ell$), we have that $|F| = \Omega(n/\alpha^2)$. ◀

As stated before, Algorithm 1 is composed by two algorithms, Algorithm 2 and Algorithm 3. Algorithm 3 was introduced in [11] as Greedy1 algorithm. In a nutshell, consider starting “at the left” of the string and adding the next shortest substring (distinct from the already selected factors) to the incumbent factorization at each step of the algorithm. See [11] for details.

■ **Algorithm 3** Greedy1.

Input: String S .

- 1 $j \leftarrow 1, F \leftarrow \emptyset$;
- 2 **for** $i \leftarrow 1$ **to** n **do**
- 3 **if** $S[j..i] \notin F$ **then**
- 4 $F \leftarrow F \cup S[j..i]$;
- 5 $j \leftarrow i + 1$;
- 6 Extend the last factor of F until the end of S ;
- 7 **return** F

It was proven in [11] that Greedy1 yields an equality-free factorization. Moreover, they prove that Greedy1 produces at least $\Omega(\sqrt{n})$ factors.

► **Theorem 11.** *Algorithm 1 is an $O(n^{1/4})$ -approximation polynomial time algorithm for the OPTEFF-s problem.*

Proof. Combining Greedy1 with Algorithm 2, we have an algorithm that produces at least $\Omega(\max((n/\alpha^2), \sqrt{n}))$ factors. This gives an approximation ratio of $O(\min(\frac{n/\alpha}{n/\alpha^2}, (n/\alpha)/\sqrt{n})) = O(\min(\alpha, \sqrt{n}/\alpha))$, which is maximized at $\alpha = \sqrt{n}/\alpha$, i.e. at $\alpha = n^{1/4}$.

Finally, notice that the both Greedy1 and Algorithm 2 run in polynomial time of at most $O(n^{1.5} \log n)$. ◀

4 The natural greedy algorithm is a $\Theta(n^{1/4})$ -approximation

In this section we prove that Greedy1 itself achieves an approximation ratio of $O(n^{1/4})$.

► **Lemma 12.** *Greedy1 is a 2-approximation of Algorithm 2.*

Proof. Let S be a string, and let ℓ be a positive integer. Let F_ℓ be a fixed ℓ length gapped factorization on S such that $|F_\ell| = \text{FLOptGEFF-s}(S, \ell)$. Let F_G be the factorization that is the output of the Greedy1 algorithm. We show that $|F_G| \geq |F_\ell|/2$.

We map each factor of F_ℓ to a factor of F_G as follows. Let $f \in F_\ell$ be a factor in F_ℓ and let $i \leq j$ be two indices such that $f = S[i..j]$. In F_G , denote the factors that cover $S[i]$ and $S[j]$ as g_i and g_j , respectively. If $g_i \neq g_j$, map f to g_j . If f is a suffix of g_j , then also map f to g_j . Otherwise, f is fully contained in a factor of F_G and f is not a suffix of g_j . Then, it must be the case that there is a factor g_s in F_G such that the suffix of g_s is exactly f , as otherwise Greedy1 would have cut the factor g_j right after index j , but f is not a suffix of g_j . Therefore, map f to g_s (if there are more than one factors with f as a suffix in F_G , map to one of them arbitrarily).

19:8 String Factorization via Prefix Free Families

Now, let g be a factor in F_G , and let $\hat{i} \leq \hat{j}$ be two indices such that $g = S[\hat{i}, \hat{j}]$. We claim that there are at most two factors in F_ℓ that are mapped to g . There is at most one factor in F_ℓ that overlaps $S[\hat{i}]$, and therefore mapped to g because of overlapping two factors in F_G . Moreover, since all the factors in F_ℓ have the same size, there is at most one factor in F_ℓ such that the suffix of g is equal to the factor. Therefore, there are at most 2 factors in F_ℓ that are mapped to g . To conclude, there are at most $2 \cdot |F_G|$ factors in F_ℓ , for every ℓ .

Let F_A be the factorization returned by Algorithm 2. There is an ℓ such that $|F_\ell| \geq |F_A|$. Since we proved that $|F_G| \geq |F_\ell|/2$ for every ℓ , we also have that $|F_G| \geq |F_A|/2$. ◀

Combining the above lemma with the Theorem 11, we conclude the following theorem.

► **Theorem 13.** *The approximation ratio of Greedy1 is $O(n^{1/4})$.*

Proof. By Lemma 12, Greedy1 is a constant approximation of Algorithm 2 and therefore Greedy1 is also a constant approximation of Algorithm 1 (that simply uses Algorithm 2 and Greedy1 and returns the maximum between them). Since by Theorem 11 Algorithm 1 is an $O(n^{1/4})$ -approximation for OPT-EFF-s we have that Greedy1 is also an $O(n^{1/4})$ -approximation. ◀

5 Tightness of Algorithm 1

In this section we prove that our analysis of Algorithm 1 is actually tight. We show that there is a case where both Greedy1 and Algorithm 2 have an approximation ratio that is at least $\Omega(n^{1/4})$.

Similar to [11], we define a string S as follows. Let n be a square of an even number, i.e. there is an integer k such that $n = (2k)^2$. Let $\Sigma = \{x_1, x_2, \dots, x_{\sqrt{n}}\}$ be an alphabet. We define variables $X_1, X_2, \dots, X_{\sqrt{n}}$ such that for each variable X_i , define $X_i = x_1 x_2 \dots x_i$. The string S is defined as $S = X_1 X_2 \dots X_{\sqrt{n}}$. Note that $|S| = \Theta(n)$.

► **Lemma 14.** *There exists a factorization of S with $\Omega(n^{3/4})$ factors.*

Proof. We first factorize S into $\Omega(n^{3/4})$ factors with gaps, and afterwards we get rid of the gaps. We factorize the variables $X_1 \dots X_{\sqrt{n}/2-1}$ using only one factor. Then, the variable $X_{\sqrt{n}/2}$ is factorized into $x_1; x_2; \dots; x_{\sqrt{n}/2}$, for a total of $\sqrt{n}/2$ factors. At least $\sqrt{n}/2 - 1$ factors are produced by the variables $X_{\sqrt{n}/2+1} X_{\sqrt{n}/2+2}$ as follows. The variable $X_{\sqrt{n}/2+1}$ is factorized into $x_1 x_2; x_3 x_4; \dots$, for a total of $\lfloor |X_{\sqrt{n}/2+1}|/2 \rfloor$ factors. The variable $X_{\sqrt{n}/2+2}$ is factorized into $x_2 x_3; x_4 x_5; \dots$, also for a total of at least $\lfloor (|X_{\sqrt{n}/2+1}|)/2 \rfloor$ factors.

In general, at each iteration i , the algorithm produces factors of length i using i variables and i offsets. Each variable is of length at least $\sqrt{n}/2$, therefore at least $i \cdot \lfloor (\sqrt{n}/2)/i \rfloor \geq \sqrt{n}/2 - i$ factors are produced by i variables. For each iteration i , the r th variable X_j of iteration i produces factors of length i starting at index r with respect to the beginning of X_j . This procedure produces an equality free factorization with gaps.

There is a constant $c > 0$ such that there are $c\sqrt{\sqrt{n}/2} = cn^{1/4}$ iterations to the process. Therefore, at least

$$\sum_{i=1}^{cn^{1/4}} \sqrt{n}/2 - i \geq cn^{3/4}/2 - c^2\sqrt{n} = \Omega(n^{3/4})$$

factors are produced in this process.

We are left with handling the gaps. Notice that there are two reasons for a gap to occur. First, (1) on iteration i and variable X_j , we produce $\lfloor |X_j|/i \rfloor$ factors, and $|X_j| - i \lfloor |X_j|/i \rfloor > 0$, so we have a gap at the end of X_j . Second, (2) on iteration i and the r th variable of the iteration X_j , when X_j is not the first variable of iteration i ($r \neq 1$), the factorization of X_j does not start from x_1 but from x_r .

For gaps of type 1, let X_j be a variable that has a gap at the end of X_j . Then, if $j \neq \sqrt{n}$, we extend the first factor of X_{j+1} backwards to close the gap. This extended factor is unique since there is only one instance of $x_j x_1$ in S . If $j = \sqrt{n}$ and we are in the last variable, we extend the last factor of X_j . This extended factor is unique since there is only one instance of this length in S .

For gaps of type 2, let X_j be a variable that has a gap at the beginning of X_j . Then, we extend the last factor of X_{j-1} forward to close the gap. This extended factor is unique since there is only one instance of $x_{j-1} x_1$ in S .

For gaps with both types 1 and 2, we just handle them as gaps with type 1. ◀

On string S , Greedy1 produces $\Theta(\sqrt{n})$ factors. Hence, and by Lemma 14, we have the following corollary.

► **Corollary 15.** *The approximation ratio of Greedy1 is $\Omega(n^{1/4})$.*

On the string S described above, Algorithm 2 produces $O(\sqrt{n})$ factors. To see this, let l be some length that is being observed in line 2 of Algorithm 2. There are (at most) \sqrt{n} x_1 's in string S . Therefore there are at most \sqrt{n} factors containing x_1 . On the other hand, every factor that does not contain x_1 , must start in a unique character (since before the extension, every factor is of length exactly l). There are (at most) \sqrt{n} unique characters in S . Therefore, there are at most \sqrt{n} factors *not* containing x_1 .

Hence, and by Lemma 14, we have the following corollary.

► **Corollary 16.** *The approximation ratio of Algorithm 2 is $\Omega(n^{1/4})$.*

Finally, since both lower bounds were based on the same case of string S , we have the following corollary.

► **Corollary 17.** *The approximation ratio of Algorithm 1 is $\Omega(n^{1/4})$.*

6 Conclusions and future work

In this paper we disproved one of the conjectures of Mincu and Popa [11] and show that the natural greedy algorithm for the OPTEFF-s problem has a $\Theta(n^{1/4})$ -approximation factor. It is, of course, a natural open question to improve the approximation ratio for OPTEFF-s using a different algorithm than the greedy. We believe that the key in succeeding to obtain such an algorithm is finding a better approximation algorithm for the case when the number of factors in an optimal solution is relatively small. Thus, the ideas introduced in Section 3, where we present an alternative $O(n^{1/4})$ approximation algorithm, represent a promising direction.

References

- 1 Amihod Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.
- 2 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

19:10 String Factorization via Prefix Free Families

- 3 Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms*, 11(2):12:1–12:30, 2014.
- 4 Anne Condon, Ján Maňuch, and Chris Thachuk. The complexity of string partitioning. *Journal of Discrete Algorithms*, 32:24–43, 2015.
- 5 Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. Pattern matching with variables: Fast algorithms and new hardness results. In *32nd International Symposium on Theoretical Aspects of Computer Science, 2015, March 4-7, 2015, Garching, Germany*, pages 302–315, 2015.
- 6 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. *The Electronic Journal of Combinatorics*, 12, 2005.
- 7 Isaac Goldstein and Moshe Lewenstein. Quick greedy computation for minimum common string partition. *Theoretical Computer Science*, 542:98–107, 2014.
- 8 Niels Grüttemeier, Christian Komusiewicz, Nils Morawietz, and Frank Sommer. String factorizations under various collision constraints. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 9 Markus L. Schmid. Computing equality-free and repetitive string factorisations. *Theoretical Computer Science*, 618:42–51, 2016.
- 10 Moshe Lewenstein. Parameterized pattern matching. In *Encyclopedia of Algorithms*, pages 1525–1530. Springer, 2016.
- 11 Radu Stefan Mincu and Alexandru Popa. The maximum equality-free string factorization problem: Gaps vs. no gaps. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 531–543. Springer, 2020.
- 12 Sebastian Ordyniak and Alexandru Popa. A parameterized study of maximum generalized pattern matching problems. *Algorithmica*, 75(1):1–26, 2016.
- 13 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *Journal of the ACM (JACM)*, 51(3):483–496, 2004.
- 14 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

Improving the Sensitivity of MinHash Through Hash-Value Analysis

Gregory Kucherov   

LIGM, CNRS/Université Gustave Eiffel, Marne-la-Vallée, France

Steven Skiena   

Dept. of Computer Science, Stony Brook University, Stony Brook, NY, USA

Abstract

MinHash sketching is an important algorithm for efficient document retrieval and bioinformatics. We show that the value of the matching MinHash codes convey additional information about the Jaccard similarity of S and T over and above the fact that the MinHash codes agree. This observation holds the potential to increase the sensitivity of minhash-based retrieval systems. We analyze the expected Jaccard similarity of two sets as a function of observing a matching MinHash value a under a reasonable prior distribution on intersection set sizes, and present a practical approach to using MinHash values to improve the sensitivity of traditional Jaccard similarity estimation, based on the Kolmogorov-Smirnov statistical test for sample distributions. Experiments over a wide range of hash function counts and set similarities show a small but consistent improvement over chance at predicting over/under-estimation, yielding an average accuracy of 61% over the range of experiments.

2012 ACM Subject Classification Theory of computation → Bloom filters and hashing

Keywords and phrases MinHash sketching, sequence similarity, hashing

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.20

Funding *Steven Skiena*: Partially supported by a travel grant from Université Paris-Est, and NSF grant IIS-1927227.

1 Introduction

MinHash sketching is an important algorithm for efficient document retrieval. It reduces a set S of size n to a smaller representation of size $m \ll n$ by applying m distinct hash functions h_1, \dots, h_m to each of the n elements of S , and identifies the smallest hash code for each h_i . This vector of minimum hash codes serves a sketch for the larger set S . A classical result [2, 3] shows that the probability that smallest hash codes of two sets S and T are equal is identical to $J(S, T)$, the Jaccard similarity of S and T . Thus the fraction of matching MinHash codes represents an unbiased estimator of $J(S, T)$.

Hash code values, by definition, are not supposed to mean anything. They represent mappings of an item x to a pseudorandom integer $h(x)$ for purpose of fast identity matching and retrieval. The relative values of $h(x)$ and $h(y)$ for items x and y have no special properties beyond that of $h(x) = h(y)$ likely implies that $x = y$ for the conventional hash functions as employed in algorithms such as MinHash.

But in this paper, we report a curious observation associated with MinHash. Suppose the MinHash values for sets $S = \{s_1, \dots, s_n\}$ and $T = \{t_1, \dots, t_n\}$ equal both a particular value, namely:

$$a = \min_{j=1}^n h_i(s_j) = \min_{j=1}^n h_i(t_j)$$

We shall show that the value of this matching MinHash value a conveys additional information about the Jaccard similarity of S and T over and above the fact that the MinHash values agree.



© Gregory Kucherov and Steven Skiena;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 20; pp. 20:1–20:12

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This observation holds the potential to increase the sensitivity of minhash-based retrieval systems. Our main results in this paper are:

- We explain why observing a larger matching MinHash value a increases the expectation of high similarity between S and T . Specifically, the expected value of a common MinHash value a for two n -element sets with intersection size i is $N/(2n - i + 1)$, presuming the underlying hash function selects an integer from $[0 \dots N]$ uniformly at random.
- We analyze the expected Jaccard similarity of two sets as a function of observing a matching MinHash value a under a reasonable prior distribution on intersection set sizes, specifically the case where pairs of n -element sets have equal probability of intersection size i for $1 \leq i \leq n$. Experimental results confirm a modest increase in the sensitivity of our hash-code weighted variant of MinHash over the original, over a range of set similarities and number of hash codes.
- We present a practical approach to using MinHash values to improve the sensitivity of traditional Jaccard similarity estimation, based on the Kolmogorov-Smirnov statistical test for sample distributions. Our techniques provide a supplemental signal suggesting whether the fraction of matching MinHashes is more likely to over-estimate or underestimate the true Jaccard similarity between two sets. Experiments over a wide range of hash counts (k) and set similarities show a small but consistent improvement over chance, specifically an average accuracy of 61% over the range of experiments.

We believe that this orthogonal view of measuring Jaccard similarity through the value of matching MinHash codes is novel, and will inspire further interest. Although the practical improvement we have demonstrated is not large, we believe that better interpretations of the underlying statistics may yield better results.

This paper is organized as follows. We begin with a survey of the literature of MinHash and related techniques in Section 2. We provide intuition as to why the value of the matching MinHash value matters through a thought experiment in Section 3. We present our analysis of the expected intersection size as a function of MinHash value for an appropriate prior distribution in Section 4, and ways to combine this information into an estimate of Jaccard similarity in Section 5. An alternate approach to interpret the values of MinHash codes using the Kolmogorov-Smirnoff statistical test is presented in Section 6. Finally, we conclude with some open problems raised by our work.

2 Prior Work

Broder [2, 3] developed MinHash as a solution to identifying similar documents (represented as sets of shingles or substrings) within a large text corpus while avoiding the quadratic-time costs of explicitly comparing each pair of documents. A function $h(x)$ is applied to each set element, mapping each element x to a pseudo-random integer. Each set S is represented by the minimum hash value among all its elements.

The *Jaccard similarity* $J(S, T)$ of two sets S and T is defined as

$$J(S, T) = \frac{|S \cup T|}{|S \cap T|}.$$

For identical sets, $J(S, T) = 1$ while for disjoint sets, $J(S, T) = 0$. Broder [2, 3] observed that the probability that two sets S and T generate the same MinHash value exactly equals the Jaccard similarity of the two sets, $J(S, T)$. The fraction of matching minimum hashes over k independent hash functions provides an unbiased estimator of $J(S, T)$, with the variance of this estimate equal to $J(S, T)(1 - J(S, T))/k$ [4]. Surveys of MinHash sketching include Cohen [9].

MinHash is one of the most important algorithms for web search and duplicate detection [12, 15, 17], social networks [8], and machine learning [7, 19]. More recently, it has been successfully applied to bioinformatics for sketching large DNA sequence datasets, starting from the seminal Mash software [21] and followed by related tools [5, 28, 1, 20]. Such applications motivate our efforts in this paper to increase the sensitivity of minimum hashing-based similarity measures.

Locality Sensitive Hashing (LSH) techniques seek to map similar data objects to the same hash codes with a higher probability than the dissimilar ones by adopting a family of randomized hash functions. Indyk et al. [16, 14, 10] introduced the notion of locality-sensitive hashing in the context of nearest-neighbor search and string similarity. MinHash can be viewed an instance of locality sensitive hashing. An extended survey of locality-sensitive hashing can be found in [24].

SimHash [6] is a LSH-based method which provides an unbiased estimator of the similarity between two vectors. Specifically, the probability that two vectors u and v generate the same SimHash value equals the Cosine similarity of u and v . Henzinger [15] performed a large-scale comparison of MinHash and SimHash on detecting similar web pages, finding that a hybrid of the two approaches yielded the best results. Srivastava and Li [22] present analysis and experiments to suggests that MinHash is more sensitive than SimHash in regions of high similarity.

Each subset element is granted equal weight in traditional MinHash schemes, but this can be generalized in weighted MinHash, perhaps to reflect the TD-IDF values of each word. Weighted MinHash algorithms are surveyed in [25].

Finally, we mention another related sketching scheme named HyperLogLog [13] primarily designed for the task of estimating the number of distinct items in a stream, but also capable of estimating the cardinality of the union of two sets and therefore their Jaccard similarity. Several works proposed unifying combinations of the two sketches [11, 27]. Note that HyperLogLog has some relationship with our work, as it employs the idea of estimating the cardinality of a random set from its minimum value. However, a direct application of this idea to MinHash has not been made, to our knowledge.

3 Thought Experiment: Why MinHash Values Matter

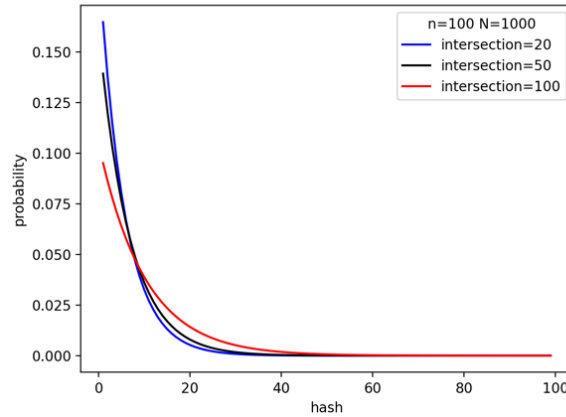
We present the following thought experiment to illustrate how the actual value of matching MinHash codes provides information about Jaccard similarity. For a set S , let $M_h(S)$ denote its MinHash value under a given hash function h , i.e. $M_h(S) = \min_{s \in S} \{h(s)\}$. We use the notation $M(S)$ when h is irrelevant (but assumed fixed across sets).

Now consider following two “extreme” situations involving pairs of sets S and T , each with n elements:

1. S and T are identical. Hence by definition, the minimum hash values must match, so $a_1 = M_h(S) = M_h(T)$,
2. S and T intersect in only one element x , which happens to be the minimum value of both under h , so $a_2 = h(x) = M_h(S) = M_h(T)$.

Now, given just the two unlabeled values for a_1 and a_2 , can we correctly assign these codes to the appropriate case above with probability greater than $1/2$?

Assume hash function h selects an integer from the range $[0 \dots N]$ uniformly at random. Now suppose that two n -element sets with intersection size i share a common MinHash value m . Then m is the smallest of the $2n - i$ values in the union. Since the expected minimum of ℓ numbers drawn uniformly at random from $[0..N]$ is $N/(\ell + 1)$, the expected value of m is



■ **Figure 1** Probability distributions that two sets of size 100 share a common MinHash value, as a function of the size of their intersection (respectively 20, 50, and 100). The probability of small matching MinHash values are increase for relatively dissimilar sets.

$N/(2n - i + 1)$. In the first case above, $i = n$, so $\mathbb{E}[a] = N/(n + 1)$, while for the second case $i = 1$ and $\mathbb{E}[b] = N/(2n)$. Thus it is more likely that $\min(a_1, a_2)$ corresponds to case (1) and $\max(a_1, a_2)$ to case (2).

The situation is illustrated in Figure 1, which shows the probability of observing a given MinHash value a for three different intersection sizes. The probability of observing a MinHash value of 0 with a possible range $[0 \dots 1000]$ is almost twice as high for two 100-element sets with a 20-element intersection than when the sets are identical. More similar pairs of sets, with larger intersection sizes, have greater probability of large matching MinHash values.

4 Expected Intersection Size as a Function of MinHash Value

In this section, we analyze the expected intersection size of two sets based on observing a particular matching MinHash value. For two n -element sets S and T where $|S \cap T| = i$, $J(S, T) = i/(2n - i)$. Thus analyzing the intersection size of S and T provides a result which can be alternately interpreted in the context of the Jaccard similarity of S and T for n -element sets.

Let S and T be two sets each of size n . We limit our attention to the case where S and T are non-disjoint, which is necessary for MinHash values to legitimately collide, so $S \cap T \neq \emptyset$. Further, we assume that range of h from $[0 \dots N]$ is sufficiently large relative to n that we can discount the possibility of spurious collisions, namely that there does not exist $s \in S$ and $t \in T$ where $h(s) = h(t)$ despite $s \neq t$.

4.1 Prior Distribution

Determining the expected intersection size as a function of matching hash values requires knowledge of a prior distribution on the value of the intersection size. In the analysis below, we base our analysis on a uniform prior distribution, that all intersection sizes between S and T are equally likely. Thus for every $i \in [1..n]$, $\mathbb{P}[|S \cap T| = i] = 1/n$.

The uniform distribution appears most natural to us as a general prior, which is why we analyze this case below. That said, the true prior distribution differs with application, particularly as to whether pairs of randomly selected sets are likely to have large or small intersection sizes. The analysis below can be repeated for any particular well specified prior distribution in an analogous fashion.

4.2 Analysis

The probability of two sets sharing the MinHash value equals the Jaccard similarity index, that is

$$\mathbb{P}[M(S) = M(T) \mid i = |S \cap T|] = \frac{i}{2n - i}. \quad (1)$$

Because all intersections are equiprobable under our prior distribution, we have

$$\mathbb{P}[|S \cap T| = i \mid M(S) = M(T)] = \frac{\frac{i}{2n-i}}{\sum_{j=1}^n \frac{j}{2n-j}} \quad (2)$$

Note that given ℓ random numbers x_1, \dots, x_ℓ uniformly drawn from $[1..N]$, for $a \in [1..N]$, we have

$$\mathbb{P}[\min\{x_1, \dots, x_\ell\} \leq a] = 1 - \mathbb{P}[x_1 > a \ \& \ \dots \ \& \ x_\ell > a] = 1 - \left(1 - \frac{a}{N}\right)^\ell. \quad (3)$$

Then, the probability the MinHash is exactly a is given by

$$\mathbb{P}[\min\{x_1, \dots, x_\ell\} = a] = \left(1 - \frac{a-1}{N}\right)^\ell - \left(1 - \frac{a}{N}\right)^\ell. \quad (4)$$

We now estimate the conditional probability $\mathbb{P}[|S \cap T| = i \mid M(S) = M(T) = a]$. We have

$$\begin{aligned} \mathbb{P}[|S \cap T| = i \mid M(S) = M(T) = a] &= \frac{\mathbb{P}[(|S \cap T| = i) \wedge (M(S) = M(T)) \wedge (a = M(S \cup T))]}{\mathbb{P}[(M(S) = M(T)) \wedge (a = M(S \cup T))]} \\ &= \frac{\mathbb{P}[|S \cap T| = i] \cdot \mathbb{P}[a = M(S \cup T) \mid |S \cap T| = i] \cdot \mathbb{P}[M(S) = M(T) \mid (|S \cap T| = i) \wedge (a = M(S \cup T))]}{\sum_{i=1}^n (\mathbb{P}[(M(S) = M(T)) \wedge (a = M(S \cup T)) \mid |S \cap T| = i] \cdot \mathbb{P}[|S \cap T| = i])} \\ &= \frac{\mathbb{P}[(a = M(S \cup T)) \mid (|S \cap T| = i)] \cdot \mathbb{P}[(M(S) = M(T)) \mid (|S \cap T| = i) \wedge (a = M(S \cup T))]}{\sum_{i=1}^n \mathbb{P}[(M(S) = M(T)) \wedge (a = M(S \cup T)) \mid |S \cap T| = i]} \end{aligned} \quad (5)$$

The last rewrite follows because $\mathbb{P}[|S \cap T| = i] = 1/n$ is the same for all i . To further simplify Eqn. 5, observe that

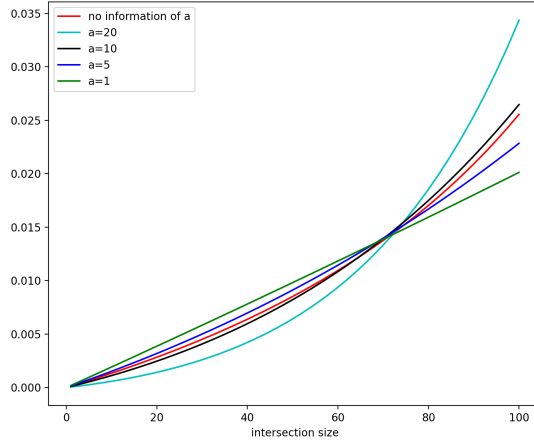
$$\mathbb{P}[M(S) = M(T) \mid (|S \cap T| = i) \wedge (a = M(S \cup T))] = \mathbb{P}[M(S) = M(T) \mid |S \cap T| = i]$$

because the event of sharing common MinHash ($M(S) = M(T)$) is independent of its value (a) for a fixed intersection size. For the same reason, in the denominator,

$$\begin{aligned} \mathbb{P}[(M(S) = M(T)) \wedge (a = M(S \cup T)) \mid |S \cap T| = i] &= \\ &= \mathbb{P}[M(S) = M(T) \mid |S \cap T| = i] \cdot \mathbb{P}[a = M(S \cup T) \mid |S \cap T| = i] \end{aligned}$$

Eqn. 5 then rewrites to

$$\frac{\mathbb{P}[a = M(S \cup T) \mid |S \cap T| = i] \cdot \mathbb{P}[M(S) = M(T) \mid |S \cap T| = i]}{\sum_{i=1}^n (\mathbb{P}[M(S) = M(T) \mid |S \cap T| = i] \cdot \mathbb{P}[a = M(S \cup T) \mid |S \cap T| = i])}. \quad (6)$$



■ **Figure 2** The probability of intersection size of two sets of size 100 sharing a common MinHash value. The red curve shows the probability of having a given intersection size (formula (2)). The other curves show the same probability conditioned on the value a of common MinHash (formula (7)), where the hash space is $[1..1000]$. Larger values of a favor larger intersection sizes.

Using (4), (1), we obtain

$$\mathbb{P}[|S \cap T| = i \mid M(S) = M(T) = a] = \frac{\frac{i}{2^{n-i}} \left(\left(1 - \frac{a-1}{N}\right)^{2n-i} - \left(1 - \frac{a}{N}\right)^{2n-i} \right)}{\sum_{j=1}^n \frac{j}{2^{n-j}} \left(\left(1 - \frac{a-1}{N}\right)^{2n-j} - \left(1 - \frac{a}{N}\right)^{2n-j} \right)} \quad (7)$$

Using (7), we can compute the expected intersection size as a function of the shared MinHash value:

$$\mathbb{E}[|S \cap T| \mid M(S) = M(T) = a] = \sum_{i=1}^n i \cdot \mathbb{P}[|S \cap T| = i \mid M(S) = M(T) = a] \quad (8)$$

As an illustration, Figure 2 shows probability distributions of intersection sizes without taking into account the common MinHash value (formula (2)) and knowing the common MinHash value (formula (7)). The figure demonstrates that larger common MinHash values provide an evidence for larger intersection sizes.

5 Hash Scoring for Sketch Similarity

In the classical MinHash scheme, the probability that two sets have matching MinHash is equal to the Jaccard similarity between the two sets. Thus, the fraction of matches taken over a number of trials provides an unbiased estimator of the Jaccard similarity. We have shown that the values of these matching MinHashes provides an orthogonal measure of similarity. The question is what the best way to combine these measures is.

We propose the following initial strategy. Traditional MinHash can be interpreted as averaging the values of 0/1 indicator variables, where a match of hash codes is represented by 1 and a mismatch by 0. We will replace the value associated with matching hashes by real values that over/underweight based on the value of shared MinHash. More specifically, a shared MinHash value will contribute with weight

$$\frac{\mathbb{E}[|S \cap T| \mid M(S) = M(T) = a]}{\mathbb{E}[|S \cap T|]}, \quad (9)$$

■ **Table 1** Improvement (in terms of the average reduction of absolute error) in estimating set intersection size by summing hash-weighted counts vs. equal weighting to estimate Jaccard similarity for different numbers of hash functions (rows) and set intersection sizes (columns). Bolded entries represent improvement over traditional MinHash estimation, representing $116/140 = 82.9\%$ of the non-trivial cells in the table.

k / int	1	2	3	5	10	15	19	20
1	.0068	.0140	.0205	.0375	.0837	.1150	.1550	-.1650
2	.0067	.0138	.0269	.0395	.0583	.0626	.1020	-.1120
3	.0092	.0250	.0258	.0466	-.0914	.0369	.0824	-.0833
4	.0128	.0250	.0308	.0496	.0042	.0093	.0648	-.0615
5	.0161	.0240	.0312	.0493	.0280	-.0799	.0684	-.0532
6	.0163	.0251	.0377	.0537	-.0471	.0012	.0550	-.0489
7	.0116	.0301	.0368	-.0277	-.0008	.0071	.0636	-.0384
8	.0143	.0257	.0356	.0119	.0282	-.0072	.0615	-.0347
9	.0159	.0201	.0350	.0200	-.0377	.0018	.0298	-.0325
10	.0139	.0283	.0416	.0233	.0029	-.0411	-.0146	-.0290
11	.0155	.0266	.0476	.0266	.0156	.0028	-.0166	-.0244
12	.0147	.0262	.0218	.0350	-.0172	-.0023	-.0004	-.0231
13	.0165	.0353	.0016	.0403	.0133	-.0044	.0035	-.0213
14	.0176	.0283	.0157	-.0133	.0162	-.0035	.0070	-.0193
15	.0168	.0280	.0125	.0077	-.0207	-.0226	.0148	-.0172
16	.0164	.0291	.0142	.0173	.0081	-.0003	.0059	-.0155
17	.0189	.0328	.0196	.0115	.0113	-.0045	.0084	-.0144
18	.0163	.0293	.0193	.0207	-.0157	.0071	.0058	-.0144
19	.0152	-.0060	.0208	.0304	.0043	.0036	.0092	-.0118
20	.0146	.0094	.0213	.0289	.0059	-.0193	-.0122	-.0115

where the numerator is defined by Eqn 8, and

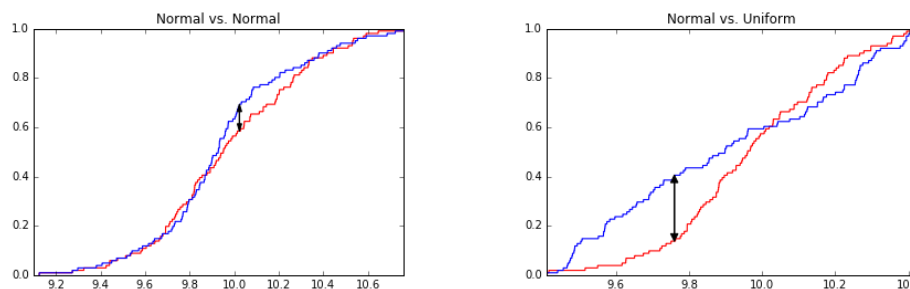
$$\mathbb{E}[|S \cap T|] = \sum_{i=1}^n \mathbb{P}(|S \cap T| = i \mid M(S) = M(T)) \quad (10)$$

is the expected intersection size independent of MinHash value, i.e. implied by the prior distribution of intersection sizes.

5.1 Experimental Results

We performed a modest experiment to evaluate the performance of this technique, with the results reported in Table 1. We limited the experiment to small sets ($n = 20$), but consider a broad range of hash function counts ($1 \leq k \leq 20 = n$) and set similarities defined by intersection sizes from $1 \leq i \leq 20 = n$. Each cell represents the average difference in absolute error in estimating intersection size between MinHash with over/underweighting and traditional 0/1 counts, where each cell is averaged over 1,000 independent random trials.

We note that the rightmost column in Table 1 (intersection size 20 out of a possible 20) corresponds to identical sets, where the traditional Jaccard (and intersection size) estimate is always correct, leaving our proposed method with no room for possible improvement. But $116/140 = 82.9\%$ of the non-trivial cells show improvement over the traditional MinHash baseline.



■ **Figure 3** The Kolmogorov-Smirnov test quantifies the difference between two probability distributions by the maximum y -distance gap between the two cumulative distribution functions. On the left, two samples from the same normal distribution. On the right, comparison of samples from uniform and normal distributions drawn over the same x -range..

6 Sketch Evaluation using the Kolmogorov-Smirnov Test

We now propose an alternate approach to improve the Jaccard similarity estimate offered by the classical MinHash approach, namely the fraction of matching MinHash values in k trials. We seek to improve this estimate by analyzing the distribution of the values of the matching hashes from these trials to decide whether it is more likely to be over or under estimating the actual similarity. A key advantage of this approach over that of Section 4 is that it does not require a prior distribution on the actual intersection sizes.

Our approach is based on the *Kolmogorov-Smirnov* (KS) statistical test [18, 23], which compares empirical cumulative distribution functions (CDFs) to assess whether two samples are drawn from the same underlying distribution. We will use it to compare the observed distribution of matching MinHash values against the theoretical distribution for the classical Jaccard estimate. The direction of the largest deviation suggests whether it is more likely an over or under estimate.

6.1 The Kolmogorov-Smirnov Test

In the KS-test, the empirical cumulative distribution functions (CDFs) of the two different samples are plotted on the same chart. If the two samples are drawn from the same distribution, the ranges of x values should largely overlap. An empirical CDF $\hat{F}(x)$ of a sample is defined as the fraction of the sample $\leq x$.

We seek to identify the value of x for which the associated values of the two CDFs differ by as much as possible. The distance $D(\hat{F}, \hat{G})$ between two empirical CDFs \hat{F} and \hat{G} is the difference of the y values at this critical x , formally stated as

$$D(\hat{F}, \hat{G}) = \max_x |\hat{F}(x) - \hat{G}(x)|$$

The more substantially that two samples differ in this fashion, the more likely it is that they were drawn from different distributions. Figure 3 (left) shows two independent samples from the same normal distribution. In contrast, Figure 3 (right) compares a sample drawn from a normal distribution against one drawn from the uniform distribution. The big gaps near the tails provide evidence that the two samples are drawn from different distributions.

The KS-test compares the value of $D(\hat{F}, \hat{G})$ against a particular target, declaring that two distributions differ at the significance level of α when:

$$D(\hat{F}, \hat{G}) > c(\alpha) \sqrt{\frac{n_1 + n_2}{n_1 n_2}}$$

where $c(\alpha)$ is a distribution-independent constant to look up in a table. In this paper, we use the ideas behind the KS-test for qualitative evaluation instead of precisely measuring statistical significance, and so will be interested in the direction of the deviation without this associated constant.

6.2 Application to MinHash Analysis

As explained above, the distribution of matching MinHash values differs as a function of the intersection size or (equivalently) Jaccard similarity between two sets of size n . Recall that for ℓ random numbers x_1, \dots, x_ℓ uniformly drawn from $[1..N]$, for $a \in [1..N]$, we have

$$F_\ell(x) = \mathbb{P}[\min\{x_1, \dots, x_\ell\} \leq x] = 1 - \mathbb{P}[x_1 > x \ \& \ \dots \ \& \ x_\ell > x] = 1 - \left(1 - \frac{x}{N}\right)^\ell \approx 1 - e^{-x\ell/N}. \quad (11)$$

This defines the CDF on matching MinHash values. Comparing two sets A and B , both of cardinality n with an intersection of size i , any common MinHash value represents the smallest of $\ell = 2n - i$ random values. Thus the distribution of matching MinHash values is defined by Eqn. 11, given an estimate for the union size ℓ . An important observation for us is that CDFs F_ℓ are majorating one another, that is if $\ell_1 > \ell_2$, then $F_{\ell_1}(x) > F_{\ell_2}(x)$ for any x .

Estimates for the union size ℓ and intersection size i follow from classical MinHash analysis. If m matching MinHash values are observed in k trials, m/k is an unbiased estimator of the Jaccard index $\frac{i}{\ell} = \frac{i}{2n-i} = \frac{2n-\ell}{\ell}$. Therefore, i and ℓ are estimated respectively by

$$\hat{i} = \frac{2nm}{k+m}, \quad \hat{\ell} = \frac{2nk}{k+m}.$$

We can now employ the idea underlying the KS-test to evaluate how well the m observed MinHash values match the estimated distribution $F_{\hat{\ell}}(x)$. In doing that, we analyze the sign of the critical deviation

$$D(F_{\hat{\ell}}, \hat{F}) = F_{\hat{\ell}}(\tilde{x}) - \hat{F}(\tilde{x}) \text{ for } \tilde{x} = \operatorname{argmax} |F_{\hat{\ell}}(x) - \hat{F}(x)|,$$

where \hat{F} is the empirical CDF obtained from the sample of matching MinHash values. When D is positive, this suggests that the regular MinHash estimate $\hat{\ell}$ is an overestimate and therefore \hat{i} is an underestimate for the true intersection size. Conversely, a negative D provides an evidence that \hat{i} is an overestimate for the true intersection size. This reasoning is supported by the above-mentioned majorating property of CDFs, as it guarantees that the sign of D correctly defines whether the estimate $\hat{\ell}$ should be increased or decreased for the KS-test statistic to be reduced and therefore for the estimated CDF to fit better the observed MinHash values. We thus propose the sign of D as a secondary signal to improve the accuracy of \hat{i} as an estimator for intersection size.

The running time of this test is $O(m \log m)$ because we must sort the observed matching hash values to compute the CDF. It is only necessary to compare the distributions at the m sample points to identify the extremal points, with each comparison efficiently done using the exponential form of Eqn. 11. The magnitude of the deviation directly maps to a confidence value in the direction of change, with p -values obtainable using tables of $c(\alpha)$ values from the standard KS-test. However, in the experiments below we propose to estimate correction direction from the sign of D independent of its magnitude.

■ **Table 2** Performance (in terms of the fraction of correct direction predictions) of the KS-test-based over/under correction, as a function of the number of hash functions k (shown in left column), and the true Jaccard similarity/intersection size (shown in first/second row). Generally speaking, the improvement is greatest at extreme values of similarity (either high or low), and with smaller numbers of hash functions.

Jaccard intersect	.961 980	.869 930	.786 880	.739 850	.667 800	.538 700	.429 600	.333 500	.258 410	.176 300	.111 200	.081 150	Avg
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	.816	.696	.596	.534	.457	.339	.988	.994	.996	.998	.999	.998	.784
3	.764	.618	.485	.419	.275	.552	.445	.340	.983	.992	.994	.998	.655
4	.711	.556	.440	.651	.561	.388	.570	.475	.361	.982	.989	.993	.640
5	.683	.519	.632	.572	.478	.551	.394	.551	.450	.974	.985	.993	.649
10	.591	.577	.590	.537	.566	.505	.474	.452	.483	.561	.399	.966	.558
20	.502	.540	.565	.573	.545	.564	.535	.528	.456	.460	.395	.514	.515
50	.506	.550	.539	.567	.550	.560	.549	.544	.541	.520	.474	.396	.525
75	.518	.535	.562	.552	.508	.553	.550	.492	.515	.506	.487	.431	.517
100	.512	.552	.556	.561	.553	.578	.552	.552	.542	.525	.468	.454	.534
200	.525	.549	.550	.557	.563	.568	.555	.557	.543	.527	.505	.491	.541
300	.522	.551	.558	.556	.535	.561	.550	.530	.543	.536	.515	.506	.539
500	.529	.550	.550	.559	.556	.574	.558	.553	.543	.534	.521	.511	.545
1000	.520	.552	.556	.567	.561	.565	.557	.548	.550	.540	.512	.512	.545
Average	.621	.596	.584	.586	.550	.561	.591	.580	.608	.690	.660	.697	.610

6.3 Experimental Results

Table 2 summarizes the performance of our KS-based correction strategy over a wide range of hash counts (from $k = 1$ to $k = 1000$) and true Jaccard similarity (from 0.081 to 0.961). For each Jaccard similarity level, we constructed 10,000 pairs of 1000-element sets, each pair constructed to the appropriate level of similarity. We then constructed k independent hash functions of these sets, and determined the number of matching MinHashes for these trials. We then performed the KS-test on the matching values to propose whether the actual Jaccard estimate should be higher or lower than the observed fraction of matches. We chose parameters of our tests (intersection size) so that to avoid the situation when the MinHash estimate exactly equals the true Jaccard similarity, making each case a fair binary trial.

Of the $14 \times 12 = 172$ entries in Table 2, 144 of them (83.7%) are greater than 0.5, meaning the adjustment breaks in the correct direction more often than not. The average accuracy ratio taken over all trials is 61.0%, substantially better than the baseline of 50%.

When employing large numbers of hash functions $k \geq 100$, our technique improves the estimate on average in 57 of 60 (95%) entries, and proves most beneficial in the middle regions where the Jaccard similarity is ≈ 0.5 . This is curious, because larger k provides greater resolution on the fraction of matching hash values, thus reducing the quantization error of classical MinHash. But the KS-analysis also improves with more samples as k increases, and continues to refine the similarity estimate even as $k = 1000$. Presumably in the limit as k grows, the improvement over baseline will disappear, but it seems durable over the range of k that appear in general applications.

That our best (and worst) performance occurs for very small k reflects issues of quantization: for an intersection size of $n/2$ and $k = 5$, the best possible estimate must be wrong by at least 10%. As a statistical significance test, the KS-test was designed to be used with a meaningful number of samples per observed distribution. There are likely other statistical tests to do better with small (and maybe even large) values of k .

7 Conclusions

We have demonstrated that the value of matching MinHash values provides additional information on the degree of similarity between pairs of sets. Our wins are small, but they are real. We believe that there exist better methods of integration to synthesize the mix of the number of matching hashes and their values into a more accurate measure of similarity and believe that this is a research direction worth pursuing. We note that even careful analysis of the values of the matching hash codes will be substantially less computationally expensive than that of obtaining the MinHash codes themselves, so these improvements will come at a little computational cost.

The MinHash values that *do not* match also contain some degree of signal concerning the similarity of two sets. Suppose the smallest hash values of two sets do not match, but are both unusually large, say a substantial fraction of the total range N . These large MinHash values signify that both sets exclude the same large fraction of possible elements from the universe, implying they must both be constructed from just a relatively small set of non-excluded elements. This conditioning increases the expected Jaccard similarity, despite the fact that the hash values do not match. We believe this signal to be very weak except in extreme cases, but its analysis may be part of a complete solution.

The theoretical success of MinHash depends strongly upon the elements of the sets being distinct. If multiplicity of elements should be taken into account, one should resort to the weighted variant of MinHash [26]. Extending our ideas to Weighted MinHash is another interesting direction of study for the future.

References

- 1 Daniel N. Baker and Ben Langmead. Dashing: fast and accurate genomic distances with HyperLogLog. *Genome Biology*, 20:265, 2019. doi:10.1186/s13059-019-1875-0.
- 2 Andrei Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society. URL: <http://dl.acm.org/citation.cfm?id=829502.830043>.
- 3 Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000, Montreal, Canada, June 21-23, 2000, Proceedings*, pages 1–10, 2000. doi:10.1007/3-540-45123-4_1.
- 4 Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, 1998.
- 5 C. Titus Brown and Luiz Irber. Sourmash: a library for minhash sketching of dna. *Journal of Open Source Software*, 1(5):27, 2016. doi:10.21105/joss.00027.
- 6 Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.
- 7 Lianhua Chi, Bin Li, and Xingquan Zhu. Context-preserving hashing for fast text classification. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 100–108. SIAM, 2014.
- 8 Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.
- 9 Edith Cohen. Min-hash sketches: A brief survey, 2016. URL: <http://www.cohenwang.com/edith/Surveys/minhash.pdf>.

- 10 Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.
- 11 Otmar Ertl. Setsketch: Filling the gap between minhash and hyperloglog. *Proc. VLDB Endow.*, 14(11):2244–2257, 2021. doi:10.14778/3476249.3476276.
- 12 Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the 12th international conference on World Wide Web*, pages 669–678, 2003.
- 13 Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), January 2007. doi:10.46298/dmtcs.3545.
- 14 Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.
- 15 Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291, 2006.
- 16 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.
- 17 Nitin Jindal and Bing Liu. Opinion spam and analysis. In *Proceedings of the 2008 international conference on web search and data mining*, pages 219–230, 2008.
- 18 Andrey Kolmogorov. Sulla determinazione empirica di una legge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 4:83–91, 1933.
- 19 Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd König. Hashing algorithms for large-scale learning. *Advances in neural information processing systems*, 24, 2011.
- 20 Brian D. Ondov, Gabriel Starrett, Anna Sappington, Aleksandra Kostic, Sergey Koren, Christopher B. Buck, and Adam M. Phillippy. Mash screen: high-throughput sequence containment estimation for genome discovery. *Genome Biology*, 20:232, 2019. doi:10.1186/s13059-019-1841-x.
- 21 Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1):1–14, 2016.
- 22 Anshumali Shrivastava and Ping Li. In defense of MinHash over SimHash. In *Artificial Intelligence and Statistics*, pages 886–894. PMLR, 2014.
- 23 Nickolay Smirnov. Table for estimating the goodness of fit of empirical distributions. *The annals of mathematical statistics*, 19(2):279–281, 1948.
- 24 Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014. arXiv:1408.2927.
- 25 Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. A review for weighted MinHash algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 34(6):2553–2573, 2020.
- 26 Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. A review for weighted minhash algorithms. *IEEE Trans. Knowl. Data Eng.*, 34(6):2553–2573, 2022. doi:10.1109/TKDE.2020.3021067.
- 27 Yun William Yu and Griffin M. Weber. Hyperminhash: Minhash in loglog space. *IEEE Trans. Knowl. Data Eng.*, 34(1):328–339, 2022. doi:10.1109/TKDE.2020.2981311.
- 28 XiaoFei Zhao. Bindash, software for fast genome distance estimation on a typical personal laptop. *Bioinformatics*, 35(4):671–673, 2019.

Suffix-Prefix Queries on a Dictionary



Grigorios Loukides  

Department of Informatics, King's College London, UK



Solon P. Pissis  

CWI, Amsterdam, The Netherlands

Vrije Universiteit, Amsterdam, The Netherlands

Sharma V. Thankachan  

North Carolina State University, Raleigh, NC, USA

Wiktor Zuba  

CWI, Amsterdam, The Netherlands

Abstract

In the *all-pairs suffix-prefix* (APSP) problem, we are given a dictionary R of k strings, S_1, \dots, S_k , of total length n , and we are asked to find the length $\text{SPL}_{i,j}$ of the *longest* string that is both a suffix of S_i and a prefix of S_j , for all $i, j \in [1, k]$. APSP is a classic problem in string algorithms with many applications in bioinformatics. When all strings of the dictionary are over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, APSP can be solved in the optimal $\mathcal{O}(n + k^2)$ time with the use of the generalized suffix tree of the dictionary [Gusfield et al., *Inf. Process. Lett.* 1992].

In many bioinformatics applications, such as in sequence assembly, the size k of dictionary R is very large. In particular, k^2 usually dominates n , and thus the k^2 factor is the bottleneck both in the time and in the space complexity of such applications. We thus initiate a holistic study on several data structure variants of APSP. In particular, we consider the following types of queries:

- **One-to-One**(i, j): output $\text{SPL}_{i,j}$.
- **One-to-All**(i): output $\text{SPL}_{i,j}$ for every $j \in [1, k]$.
- **Report**(i, ℓ): output all distinct $j \in [1, k]$ such that $\text{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- **Count**(i, ℓ): output the number of distinct $j \in [1, k]$ such that $\text{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- **Top**(i, K): output K distinct $j \in [1, k]$ with the highest values of $\text{SPL}_{i,j}$ breaking ties arbitrarily.

We assume the standard word RAM model of computation with word size $w = \Omega(\log n)$ and an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. We show the following upper bounds:

Query	Space (words)	Query time	Note
One-to-One(i, j)	$\mathcal{O}(n)$	$\mathcal{O}(\log \log k)$	Theorem 11
One-to-All(i)	$\mathcal{O}(n)$	$\mathcal{O}(k)$	Theorem 14
Report(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n / \log \log n + \text{output})$	Theorem 19(i)
Count(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n / \log \log n)$	Theorem 19(ii)
Top(i, K)	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n / \log \log n + K)$	Theorem 22

We also present efficient algorithms for constructing these data structures.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases all-pairs suffix-prefix, suffix-prefix queries, internal pattern matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.21

Funding This work is supported in part by the Royal Society grant IES\R3\193209.

Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

Sharma V. Thankachan: Supported by the U.S. National Science Foundation (NSF) grant CCF-2146003.

Wiktor Zuba: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.



© Grigorios Loukides, Solon P. Pissis, Sharma V. Thankachan, and Wiktor Zuba; licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 21; pp. 21:1–21:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *all-pairs suffix-prefix* problem (APSP, in short) is a classic problem in string algorithms. APSP finds numerous applications in bioinformatics because it is the first step in sequence assembly [26, 37, 46, 8, 11]. Given a dictionary R of k strings, S_1, \dots, S_k , of total length n , the APSP problem asks us to find, for each string S_i , $i \in [1, k]$, its longest suffix that is a prefix of string S_j , for all $j \neq i$, $j \in [1, k]$. Gusfield et al. [27] presented an algorithm running in the optimal $\mathcal{O}(n + k^2)$ time for solving APSP, assuming all strings in R are over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. The algorithm is based on the generalized suffix tree [53] of R . Ohlebusch and Gog [39] gave another optimal algorithm which is based on the generalized suffix array [36] of R . Tustumi et al. [49] gave yet another optimal algorithm based on the generalized suffix array of R . Thus the common denominator of all existing optimal algorithms for APSP is that they rely on sorting the suffixes of all strings in R , and therefore they require space $\Omega(n)$ in any case and for any alphabet. In a very recent work, Loukides and Pissis [34] presented a different optimal algorithm, which is based on the Aho-Corasick automaton of R [1], and it thus requires space linear in the size of the automaton.

Due to the practical relevance of APSP, there also exists a large body of works devoted to implementing algorithms for APSP that are suboptimal but practically fast on real-world datasets; see [25, 42, 33] and references therein for some of the state-of-the-art implementations. For a parallel implementation of the algorithm by Tustumi et al. see [35]. For approximate variants of APSP, under the Hamming or edit distance, see [44, 52, 32, 5, 47].

In many bioinformatics applications, such as in sequence assembly, the size k of dictionary R is very large. In particular, k^2 usually dominates n , and thus the k^2 factor is the bottleneck both in the time and the space complexity of such applications. For instance, in typical benchmark datasets¹ for genome assembly using short DNA reads (fragments), k is in the order of 10^6 to 10^8 and n is in the order of 10^8 to 10^{10} . Hence k^2 dominates n significantly.

We thus initiate a holistic study on several data structure variants of APSP. Let $\text{SPL}_{i,j}$ (short for suffix-prefix length), for any $i, j \in [1, k]$, denote the length of the *longest* string that is both a suffix of S_i and a prefix of S_j . We consider the following types of queries:

- **One-to-One**(i, j): output $\text{SPL}_{i,j}$.
- **One-to-All**(i): output $\text{SPL}_{i,j}$ for every $j \in [1, k]$.
- **Report**(i, ℓ): output all distinct $j \in [1, k]$ such that $\text{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- **Count**(i, ℓ): output the number of distinct $j \in [1, k]$ such that $\text{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- **Top**(i, K): output K distinct $j \in [1, k]$ with the highest values of $\text{SPL}_{i,j}$ breaking ties arbitrarily.

By being able to answer different types of such queries efficiently, one may be able to design alternative algorithms, depending on the application in scope, which avoid the k^2 factor in their time or space complexity. Indeed, we stress that most works studying APSP from a practical perspective (e.g., [25, 42, 33]), in fact considered the ℓ -APSP problem in their experimental part; namely, the problem in which we are asked to output only the $\text{SPL}_{i,j}$ values with $\text{SPL}_{i,j} \geq \ell$, for some integer $\ell \geq 0$, which, however, is given *a priori* and is *fixed for all pairs* S_i, S_j . This inflexibility would be surpassed should one have *space-efficient* (e.g., linear-space) data structures for answering these different types of queries *fast*.

¹ For example, see <http://gage.cbcb.umd.edu/data/index.html>.

Our Results. We assume the standard word RAM model of computation with word size $w = \Omega(\log n)$ and an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$. We show the following upper bounds:

Query	Space (words)	Query time	Note
One-to-One(i, j)	$\mathcal{O}(n)$	$\mathcal{O}(\log \log k)$	Theorem 11
One-to-All(i)	$\mathcal{O}(n)$	$\mathcal{O}(k)$	Theorem 14
Report(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n / \log \log n + \text{output})$	Theorem 19(i)
Count(i, ℓ)	$\mathcal{O}(n)$	$\mathcal{O}(\log n / \log \log n)$	Theorem 19(ii)
Top(i, K)	$\mathcal{O}(n)$	$\mathcal{O}(\log^2 n / \log \log n + K)$	Theorem 22

We also provide efficient construction algorithms for Theorems 11 and 14: Theorem 11 can be implemented in $\mathcal{O}(n \log \log k)$ time and Theorem 14 can be implemented in $\mathcal{O}(n)$ time. For Theorems 19 and 22, no guaranteed construction time is provided: the query times for Report, Count, and Top rely on the construction of a 2D rectangle stabbing data structure for reporting [45] and counting [28], but unfortunately the construction times for these data structures are not mentioned in [45] or [28]. However, by constructing the classic data structure for 2D rectangle stabbing [15], we obtain $\mathcal{O}(n \log n)$ construction time, $\mathcal{O}(n)$ words of space, $\mathcal{O}(\log n + \text{output})$ query time for Report, $\mathcal{O}(\log n)$ query time for Count, and $\mathcal{O}(\log^2 n + K)$ query time for Top. We also make the following straightforward observation.

► **Observation 1.** *The symmetric versions of One-to-All, Report, Count and Top, where we are given string S_j as the query and we are asked to output information about $\text{SPL}_{i,j}$, for all $i \in [1, k]$, can be addressed by constructing the corresponding data structures for the dictionary R^r of k strings S_1^r, \dots, S_k^r , where $S^r = S[|S|] \cdots S[2]S[1]$ denotes the reverse of string $S = S[1]S[2] \cdots S[|S|]$. Hence, the same space/query-time trade-offs can be achieved.*

Related Work. In addition to the data structure variants of APSP that are studied here, two other versions of APSP have been studied in the literature. The first version consists in enumerating all pairwise suffix-prefix matches (not necessarily the longest ones) in decreasing order of their lengths. This version of the problem was solved by Ukkonen [50], who used this solution as the crux of his classic linear-time implementation of the greedy algorithm for constructing approximate shortest common superstrings. The second APSP version studied consists in enumerating the *set* of longest suffix-prefix matches (not however their association with the corresponding pairs of strings) [12]. Since any suffix-prefix match in this set is a prefix of some input string, the size of this set is $\mathcal{O}(n)$. This version of the problem was solved in the optimal $\mathcal{O}(n)$ time, independently, by Park et al. [40] and by Khan [29].

Although our work is inspired by real-world applications, the underlying data structure problems are also appealing from a theoretical perspective: (i) they are analogous to *distance oracles* for networks [48, 41, 17, 16, 13]; and (ii) they are special types of *internal pattern matching* (IPM) data structures [31, 30, 3, 14, 4]. For instance, an existing, more general, IPM data structure [30, 31] can be employed to answer One-to-One queries in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n)$ words of space; see Section 2.3 for more details. By designing a specialized data structure for One-to-One, we obtain $\mathcal{O}(\log \log k)$ query time using $\mathcal{O}(n)$ words of space.

Paper Organization. In Section 2, we provide basic definitions and notation on strings. We also describe basic data structures for representing a dictionary, some more advanced data structures that are necessary to obtain our upper bounds, and a few previous solutions to APSP (variants). In Section 3, we provide the solution to One-to-One queries. In Section 4, we provide the solution to One-to-All queries. In Section 5, we provide the solutions to Report and Count queries. Finally, in Section 6, we provide the solution to Top queries.

2 Preliminaries

An *alphabet* Σ is a finite nonempty set of $\sigma = |\Sigma|$ elements called *letters*. By Σ^* we denote the set of all strings over Σ including the *empty string* ε of length 0. A *string* S over Σ is a sequence of letters of Σ . For a string $S = S[1] \cdots S[n]$ over Σ , by $n = |S|$ we denote its length. The fragment $S[i..j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We also say that P *occurs* at (*starting*) *position* i in S . A *prefix* of S is a fragment of S of the form $S[1..j]$ and a *suffix* of S is a fragment of S of the form $S[i..n]$.

Let M be a finite nonempty set of strings over Σ of total length m . We call M a *dictionary*. We define the *trie* of M , denoted by $\text{TR}(M)$, as a deterministic finite automaton that recognizes M . 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 is M ; and transitions (edges) are of the form $\delta(u, \alpha) = u\alpha$, where u and $u\alpha$ are nodes and $\alpha \in \Sigma$. The size of $\text{TR}(M)$ is thus $\mathcal{O}(m)$. The *compacted trie* of M , denoted by $\text{CT}(M)$, contains the root, the branching nodes, and the terminal nodes of $\text{TR}(M)$. The term *compacted* refers to the fact that $\text{CT}(M)$ reduces the number of nodes by replacing each maximal branchless path segment with a single edge, and that it uses a fragment of a string from M to represent the label of this edge in $\mathcal{O}(1)$ words of space. The nodes of $\text{TR}(M)$ that are included in $\text{CT}(M)$ are called *explicit*; all other nodes are called *implicit*. The size of $\text{CT}(M)$ is thus $\mathcal{O}(|M|)$. The most well-known form of compacted trie is the suffix tree described next.

2.1 Suffix Tree and Aho-Corasick Automaton

We are given a dictionary R of k strings, S_1, S_2, \dots, S_k , whose total length is $n = |S_1| + |S_2| + \dots + |S_k|$. Every string in R is over an integer alphabet Σ whose size σ is polynomial in n , i.e., $\Sigma = \{1, 2, \dots, n^{\mathcal{O}(1)}\}$ and thus $\sigma \leq n^{\mathcal{O}(1)}$. For constructing specialized data structures and answering internal pattern matching queries, non-trivial representations of R (different than a simple set of strings) are usually more efficient.

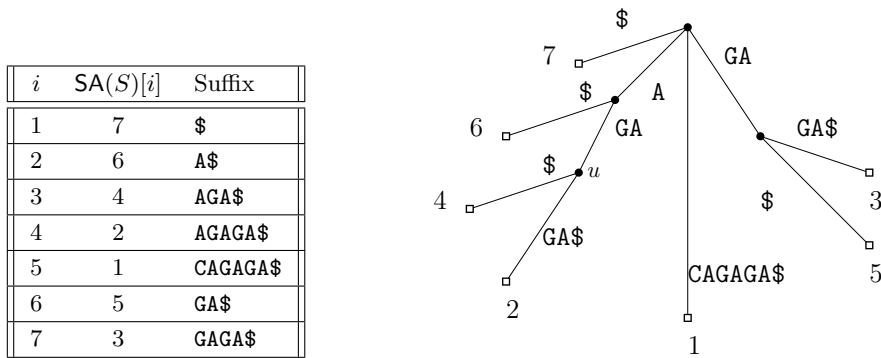
Let us set $T_R := S_1\$1S_2\$2 \cdots S_k\$k$, where $\$1 < \$2 < \dots < \$k$ are letters that are strictly lexicographically smaller than any letter from Σ (and as such they do not belong to Σ).

Let $\text{ST}(S)$ denote the *suffix tree* of string S , that is the compacted trie of all the suffixes of S . For any node v of $\text{ST}(S)$, by $\text{str}(v)$ we denote the concatenation of the edge labels on the path from the root to v , and by $d(v) = |\text{str}(v)|$ we denote the *string depth* of v . The *suffix array* $\text{SA}(S)$ of S is the lexicographically sorted array of the set of suffixes of S , represented by their starting positions; see Figure 1 for an example.

► **Lemma 2** ([53, 22]). *For any string S of length m over an integer alphabet of size $\sigma \leq m^{\mathcal{O}(1)}$, the suffix tree and the suffix array of S can be constructed in $\mathcal{O}(m)$ time.*

We also denote $\text{ST}_i = \text{ST}(S_i\$i)$ and $\text{ST}_R = \text{ST}(S_1\$1, \dots, S_k\$k)$; that is ST_R is the generalized suffix tree [51] of the k strings from R . The generalized suffix tree can be built in linear time; here, however, this more complicated construction is not needed since this compacted trie is equivalent to $\text{ST}(T_R)$ as the letters $\$i$ occur uniquely in this string (and hence a compacted edge containing any label $\$i$ must end at a leaf node).

Another useful representation of R is given by its Aho-Corasick (AC) automaton [1]; the set of states of the AC automaton of R , denoted by $\text{AC}(R)$, corresponds to the set of the prefixes of the strings in R . Let $\text{node}(S)$ denote the node corresponding to string S . After reading an input string the automaton must be in a state corresponding to a suffix of this string (the longest one that is also a prefix of some string in R and has a corresponding state); such a state always exists as ε is always represented (recall ε is the string of length 0). As such, the automaton $\text{AC}(R)$ is often represented by the trie $\text{TR}(R)$ with transitions $\delta(\text{node}(S), \alpha) =$



■ **Figure 1** Suffix array $SA(S)$ and suffix tree $ST(S)$ of string $S = CAGAGA\$$, where $\$$ is a terminal letter, which is the lexicographically smallest letter occurring in S . For node u in $ST(S)$, $str(u) = AGA$ and $d(u) = 3$.

$\{\text{node}(S\alpha)\}$ if $S\alpha$ is a prefix of a string in R , and $\delta(\text{node}(S), \varepsilon) = \{\text{node}(S')\}$, where S' is the longest suffix of S which is also a prefix of a string in R . The ε -transitions are called *failure transitions*. The existence of ε -transitions makes the automaton nondeterministic, and even though this nondeterminism can be avoided, we are going to actually employ those ε -transitions to construct the data structure for **One-to-All** queries.

► **Lemma 3** ([1, 20]). *For any dictionary R of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, $AC(R)$ can be constructed in $\mathcal{O}(n)$ time.*

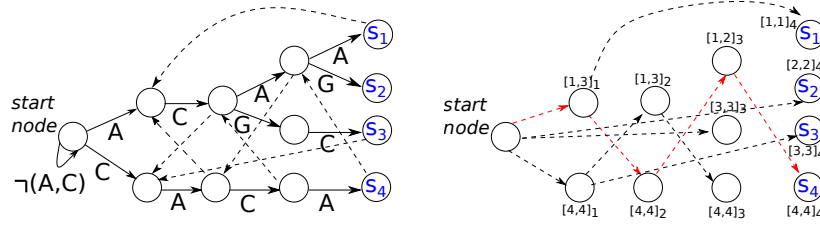
By $FT(R)$ we denote the so-called *Failure Transition tree* (FTtree) of R , introduced by Loukides and Pissis in [34] for solving the APSP problem: the FTtree nodes correspond to the states of the AC automaton (that is, to prefixes of strings in R), and the edges correspond to its ε -transitions with *reversed* direction. Notice that, since every state of $AC(R)$ has *exactly one* outgoing failure transition, $FT(R)$ is indeed a tree rooted at $\text{node}(\varepsilon)$. We additionally decorate every node u of $FT(R)$ by a labeled interval $I_u = [i, j]_d$: S_i, S_{i+1}, \dots, S_j have as a common prefix the string of length d represented by node u ; see [34]. We will generally assume that R is given lexicographically sorted at construction time; otherwise, the sorted version of R can be produced in linear time using, for example, Lemma 3 or Lemma 2.

► **Example 4.** Let $R = \{S_1, S_2, S_3, S_4\} = \{ACAA, ACAG, ACGC, CACA\}$ be a dictionary of $k = 4$ strings. The AC automaton and the FTtree of R is shown in Figure 2. Consider the path from the root to leaf node S_4 (shown in red) in the FTtree of R , where the non-root nodes have the following labeled intervals $[i, j]_d$: $[1, 3]_1$, $[4, 4]_2$, $[1, 2]_3$, $[4, 4]_4$. By recording the largest string depth d of an interval containing j , for every $j \in [1, k]$, along this path, we compute all $SPL_{4,j}$: $SPL_{4,1} = 3$, $SPL_{4,2} = 3$, $SPL_{4,3} = 1$, and $SPL_{4,4} = 4$. Loukides and Pissis [34] showed how to compute this information, for all i , in $\mathcal{O}(n + k^2)$ total time, thus solving the APSP problem optimally using only the FTtree of R .

2.2 Advanced Data Structures

Let T be a rooted tree. A *lowest common ancestor* (LCA) query on T for two given nodes u and v , denoted by $w = LCA_T(u, v)$, returns the last (i.e., the lowest) common node w on their paths from the root.

► **Lemma 5** ([9]). *For any rooted tree T with m nodes, after $\mathcal{O}(m)$ -time preprocessing, we can answer LCA_T queries in $\mathcal{O}(1)$ time per query.*



■ **Figure 2** The AC automaton $AC(R)$ (on the left) and FTtree $FT(R)$ (on the right) of the dictionary of strings $R = \{S_1, S_2, S_3, S_4\} = \{ACAA, ACAG, ACGC, CACA\}$. In $AC(R)$, solid arrows correspond to transitions and dashed arrows to failure transitions. To avoid cluttering the figure, failure transitions to the start node in $AC(R)$ have been omitted.

A *rank and select* data structure (also known as *succinct indexable dictionary* [43]) is a classic data structure, constructed over an array A of length m over alphabet $[1, \sigma]$, which supports two types of queries:

- $\text{rank}_A(i, x) = |\{\ell \in [1, x] : A[\ell] = i\}|$, for $i \in [1, \sigma]$ and $x \in [1, m]$;
- $\text{select}_A(i, x) = \min\{\ell \in [1, m] : \text{rank}_A(i, \ell) = x\}$, for $i \in [1, \sigma]$ and $x \in [1, m]$.

In other words, $\text{rank}_A(i, x)$ returns the number of elements with value equal to i occurring at positions in $[1, x]$ of S , while $\text{select}_A(i, x)$ returns the position of the x th element of A with value equal to i .

► **Lemma 6** ([7, 38, 18]). *For any array $A = A[1..m]$ over $[1, \sigma]$, $\sigma \leq m$, after $\mathcal{O}(m \log \log \sigma)$ -time preprocessing, we can construct a data structure of $\mathcal{O}(m)$ words of space that supports $\mathcal{O}(\log \log \sigma)$ -time rank and select queries on A .*

Let T be a rooted tree of m nodes with integer weights on nodes. Further assume that the weight of every node of T satisfies the *min-heap property*: the weight of each node is greater than or equal to the value of its parent (the smallest weight is hence at the root). A *weighted ancestor* (WA) query for a given node u of T and an integer d , denoted by $w = \text{WA}_T(u, d)$, returns its deepest ancestor w whose weight is at most d [23]. This problem is the generalization of the classic *predecessor search* problem on rooted trees. In the special case when T is a suffix tree and the nodes are weighted by *string depth*, the problem admits an optimal solution due to the recent result of Belazzougui et al. [6] (see also [24]).

► **Lemma 7** ([6]). *For any suffix tree T with m nodes weighted by string depth, after $\mathcal{O}(m)$ -time preprocessing, we can answer WA_T queries in $\mathcal{O}(1)$ time per query.*

In this special case, the ancestor at string depth exactly d may be an implicit node of T , in which case the query outputs its closest explicit ancestor.

2.3 Previous Solutions

$\mathcal{O}(n + k^2)$ -time Algorithm for APSP. We describe the optimal solution to APSP given by Gusfield et al. in [27]. We set $T_R := S_1\$1S_2\$2 \cdots S_k\$k$, where $\$1 < \$2 < \cdots < \$k$ are letters that are strictly lexicographically smaller than any letter from Σ . We start by constructing the suffix tree $\text{ST}_R = \text{ST}(T_R)$. Using a DFS traversal on ST_R , we construct lists $L(v)$ for all nodes v of ST_R : $L(v)$ stores all i such that the suffix of length $d(v)$ of string S_i is $\text{str}(v)$. Consider a string S_j from R and focus on the path P_j from the root of ST_R to the leaf node representing the longest suffix of S_j , i.e., the entire string S_j . Let v be a node on P_j . A suffix of string S_i of length $d(v)$ is a prefix of string S_j of the same length if and only if i is

in $L(v)$. However, for each index i , we want to record the *deepest* node v on P_j such that i is in $L(v)$. It then follows that $d(v) = \text{SPL}_{i,j}$. In order to achieve a linear-time complexity, we perform another DFS maintaining k stacks (one for each S_i). Upon visiting v , we push it on stack i for every $i \in L(v)$. When the leaf node representing the entire string S_j is reached, we scan the k stacks and record, for each index i , the current top of the i th stack. When v is reached in a backward edge traversal, we pop the top of any stack whose index is in $L(v)$. We obtain the following result.

► **Lemma 8** ([27]). *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, APSP can be solved in the optimal $\mathcal{O}(n + k^2)$ time.*

In what follows, we assume that $k \geq \sqrt{n}$; otherwise, when $k < \sqrt{n}$, Lemma 8 implies an optimal solution to our data structure problems (linear preprocessing time, linear size and time-optimal queries), which precomputes and stores all answers.

Internal Prefix-Suffix Queries for One-to-One. Kociumaka considered the following data structure problem in [30]: Given two fragments x and y of a string T and a positive integer d , report all suffixes of y of length between d and $2d - 1$ that also occur as prefixes of x (represented as an arithmetic progression of their lengths). This is the *Internal Prefix-Suffix Queries* problem. Kociumaka showed the following result (see also [31]).

► **Lemma 9** (Theorem 1.1.3 in [30]). *For any string T of length m over an integer alphabet of size $\sigma \leq m^{\mathcal{O}(1)}$, after $\mathcal{O}(m)$ -time preprocessing, we can answer Internal Prefix-Suffix Queries in $\mathcal{O}(1)$ time per query.*

By employing Lemma 9 on T_R , after an $\mathcal{O}(n)$ -time preprocessing, we can answer One-to-One queries in $\mathcal{O}(\log(\min(|S_i|, |S_j|))) = \mathcal{O}(\log n)$ time. In particular, we query for $x = S_j$, $y = S_i$, and $d = 2^\ell$, for all integers $0 \leq \ell \leq \log \min(|S_i|, |S_j|)$, to compute a representation of all the suffixes of S_i that are also prefixes of S_j and then return the length of the longest one as $\text{SPL}_{i,j}$. We obtain the following result, which we improve in Section 3.

► **Corollary 10.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering One-to-One queries in $\mathcal{O}(\log n)$ time.*

3 Answering One-to-One Queries

Main Idea. Say we want to find the longest suffix of S_i that is a prefix of S_j . We first find the maximal longest common prefix between S_j and any suffix of S_i . Say this suffix is $S_i[q..|S_i|]$ and we have that $S_i[q..q+r-1] = S_j[1..r]$ is this longest common prefix. If this prefix is the whole $S_i[q..|S_i|]$, i.e., $|S_i| = q+r-1$, then r is clearly the answer. If this longest common prefix is not a suffix of S_i , i.e., $|S_i| > q+r-1$, then the answer is the longest prefix of $S_i[q..q+r-1]$, that is also a suffix of S_i .

Recall that $\text{ST}_i = \text{ST}(S_i\$_i)$ and $\text{ST}_R = \text{ST}(T_R)$. Consider the path in ST_R obtained by reading $S_j\$_j$ from its root (this path ends in a leaf node). When spelling any suffix of S_i that is also a prefix of S_j in ST_R we use exactly the same path and end by going out of it when reading $\$_i$. This means, that $\text{SPL}_{i,j}$ is represented by the lowest node on this path that has an outgoing edge with label $\$_i$.

In the following we focus on enhancing ST_R and ST_i , for all $i \in [1, k]$, to obtain a data structure that allows finding the string depth of such a node (equal to $\text{SPL}_{i,j}$) efficiently. We will prove the following result.

► **Theorem 11.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering **One-to-One** queries in $\mathcal{O}(\log \log k)$ time. The data structure can be constructed in $\mathcal{O}(n \log \log k)$ time.*

Let us start with a straightforward auxiliary lemma.

► **Lemma 12.** *For any dictionary of k strings S_1, \dots, S_k of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, in $\mathcal{O}(n)$ time we can construct a data structure of $\mathcal{O}(k)$ words of space that answers queries of the type “Is S_j a suffix of S_i ?” in $\mathcal{O}(1)$ time.*

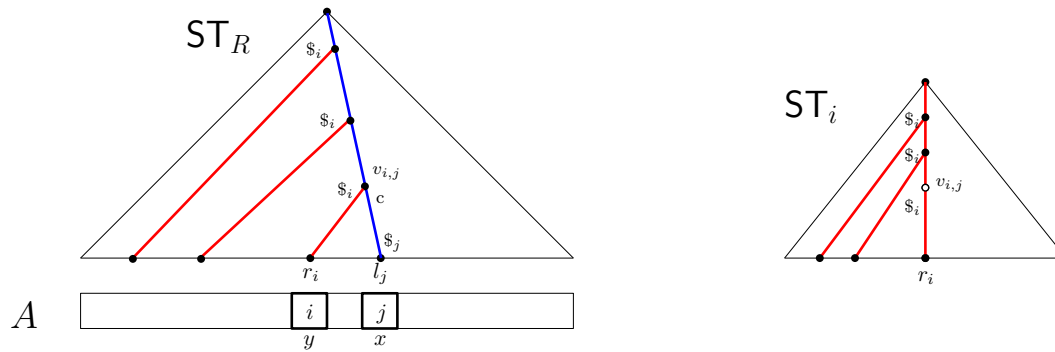
Proof. Let X^r denote the *reverse* of string X , i.e., $X^r = X[|X|] \dots X[1]$. We first sort S_1^r, \dots, S_k^r lexicographically, and store for each $j \in [1, k]$ a value $\text{rlex}[j] \in [1, k]$ equal to the rank of S_j^r in this sorted list. S_j is a suffix of S_i if and only if S_j^r is a prefix of S_i^r . The crucial property of this ordering is that all the strings such that S_j^r is their prefix form an interval from the position $\text{rlex}[j]$ to a position $\text{rlex}[j] + l[j] - 1$, where $l[j]$ is the total number of strings S_1^r, \dots, S_k^r starting with S_j^r ; that is, $\text{rlex}[j] + l[j]$ is the position of the first string having a longest common prefix with S_j^r shorter than $|S_j^r|$. The values $\text{rlex}[j]$ and $l[j]$, for all $j \in [1, k]$, can be computed in $\mathcal{O}(n)$ time [19].

As for the querying, for any i, j , we have that S_j is a suffix of S_i if and only if $\text{rlex}[j] \leq \text{rlex}[i] < \text{rlex}[j] + l[j]$, which is checked in $\mathcal{O}(1)$ time. The total size of arrays l and rlex is $\Theta(k)$. ◀

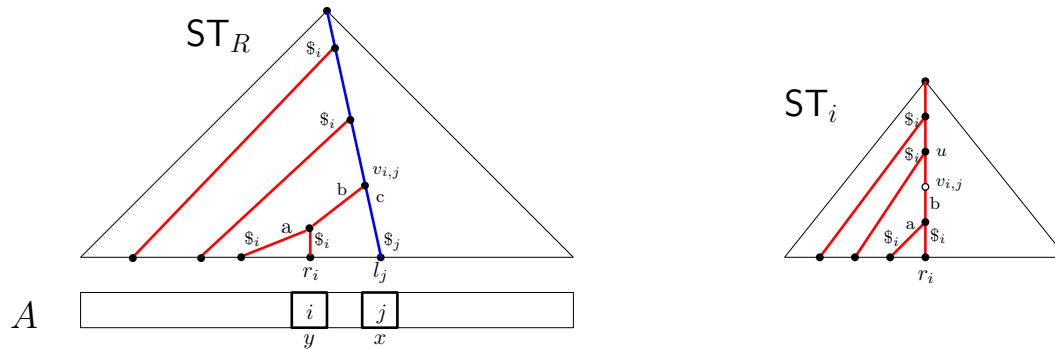
Construction. We start the construction of the data structure by constructing the data structure underlying Lemma 12. We also construct ST_R and ST_i , for all $i \in [1, k]$, using Lemma 2. We enhance ST_R with the data structure for LCA queries underlying Lemma 5, and link the leaf nodes originating from suffixes of $S_i \$i$ with the corresponding leaf nodes of ST_i , for all $i \in [1, k]$. We construct an array $A = A[1..|T_R|]$ over $[1, k]$ such that $A[\ell] = i$ if the ℓ th leaf node (from the left) of ST_R originates from a suffix of $S_i \$i$; since the leaf nodes are ordered according to the lexicographic order of the suffixes they originate from, array A can be easily extracted from $\text{SA}(T_R)$ constructed by means of Lemma 2. We enhance array A with the rank and select data structure underlying Lemma 6. We link the leaf nodes of ST_R with the corresponding elements of A . For each ST_i , we construct the data structure for WA queries underlying Lemma 7. For every node w of ST_i , we store the string depth of its closest ancestor (including w itself) that has an outgoing edge with label $\$i$ and hence corresponds to a suffix of S_i ; since the root always has such an edge, this assignment is always well-defined. In order to efficiently compute and store all those values, we simply process the information through the tree in a top-down manner. This completes the construction.

The part of the data structure that relies on Lemmas 2, 5, 7, and 12 is implemented in $\mathcal{O}(n)$ time and it occupies $\mathcal{O}(n)$ words of space. By Lemma 6, array A occupies $\mathcal{O}(n)$ words of space, and it can be implemented in $\mathcal{O}(n \log \log k)$ time as it stores k distinct values.

Querying. Consider a **One-to-One**(i, j) query; that is, we want to compute $\text{SPL}_{i,j}$, the length of the longest suffix of S_i that is a prefix of S_j . Let x be the position in array A that corresponds to the leaf node l_j of ST_R reached after conceptually reading $S_j \$j$. We first check if the entire S_j is a suffix of S_i by means of Lemma 12. If this is the case then we return $\text{SPL}_{i,j} = |S_j|$. If this is not the case (inspect Figure 3), we perform the following sequence of queries, $\text{select}_A(i, \text{rank}_A(i, x))$, which finds the position y in array A that corresponds to the leaf node r_i ; this corresponds to the suffix of $S_i \$i$ that is closest to the left of l_j . We then compute the lowest common ancestor of r_i and l_j : $v_{i,j} = \text{LCA}_{\text{ST}_R}(r_i, l_j)$. If node $v_{i,j}$ has an outgoing edge labeled with $\$i$, which ends at r_i , then we return $\text{SPL}_{i,j} = d(v_{i,j})$ (this



■ **Figure 3** An illustration of the **One-to-One**(i, j) query algorithm. The node $v_{i,j}$, which is explicit in ST_R but implicit in ST_i , has an outgoing edge labeled with $\$_i$ and hence the string depth $d(v_{i,j})$ of node $v_{i,j}$ is the answer to the query.



■ **Figure 4** An illustration of the **One-to-One**(i, j) query algorithm. The closest ancestor of node $v_{i,j}$, which is explicit in ST_R but implicit in ST_i , with an outgoing edge labeled with $\$_i$ is node u and hence the string depth $d(u)$ of node u is the answer to the query.

is the case in Figure 3). We check this by checking whether $d(r_i) = d(v_{i,j}) + 1$. If $v_{i,j}$ does not have such an outgoing edge (this is the case in Figure 4), we locate the explicit node corresponding to $v_{i,j}$ in ST_i (or its closest explicit ancestor if it is implicit) by asking a WA query: $w = WA_{ST_i}(r_i, d(v_{i,j}))$. Finally, we return the string depth of the closest ancestor of w with an outgoing edge labeled $\$_i$ as $SPL_{i,j}$; recall that every node of ST_i stores this information.

The time complexity of the query is $\mathcal{O}(\log \log k)$; the bottleneck is the complexity of the rank and select queries on A – all other operations take constant time. Let us now explain why the faster $\mathcal{O}(1)$ -time select and $\mathcal{O}(1 + \log \frac{\log k}{\log w})$ -time rank queries presented in [7], where w is the machine word, cannot improve our query time further. The size of the problem is $\Theta(n)$, hence the size of the machine word in the word-RAM model is $\Theta(\log n)$, thus the query time equals $\mathcal{O}(1 + \log \frac{\log k}{\log \log n})$. However, we have assumed that $k \geq \sqrt{n}$ (otherwise the structure of Lemma 8 implies an optimal solution – linear size and constant time queries – for the **One-to-One** queries), hence this is equal to $\mathcal{O}(1 + \log \log k) = \mathcal{O}(\log \log k)$ as stated.

Correctness. Recall that the answer to **One-to-One**(i, j) equals to the string depth of the closest ancestor of l_j in ST_R that has an outgoing edge labeled with $\$_i$. By construction, this ancestor ends on the right of l_j only if the entire S_j is a suffix of S_i , which we check separately. Otherwise, this ancestor is also an ancestor of r_i (which is on the left of l_i) as $\$_i$

goes out of the path from the root to l_j to the left (by construction, it is lexicographically smaller than the next letter on this path), and hence this edge labeled with s_i must end either in r_i or further to the left (by the definition of r_i). As an ancestor of l_j and r_i , it is also the closest ancestor of $v_{i,j}$ with such an outgoing edge; the latter actually exists (possibly as an implicit node) in ST_i (unlike l_j). The final steps of the query algorithm find the string depth of the node corresponding to the searched ancestor in ST_i (string depth is a shared property of the corresponding nodes).

We have arrived at Theorem 11. Note that the construction time for our data structure is $\mathcal{O}(n \log \log k)$. The bottleneck for the construction time is the construction time for the rank and select data structure (Lemma 6).

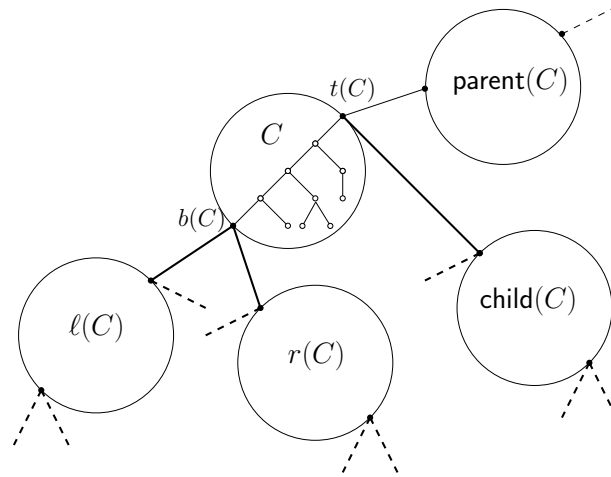
4 Answering One-to-All Queries

The spine of the data structure described in this section is $\text{FT}(R)$, the FTtree of R (see Section 2). Recall that for each node in $\text{FT}(R)$ (representing each prefix of a string S_i), we store information about which strings from R it is a prefix of (see Figure 2).

Main Idea. The Aho-Corasick lemma [1] states that for any two nodes, $\text{node}(U)$ and $\text{node}(V)$, in $\text{AC}(R)$, we have a failure transition from $\text{node}(U)$ to $\text{node}(V)$ if and only if V is the longest suffix of U that is also a prefix of some string in R . As a consequence, in $\text{FT}(R)$, $\text{node}(S)$ is an ancestor of $\text{node}(S')$ if and only if S is a suffix of S' (and both are prefixes of some strings from R as nodes of $\text{FT}(R)$). Thus the path from $\text{node}(\varepsilon)$ (the root) to $\text{node}(S_i)$ in $\text{FT}(R)$ contains exactly the nodes $\text{node}(S)$ such that S is a suffix of S_i and a prefix of some string in R . Those nodes are ordered according to the string length, hence the nodes closer to $\text{node}(S_i)$ on this path will correspond to *longer* suffix-prefix matches.

A One-to-All(i) query can thus be answered by simply reading the path from the root to $\text{node}(S_i)$ recording, for each $j \in [1, k]$, the last node on the path corresponding to a prefix of S_j . The space occupied by $\text{FT}(R)$ is in $\mathcal{O}(n)$; and such a query algorithm can take $\Theta(|S_i|)$, that is even $\Theta(n)$ time. Hence, by such an algorithm, we would not really gain anything from constructing $\text{FT}(R)$ in the preprocessing. On the other extreme, by running this algorithm not for a single path, but for the whole $\text{FT}(R)$ using a DFS traversal, we can precompute the answers for all the values of $i \in [1, k]$ in $\mathcal{O}(n + k^2)$ total time (and space), and then answer a query in $\mathcal{O}(k)$ time by simply outputting the k stored values; this would not be faster than using the algorithm by Gusfield et al. [27] or the one by Loukides and Pissis [34]. We will augment $\text{FT}(R)$ to obtain a more efficient solution combining the space efficiency of the first approach with the low query time of the second one.

A τ -micro-macro decomposition, introduced for rooted binary trees in [2], and then generalized for rooted general trees in [10] (after an appropriate mapping), is a partition of a rooted tree T of N nodes into $\mathcal{O}(N/\tau)$ connected subtrees, called *micro trees*. In the case of binary trees each micro tree is of size at most τ and at most two of its nodes are adjacent to nodes in other micro trees. These nodes are referred to as *top* and *bottom boundary* nodes of the micro tree. The top boundary node is chosen as the root of the micro tree. The *macro tree* is a rooted tree of size $\mathcal{O}(N/\tau)$ whose nodes correspond to micro trees as follows (inspect Figure 5): The top boundary node $t(C)$ of a micro tree C is connected to a boundary node $\text{parent}(C)$ in the parent micro tree (apart from the root). The boundary node $t(C)$ might also be connected to a top boundary node of a child micro tree, which we denote by $\text{child}(C)$. Such a τ -micro-macro decomposition can be computed in $\mathcal{O}(N)$ time for binary [2] and general [10] rooted trees. We summarize the above discussion in the lemma below.



■ **Figure 5** The structure of a micro-macro decomposition of a rooted binary tree.

► **Lemma 13** ([2, 10]). *For any rooted tree T with N nodes and for any integer $\tau \in [1, N]$, the τ -micro-macro decomposition of T can be computed in $\mathcal{O}(N)$ time.*

We will prove the following result.

► **Theorem 14.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering One-to-All(i) queries in $\mathcal{O}(k)$ time. The data structure can be constructed in $\mathcal{O}(n)$ time.*

Construction. We start the construction of the data structure by constructing $\text{FT}(R)$ from $\text{AC}(R)$ using Lemma 3. We compute the τ -micro-macro decomposition of $\text{FT}(R)$, for a parameter τ defined later, using Lemma 13. For each node u of the $\text{FT}(R)$, corresponding to a prefix S of some string S_i in R , we store the labeled interval I_u . For each boundary node in the τ -micro-macro decomposition of $\text{FT}(R)$, we store an array of k integers, which for each $i \in [1, k]$, stores the string depth of its lowest ancestor $\text{node}(S)$ such that S is a prefix of S_i . The additional size for storing this information in all the boundary nodes is $\mathcal{O}(k \cdot n/\tau)$. We compute these arrays by performing a DFS over $\text{FT}(R)$ with a set of k stacks, one for every string in R , storing the string depths of ancestors of the visited node of each type (which S_i they originate from). As there are only $2n$ updates of the stacks (each prefix of a string S_i is stored and removed once from the i th stack) and the information is stored by simply reading the top values of the k stacks, the total computation time is bounded by $\mathcal{O}(n + k \cdot n/\tau)$.

Querying. Let us start with the following observation from [34] (inspect also Figure 2).

► **Observation 15** ([34]). *Let u and v be two non-root nodes of $\text{FT}(R)$ with labeled intervals $I_u = [i_u, j_u]_{d(u)}$ and $I_v = [i_v, j_v]_{d(v)}$, respectively, and such that u is an ancestor of v . Then $d(u) < d(v)$ and either $[i_u, j_u]$ contains $[i_v, j_v]$ or $[i_u, j_u]$ and $[i_v, j_v]$ do not intersect.*

Consider a One-to-All(i) query; that is, we want to compute an array of length k , which stores $\text{SPL}_{i,j}$, for all $j \in [1, k]$. We start by finding the closest boundary node on the path from the root to $\text{node}(S_i)$; that is, the top boundary node of the micro tree containing $\text{node}(S_i)$. On the path between this top boundary node and $\text{node}(S_i)$, there are at most τ nodes. We compute the information coming from just those nodes in $\mathcal{O}(k + \tau)$ time with a

21:12 Suffix-Prefix Queries on a Dictionary

sweep line approach: there are $\mathcal{O}(\tau)$ (labeled) intervals from $[1, k]$, the intervals are labeled by different values (string depth), but, by Observation 15, two intervals are either disjoint or the one with the larger string depth is contained in the one with the smaller one. Thus, it is enough to hold the active intervals on a stack to keep track of the longest possible suffix-prefix match: the interval on the top of the stack has the highest value and will end the soonest. The solution is then obtained as the position-wise maximum of the computed array and the array stored in the top boundary node, which we compute in $\mathcal{O}(k)$ time.

Correctness. The correctness of the algorithm follows by the Aho-Corasick lemma (see also the discussion of the “main idea” paragraph above).

The data structure occupies $\mathcal{O}(n + k \cdot n/\tau)$ words of space and supports **One-to-All** queries in $\mathcal{O}(k + \tau)$ time. By setting τ to k (or to ck , for some positive constant c that balances the operation costs more efficiently) we obtain the complexities claimed in Theorem 14. Note that the data structure is constructed in $\mathcal{O}(n + k \cdot n/\tau)$ time, which is $\mathcal{O}(n)$ for $\tau = \Theta(k)$. Thus the presented data structure for **One-to-All** queries is optimal.

5 Answering Report and Count Queries

In this section we are going to use ST_R again. This time, however, instead of augmenting ST_R with an LCA data structure and linking its nodes with the rank and select array, we are going to link the nodes with rectangles and employ classic results from computational geometry for reporting (see Lemma 16) and counting (see Lemma 17).

Let $[x_1, x_2] \times [y_1, y_2]$ denote a rectangle in a 2D space with edges parallel to the axes, where the intervals $[x_1, x_2]$ and $[y_1, y_2]$ are the projections of this rectangle to the x -axis and y -axis, respectively. In the reporting version of the *2D rectangle stabbing* problem [15], we are given a set S of n rectangles to preprocess, so that when we are given a query point $q = (x, y)$, we report the subset $Q \subseteq S$ of rectangles $[x_1, x_2] \times [y_1, y_2]$ that contain q : $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. In the counting version of 2D rectangle stabbing, we are asked to return $|Q|$.

► **Lemma 16** ([45]). *For any set S of n rectangles, we can construct a data structure of $\mathcal{O}(n)$ words of space answering 2D rectangle stabbing reporting queries in $\mathcal{O}(\log n / \log \log n + f)$ time, where f is the output size $|Q|$.*

2D rectangle stabbing counting is known to be reducible to 2D orthogonal range counting [21], and such a data structure for 2D orthogonal range counting can be found in [28].

► **Lemma 17** ([21, 28]). *For any set S of n rectangles, we can construct a data structure of $\mathcal{O}(n)$ words of space answering 2D rectangle stabbing counting queries in $\mathcal{O}(\log n / \log \log n)$ time.*

Main Idea. For every suffix S of a string in R that is represented by a node in ST_R , we define a rectangle in 2D space: the x dimension corresponds to the lexicographically sorted list of all suffixes of strings in R whose prefix is S ; and the y dimension corresponds to interval $[0, |S|]$. A **Report** (resp. a **Count**) query is defined by two parameters, which form a point in the 2D space: i corresponds to string S_i in the same sorted list (x dimension) and ℓ corresponds to the smallest length of interest (y dimension). By reporting (resp. counting) all rectangles *enclosing this point* (Lemmas 16 and 17), we locate all suffix-prefix matches. Extra care, however, needs to be taken in order to avoid double reporting (resp. counting).

Construction. We start the construction of the data structure by constructing ST_R using Lemma 2. Let u be an explicit or implicit node of ST_R that is the parent of a leaf node reached with $\$i$: the labels of the path from root to u form a suffix of S_i . For every such node u and every i , we create a tuple $(L(u), R(u), d(u), i)$, where $L(u)$ and $R(u)$ are the (pre-order rank of) the leftmost and the rightmost leaf node under u , respectively.² Note that such a node may correspond to multiple tuples for different i values – this occurs when distinct elements of R share the same suffix. There are exactly n such tuples (one for every suffix) coming from ST_R and we can compute them in $\mathcal{O}(n)$ total time using a DFS traversal.

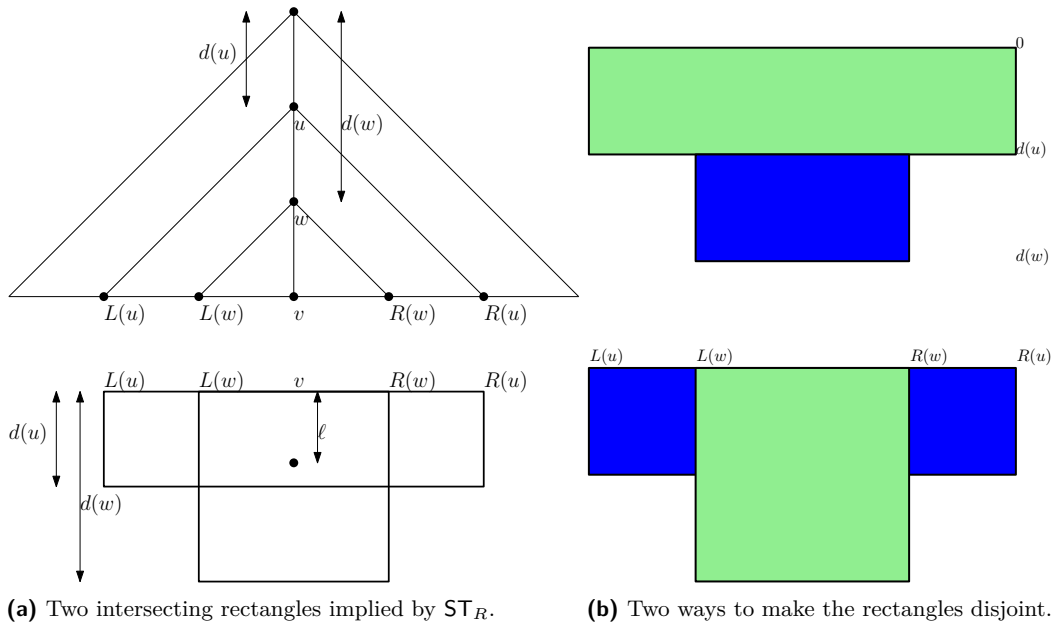
Recall that if we spell $S_j\$j$ in ST_R and the obtained leaf node v has an ancestor of string depth ℓ which has an outgoing edge with label $\$i$, then $\text{SPL}_{i,j} \geq \ell$. The same property ($\text{SPL}_{i,j} \geq \ell$) can be expressed by $L(v) \in [L(u), R(u)]$ (namely, u is an ancestor of v), and $\ell \in [0, d(u)]$ (namely, the string depth of u is at least ℓ) for a tuple $(L(u), R(u), d(u), i)$. Now note that $(L(u), R(u), d(u), i)$ forms a rectangle, whose identifier is i . In particular, $(L(u), R(u), d(u), i)$ can be viewed as rectangle $[L(u), R(u)] \times [0, d(u)]$ with satellite data i .

Now consider constructing the 2D rectangle stabbing data structure for reporting (resp. counting) for these n rectangles, and then ask the query for a point $(L(v), \ell)$, where v is the leaf node reached from the root by conceptually reading $S_j\$j$. The data structure will report (resp. count) all of the suffixes of S_i , for $i \in [1, k]$, of length at least ℓ that are also prefixes of S_j . Unfortunately, such a solution differs from the expected results of $\text{Report}(i, \ell)$ and $\text{Count}(i, \ell)$ in the following two ways:

1. Instead of finding all $j \in [1, k]$ such that $\text{SPL}_{i,j} \geq \ell$ for a given i , we find all such $i \in [1, k]$ for a given j . This issue is addressed by Observation 1, which states that $\text{Report}(i, \ell)$ and $\text{Count}(i, \ell)$ reduce trivially to the problems considered here, denoted by $\text{Report}^r(i, \ell)$ and $\text{Count}^r(i, \ell)$, respectively (recall that the r superscript refers to reversing the input strings);
2. If there are multiple prefixes of S_j of length at least ℓ that are also suffixes of S_i , then we will report (resp. count) each of them leading to double reporting (resp. counting). Although one may actually be interested in reporting or counting those multiple suffix-prefixes, in this paper, we are only interested in the *longest* ones. We address this issue by modifying the rectangles before the construction.

As mentioned earlier the first issue is resolved by Observation 1. To solve the second issue, we have to make the set of rectangles, for a single $i \in [1, k]$, pairwise disjoint while leaving their union unchanged. Notice that two such non-disjoint rectangles must come from a pair of nodes u and w in an ancestor-descendant relationship. An easy solution is to take, for every node w which has an outgoing edge with label $\$i$, its closest ancestor u which also has an outgoing edge with label $\$i$, and change the $[L(w), R(w)] \times [0, d(w)]$ rectangle into $[L(w), R(w)] \times [d(u) + 1, d(w)]$; inspect Figure 6. Since the part $[L(w), R(w)] \times [0, d(u)]$ is already contained in $[L(u), R(u)] \times [0, d(u)]$ the union remains unchanged, and since u is the closest such ancestor, the other rectangles (for this i) cannot have a nonempty intersection with the newly obtained one (the intersection with the ones coming from the descendants of w is empty after the modification of those rectangles). We can perform these modifications with a single DFS traversal with k stacks of nodes on the path from the root to the currently processed node, which has an outgoing edge with label $\$i$, $i \in [1, k]$. A more complicated solution is obtained by replacing the two rectangles $[L(u), R(u)] \times [0, d(u)]$ and $[L(w), R(w)] \times [0, d(w)]$ with three rectangles: $[L(u), L(w) - 1] \times [0, d(u)]$, $[L(w), R(w)] \times$

² $[L(u), R(u)]$ is also known as the suffix array interval of node u .



■ **Figure 6** On the bottom left part, the rectangles obtained from two nodes u and w of ST_R (top left), both having an outgoing edge with label $\$i$, forming a suffix-prefix match of S_i and S_j for node v reached by reading $S_j\$j$ from the root. The rectangles have a nonempty intersection. To avoid double reporting (or double counting), we make the rectangles disjoint while leaving their union unchanged. We can do this (by taking the intersection *once*) in two ways (on the right): a simple one (top) or a more complicated one (bottom), which allows us to efficiently output $SPL_{i,j}$.

$[0, d(w)]$ and $[R(w) + 1, R(u)] \times [0, d(u)]$; inspect Figure 6. Unlike the previous construction, a single rectangle can be spliced into smaller ones many times (a node can be a direct ancestor of many other nodes); at the same time a single rectangle can splice only its direct ancestor, hence the number of rectangles obtained this way is bounded from above by $2n$. This set of modified intervals can be obtained similarly: in a DFS traversal, when a node which has an outgoing edge with label $\$i$ is reached, we access its closest ancestor, which also has an outgoing edge with label $\$i$, and splice its rectangle. As such descendants of a node are visited from left to right, we always know which part of the rectangle will be spliced next, hence each such splice takes $\mathcal{O}(1)$ time leading to computing $\mathcal{O}(n)$ such modified rectangles in $\mathcal{O}(n)$ total time.

In order to finalize the construction of our data structure, we compute the set of modified rectangles of one of the two types described above, and construct for them the 2D rectangle stabbing data structures for reporting (Lemma 16) and counting (Lemma 17).

Querying. To answer a $\text{Report}^r(j, \ell)$ or a $\text{Count}^r(j, \ell)$ query, we simply ask the corresponding 2D rectangle stabbing data structure for the point $(L(v), \ell) = (R(v), \ell)$, where v is the node reached in ST_R from the root by conceptually reading $S_j\$j$. In case of a reporting query, the data structure returns a set of rectangles $[x, y] \times [\ell_1, \ell_2]$ labeled with distinct values $i \in [1, k]$. We can simply report the set of these i values. In case of a counting query, the result is simply an integer which we output. The two constructions of modified rectangles have additional nice properties however – each value i is associated with a value ℓ_2 . In case of the first construction, this ℓ_2 is the length of the shortest suffix of S_i which is also a prefix of S_j of length at least ℓ ; in case of the second construction, ℓ_2 is the length of the longest such suffix, that is $\ell_2 = \text{SPL}_{i,j}$.

Correctness. The correctness of the algorithm follows by the fact that point $(L(v), \ell) = (R(v), \ell)$ is enclosed by a rectangle $[L(u), R(u)] \times [0, d(u)]$ if and only if $S_j\$j$ has a prefix of length at least ℓ that is also a suffix of S_i ; and by the fact that the set of rectangles originating from a single i are made pairwise disjoint while their union remains unchanged.

We have thus arrived at the following lemma.

► **Lemma 18.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\text{Report}^r(j, \ell)$ queries in $\mathcal{O}(\log n / \log \log n + f)$ time, where f is the size of the output; and (ii) $\text{Count}^r(j, \ell)$ queries in $\mathcal{O}(\log n / \log \log n)$ time.*

By combining Lemma 18 with Observation 1 we obtain the main result of this section.

► **Theorem 19.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\text{Report}(i, \ell)$ queries in $\mathcal{O}(\log n / \log \log n + f)$ time, where f is the output size; and (ii) $\text{Count}(i, \ell)$ queries in $\mathcal{O}(\log n / \log \log n)$ time.*

Let us remark that the construction time for our data structures, excluding the implementation of the data structures underlying Lemmas 16 and 17, is $\mathcal{O}(n)$. Unfortunately, the construction time of the latter data structures (Lemmas 16 and 17) is not mentioned in [28, 45]. However, by using the construction from [15], we obtain $\mathcal{O}(n \log n)$ construction time, $\mathcal{O}(n)$ words of space, $\mathcal{O}(\log n + f)$ time for reporting, and $\mathcal{O}(\log n)$ time for counting.

► **Theorem 20.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering: (i) $\text{Report}(i, \ell)$ queries in $\mathcal{O}(\log n + f)$ time, where f is the output size; and (ii) $\text{Count}(i, \ell)$ queries in $\mathcal{O}(\log n)$ time. The data structure construction time is $\mathcal{O}(n \log n)$.*

Let us also remark that $\text{Report}(i, 0)$ (with the second construction of disjoint rectangles) actually answers any $\text{One-to-All}(i)$ query within the same asymptotic time: $\mathcal{O}(\log n + f) = \mathcal{O}(\log n + k) = \mathcal{O}(k)$ as $k \geq \sqrt{n}$. While the data structure for answering Report queries occupies $\mathcal{O}(n)$ words of space, like the data structure for One-to-All queries, the construction time for the former is more expensive – and it is likely much slower in practice.

6 Answering Top Queries

Recall that a $\text{Top}(i, K)$ query returns exactly K elements j for which $\text{SPL}_{i,j}$ is the largest, breaking ties arbitrarily. In case we are given an additional bound $K' \leq k$ such that $K \leq K'$ (e.g., we are only interested in finding $\mathcal{O}(1)$ many such top elements), the obvious data structure would be to store, for each $i \in [1, k]$, the sorted list of size K' of the best answers. Such a data structure allows answering $\text{Top}(i, K)$ queries, for $K \leq K'$, in the optimal $\mathcal{O}(K)$ time, but it requires $\mathcal{O}(kK')$ space, which for small K' may be $\mathcal{O}(n)$, but in general (i.e., when $K' = k$) leads back to the $\mathcal{O}(n + k^2)$ -time APSP algorithm. We show how to use our results from Section 5 to answer $\text{Top}(i, K)$ queries using $\mathcal{O}(n)$ space without this K' bound.

Clearly, we can assume that $K < k$. We start by making the following crucial observation.

► **Observation 21.** *For any $\text{Top}(i, K)$ query, with $K < k$, there exists an integer $\ell \in [0, n - 1]$ such that $\text{Count}(i, \ell + 1) \leq K < \text{Count}(i, \ell)$.*

Using the results from Section 5, we can find such an ℓ in $\mathcal{O}(\log^2 n / \log \log n)$ time using binary search on $\ell \in [0, n - 1]$ and the data structure for Count queries. Next we can simply compute $\text{Report}(i, \ell + 1)$ to be left with only choosing the remaining $(K - \text{Count}(i, \ell + 1))$

elements out of all $j \in [1, k]$ such that $\text{SPL}_{i,j} = \ell$. Unfortunately, there can be many such elements (even k), and we do not want this to influence the query time. We have to report the remaining elements out of the ones such that $\text{SPL}_{i,j} = \ell$ without computing or explicitly accessing all of them. Recall that, in ST_R , a list of elements i such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j can be accessed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing by finding the ancestor of the node reached by conceptually reading $S_j\$j$ at string depth ℓ (using a WA query) and reading the first letters of its outgoing edges from left to right; since $\$1 < \dots < \k are smaller than any element of Σ those values form a sorted list. Analogously, to access the list of elements j such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j , we simply use the symmetric data structure by Observation 1.

Unfortunately, this list may contain elements j such that $\text{SPL}_{i,j} > \ell$, and we do not want to report them again. This, however, can be fixed by maintaining a bitvector of size k as an integral part of our data structure; for each element $j \in \text{Report}(i, \ell + 1)$, we set the j th element of the bitvector to 1 in $\mathcal{O}(\text{Count}(i, \ell + 1)) = \mathcal{O}(K)$ time. When accessing the elements of the sorted list one-by-one, we simply check if the element was already outputted using the bitvector in $\mathcal{O}(1)$ time. In total, we can check up to K such elements, hence the total time of merging those two parts of the output is $\mathcal{O}(K)$ (including the bitvector reset). We summarize the solution in Theorem 22, which is the main result of this section.

► **Theorem 22.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering $\text{Top}(i, K)$ queries in $\mathcal{O}(\log^2 n / \log \log n + K)$ time.*

Proof. We start the construction of the data structure by constructing the data structures for $\text{Report}(i, \ell)$ and $\text{Count}(i, \ell)$ using Theorem 19. We also construct a data structure to find the list of elements j such that S_i has a suffix-prefix match of length ℓ with S_j in $\mathcal{O}(1)$ time using Lemmas 2 and 7 and Observation 1. Finally, we also maintain a bitvector of size $k = \mathcal{O}(n)$. The space required by our data structure is $\mathcal{O}(n)$ words.

Consider a $\text{Top}(i, K)$ query. We ask $\mathcal{O}(\log n)$ Count queries and a single Report query in $\mathcal{O}(\log^2 n / \log \log n + K)$ total time, as the output is bounded by K . We index the Report result in the bitvector. We find the list (without reading its content) of elements j such that S_i has a suffix of length exactly ℓ which is also a prefix of S_j in $\mathcal{O}(1)$ time. Finally, we access and check at most K elements from the list in $\mathcal{O}(K)$ total time.

The correctness of the algorithm follows by Observation 21 and Theorem 19. ◀

Similar to Section 5, the construction time for our data structure, excluding the implementation of Theorem 19, is $\mathcal{O}(n)$. If instead of Theorem 19, we employ Theorem 20, we obtain $\mathcal{O}(n \log n)$ construction time, $\mathcal{O}(n)$ words of space, and $\mathcal{O}(\log^2 n + K)$ query time.

► **Theorem 23.** *For any dictionary of k strings of total length n over an integer alphabet of size $\sigma \leq n^{\mathcal{O}(1)}$, we can construct a data structure of $\mathcal{O}(n)$ words of space answering $\text{Top}(i, K)$ queries in $\mathcal{O}(\log^2 n + K)$ time. The data structure construction time is $\mathcal{O}(n \log n)$.*

References

- 1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
- 2 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Minimizing diameters of dynamic trees. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer, 1997. doi:10.1007/3-540-63165-8_184.

- 3 Amihoud 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.
- 4 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theor. Comput. Sci.*, 922:271–282, 2022. doi:10.1016/j.tcs.2022.04.029.
- 5 Carl Barton, Costas S. Iliopoulos, Solon P. Pissis, and William F. Smyth. Fast and simple computations using prefix tables under hamming and edit distance. In Jan Kratochvíl, Mirka Miller, and Dalibor Fronček, editors, *Combinatorial Algorithms – 25th International Workshop, IWOCA 2014, Duluth, MN, USA, October 15-17, 2014, Revised Selected Papers*, volume 8986 of *Lecture Notes in Computer Science*, pages 49–61. Springer, 2014. doi:10.1007/978-3-319-19315-1_5.
- 6 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.
- 7 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.
- 8 Ilan Ben-Bassat and Benny Chor. String graph construction using incremental hashing. *Bioinform.*, 30(24):3515–3523, 2014. doi:10.1093/bioinformatics/btu578.
- 9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- 10 Philip Bille and Inge Li Gørtz. The tree inclusion problem: In linear space and faster. *ACM Trans. Algorithms*, 7(3):38:1–38:47, 2011. doi:10.1145/1978782.1978793.
- 11 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. FSG: fast string graph construction for de novo assembly. *J. Comput. Biol.*, 24(10):953–968, 2017. doi:10.1089/cmb.2017.0089.
- 12 Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Inf. Process. Lett.*, 155, 2020. doi:10.1016/j.ipl.2019.105862.
- 13 Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. 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 138–151. ACM, 2019. doi:10.1145/3313276.3316316.
- 14 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/s00453-021-00821-y.
- 15 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.
- 16 Shiri Chechik. Approximate distance oracles with constant query time. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 – June 03, 2014*, pages 654–663. ACM, 2014. doi:10.1145/2591796.2591801.
- 17 Shiri Chechik. Approximate distance oracles with improved bounds. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 1–10. ACM, 2015. doi:10.1145/2746539.2746562.

- 18 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages. part I. *CoRR*, abs/2208.04931, 2022. doi:10.48550/arXiv.2208.04931.
- 19 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 20 Shiri Dori and Gad M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006. doi:10.1016/j.ipl.2005.11.019.
- 21 Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Inf. Process. Lett.*, 14(3):124–127, 1982. doi:10.1016/0020-0190(82)90068-0.
- 22 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
- 23 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996. doi:10.1007/3-540-61258-0_11.
- 24 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms – ESA 2014 – 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.
- 25 Giorgio Gonnella and Stefan Kurtz. Readjoinder: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinform.*, 13:82, 2012. doi:10.1186/1471-2105-13-82.
- 26 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 27 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. doi:10.1016/0020-0190(92)90176-V.
- 28 Joseph F. JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004. doi:10.1007/978-3-540-30551-4_49.
- 29 Shahbaz Khan. Optimal construction of hierarchical overlap graphs. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wroclaw, Poland*, volume 191 of *LIPICs*, pages 17:1–17:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.17.
- 30 Tomasz Kociumaka. Efficient data structures for internal queries in texts. *PhD thesis, University of Warsaw, October 2018.*, 2018. URL: <https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 31 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *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.
- 32 Gregory Kucherov and Dekel Tsur. Improved filters for the approximate suffix-prefix overlap problem. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2014. doi:10.1007/978-3-319-11918-2_14.

- 33 Jihyuk Lim and Kunsoo Park. A fast algorithm for the all-pairs suffix-prefix problem. *Theor. Comput. Sci.*, 698:14–24, 2017. doi:10.1016/j.tcs.2017.07.013.
- 34 Grigorios Loukides and Solon P. Pissis. All-pairs suffix/prefix in optimal time using Aho-Corasick space. *Inf. Process. Lett.*, 178:106275, 2022. doi:10.1016/j.ipl.2022.106275.
- 35 Felipe A. Louza, Simon Gog, Leandro Zanotto, Guido Araujo, and Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval – 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, volume 9954 of *Lecture Notes in Computer Science*, pages 122–132, 2016. doi:10.1007/978-3-319-46049-9_12.
- 36 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 37 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, September 2005. doi:10.1093/bioinformatics/bti1114.
- 38 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>.
- 39 Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Inf. Process. Lett.*, 110(3):123–128, 2010. doi:10.1016/j.ipl.2009.10.015.
- 40 Sangsoo Park, Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. A linear time algorithm for constructing hierarchical overlap graphs. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPICs*, pages 22:1–22:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.22.
- 41 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.
- 42 Maan Haj Rachid and Qutaibah Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed Res. Int.*, 2015(905261), 2015. doi:10.1155/2015/905261.
- 43 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 233–242. ACM/SIAM, 2002. URL: <http://dl.acm.org/citation.cfm?id=545381.545411>.
- 44 Kim R. Rasmussen, Jens Stoye, and Eugene W. Myers. Efficient q -gram filters for finding all ϵ -matches over a given length. *J. Comput. Biol.*, 13(2):296–308, 2006. doi:10.1089/cmb.2006.13.296.
- 45 Qingmin Shi and Joseph F. JáJá. Novel transformation techniques using q -heaps with applications to computational geometry. *SIAM J. Comput.*, 34(6):1474–1492, 2005. doi:10.1137/S0097539703435728.
- 46 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinform.*, 26(12):367–373, 2010. doi:10.1093/bioinformatics/btq217.
- 47 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology – 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2018. doi:10.1007/978-3-319-89929-9_14.
- 48 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.

- 49 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. doi:10.1016/j.jda.2016.04.002.
- 50 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990. doi:10.1007/BF01840391.
- 51 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 52 Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Inf. Comput.*, 213:49–58, 2012. doi:10.1016/j.ic.2012.02.002.
- 53 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

Merging Sorted Lists of Similar Strings

Gene Myers   

Okinawa Institute of Science and Technology, Japan

MPI for Molecular Cell Biology and Genetics, Dresden, Germany

Abstract

Merging T sorted, non-redundant lists containing M elements into a single sorted, non-redundant result of size $N \geq M/T$ is a classic problem typically solved practically in $O(M \log T)$ time with a priority-queue data structure the most basic of which is the simple *heap*. We revisit this problem in the situation where the list elements are *strings* and the lists contain many *identical or nearly identical elements*. By keeping simple auxiliary information with each heap node, we devise an $O(M \log T + S)$ worst-case method that performs no more character comparisons than the sum of the lengths of all the strings S , and another $O(M \log(T/\bar{e}) + S)$ method that becomes progressively more efficient as a function of the fraction of equal elements $\bar{e} = M/N$ between input lists, reaching linear time when the lists are all identical. The methods perform favorably in practice versus an alternate formulation based on a trie.

2012 ACM Subject Classification Theory of computation \rightarrow Theory and algorithms for application domains

Keywords and phrases heap, trie, longest common prefix

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.22

Related Version *Prepublication*: <https://arxiv.org/abs/2208.09351>

Supplementary Material *Software*: <https://github.com/thegenemyers/STRING.HEAP>
archived at `swh:1:dir:4f72ad3dfb64ad16ba647ca815cd76a15bb6e5be`

Acknowledgements I would like to acknowledge Richard Durbin and Travis Gagie, for their encouragement to write up this work, Shane McCarthy for providing the data sets that made the need for a collision heap apparent, Gonzalo Navarro for suggesting the trie approach and pointing at Thorup’s work, and Shinichi Morishita and Yoshihiko Suzuki for their many helpful comments and review of the work.

1 Introduction & Summary

Producing a sorted list, possibly with duplicate elements removed, from a collection of T sorted input lists is a classic problem [4]. Moreover, with today’s massive data sets, where an in-memory sort would require an excessively large memory, this problem gains in importance as a component of an external, disk-based sort. Our motivating example is modern DNA sequencing projects that involve anywhere from 100 billion to 5 trillion DNA bases of data in the form of sequencing reads that are conceptually strings over the 4-letter alphabet A, C, G, T [8]. In particular, the problem of producing a sorted table of all the k -mers (substrings of length exactly k) and their counts has been the focus of much study and is used in many analysis methods for these data sets [5, 6, 7].

Priority queue implementations such as a heap, take $O(\log T)$ to extract the next minimum and insert its replacement, giving an $O(M \log T)$ merge time where M is the sum of the lengths of the input lists [2]. However when the domain of the merge is strings, as opposed to say integers, then one must consider the time taken for each of the $O(\log T)$ string comparisons, which is not $O(1)$ but conceptually the average length of the longest common prefix (*lcp*) between all the compared strings. For example, this is $O(\log_{\Sigma} M)$ in the “*Uniform Scenario*”



© Gene Myers;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 22; pp. 22:1–22:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

where the characters of the strings are chosen with equal probability over an alphabet of size Σ . But it can be much worse, for example, when merging lists of say 21-mers each obtained from a portion of a 40X coverage DNA sequencing data set, where many strings are identical.

In the worst case, one can only assert that the time to merge the list of strings is $O(S \log T)$ where S is the total number of characters in the input lists, e.g. Mk for lists of k -mers. Assuming the Uniform Scenario, one can more accurately characterize the efficiency as $O(M \log T \log M)$ *expected* time. In this paper we present a method that is guaranteed to take $O(M \log T + S)$ time by modifying the heap data structure so that the *amortized* time spent on comparing the characters of any string while it is in the heap is never more than its length. Moreover, in the Uniform Scenario, the efficiency is $O(M(\log T + \log M))$ *expected* time. We call such a modified heap a *string heap*. Interestingly, a binary search tree augmented by a generalized list structure that also leverages *lcp*'s was developed by Amir et al. [1] and also achieves the bounds above, albeit with a different logic/design.

It is further true in the case of DNA sequencing data sets, that often the number of elements N in the merged list is much smaller than M when duplicate elements are removed. Specifically, N can be as small as M/T assuming the input lists themselves do not contain equal elements. With another modification to a heap, not specific to strings *per se*, we will achieve here an algorithm that takes $O(M \log(T/\bar{e}))$ time where $\bar{e} = M/N$ is the average number of distinct input lists a given element is in. So when all the input elements are unique the time is as usual $O(M \log T)$ but as \bar{e} increases less time is taken, reaching $O(M)$ when all the input lists are identical, that is, $\bar{e} = T$. We call such a modified heap a *collision heap*. We show it can easily be combined with a string heap to give an $O(M \log(T/\bar{e}) + S)$ algorithm for string merging.

While the focus of this paper is on modifying a heap to support string elements, an orthogonal approach to realizing a priority queue (PQ) of strings appeared in a comprehensive paper by Thorup ([9]) that is primarily focused on integer PQs, but which in Section 6 uses a trie [3] to merge strings of, potentially large, integers in $O(M \log \log T + S)$ time. In bioinformatics, strings are generally over alphabets of small size Σ , e.g. 4 for DNA, so taking Thorup's algorithm, but replacing the general integer priority queue with van Emde Boas small integer PQs [10] over domain Σ , one obtains an $O(M \log \log \Sigma + S)$ time algorithm. The implementation of either of these methods encounters rather larger overheads compared to simply realizing the basic approach of Thorup's algorithm with a compact trie with Σ -element arrays for the out-edges. Moreover, because adding to the trie then becomes linear, the complexity of this simplified approach is $O(N\Sigma + S)$. Given limited values of Σ , e.g. say up to 20 for protein sequences, the trie approach is very competitive, especially for the cases where N is significantly smaller than M .

We implemented programs to merge files of sorted strings using a regular heap, a string heap, a collision heap, a combination of the string and collision heap, and a simple compact trie and performed timing experiments on both simulated and real DNA sequencing k -mer data to determine their relative performance. The codes are available at github.com/theenemyers/HEAPS. Amongst the heap-based algorithms, the string heap proves superior as the average *lcp* between consecutive output strings increases, and the collision heap proves superior as the collision ratio \bar{e} increases. Also, the combination heap tracked the behavior of whichever of the string or collision heap proves superior, but at an overhead of roughly 5%. Against the trie approach, the string heap is faster in the uniform scenario until T becomes quite large, e.g. 256 in our experiments. In scenarios where $N \ll M$ due to a *uniform* collision rate the trie proved fastest save for small values of T . For real data sets, where the collision rate is highly variable, the collision and combination heaps gave the best times. In short, the new heap methods are of both theoretical and practical interest.

2 Preliminaries: Definitions and a Short Recap of Heaps

Consider T sorted lists of strings $S_t = s_1^t, s_2^t, \dots, s_{N_t}^t$ of lengths N_t . We assume that the elements are distinct, i.e. $s_j^t < s_{j+1}^t$, and let $s_j^t = a_0^{j^t} a_1^{j^t} \dots a_{n_j^t-1}^{j^t}$. Note carefully, that the first character of a string is at index 0. The problem is to produce a single sorted list $R = r_1, r_2, \dots, r_N$ of length N with any duplicates *between* the lists removed. That is, while each input list has unique strings, the same string, can occur in up to T different lists. Let $e_i \in [1, T]$ be the number of different queues the string r_i occurs in. Letting $M = \sum_{t=1}^T N_t$ be the sum of the lengths of the input lists, note that N is in the range $[M/T, M]$.

A T element heap is a complete binary tree of T nodes containing or referring to domain values to be prioritized. A heap further has the *heap property* when for every node, the domain values of its children are not less than its domain value. A heap can be very simply implemented as an array $H[1..T]$ where $H[i]$ is the datum for node i , its left child is $2i$, and its right child is $2i + 1$ (if they exist, i.e. are $\leq T$).

In the case of merging T input lists, we will let each heap node contain the index $t \in [1, T]$ of an input list and another array, $V[0..T]$ will contain the current value for that list in $V[t]$ (the role of $V[0]$ is discussed in the next paragraph). If all the nodes greater than i have the heap property, then recall that the simple routine *Heapify*(i, x, t) in Figure 1 below will add the value x from list t to the heap guaranteeing that H has the heap property for all nodes greater than $i-1$. The routine takes time proportional to at most the height of i in the heap which is $O(\log T)$ for all i .

Let $S[t]$ denote the t^{th} sorted input list and assume it operates as a one-sided queue where one can *Pop* the next element from the list and ask if the queue is *Empty*. We will also assume that *INFINITY* is an infinitely large string value greater than all those encountered as input and when a list is exhausted place this value at $V[0]$ so that $H[1]$ becomes 0 when *all* the lists are exhausted. Finally, for simplicity we assume each list has at least one element, i.e. $N_t > 0$ for all t . Then a complete pseudo-code for the basic priority queue approach to merging T sorted lists *while removing duplicate values* is shown at right in Figure 1.

```

int      T
int      H[1..T]
domain  V[0..T]

Heapify(i,x,t)
{ c = i
  while (u=2c) <= T do
    { if u < T and V[H[u+1]] < V[H[u]] then
      u = u+1
      if x <= V[H[u]] then
        break
      H[c] = H[u]
      c = u
    }
  (H[c],V[t]) = (t,x)
}

domain_list S[1..T]
1. for t = T downto 1 do
2.   Heapify(t,Pop(S[t]),t)

3. last = INFINITY
4. while (t = H[1]) > 0 do
5.   { x = V[t]
6.     if x != last then
7.       output (last=x)
8.     if Empty(S[t]) then
9.       Heapify(1,INFINITY,0)
10.    else
11.      Heapify(1,Pop(S[t]),t)
}
```

■ **Figure 1** The *Heapify* routine (left) and the overall merge algorithm (right).

In lines 1 and 2, the first element of each list is *Pop*'d and placed in the heap in reverse order of the nodes so that the entire heap has the heap property upon completion. The total time taken for this setup is $O(T)$ as the sum of the heights of the nodes in a complete binary tree is of this order. Then in the while-loop of line 4, the list t with the next smallest element is $H[1]$ and if this value is not zero (indicating the exhaustion of all the queues), then the element $x = V[t]$ is processed in the loop body. If the value x is not a duplicate of

the last element output then it is output (lines 5 - 7). If list t is not empty then its next element replaces the element just output and the heap property is restored at node 1 (lines 8 & 11). Otherwise the element is replaced with the largest possible value $INFINITY$ (line 9) in “queue” 0 so that when all T lists are exhausted the extraction of 0 as the queue index marks the end of the merge.

Given that *Heapify* takes $O(\log T)$ time and an input element is processed with each iteration of the loop, the algorithm clearly operates in $O(M \log T)$ time assuming domain comparisons are $O(1)$. As discussed in the introduction this assumption is not necessarily true when the values are strings and we address this in the next section.

3 The String Heap

The idea for a string heap is very simple, namely, for each node also record and keep current the length of the longest common prefix between the string at the node and the string at its parent (except for the root). Let $lcp(u, v)$ be the longest common prefix between strings u and v . Then more formally, a string heap also maintains a third array $P[1..T]$ such that $P[i] = lcp(V[H[i]], V[H[\lfloor i/2 \rfloor]])$ for $i > 1$. The interesting and complex part of this extension is maintaining this property during the induction of *Heapify* and using it to accelerate the comparison of string values by limiting the number of character comparisons involved.

Intuitively, $Heapify(i, x, t)$ traverses the maximal left most path starting at i , all of whose elements are less than x and not more than their siblings until a node c^* is reached that is either a leaf or for which all its children are not less than x . The values along this path are shifted up to the node above during each iteration until x is placed at node c^* at the last. To help argue the induction to follow, it conceptually simplifies matters to think of x as being explicitly placed at the node indexed by the variable c (i.e. $H[c] = t$) as the algorithm descends from node i to the final placement of x . From this viewpoint, at the start of each iteration of the loop of *Heapify*, the heap satisfies the heap property at every node in the subtree rooted at i except c where x conceptually currently resides. For the array P realizing a string heap, the loop invariant is that P is correct except possibly at nodes $2c$ and $2c + 1$ as x has just been placed at their parent node c . Our goal is to maintain this invariant through the next iteration of the loop where either x is found to be not greater than the children of c and the loop exits, or the algorithm descends to one of the children of c swapping x with the child’s value.

To facilitate a simpler logic around the comparison of strings, we will assume that every string ends with a special terminating character $\$$ that is less than any ordinary character (e.g. 0 for C-strings). With this convention, finding the lcp of two strings x and y is simply a matter of finding the first index ρ for which the strings have unequal characters or both are $\$$. Moreover note that $x < y$ iff $x[\rho] < y[\rho]$.

For all but the first iteration of the loop of *Heapify*, note that the value that was at the current node c is now at $\lfloor c/2 \rfloor$ having been exchanged with x as it has a smaller value. In what follows, we will let $o = V[H[\lfloor c/2 \rfloor]] < x$ be this value and also let $v_l = V[H[2c]]$ and $v_r = V[H[2c + 1]]$ be the strings currently at the left and right children of c . Observe that it must be that $P[2c] = lcp(o, v_l)$ and $P[2c + 1] = lcp(o, v_r)$ as these values are unchanged since the previous iteration when o was at node c . Let p_l and p_r denote these values, respectively, and further let $p = P[c] = lcp(o, x)$ in the proof/analysis that follows.

► **Theorem 1.** *Given that insertions are monotone, i.e. the next value inserted is not less than the value just extracted, the *Heapify* routine of Figure 2 is correct once initialized. To start, it suffices to set, $H[i] = 0$ for all i and $V[0] = \$$ and then perform Lines 1 and 2 of the merge given in Figure 1. After calling *Heapify*, $P[1]$ is the lcp of the the last value extracted and the value at the root.*

Proof. First consider the situation when the heap has been correctly initialized and one is now inserting a new element x as in Line 11 of the merge algorithm in Figure 1. So upon entry to *Heapify*, $c = i = 1$ and observe that it will be the case that $P[2]$ and $P[3]$ will have the value $lcp(o, H[V[2]])$ and $lcp(o, H[V[3]])$ where o is the value that was just extracted from the root of the heap and which x is now about to replace. So in order to get started we need to set p to $lcp(o, x)$ where $o < x$ by the monotonicity condition. Given that nothing has been placed at $V[H[1]]$, it still has o as its value, and so in Figure 2, *Heapify* starts correctly by setting p to $LCP2(V[H[1]], x, 0)$ before initiating its loop. Indeed one could imagine that o is at the virtual father of the root 1.

(The reader should observe that if $o > x$, i.e. the context is not monotone, then getting the induction started also requires readjusting $P[2]$ and $P[3]$ downward to p if they happen to be larger than p . In this case, we can no longer place a bound on the total number of character comparison made during the operation of the heap, but it will still operate correctly.)

In the case that the heap is being initialized, i.e. $i > 1$ in Line 2 of the merge algorithm, it suffices to let o be the empty string, ϵ , so that $P[i] = P[2i] = P[2i + 1] = 0$. The conditions of the theorem correctly guarantee then that $LCP2(V[H[1]], x, 0) = LCP2(\$, x, 0) = 0$.

We now proceed to analyze the numerous cases that arise to maintain the induction during the iterations of *Heapify*'s loop in terms of the relationships between the quantities p , p_l , and p_r . To further simplify matters observe that the treatment of the left and right children of c is symmetric, so we only consider the left case, $p_r \leq p_l$, in the enumeration below knowing that the right case, $p_l < p_r$, is handled simply by exchanging the roles of left and right. Furthermore, we repeatedly use the logic that if $lcp(x, s) < lcp(s, y)$ then $lcp(x, y) = lcp(x, s)$ and $x < s$ iff $x < y$.

Case 1: $p_r < p_l$ and $p < p_l$. By the case condition $lcp(x, o) = p < p_l = lcp(o, v_l)$ and since we know $x > o$ we can conclude that $x > v_l$ and $lcp(x, v_l) = lcp(x, o) = p$. Similarly $lcp(v_r, o) = p_r < p_l = lcp(o, v_l)$ and we know $v_r \geq o$ allowing us to conclude that $v_r > v_l$ and $lcp(v_r, v_l) = lcp(v_r, o) = p_r$. So v_l is the smallest of x , v_l , and v_r implying that the loop should descend to $2c$ with v_l being placed at c . Moreover, $P[c]$ should be set to p_l , while p and $P[2c + 1]$ can remain unchanged having already the correct values for the next iteration.

Case 2: $p_r \leq p_l$ and $p > p_l$. By the case condition $lcp(v_l, o) = p_l < p = lcp(o, x)$ and since we know $v_l \geq o$ we can conclude that $x < v_l$ and $lcp(v_l, x) = lcp(v_l, o)$. Similarly $lcp(v_r, o) = p_r \leq p_l < p = lcp(o, x)$ and we know $v_r \geq o$ allowing us to conclude that $x < v_r$ and $lcp(v_r, x) = lcp(v_r, o)$. So the loop can terminate with x being placed at node c . Moreover, $P[2c]$ and $P[2c + 1]$ remain unchanged having yet the correct values.

Case 3: $p_r < p_l$ and $p = p_l$. First compute $p_x = p + lcp(v_l + p, x + p)$ where $s + j$ is the suffix of string s beginning at position j . Clearly $p_x = lcp(v_l, x)$ and if $v_l[p_x] < x[p_x]$ then $v_l < x$, otherwise $v_l \geq x$. We have two subcases:

Subcase 3a: $v_l[p_x] < x[p_x]$. The condition $p_r < p_l$ implies $lcp(v_r, o) < lcp(o, v_l)$ and we know $v_r \geq o$ allowing us to conclude that $v_r > v_l$ and $lcp(v_r, v_l) = lcp(v_r, o)$. Thus v_l is smaller than both x and v_r . So the loop should descend to $2c$ with v_l being placed at c . Therefore, $P[c]$ should be set to p_l and p to p_x , while $P[2c + 1]$ has the correct value.

Subcase 3b: $v_l[p_x] \geq x[p_x]$. By the case conditions we know $p_r < p$ implying $lcp(v_r, o) < lcp(o, x)$ and we know $v_r \geq o$ allowing us to conclude that $v_r > x$ and $lcp(v_r, x) = lcp(v_r, o)$. Thus x is not less than both v_l and v_r . So the loop can terminate with x being placed at node c . While $P[2c + 1]$ remains correct, $P[2c]$ needs to be updated to p_x .

Case 4: $p_r = p_l$ and $p < p_l$. Compute $p_x = p_l + \text{lcp}(v_l + p_l, v_r + p_l)$ which is clearly $\text{lcp}(v_l, v_r)$. If $v_r[p_x] < v_l[p_x]$ then $v_r < v_l$, otherwise $v_r \geq v_l$. WLOG let's assume $v_l \leq v_r$, as the case $v_r < v_l$ is symmetric. As in cases before $p < p_l$ and $x > o$ allow us to surmise that $x > v_l$ and $\text{lcp}(x, v_l) = \text{lcp}(x, o)$. Therefore v_l is smaller than x and not larger than v_r , implying that the loop should descend to $2c$ with v_l being placed at c . Therefore, $P[c]$ should be set to p_l , $P[2c + 1]$ to p_x , while p continues to have the correct value.

Case 5: $p_r = p_l = p$. First compute $p_x = p + \text{lcp}\beta(v_l + p, v_r + p, x + p)$ where $\text{lcp}\beta$ is the 3-way common prefix, and this by the case conditions is clearly equal to $\text{lcp}\beta(v_l, v_r, x)$. There now arise numerous subcases based on the relationships between $x[p_x]$, $v_l[p_x]$, and $v_r[p_x]$ in direct analogy to the subcases based on the relationships between p , p_l , and p_r , so we will number these 5.1, 5.2, and so on:

Subcase 5.1: $v_r[p_x] > v_l[p_x]$ and $x[p_x] > v_l[p_x]$. The case conditions imply $x > v_l$ and $v_r > v_l$ and $\text{lcp}(v_l, x) = \text{lcp}(v_l, v_r) = p_x$. So v_l is the smallest of x , v_l , and v_r implying that the loop should descend to $2c$ with v_l being placed at c . So $P[c]$ should be set to p_l , while p and $P[2c + 1]$ are now clearly p_x .

Subcase 5.2: $v_r[p_x] \geq v_l[p_x]$ and $x[p_x] \leq v_l[p_x]$. In this subcase, clearly x is not more than both v_l and v_r and $\text{lcp}(v_l, x) = \text{lcp}(v_r, x) = p_x$. So the loop should terminate and both $P[2c]$ and $P[2c + 1]$ should be updated to p_x .

Subcase 5.3: $v_r[p_x] > v_l[p_x]$ and $x[p_x] = v_l[p_x]$. First compute $p_y = p_x + \text{lcp}(v_l + p_x, x + p_x)$ which is clearly $\text{lcp}(v_l, x)$ note that the conditions to this point imply $\text{lcp}(v_l, v_r) = \text{lcp}(x, v_r) = p_x$.

Subcase 5.3a: $v_l[p_y] < x[p_y]$. So v_l is the smaller than x and v_r implying the loop should descend to $2c$ with v_l being placed at c . So $P[c]$ should be set to p_l and the correct new values for p and $P[2c + 1]$ are p_y and p_x , respectively.

Subcase 5.3b: $v_l[p_y] \geq x[p_y]$. So x is not smaller than v_l and v_r implying the loop can terminate at c ,

Subcase 5.4: $v_r[p_x] = v_l[p_x]$ and $x[p_x] > v_l[p_x]$. Compute $p_y = p_x + \text{lcp}(v_l + p_x, v_r + p_x)$ which is clearly $\text{lcp}(v_l, v_r)$. If $v_r[p_y] < v_l[p_y]$ then $v_r < v_l$, otherwise $v_r \geq v_l$. WLOG let's assume $v_l \leq v_r$, as the case $v_r < v_l$ is symmetric. By the case conditions $x > v_l$ and $\text{lcp}(x, v_l) = p_x$. Therefore v_l is smaller than x and not larger than v_r , implying that the loop should descend to $2c$ with v_l being placed at c . Therefore, $P[c]$ should be set to p_l , $P[2c + 1]$ to p_y , and p to p_x . ◀

Figure 2 presents the complete algorithm for the string version of *Heapify* embodying the case analysis above so that the P -array values are correctly maintained. Note carefully, that the lcp information in the P -array is used both to determine the relative values of the heap elements and hence direct the path that *Heapify* takes to insert a new element x , but further also saves time on the number of character comparisons performed by only computing new lcp 's in terms of an initial lcp -offset that is common to *all* of the arguments to LCP2 or LCP3. So as regards complexity, the algorithm for *Heapify* takes $O(\log T)$ time **plus** the time spent in LCP2 or LCP3 for character comparisons. Note carefully the code assumes that we are merging *sorted* string lists, so the value of x is not less than the value of the previous element $o = V[H[1]]$ on the same queue $H[1]$, i.e. the computation is *monotone*. We make an amortization argument to bound the total number of character comparisons as follows:

```

Heapify(int i, string x, int t)
{ c = i
  p = LCP2(V[H[i]],x,0)
  while (l = 2c) <= T do
    { (hl,pl) = (H[l],P[l])
      if l < T then
        (hr,pr) = (H[l+1],P[l+1])
      else
        pr = -1
      if pr < pl then
        { if p < pl then # Case 1L
          (H[c],P[c],c) = (hl,pl,l)
          else if p > pl then # Case 2L
            break
          else
            { v1 = V[hl]
              px = LCP2(v1,x,pl)
              if v1[px] < x[px] then # Case 3La
                (H[c],P[c],p,c) = (hl,pl,px,l)
              else # Case 3Lb
                { P[l] = px
                  break
                }
            }
          }
        else if pr > pl then
          { # Case 1R, 2R, 3Ra, 3Rb
            ...
          }
        else if p > pl then # Case 2
          break
        else
          { (v1,vr) = (V[hl],V[hr])
            if p < pl then # Case 4
              { px = LCP2(vr,v1,pl)
                if (v1[px] <= vr[px])
                  (H[c],P[c],P[l+1],c) = (hl,pl,px,l)
                else
                  (H[c],P[c],P[l],c) = (hr,pr,px,l+1)
                }
            else # Case 5
              { px = LCP3(v1,vr,x,p)
                if vr[px] > v1[px] then
                  { if x[px] > v1[px] then # Case 5.1L
                    (H[c],P[c],P[l+1],p,c) = (hl,pl,px,px,l)
                    else if x[px] < v1[px] then # Case 5.2L
                      { P[l] = P[l+1] = px
                        break
                      }
                    }
                  else
                    { py = LCP2(v1,x,px)
                      if v1[py] < x[py] then # Case 5.3La
                        (H[c],P[c],P[l+1],p,c) = (hl,pl,px,py,l)
                      else # Case 5.3Lb
                        { (P[l],P[l+1]) = (py,px)
                          break
                        }
                    }
                }
            }
          }
        else if vr[px] < v1[px] then
          { # Case 5.1R, 5.2R, 5.3Ra, 5.3Rb
            ...
          }
        else if x[px] <= v1[py] then # Case 5.2
          { P[l] = P[l+1] = px
            break
          }
        else # Case 5.4
          { py = LCP2(v1,vr,px)
            if v1[py] < vr[py] then
              (H[c],P[c],P[l+1],p,c) = (hl,pl,py,px,l)
            else
              (H[c],P[c],P[l],p,c) = (hr,pr,py,px,l+1)
            }
          }
      }
    }
  (H[c],V[t],P[c]) = (t,x,p)
}

```

```

int LCP2(string x, string y, int n)
{ while true do
  { (a,b) = (x[n],y[n])
    if a != b or a == $ then
      return n
    n += 1
  }
}

int LCP3(string x, string y, string z, int n)
{ while true do
  { (a,b,c) = (x[n],y[n],z[n])
    if a != b or a != c or a == $ then
      return n
    n += 1
  }
}

```

■ **Figure 2** The *Heapify* algorithm for a string heap.

► **Theorem 2.** *The merge algorithm of Figure 1 when using the string heap of Figure 2¹ takes $O(M \log T + X)$ time where $X = |s_1| + \sum_{i=2}^M \text{lcp}(s_{i-1}, s_i)$ and $R^+ = s_1, s_2, \dots, s_M$ is the sequence of M strings extracted from the heap over the course of the list merge, i.e. the output list if duplicates were not removed. Thus it takes $O(M \log T + S)$ worst-case time and $O(M(\log T + \log M))$ expected time in the Uniform Scenario.*

Proof. First, observe that every string value has an *lcp*-value associated with it, namely, for the string $V[H[i]]$ it is $P[i]$ and it represents the number of character comparisons “charged” to its string. Examination of the case conditions reveals that when LCP2 or LCP3 is called, all the string arguments have the same *lcp*-value at the time of the call. Afterwards, all but one of the arguments will have its *lcp*-value increased to the returned value, effectively charging the comparisons of the *lcp* call to those arguments (NB: for LCP3 **two** comparisons per *lcp* increment are made). The total time taken then over the course of the merge is the sum of the maximum *lcp*-value of every string that passes through the heap. Since the *lcp*-value of each string is never more than the length of the string, we have our $O(S)$ bound on the total number of character comparisons.

We can more accurately characterize the number of comparisons with the observation that the maximum *lcp*-value that each string reaches when it is extracted from the root of the heap is its *lcp* with the string value extracted just before it. To see this simply review WLOG the logic involved in a value moving from node 2 to node 1 where, in all relevant cases, $P[1]$ is assigned to $p_l = \text{lcp}(o, v_l)$ where o is the last value extracted as explained previously. Further note that the comparisons for the first element extracted equals its length as its conceptual predecessor is the empty string. So the total number of character comparisons is $|s_1| + \sum_{i=2}^M \text{lcp}(s_{i-1}, s_i)$ over the M string in R^+ . This expression clearly reveals that the time spent comparing strings in a string heap is a function of the consecutive similarity of the strings in the final list, and immediately proves the expected time complexity claim for the Uniform Scenario as the average *lcp* value is $O(\log_\Sigma M)$ in this scenario. ◀

4 The Collision Heap

One might think that when merging sorted string lists that themselves have no duplicates, that there would be in expectation very few duplicates between the lists. This would be correct for the Uniform Scenario. But this is not true, for instance, when the problem is to merge lists of k -mers generated from a shotgun data set. To wit, in a coverage c , say $40\times$, data set, every part of the underlying target sequence/genome has been sampled on average 40 times and so we expect non-erroneous k -mers from unique parts of the target to occur on average 40 times, and a multiple thereof if from repetitive regions. So if one were to partition the data into T equal sized parts, sort the k -mers in each part, and then merge those lists, one quite often sees the same k -mer in different lists. More precisely, the chance that a given k -mer that occurs c times in the data set is not in a given input queue is $(1 - 1/T)^c$, so we expect the k -mer to be in $\bar{e} = T(1 - (1 - 1/T)^c) \approx T(1 - e^{-c/T})$ of the input lists. So if T is say 10, then a non-erroneous, unique k -mer will be found in $\bar{e} = 6.5, 8.8, 9.6,$ or 9.85 of the lists if $c = 10, 20, 30,$ or $40,$ respectively. It was this specific use-case, that we call the “*Shotgun Scenario*”, that motivated the development of a collision heap.

¹ The test $x \neq \text{last}$ is simply replaced by $P[1] < |x|$ as $P[1]$ is the *lcp* of x and the previous extracted element per Theorem 1.

The idea behind a collision heap is also very simple, namely, for each node one also records whether the value at that node is equal to its left child and its right child with a pair of boolean flags in auxiliary (bit) arrays $L[1..T]$ and $R[1..T]$. Formally, $L[i]$ has the value of the predicate $V[H[i]] = V[H[2i]]$ and $R[i]$ has the value of the predicate $V[H[i]] = V[H[2i + 1]]$. Again the interesting and somewhat complex part of this extension is maintaining these values during the induction of *Heapify* and using them to accelerate the handling of duplicate entries.

► **Theorem 3.** *The Heapify routine of Figure 3 correctly maintains the L and R arrays.*

Proof. The inductive invariant for the loop of *Heapify* is basically that all values are correct or *will be correct* once the algorithm is complete, save for $H[c]$, $L[c]$ and $R[c]$ which need to be determined depending on the relative values of x , conceptually at c , and those of its current children. Let $v_l = H[V[2c]]$ and $v_r = H[V[2c + 1]]$ be the strings currently at the left and right children of c . There are 9 cases depending on the relative magnitudes of x , v_l , and v_r , where the three that entail the condition $v_l > v_r$ are treated by symmetry:

Case 1: $v_l < v_r$ and $x > v_l$. By the conditions, v_l will move to node c and the path followed descends to $2c$. $v_l < v_r$ implies that $R[c]$ should be *false*. However, $v_l < x$ does not imply the same for the new value of $L[c]$ as v_l could be equal to the element at $2(2c)$ or $2(2c) + 1$ or both and if so, then those elements are also less than x implying one or the other will replace x at $2c$. Therefore $L[c]$ should be *true* as it will be correct and remain correct after the next loop iteration. So to recapitulate, if $L[2c]$ or $R[2c]$ are true then $L[c]$ should be set to true otherwise it should be false.

Case 2: $v_l < v_r$ and $x = v_l$. By the conditions, the loop will terminate with x finally resting at node c . By the case conditions it is then clear that $L[c]$ is true and $R[c]$ is false.

Case 3: $v_l \leq v_r$ and $x < v_l$. Again a very simple case where the loop stops and clearly $L[c] = R[c] = \textit{false}$.

Case 4: $v_l = v_r$ and $x > v_l$. In this case, v_l moves up to occupy c and x moves down to node $2c$. Clearly $R[c]$ should then be true as $v_l = v_r$. As argued in Case 1, if v_l equals either of its children then the value of $L[c]$ needs to be true as one of these children is smaller than x . Otherwise $L[c]$ should be false.

Case 5: $v_l = v_r$ and $x = v_l$. Then the loop terminates and both $L[c]$ and $R[c]$ are true. ◀

Figure 3 presents the complete algorithm for the collision version of *Heapify* embodying the case analysis above so that the L and R array values are correctly maintained when a heap update occurs. The code is further obviously $O(\log T)$.

The value of the additional L and R flags is that when the top element, say x , is about to be extracted as the current minimum in the heap, one can find all the additional elements equal to x by recursively visiting the children that are marked as equal according to the relevant L and R flags. In Figure 3, the routine *PopHeap* calls the recursive routine *cohort* that makes a post order traversal of the subtree of the heap of all elements equal to x , and places the indices of these nodes in post order in an array G , returning how many of them there are. Thus after calling *PopHeap* the array $G[1..PopHeap()]$ contains the next group of equal elements. The routine clearly takes time proportional to the number of equal elements found. The interesting part is how to replace the cohort in the heap and the time taken to do so, which we treat in the following:

► **Theorem 4.** *The merge algorithm of Figure 3 correctly merges the lists, outputting a unique element in each iteration and takes $O(M \log(T/\bar{e}))$ worst-case time where $\bar{e} = M/N$.*

22:10 Merging String Lists

```

int    H[1..T]
value  V[1..T]
boolean L[1..T]
boolean R[1..T]

static void Heapify(int i, value x, int t)
{ V[t] = x
  c    = i
  while ((l = (2c)) <= T)
    { hl = H[l]
      vl = V[hl]
      if (l >= T)
        vr = INFINITY
      else
        { hr = H[l+1]
          vr = V[hr]
        }
      if (vr > vl)
        { if (x > vl)          # Case 1L
          { H[c] = hl
            L[c] = L[l] or R[l]
            R[c] = false
            c    = l
          }
          else if (x == vl)   # Case 2L
            { H[c] = t
              L[c] = true
              R[c] = false
              return
            }
          else                # Case 3L
            break
        }
      else if (vr < vl)
        { # Cases 1R, 2R, 3R
          . . .
        }
      else
        { if (x > vl)        # Case 4
          { H[c] = hl
            L[c] = L[l] or R[l]
            R[c] = true
            c    = l
          }
          else if (x < vl)   # Case 3
            break
          else                # Case 5
            { H[c] = t
              L[c] = R[c] = true
              return
            }
        }
    }
  H[c] = t
  L[c] = R[c] = false
  return
}

int G[1..T]

int cohort(int c, int len)
{ if (R[c])
  len = cohort(2c+1,len)
  if (L[c])
  len = cohort(2c,len)
  len += 1
  G[len] = c
  return len
}

domain_list S[1..T]

1. for t = T downto 1 do
2.   Heapify(t,Pop(S[t]),t)

3. while (t = H[1]) > 0 do
4.   { output V[t]
5.     len = cohort(1,0)
6.     for k = 1 to len do
7.       { i = G[k]
8.         t = H[i]
9.         if Empty(S[t]) then
10.          Heapify(i,INFINITY,0)
11.         else
12.          Heapify(i,Pop(S[t]),t)
        }
    }
}

```

■ **Figure 3** The *Heapify* (left) and *cohort* (upper right) and top-level merge (lower right) algorithms for a collision heap. In the main algorithm $cohort(1,0)$ identifies all of the equal next elements to be output in Line 5, and then Lines 6-12 carefully replace each of these with its list successor.

Proof. While the flags allow us to easily identify the next cohort of equal elements to extract from the heap, there remains the somewhat more subtle problem of replacing all of them with their list successors. Lines 6-12 of the psuedo-code for the top-level merge at the bottom right of Figure 3 details how this is done. Because the nodes in the cohort G are in post-order, calling *Heapify* on each listed node in that order guarantees a proper heap after all the elements have been replaced. In terms of complexity suppose e nodes are in the cohort for a given iteration of the loop. While the time taken in Lines 6-12 is certainly $O(e \log T)$ we can bound this more tightly by observing that the most time is taken when the e nodes form a complete binary subtree of the heap, that is, every node has the highest height possible. In this case the lowest nodes are at height $\log T - \log e$ and the sum over all e nodes is dominated by this as the sum telescopes (e.g. as for the time analysis for establishing the heap in Lines 1 and 2). Thus the time taken is more accurately $O(e \log(T/e))$. Observe that when $e = T$ the time is $O(e)$ and when $e = 1$ the time taken is $O(\log T)$.

Looking at the overall time to produce the final list R where r_i occurs in e_i of the lists, the total time is $O(\sum_{i=1}^N e_i (\log T - \log e_i))$. By the convexity of the log-function $\sum_i e_i \log_2 e_i \geq N \bar{e} \log_2 \bar{e}$ where $\bar{e} = \sum_i e_i / N$ is the average value of e_i . It thus follows that the total time is $O(\sum_i e_i \log T - N \bar{e} \log \bar{e}) = O(M \log(T/\bar{e}))$. So when $\bar{e} = 1$, i.e. every input element is unique, then the time is $O(M \log T)$ as usual. But this gradually decreases as \bar{e} approaches T where upon the time is $O(M)$. ◀

5 The String Collision Heap

Observing that the idea of a string heap and a collision heap are independent, one can combine the ideas obtaining an $O(M \log(T/\bar{e}) + S)$ time algorithm. Further observe that the L - and R -arrays are not necessarily needed as $L[c]$ is the same as the predicate $V[H[x]][P[x]] = \$$ where $x = 2c$ and $R[c]$ is similarly $V[H[x]][P[x]] = \$$ where $x = 2c + 1$. In words, the string of a child equals the string of its' parent iff the character at its' *lcp*-value is the end of its' string. If one has the length $Len[t]$ of the current string from the t^{th} input list, then the test is simply, $P[x] = Len[H[x]]$ where x is either $2c$ or $2c + 1$.

6 A Trie-Based Priority Queue for Strings

We briefly review trie-based implementations of a string priority queue in order to explain which approach we chose to compare against the modified heap algorithms of this paper. Given a basic Fredkin trie, adding a new string is a matter of following the path from the root of the trie spelling the common prefix with the new string, until its remaining suffix diverges at some node x . A new out edge labelled with the first character of the remaining suffix is added to node x and trie nodes for the suffix are linked in. Finding the minimum string in a trie is simply a matter of following the out edge with the smallest character from each node. To delete this minimum, one finds the last node along the minimum path that has out degree greater than one, and then removes the minimum out edge from this divergent node and the suffix that follows.

If the out edges of each node are realized with a van Emde Boas priority queue for which add and delete are $O(\log \log \Sigma)$ and finding the minimum is $O(1)$ then adding and deleting from the queue are both $O(\log \log \Sigma + s)$ where s is the length of the string being added or deleted. Finding the minimum element is $O(s)$. This gives the $O(M \log \log \Sigma + S)$ bound for the entire merge. If one further realizes a compact trie, wherein all nodes with out degree 1

22:12 Merging String Lists

are collapsed into their successor so that nodes are now labeled with string fragments, the trie is guaranteed to have $O(T)$ nodes and thus the space requirement for the trie is $O(T\Sigma)$ (excluding the space for the strings themselves).

Empirically we found that for typical values of Σ it is actually more efficient to simply realize the out edge PQ with a Σ element array that is directly indexed with a character. In addition, one keeps the current out-degree of the node and the current minimum out-edge. With this information finding the minimum and adding a new string is just $O(s)$. Deletion however does require traversing the out-edge array at the divergent node looking for the new minimum out-edge and so is $O(\Sigma + s)$. Offsetting this is the fact that the number of strings deleted/extracted from the trie is N and not M , so the total complexity for this simple implementation is $O(N\Sigma + S)$ and as will be seen this empirically gives very good performance for the Shotgun Scenario.

■ **Table 1** Performance for the Uniform Scenario.

$M, \Sigma, \bar{l}_{cp}, \bar{e}$	T	Time (in sec.) Heap	Time (in sec.) String Heap	Time (in sec.) Collision Heap	Time (in sec.) Combo' Heap	Time (in sec.) Trie
10M, 4, 10.8, 1.000	4	.654	.506	.597	.542	.984
	8	.848	.633	.851	.673	1.022
	16	1.057	.776	1.130	.810	1.060
	32	1.268	.907	1.407	.950	1.097
	64	1.479	1.053	1.658	1.090	1.144
	128	1.650	1.183	1.914	1.230	1.249
	256	1.897	1.323	2.201	1.373	1.331
100M, 4, 12.5, 1.000	4	7.04	5.21	6.29	5.54	9.99
	8	8.99	6.47	8.96	6.85	10.81
	16	11.18	7.88	11.84	8.12	11.10
	32	13.57	9.15	14.75	9.60	11.50
	64	16.16	10.58	17.86	10.90	12.01
	128	18.69	12.05	21.18	12.34	13.09
	256	21.63	13.64	24.53	14.00	13.77
1000M, 4, 14.1, 1.000	4	72.9	52.5	66.0	55.7	99.7
	8	94.7	65.5	95.2	68.9	107.9
	16	115.9	77.7	121.9	81.0	111.6
	32	140.2	91.1	154.6	95.6	116.2
	64	168.5	106.6	187.4	111.2	123.4
	128	195.5	121.1	221.8	125.7	132.2
	256	229.5	139.2	265.3	142.9	143.2
1000M, 8, 9.3, 1.000	4	61.4	51.0	57.2	53.8	89.7
	8	77.7	64.3	81.1	67.3	93.6
	16	94.2	76.7	103.0	80.0	95.7
	32	108.5	90.5	126.6	94.1	98.3
	64	126.4	105.4	149.3	110.2	102.4
	128	145.9	118.9	174.1	123.9	110.2
	256	168.5	133.5	202.4	137.8	113.9
1000M, 16, 6.8, 1.000	4	56.1	50.0	53.4	50.8	85.2
	8	69.2	60.2	71.1	63.9	85.0
	16	81.4	74.1	92.0	75.2	87.7
	32	95.1	88.1	110.5	89.2	88.5
	64	109.6	99.7	132.7	103.9	91.3
	128	126.7	113.6	157.0	119.1	100.2
	256	150.0	129.3	181.0	134.6	101.7

7 Empirical Performance

We implemented string list merging programs using a regular heap, **Heap**, a string heap, **Sheap**, a collision heap, **Cheap**, a string-collision heap, **SCheap**, and a simple, compact trie, **Trie**, and measured their performance on a 2019 Mac Pro with a 2.3 GHz Intel Core i9 processor, 64GB of memory, and 8TB of SSD disk. All the codes are available at GitHub at the url github.com/thegenemyers/STRING.HEAP.

In the first set of timing experiments over synthetic data, for a given setting of parameters M , T , and Σ we generated T input files, each with M 20-mers where every 20-mer over a Σ character ASCII alphabet occurs with equally likelihood, that is, the Uniform Scenario introduced in the introduction. We chose 20 as the k -mer size as it is greater than the lcp seen in any of the experiments. For such data we expect the lcp between successive elements in the output list to be on average $\log_{\Sigma} M$ and \bar{e} to be 1 given that the average lcp is less than 20 for all trials considered. In Table 1, we present timings where Σ was set to 4, T was set from 4 to 256 in steps of 2x, and M was set to 10^x for $x = 7, 8, \text{ and } 9$. In addition, for $M = 10^9$, we also generated data sets where Σ was also set to 8 and 16 to see the dependence of the programs, especially *Trie*, on Σ .

■ **Table 2** Performance for the Shotgun Scenario.

T	c	\bar{lcp}	\bar{e}	Time (in sec.) Heap	Time (in sec.) String Heap	Time (in sec.) Collision Heap	Time (in sec.) Combo' Heap	Time (in sec.) Trie
4	0	14.1	1.0	70.6	54.2	65.8	58.0	98.4
	1	15.7	1.4	73.7	53.6	59.8	56.3	84.6
	2	16.8	2.0	69.7	50.1	49.9	50.7	66.5
	4	17.7	2.8	63.2	42.6	38.6	43.3	48.0
	6	18.0	3.3	60.2	38.4	34.0	38.7	41.0
	8	18.1	3.6	58.5	35.8	31.7	35.1	36.5
	12	18.2	3.9	57.0	34.0	29.8	32.8	33.9
8	0	14.1	1.0	95.0	66.4	96.7	70.1	113.5
	2	17.0	2.1	93.6	64.7	79.1	65.8	79.8
	4	18.1	3.5	88.4	57.8	60.9	56.0	61.1
	8	18.7	5.3	82.5	48.1	45.9	46.9	45.9
	12	18.9	6.4	79.1	42.5	38.8	40.6	39.6
	16	19.0	7.1	77.9	39.3	34.9	37.0	35.9
	24	19.0	7.7	76.9	35.4	31.7	33.2	33.5
16	0	14.1	1.0	117.9	79.3	124.6	82.8	113.7
	4	18.2	3.9	109.8	73.0	85.6	70.7	63.0
	8	18.9	6.6	101.3	63.1	65.1	59.3	50.3
	16	19.3	10.4	97.3	54.3	48.8	47.5	40.1
	24	19.4	12.6	94.9	48.1	41.7	40.9	34.9
	32	19.4	14.0	93.8	44.0	37.9	36.8	33.2
	48	19.5	15.3	91.8	39.1	34.1	33.5	30.6

The timings confirm that all algorithms are linear in M and that the heap algorithms are linear in $\log T$. As M becomes larger or Σ becomes smaller the average lcp between consecutive strings increases and so as expected the string heap becomes progressively faster than a regular heap. The combination heap tracks the performance of the string heap but lags by about 5% for all parameter values due to the additional overhead of maintaining information about collisions, which in these experiments basically do not occur. The trie's behavior is basically constant, edging up slightly with T due to an increase in the branching layers in the prefix of the trie. Counter intuitively, the trie becomes faster with larger Σ as this reduces the expected number of branching layers in the trie which dominates the minor cost of searching for the smallest out-edge of a single node when deleting an entry. Thus, ultimately as T increases the trie becomes the fastest, at 256 for $\Sigma = 4$, 64 for $\Sigma = 8$, and 32 for $\Sigma = 16$.

In the second set of timing experiments, we produced synthetic data sets of k -mers where they followed the Shotgun Scenario. We fixed M at 10^9 , k at 20, and then for each of $T = 4, 8, \text{ and } 16$, we varied the coverage c such that $c/T = .25, .5, 1.0, 1.5, 2.0, \text{ and } 3.0$.

As the number of collisions increases, the collision heap overtakes the string heap, with again the combination heap tracking the better of the two with an overhead of 5% or so. But our trie implementation does become faster for larger values of T due to its $O(N\Sigma + S)$ complexity. The collision heap, in terms of N , has complexity $O(N\bar{e} \log(T/\bar{e}))$ which explains the behavior. Basically, the number of elements in the trie decreases rapidly from T toward 1 as collisions occur greatly accelerating its operation. Nonetheless, the table reveals that for smaller values of T the heap algorithms are generally superior.

The final set of experiments were for k -mers from a real shotgun sequencing data set, the motivating example for this work. For high-accuracy read data sets k , is typically chosen at 40 or more, as k -mers of that size are well conserved. The other difference with the synthetic Shotgun Scenario is that the k -mers occur with a complex frequency profile wherein some k -mers occur with frequency about C , but for example, 80% of the k -mers contain errors and occur once, others from a haplotype region occur roughly $C/2$ times, and so on. So in Table 3 below one will see that \bar{e} is significantly less than for the synthetic examples, yet still substantially elevated. Interestingly in these cases the collision heap or combined heap perform best because they respond continuously to the collisions, beating out the string heap, and the lcp is very near k , thus beating out the trie.

■ **Table 3** Performance on real sequencing data with $k = 40$.

M	T	c	\bar{lcp}	\bar{e}	Time (in sec.) Heap	Time (in sec.) String Heap	Time (in sec.) Collision Heap	Time (in sec.) Combo' Heap	Time (in sec.) Trie
333M	4	4	28.3	2.1	33.2	23.1	22.4	24.8	29.1
666M	8	8	32.9	3.4	85.7	56.1	55.0	55.0	60.5
1333M	16	16	35.4	4.9	210.2	123.3	118.5	114.7	121.8
484M	4	8	30.3	2.5	50.7	32.5	29.9	33.7	38.3
968M	8	16	33.6	3.6	127.4	74.9	71.3	73.6	84.0
1936M	16	32	35.4	4.6	314.9	175.0	164.7	159.2	183.4
566M	4	12	30.4	2.4	59.6	37.6	35.2	39.2	44.0
1232M	8	24	33.3	3.3	149.9	87.1	84.3	86.3	98.5
2464M	16	48	34.8	4.0	382.2	212.0	205.3	194.7	230.4

In summary, the string heap performs best when the average lcp value increases, and the collision ratio is low. The collision heap always performs better than the string heap when collisions become high. The combination heap tracks the better of the two combined methods, lagging by about 5%. The trie data structure is generally the best for large T or pure collision scenarios, but on real high-fidelity shotgun data sets the collision and combination heaps proved superior.

References


- 1 A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Managing unbounded-length keys in comparison-driven data structures with application to online indexing. *SIAM J. on Computing*, 43:1396–1416, 2014.
- 2 T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.
- 3 E. Fredkin. Trie memory. *Comm. of the ACM*, 3:490–499, 1960.
- 4 Donald E. Knuth. *The Art of Computer Programming Vol. 3*. Addison Wesley, 1998.
- 5 M. Kokot, M. Dlugosz, and S. Deorowicz. Kmc3: Counting and manipulating k -mer statistics. *Bioinformatics*, 33:2759–2761, 2017.

- 6 G. Marcais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k -mers. *Bioinformatics*, 27:764–770, 2011.
- 7 Eugene Myers. <https://github.com/thejenemyers/FASTK>, 2020.
- 8 Arang Rhie, Shane McCarthy, Olivier Frederigo, (others), Eugene Myers, Richard Durbin, Adam Phillippy, and Erich Jarvis. Towards complete and error-free genome assemblies of all vertebrate species. *Nature*, 592:737–746, 2021.
- 9 M. Thorup. On ram priority queues. *SIAM J. on Computing*, 30:86–109, 2000.
- 10 P. van Emde Boas, R. Kaas, , and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematics Systems Theory*, 10:99–127, 1977.

PalFM-Index: FM-Index for Palindrome Pattern Matching

Shinya Nagashita ✉

Kyushu Institute of Technology, Fukuoka, Japan

Tomohiro I ✉ 

Kyushu Institute of Technology, Fukuoka, Japan

Abstract

The palindrome pattern matching (pal-matching) is a kind of generalized pattern matching, in which two strings x and y of same length are considered to match (pal-match) if they have the same palindromic structures, i.e., for any possible $1 \leq i < j \leq |x| = |y|$, $x[i..j]$ is a palindrome if and only if $y[i..j]$ is a palindrome. The pal-matching problem is the problem of searching for, in a text, the occurrences of the substrings that pal-match with a pattern. Given a text T of length n over an alphabet of size σ , an index for pal-matching is to support, given a pattern P of length m , the counting queries that compute the number occ of occurrences of P and the locating queries that compute the occurrences of P . The authors in [I et al., Theor. Comput. Sci., 2013] proposed an $O(n \lg n)$ -bit data structure to support the counting queries in $O(m \lg \sigma)$ time and the locating queries in $O(m \lg \sigma + \text{occ})$ time. In this paper, we propose an FM-index type index for the pal-matching problem, which we call the PalFM-index, that occupies $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space and supports the counting queries in $O(m)$ time. The PalFM-indexes can support the locating queries in $O(m + \Delta \text{occ})$ time by adding $\frac{n}{\Delta} \lg n + n + o(n)$ bits of space, where Δ is a parameter chosen from $\{1, 2, \dots, n\}$ in the preprocessing phase.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Palindrome matching, Generalized string pattern matching, Indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.23

Related Version *Previous Version:* <https://arxiv.org/abs/2206.12600>

Funding This work was supported by JSPS KAKENHI (Grant Number 19K20213).

Tomohiro I: KAKENHI (Grant Numbers 19K20213).

1 Introduction

A palindrome is a string that can be read same backward as forward. Palindromic structures in a string are one of the most fundamental structures in the string and have been extensively studied. For example, it is known that any string w contains at most $|w| + 1$ distinct palindromic substrings [6], and the strings reaching the maximum values have some intriguing properties [15, 28]. Another concept regarding palindromic structures is the palindrome complexity [1, 4, 2], which is the number of distinct palindromic substrings of a given length in a string.

Instead of thinking about distinct palindromic substrings, one might be interested in occurrences of palindromic substrings. The palindromic structures in such a sense are captured by the maximal palindromes from all possible “centers” in a string. Manacher’s algorithm [26], originally proposed for computing a prefix-palindrome, can be extended to compute all the maximal palindromes in $O(|w|)$ time for a string w . The authors in [18] considered the problem of inferring strings from a given set of maximal palindromes and showed that the problem can be solved in $O(|w|)$ time.



© Shinya Nagashita and Tomohiro I;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 23; pp. 23:1–23:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In [19], a new concept called *palindrome pattern matching* was introduced as a generalized pattern matching. Two strings x and y of the same length are said to *palindrome pattern match* (*pal-match* in short) iff they have the same palindromic structures, i.e., the following condition holds: for any possible $1 \leq i < j \leq |x| = |y|$, $x[i..j]$ is a palindrome iff $y[i..j]$ is a palindrome. We remark that x and y themselves are not necessarily palindromes. The palindrome pattern matching has potential applications to genomic analysis, in which some palindromic structures play an important role to estimate RNA secondary structures [21].

The pal-matching problem is to search for, in a text, the occurrences of the substrings that pal-match with a pattern. Given a text T of length n and a pattern P of length m , a Morris-Pratt type algorithm for solving the pal-matching problem in $O(n)$ time was proposed in [19]. The method in [19] is based on the *lpal*-encoding of a string w , denoted as lpal_w , that is the integer array of length $|w|$ such that $\text{lpal}_w[i]$ is the length of the longest suffix palindrome of $w[1..i]$. The *lpal*-encoding is helpful because two strings x and y pal-match iff $\text{lpal}_x = \text{lpal}_y$. When T is large and static, and patterns come online later, one might think of preprocessing T to construct an index for pal-matching. An index for pal-matching is to support the counting queries that compute the number occ of occurrences of P and the locating queries that compute the occurrences of P . For this purpose, I et al. [19] proposed the *palindrome suffix tree* of T , which is a compacted tree of the *lpal*-encoded suffixes of T . The palindrome suffix tree takes $O(n \lg n)$ bits of space and supports the counting queries in $O(m \lg \sigma)$ time and the locating queries in $O(m \lg \sigma + \text{occ})$ time, where σ is the size of the alphabet from which characters in T are taken and occ is the number of occurrences.

In this paper, we present a new index, named the *PalFM-index*, by applying the technique of the FM-index [7] to the pal-matching problem. In so doing we introduce a new encoding, named the *ssp*-encoding, that is based on the non-trivial shortest suffix-palindrome of each prefix. In contrast to the *lpal*-encoding, the *ssp*-encoding has a good property to design the PalFM-index. The PalFM-index occupies $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space and supports the counting queries in $O(m)$ time. The locating queries can be supported in $O(m + \Delta \text{occ})$ time by adding $\frac{n}{\Delta} \lg n + n + o(n)$ bits of space, where Δ is a parameter chosen from $\{1, 2, \dots, n\}$ in the preprocessing phase.

1.1 Related work

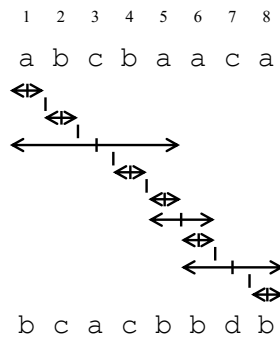
One of the well-studied algorithmic problems related to palindromes is factorizing a string into non-empty palindromes, or in other words, recognizing a string that is obtained by concatenating a certain number of non-empty palindromes [26, 24, 12, 9, 20, 25, 3, 29]. The combinatorial properties discovered during tackling this factorization problem are useful to work on palindromes-related problems.

Developing techniques of designing space-efficient indexes for generalized pattern matching is of great interest. Our PalFM-index was inspired by that of Kim and Cho [23], which is a simplified version of the FM-index for parameterized pattern matching [13]. Indexes based on the FM-index for other generalized pattern matching problems were considered in [14, 11, 22].

2 Preliminaries

2.1 Notations

An integer interval $\{i, i + 1, \dots, j\}$ is denoted by $[i..j]$, where $[i..j]$ represents the empty interval if $i > j$.



■ **Figure 1** Illustration of the palindromic structures for pal-matching strings `abcbaaca` and `bcacbbdb`. Check that the radii of their maximal palindromes for all possible centers, which are illustrated by two-headed arrows, coincide.

Let Σ be a finite *alphabet*, a set of characters. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. The concatenated string of two strings x and y are denoted as $x \cdot y$ or simply xy . The i -th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the *substring* of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$, i.e. $w[i..j] = w[i]w[i+1] \dots w[j]$. For convenience, let $w[i..j] = \varepsilon$ if $i > j$. A substring of the form $w[1..j]$ (resp. $w[i..|w|]$) is called a *prefix* (resp. *suffix*) of w and denoted as $w[..j]$ (resp. $w[i..]$) in shorthand. Note that ε is a substring/prefix/suffix of any string w . A substring of w is called *proper* if it is not w itself. When needed we use parentheses to indicate positions in a concatenated string, for example, $(xy)[i]$ refers to the i -th character of the string xy . Hence, $(xy)[i]$ should be distinguished from $xy[i]$, which can be interpreted as the concatenated string of x and $y[i]$.

Let \prec denote the total order over an alphabet we consider. In particular, we will consider strings over a set consisting of integers and ∞ , in which natural total order based on their values is employed. We extend \prec to denote the lexicographic order of strings over the alphabet. For any strings x and y that do not match, we say that x is lexicographically smaller than y and denote it by $x \prec y$ iff $x[i+1] \prec y[i+1]$ for largest integer i with $x[..i] = y[..i]$, where we assume that $x[i+1]$ or $y[i+1]$ refers to the lexicographically smallest character $\$$ if it points to out of bounds.

For any string w , let w^R denote the reversed string of w , that is, $w^R = w[|w|] \dots w[2]w[1]$. A string w is called a *palindrome* if $w = w^R$. The *radius* of a palindrome w is $\frac{|w|}{2}$. The *center* of a palindromic substring $w[i..j]$ of a string w is $\frac{i+j}{2}$. A palindromic substring $w[i..j]$ is called the *maximal palindrome* at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $w[i..j]$, i.e., if $w[i-1] \neq w[j+1]$, $i = 1$, or $j = |w|$.

Two strings x and y of same length are said to *palindrome pattern match* (*pal-match* in short) iff they have the same palindromic structures, i.e., the following condition holds: for any possible $1 \leq i < j \leq |x| = |y|$, $x[i..j]$ is a palindrome iff $y[i..j]$ is a palindrome. For example, `abcbaaca` and `bcacbbdb` pal-match since their palindromic structures coincide (see Figure 1). Note that pal-matching induces a substring consistent equivalent relation [27], i.e., if x and y pal-match then $x[i..j]$ and $y[i..j]$ pal-match for any possible $1 \leq i < j \leq |x| = |y|$.

The pal-matching problem is to search for, in a text string T , the occurrences of the substrings that pal-match with a pattern P . In the pal-matching problem, an occurrence of P refers to a position i such that $T[i..i+|P|-1]$ and P pal-match. Throughout this paper we consider indexing a text T of length n over an alphabet Σ of size σ .

2.2 Toolbox

As a component of our PalFM-index, we use a data structure for a string w over an integer alphabet U supporting the following queries.

- $\text{rank}_w(i, c)$: return the number of occurrences of character $c \in U$ in $w[1..i]$.
- $\text{select}_w(i, c)$: return the i -th smallest position of the occurrences of character $c \in U$ in w .
- $\text{rangeCount}_w(i, j, c, d)$: return the number of the occurrences of any character in $[c..d] \subseteq U$ in $w[i..j]$.

The Wavelet tree [17] supports these queries in $O(\lg |\Sigma|)$ time using $|w|\mathcal{H}_0(w) + o(|w| \lg |U|)$ bits of space, where $\mathcal{H}_0(w) = O(\lg |U|)$ is the 0-th order empirical entropy of w . The subsequent studies [8, 16] improved the complexities, resulting in the following theorem.

► **Theorem 1** ([16]). *For a string w over an integer alphabet U , there is a data structure in $|w|\mathcal{H}_0(w) + o(|w|)$ bits of space that supports rank , select and rangeCount in $O(1 + \frac{\lg |U|}{\lg |w|})$ time.*

We also use a data structure for the *Range Maximum Queries (RMQs)* over an integer array V . Given an interval $[i..j]$ over V , a query $\text{RMQ}_V(i, j)$ returns a position in $[i..j]$ that has the maximum value in $V[i..j]$, that is, $\text{RMQ}_V(i, j) = \arg \max_{k \in [i..j]} V[k]$. We use the following result.

► **Theorem 2** ([10]). *For an integer array V of length n , there is a data structure with $2n + o(n)$ bits of space that supports the RMQs in $O(1)$ time.*

2.3 FM-index

The suffix array SA of T is the integer array of length $n + 1$ such that $\text{SA}[i]$ is the starting position of the lexicographically i -th suffix of T .¹ We define the string L (a.k.a. the *Burrows-Wheeler Transform (BWT)* [5] of T) of length $n + 1$ as follows:

$$\text{L}[i] = \begin{cases} \$ & (\text{SA}[i] = 1), \\ T[\text{SA}[i] - 1] & (\text{SA}[i] > 1). \end{cases}$$

We define the string F of length $n + 1$ as $\text{F} = T[\text{SA}[1]]T[\text{SA}[2]] \cdots T[\text{SA}[n + 1]]$. The so-called *LF-mapping* LF is the function defined to map a position i to j such that $\text{SA}[j] = \text{SA}[i] - 1$ (with the corner case $\text{LF}(i) = 1$ for $\text{SA}[i] = 1$). A crucial point is that LF -mapping can be efficiently implemented by rank queries on L and select queries on F with $\text{LF}(i) = \text{select}_{\text{F}}(\text{rank}_{\text{L}}(i, \text{L}[i]), \text{L}[i])$.² The occurrences of pattern P in T can be answered by finding the maximal interval $[P_b..P_e]$ in the SA array such that $T[\text{SA}[i]..]$ is prefixed by P if $i \in [P_b..P_e]$, and computing the SA -values in the interval. For a string w and character c , the so-called *backward search* computes the maximal interval in the SA prefixed by cw from that of w using a similar mechanism of the LF -mapping (see [7] for more details).

¹ Against convention, we include the empty string that starts with the position $n + 1$ to SA . In particular, $\text{SA}[1] = n + 1$ holds as the empty string is always the smallest suffix.

² In the plain LF -mapping, select queries on F can be implemented by a simple table that counts, for each character c , the number of occurrences of characters smaller than c in T , but it is not the case in our generalized LF -mapping for pal-matching.

■ **Table 1** A comparison between lpal and ssp for $w = \text{abbbabb}$ and $w' = \text{bw} = \text{babbbabb}$. The values that change when prepending b to w are underlined.

$w =$		a	b	b	b	a	b	b
$\text{lpal}_w =$		1	1	2	3	5	3	5
$\text{ssp}_w =$		∞	∞	2	2	5	3	2
$w' =$	b	a	b	b	b	a	b	b
$\text{lpal}_{w'} =$	1	1	<u>3</u>	2	3	5	<u>7</u>	5
$\text{ssp}_{w'} =$	∞	∞	<u>3</u>	2	2	5	3	2

3 Encodings for pal-matching

The pal-matching algorithms in [19] are based on the lpal -encoding of a string w , denoted as lpal_w . lpal_w is the integer array of length $|w|$ such that, for any position $1 \leq i \leq |w|$, $\text{lpal}_w[i]$ is the length of the longest suffix-palindrome of $w[1..i]$. See Table 1 for example.

► **Lemma 3** (Lemma 2 in [19]). *For any strings x and y , x and y pal-match iff $\text{lpal}_x = \text{lpal}_y$.*

Although Lemma 3 is sufficient to design suffix-tree type indexes, it seems that the lpal -encoding is not suitable to design FM-index type indexes. For example, more than one position could change when a character is prepended (see Table 1) and this unstable property makes messes up lexicographic order of lpal -encoded suffixes, which prevents us to implement LF-mapping space efficiently.

In this paper, we introduce a new encoding suitable to design FM-index type indexes for pal-matching. Our new encoding is based on the shortest suffix-palindrome for each prefix, where the shortest suffix is chosen excluding the trivial palindromes of length ≤ 1 . We call the encoding the *shortest suffix-palindrome encoding* (the ssp -encoding in short). For any string w , the ssp -encoding ssp_w of w is the integer array of length $|w|$ such that, for any position $1 \leq i \leq |w|$, $\text{ssp}_w[i]$ is the length of the non-trivial shortest suffix-palindrome of $w[1..i]$ if such exists, and otherwise ∞ . See Table 1 for example.

► **Lemma 4.** *Two strings x and y pal-match iff $\text{ssp}_x = \text{ssp}_y$.*

Proof. Since the ssp -encoding relies only on palindromic structures, the direction from left to right is clear.

In what follows, we focus on the opposite direction; x and y pal-match if $\text{ssp}_x = \text{ssp}_y$. Assume for contrary that x and y does not pal-match. Without loss of generality, we can assume that there are positions i and j such that $x[i..j]$ is a palindrome but $y[i..j]$ is not, with smallest j if there are many. Note that the smallest assumption on j implies that $y[i+1..j-1]$ is a palindrome: If $y[i+1..j-1]$ is not a palindrome (clearly $|y[i+1..j-1]| > 1$ in such a case), $j-1$ must be a smaller position that satisfies the above condition because $x[i+1..j-1]$ is a palindrome. Let $k = \text{ssp}_x[j] = \text{ssp}_y[j]$. Since $x[i..j]$ is a palindrome, it holds that $1 < k \leq |x[i..j]|$. Moreover, $k \neq |y[i..j]|$ as $y[i..j]$ is not a palindrome. Since the palindrome $x[i..j]$ has a suffix-palindrome of length k , the prefix $x[i..i+k-1]$ of length k is a palindrome, too. On the other hand, since $y[i..j]$ is not a palindrome that has a suffix-palindrome of length k , the prefix $y[i..i+k-1]$ of length k cannot be a palindrome. This contradicts the smallest assumption on j because $i+k-1$ is a smaller position such that $x[i..i+k-1]$ and $y[i..i+k-1]$ disagree on their palindromic structures. ◀

In contrast to the lpal -encoding, the ssp -encoding has a stable property when prepending a character.

► **Lemma 5.** *For any string w and character c , there is at most one position i ($1 \leq i \leq |w|$) such that $\text{ssp}_w[i] \neq \text{ssp}_{cw}[i+1]$. Moreover, if such a position i exists, $\text{ssp}_w[i] = \infty$ and $\text{ssp}_{cw}[i+1] = i+1$.*

Proof. By definition it is obvious that $\text{ssp}_w[i] = \text{ssp}_{cw}[i+1]$ if $\text{ssp}_w[i] \neq \infty$. In what follows, we assume for contrary that there exist two positions i and i' with $1 \leq i < i' \leq |w|$ such that $\text{ssp}_w[i] = \infty > \text{ssp}_{cw}[i+1]$ and $\text{ssp}_w[i'] = \infty > \text{ssp}_{cw}[i'+1]$. Note that $\text{ssp}_{cw}[i+1] = i+1$ and $\text{ssp}_{cw}[i'+1] = i'+1$ by definition, and $(cw)[..i+1]$ and $(cw)[..i'+1]$ are palindromes. Since $(cw)[..i+1]$ is a prefix-palindrome of $(cw)[..i'+1]$, it is also a suffix-palindrome of $(cw)[..i'+1]$. It contradicts that $(cw)[..i'+1]$ is the non-trivial shortest suffix-palindrome of $(cw)[..i'+1]$. ◀

We consider yet another encoding based on the shortest suffix of $w[..i-1]$ that is extended outwards when appending a character $w[i]$. The concept is closely related to the **ssp**-encoding because the extended palindrome is the non-trivial shortest suffix-palindrome of $w[..i]$. An advantage of this new encoding is that we can reduce the number of distinct integers to be used to $O(\min(\sigma, \lg |w|))$, which will be used (in a symmetric way) to define \mathbb{L}_{pal} and obtain a space-efficient FM-index specialized for pal-matching.

For any string w we partition the suffix-palindromes (including the empty suffix) by the characters they have immediately to their left and call each group a *suffix-pal-group* for w . We utilize the following lemma.

► **Lemma 6.** *For any string w , the number of suffix-pal-groups for w is $O(\min(\sigma, \lg |w|))$.*

Proof. It is obvious that the number of suffix-pal-groups is at most σ because each character is associated to at most one suffix-pal-group. Also it is known that the lengths of the suffix-palindromes can be represented by $O(\lg |w|)$ arithmetic progressions and each arithmetic progression induces a period in the involved suffix (e.g., see [20]). Then we can see that every suffix-palindrome represented by an arithmetic progression is in the same group. Hence there are $O(\lg |w|)$ groups. ◀

The next lemma shows that pal-matching strings share the same structure of suffix-pal-groups.

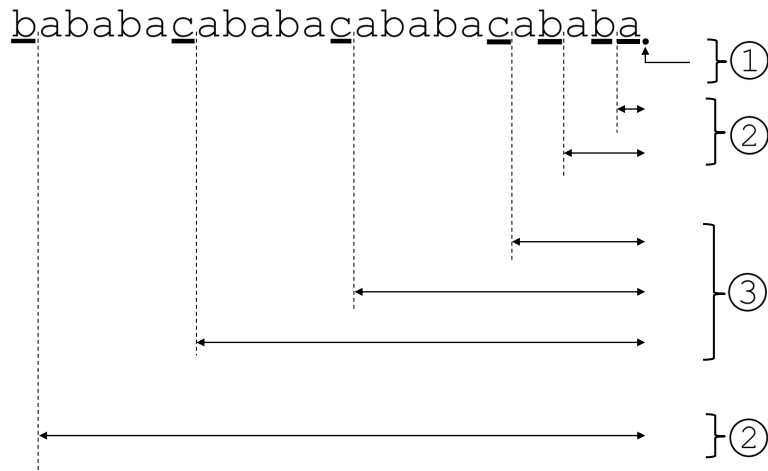
► **Lemma 7.** *Let x and y be strings that pal-match and let i and j be integers with $1 \leq i < j \leq |x| = |y|$. If $x[i+1..]$ and $x[j+1..]$ are palindromes with $x[i] = x[j]$, then $y[i+1..]$ and $y[j+1..]$ are palindromes with $y[i] = y[j]$.*

Proof. Since the palindrome $x[i+1..]$ has a suffix-palindrome of length $k = |x[j+1..]|$, it also has a prefix-palindrome of length k , that is, $x[i+1..i+k]$ is a palindrome. Also, $x[i+k+1] = x[j]$ holds. Since $x[i] = x[j] = x[i+k+1]$, $x[i..i+k+1]$ is a palindrome.

Since x and y pal-match, $y[i+1..]$, $y[j+1..]$ and $y[i..i+k+1]$ are palindromes. By transition of equivalence induced by the palindromes $y[i..i+k+1]$ and $y[i+1..]$, we can see that $y[i] = y[i+k+1] = y[j]$. Thus the claim holds. ◀

Let the shortest palindrome in a suffix-pal-group be the representative of the group. We assign consecutive integer identifiers starting from 1 to the suffix-pal-groups in increasing order of their representative's lengths. See Figure 2 for example.

For any string w , we define the *shortest suffix-pal-group encoding* ssp_{g}_w of w as the integer array of length $|w|$ such that, for any position $1 \leq i \leq |w|$, $\text{ssp}_{\text{g}}_w[i]$ is the identifier assigned to the suffix-pal-group of the suffix-palindrome in $w[..i-1]$ that is extended outwards by appending $w[i]$, if such exists, and otherwise ∞ . See Table 2 and Figure 3 for example. Since



■ **Figure 2** An example of suffix-pal-groups for `bababababacababacababacababa`. The number enclosed in a circle denotes the pal-group-id. The suffix-palindromes in the suffix-pal-group with identifier 1 (resp. 2 and 3) have `a` (resp. `b` and `c`) immediately to their left. The identifiers are given in increasing order of their representative’s lengths, that is, $|\varepsilon| = 0$, $|\mathbf{a}| = 1$ and $|\mathbf{ababa}| = 5$.

the non-trivial shortest suffix of $w[..i]$ is extended outwards from the representative of the suffix-pal-group for $w[1..i - 1]$ that has $w[i]$ immediately to the left, $\text{ssp}_w[i]$ has essentially equivalent information to $\text{ssp}_w[i]$. Formally the next lemma holds.

► **Lemma 8.** *For any string x of length k , suppose we have the set of lengths of the representatives of suffix-pal-groups of $x[..k - 1]$. Given $\text{ssp}_x[k]$ we can identify $\text{ssp}_x[k]$, and vice versa.*

Proof. It is clear that $\text{ssp}_x[k] = \infty$ iff $\text{ssp}_x[k] = \infty$. Given $\text{ssp}_x[k] \neq \infty$ we can identify $\text{ssp}_x[k]$ from the representative of the suffix-pal-group with identifier $\text{ssp}_x[k]$. Given $\text{ssp}_x[k] \neq \infty$ we can identify $\text{ssp}_x[k]$ from the representative that has length $\text{ssp}_x[k] - 2$. ◀

The next lemma shows that the ssp_g -encoding is another encoding for pal-matching, and induces the same lexicographic order with the ssp -encoding.

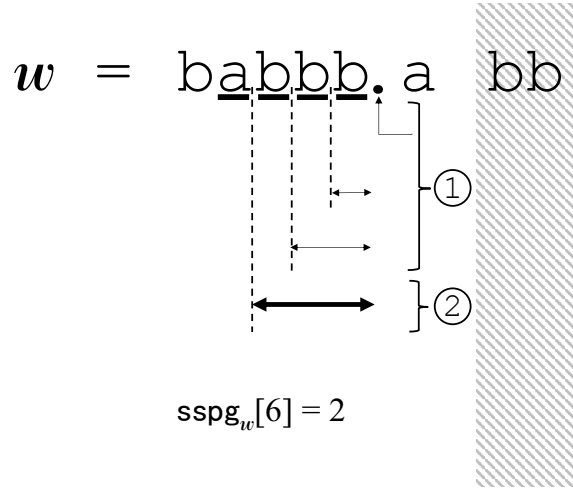
► **Lemma 9.** *Let x and y be strings of length k such that $\text{ssp}_x[..k - 1] = \text{ssp}_y[..k - 1]$. Then, $\text{ssp}_x[k] = \text{ssp}_y[k]$ iff $\text{ssp}_x[k] = \text{ssp}_y[k]$. Also, $\text{ssp}_x[k] < \text{ssp}_y[k]$ iff $\text{ssp}_x[k] < \text{ssp}_y[k]$.*

Proof. It follows from Lemma 7 that $x[..k - 1]$ and $y[..k - 1]$ have the same structure of suffix-pal-groups. By Lemma 8, $\text{ssp}_x[k] = \text{ssp}_y[k]$ if $\text{ssp}_x[k] = \text{ssp}_y[k]$, and vice versa. Since the identifiers of suffix-pal-groups are given in increasing order of their representative’s lengths, it holds that $\text{ssp}_x[k] < \text{ssp}_y[k]$ if and only if $\text{ssp}_x[k] < \text{ssp}_y[k]$. ◀

For any string w , let $\pi(w) = \text{ssp}_{w^R}[|w|]$. Intuitively, $\pi(w)$ holds the information from which prefix-palindrome of $w[2..]$ the non-trivial shortest prefix-palindrome of w is extended, and the information is encoded with the identifier defined in the completely symmetric way as the case of the suffix-pal-groups. The function $\pi(\cdot)$ will be applied to the suffixes of T to define F_{pal} and L_{pal} , and the next lemma is a key to implement LF-mapping for our PalFM-index.

■ **Table 2** A comparison between ssp_w and sspg_w for $w = \text{babbbabb}$. $\text{ssp}_w[6] = 5$ because the non-trivial shortest suffix-palindrome of $w[1..6] = \text{babbbba}$ is abbba , which is of length 5. On the other hand, $\text{sspg}_w[6] = 2$ because the shortest suffix-palindrome abbba ending at 6 is extended from bbb and the suffix-pal-group to which bbb belongs for $w[1..5] = \text{babbb}$ has the identifier 2.

$w =$	b	a	b	b	b	a	b	b
$\text{ssp}_w =$	∞	∞	3	2	2	5	3	2
$\text{sspg}_w =$	∞	∞	2	1	1	2	2	2



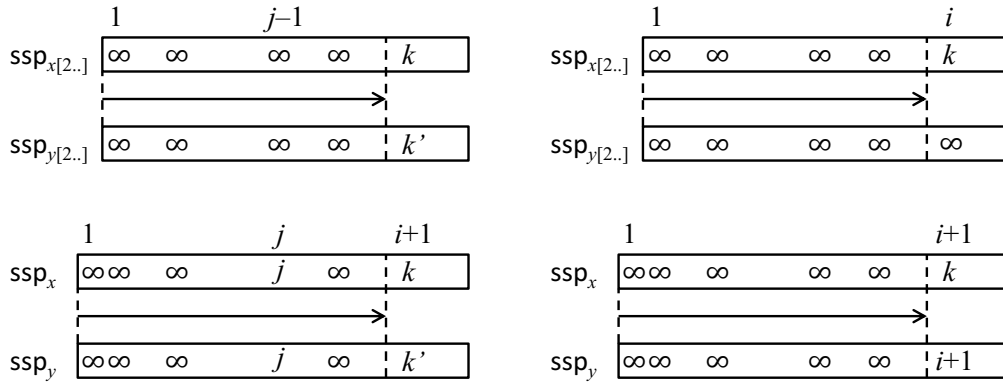
■ **Figure 3** Illustration to show $\text{sspg}_w[6] = 2$ for $w = \text{babbbabb}$.

► **Lemma 10.** Let x and y be strings of length ≥ 1 such that $\pi(x) = \pi(y)$. Then, $\text{ssp}_x \prec \text{ssp}_y$ iff $\text{ssp}_{x[2..]} \prec \text{ssp}_{y[2..]}$.

Proof. Let i be the largest integer such that $x[2..i]$ and $y[2..i]$ pal-match. Since $\pi(x) = \pi(y)$, using Lemma 9 in a symmetric way, it holds that $x[1..i]$ and $y[1..i]$ pal-match. Recall Lemma 5 that at most one ∞ in $\text{ssp}_{x[2..]}$ (resp. $\text{ssp}_{y[2..]}$) turns into the largest possible integer at the changed position when prepending $x[1]$ (resp. $y[1]$). We analyze the cases focusing on the changed positions:

1. The claim clearly holds if neither ssp_x nor ssp_y has the changed position less than or equal to $i + 1$.
2. If both of ssp_x and ssp_y have the changed position at $j \leq i + 1$, it holds that $\text{ssp}_x[j] = \text{ssp}_y[j] = j$ and $\text{ssp}_{x[2..]}[j - 1] = \text{ssp}_{y[2..]}[j - 1] = \infty$, which also indicates that $j < i + 1$. Since this change does not affect the lexicographic order, the claim holds. See the left part of Figure 4 for an illustration of this case.
3. Assume ssp_y has the changed position at $j \leq i + 1$, but ssp_x does not. Since $x[1..i]$ and $y[1..i]$ pal-match, j cannot be less than $i + 1$, and hence, $j = i + 1$ and $\text{ssp}_x[i + 1] = \text{ssp}_{x[2..]}[i] \prec i + 1 = \text{ssp}_y[i + 1] \prec \infty = \text{ssp}_{y[2..]}[i]$. Note that the lexicographic order between ssp_x and ssp_y (resp. $\text{ssp}_{x[2..]}$ and $\text{ssp}_{y[2..]}$) is determined by that between $\text{ssp}_x[i + 1]$ and $\text{ssp}_y[i + 1]$ (resp. $\text{ssp}_{x[2..]}[i]$ and $\text{ssp}_{y[2..]}[i]$). Since the lexicographic order between $\text{ssp}_x[i + 1]$ and $\text{ssp}_y[i + 1]$ is the same as that between $\text{ssp}_{x[2..]}[i]$ and $\text{ssp}_{y[2..]}[i]$, the claim holds. See the right part of Figure 4 for an illustration of this case.

Thus, we conclude that the lemma holds. ◀



■ **Figure 4** The left (resp. right) figure illustrates the second (resp. third) case in the proof of Lemma 10.

4 Computational results for new encodings

In this section, we show that the ssp - and sspg -encodings can be computed in linear time for a given string.

We use the following known results.

► **Lemma 11** ([26]). *For any string w , we can compute all the maximal palindromes in $O(|w|)$ time.*

► **Lemma 12** (Lemma 3 in [19]). *For any string w , we can compute lpal_w in $O(|w|)$ time.*

Using Lemmas 11 and 12, we obtain:

► **Lemma 13.** *For any string w , we can compute ssp_w in $O(|w|)$ time.*

Proof. Manacher's algorithm [26] can compute the radius of the maximal palindrome in increasing order of centers in linear time. It can be extended to compute the length $\text{lpal}_w[i]$ of the longest palindrome ending at each position i because the maximal palindrome with the smallest center that ends at position $\geq i$ gives us the longest suffix-palindrome ending at i by truncating the palindrome at i (e.g., see Lemma 3 of [19]). In a similar way, we can compute the length $\text{lpal}'_w[i]$ of the second longest palindrome ending at i .

Using lpal_w and lpal'_w , we can compute $\text{ssp}_w[i]$ in increasing order as follows:

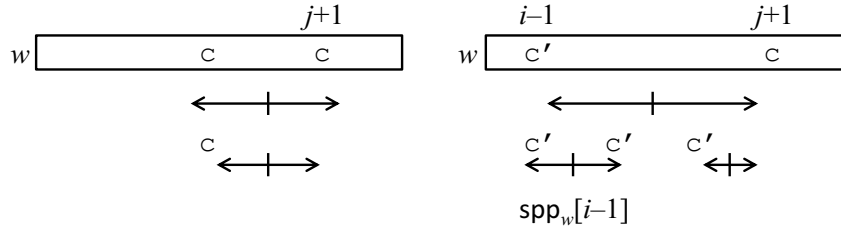
1. If $\text{lpal}_w[i] = 1$, then $\text{ssp}_w[i] = \infty$.
2. If $\text{lpal}_w[i] > 1$ and $\text{lpal}'_w[i] = 1$, then $\text{ssp}_w[i] = \text{lpal}_w[i]$.
3. If $\text{lpal}_w[i] > 1$ and $\text{lpal}'_w[i] > 1$, then $\text{ssp}_w[i] = \text{ssp}_w[i - \text{lpal}_w[i] + \text{lpal}'_w[i]]$.

In the third case, we use the fact that the non-trivial shortest suffix-palindrome ending at i has length $\leq \text{lpal}'_w[i]$ and it ends at $i - \text{lpal}_w[i] + \text{lpal}'_w[i]$, too.

Clearly all can be done in $O(|w|)$ time. ◀

For any string w , let \mathbf{G}_w denote the array of length $|w|$ such that $\mathbf{G}_w[i]$ stores the number of suffix-pal-groups for $w[..i]$.

► **Lemma 14.** *For any string w , we can compute \mathbf{G}_w in $O(|w|)$ time.*



■ **Figure 5** The left figure illustrates the case with $\text{lpal}_w[j+1] > 1$, in which we see that there is a suffix-pal-group for $w[..j]$ that has $w[j+1] = c$ immediately to their left. The right figure illustrates the case with $\text{spp}_w[i-1] \leq |w[i-1..j]|$, in which we see that the maximal palindrome $w[i..j]$ is not the representative because there is a shorter palindrome that ends at j and has the same character c' immediately to the left.

Proof. Let spp_w be the array defined in a symmetric way of ssp_w such that $\text{spp}_w[i]$ stores the length of the non-trivial shortest prefix-palindrome starting at i (or ∞ if such a palindrome does not exist). Using Lemma 13 in a symmetric way, we can compute spp_w in $O(|w|)$ time.

Let us focus on the palindromes involved in $\mathbf{G}_w[j]$. First, there is a suffix-pal-group for $w[..j]$ that has $w[j+1]$ immediately to their left iff $\text{lpal}_w[j+1] > 1$. Next observe that the palindromes in other suffix-pal-groups for $w[..j]$, which do not have $w[j+1]$ immediately to their left, are the maximal palindromes ending at j . Also, a maximal palindrome $w[i..j]$ is the representative (i.e., the shortest palindrome) in a suffix-pal-group to which it belongs, if and only if $\text{spp}_w[i-1] > |w[i-1..j]|$ or $i = 1$. See Figure 5 for illustrations of these observations.

Based on the above observations, we compute \mathbf{G}_w as follows: First, we compute the maximal palindromes and lpal_w in $O(|w|)$ time by Lemmas 11 and 12. Next we check every maximal palindrome and assign it to its ending position if it is a representative, which can be done in $O(|w|)$ time in total. We also check if $\text{lpal}_w[j+1] > 1$ for all positions j in $O(|w|)$ time to count a suffix-pal-group that has $w[j+1]$ immediately to their left. To sum up, \mathbf{G}_w can be computed in $O(|w|)$ time. ◀

Generalizing the algorithm presented in the proof of Lemma 14, we obtain:

► **Lemma 15.** *For any string w , we can compute sspg_w in $O(|w|)$ time.*

Proof. We modify the algorithm presented in the proof of Lemma 14 slightly. Now the task is to count, for every position $j+1$, the number of suffix-pal-groups for $w[..j]$ whose representative is shorter than $\text{ssp}[j+1] - 1$ because the number is exactly $\text{sspg}_w[j+1]$ by definition. We check every maximal palindrome $w[i..j]$ and assign it to its ending position j if $\text{spp}_w[i-1] > |w[i-1..j]|$ and $\text{ssp}[j+1] - 1 > j - i + 1$. Finally the number of representatives assigned to j plus one is $\text{sspg}_w[j+1]$. Similarly to the proof of Lemma 14, all can be done in $O(|w|)$ time. ◀

5 PalFM-index

The PalFM-index of T conceptually sort the suffixes of T in lexicographic order of their ssp -encodings (or equivalently sspg -encodings). Let SA_{pal} be the integer array of length $n+1$ such that $\text{SA}_{\text{pal}}[i]$ is the starting position of the i -th suffix of T in ssp -encoded order. We define the strings F_{pal} and L_{pal} of length $n+1$ based on π function applied to the sorted suffixes. Formally, for any position i ($1 \leq i \leq n+1$) we define:

i	$T[i..]$	$\text{ssp}_{T[i..]}$	$\text{ssp}_{T[\text{SA}_{\text{pal}}[i]..]}$	$\text{SA}_{\text{pal}}[i]$	$\text{F}_{\text{pal}}[i]$	$\text{L}_{\text{pal}}[i]$	$\text{LF}_{\text{pal}}(i)$
1	abbabbcbc	$\infty\infty 2432\infty 33$	ε	10	\$	∞	2
2	bbabbcbc	$\infty 2\infty 32\infty 33$	∞	9	∞	∞	5
3	babbcbc	$\infty\infty 32\infty 33$	$\infty 2\infty 32\infty 33$	2	1	2	6
4	abbcbc	$\infty\infty 2\infty 33$	$\infty 2\infty 33$	5	1	∞	7
5	bbcbc	$\infty 2\infty 33$	$\infty\infty$	8	∞	2	8
6	bcbc	$\infty\infty 33$	$\infty\infty 2432\infty 33$	1	2	\$	1
7	cbc	$\infty\infty 3$	$\infty\infty 2\infty 33$	4	∞	2	9
8	bc	$\infty\infty$	$\infty\infty 3$	7	2	2	10
9	c	∞	$\infty\infty 32\infty 33$	3	2	1	3
10	ε	ε	$\infty\infty 33$	6	2	1	4

■ **Figure 6** An example of $\text{SA}_{\text{pal}}[i]$, $\text{F}_{\text{pal}}[i]$ and $\text{L}_{\text{pal}}[i]$ for $T = \text{abbabbcbc}$.

$$\text{F}_{\text{pal}}[i] = \begin{cases} \$ & \text{if } i = 1, \\ \pi(T[\text{SA}_{\text{pal}}[i]..]) & \text{otherwise.} \end{cases}$$

$$\text{L}_{\text{pal}}[i] = \begin{cases} \$ & \text{if } \text{SA}_{\text{pal}}[i] = 1, \\ \pi(T[\text{SA}_{\text{pal}}[i] - 1..]) & \text{otherwise.} \end{cases}$$

See Figure 6 for example.

As in the case of LF, we define a function $\text{LF}_{\text{pal}} : i \mapsto j$ so that $\text{SA}_{\text{pal}}[j] = \text{SA}_{\text{pal}}[i] - 1$ (with the corner case $\text{LF}_{\text{pal}}(i) = 1$ for $\text{SA}_{\text{pal}}[i] = 1$). Thanks to Lemma 10, for any value c , the suffixes used to obtain i -th k in L_{pal} and in F_{pal} are the same, which enables us to implement the LF_{pal} function by $\text{LF}_{\text{pal}}(i) = \text{select}_{\text{F}_{\text{pal}}}(\text{rank}_{\text{L}_{\text{pal}}}(i, \text{L}_{\text{pal}}[i]), \text{L}_{\text{pal}}[i])$. See Figure 7 for an illustration.

For any string w , let w -interval refer to the maximal interval $[b..e]$ such that $\text{ssp}_{T[\text{SA}_{\text{pal}}[i]..]}$ is prefixed by ssp_w , where w -interval is empty if there is no substring of T that pal-matches with w . Notice that the substring of T of length $|w|$ starting at $\text{SA}_{\text{pal}}[i]$ pal-matches with w iff $i \in [b..e]$. A single step of backward search computes cw -interval from w -interval for some character c .

The following theorems are the main contributions of this paper.

► **Theorem 16.** *Let T be a string of length n over an alphabet of size σ . There is a data structure of $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space to support the counting queries for the pal-matching problem in $O(m)$ time, where m is the length of a given pattern P .*

Proof. We use the data structures of Theorem 1 for L_{pal} and F_{pal} , and the RMQ data structure of Theorem 2 for the integer array V with $V[i] = \text{LF}_{\text{pal}}(i)$. Since the number of distinct symbols in L_{pal} and F_{pal} are $O(\min(\sigma, \lg n))$ by Lemma 6, the data structures occupy $2n \lg \min(\sigma, \lg n) + 2n + o(n)$ bits of space in total and all queries (rank, select, rangeCount and RMQ) can be supported in $O(1)$ time.

The number of occurrences of P can be answered by computing the width of P -interval. Thus we focus on a single step of backward search. In a general setting, for any string w and a character c , we show how to compute cw -interval $[b'..e']$ in $O(1)$ time from w -interval $[b..e]$, $\pi(cw)$ and the number g of prefix-pal-groups of w . The procedure differs depending on $\pi(cw) = \infty$ or not.

$T[\text{SA}[i]..]$	$F_{\text{pal}}[i]$	$LF_{\text{pal}}(i)$	$L_{\text{pal}}[i]$	$T[\text{SA}[i]-1..]$
ε	$\$$		∞	c
c	∞		∞	b c
b b a b b c b c	1		2	a b b a b b c b c
b b c b c	1		∞	a b b c b c
b c	∞		2	c b c
a b b a b b c b c	2		$\$$	
a b b c b c	∞		2	b a b b c b c
c b c	2		2	b c b c
b a b b c b c	2		1	b b a b b c b c
b c b c	2		1	b b c b c

■ **Figure 7** An illustration for $F_{\text{pal}}[i]$, $L_{\text{pal}}[i]$ and $LF_{\text{pal}}(i)$. Except the corner cases that have $\$$, $F_{\text{pal}}[i]$ and $L_{\text{pal}}[i]$ are defined by $\pi(T[\text{SA}_{\text{pal}}[i]..])$ and $\pi(T[\text{SA}_{\text{pal}}[i]-1..])$, respectively. Since $\pi(w)$ encodes the information about the non-trivial shortest prefix of w , in each row the non-trivial shortest prefix is shown in grayed background. For example, $\pi(\text{abbabbcbc}) = 2$ because its non-trivial shortest prefix-palindrome **abba** is extended from the prefix-palindrome **bb** of **bbabbcbc** and **bb** belongs to the prefix-pal-group with the identifier 2. Observe that F_{pal} is a permutation of L_{pal} since both F_{pal} and L_{pal} use every suffix w of T exactly once to obtain $\pi(w)$. Roughly speaking, $LF_{\text{pal}}(\cdot)$ is meant to map a row having a suffix w in the $T[\text{SA}_{\text{pal}}[i]-1..]$ column to the row having the same suffix w in the $T[\text{SA}_{\text{pal}}[i]..]$ column. Thanks to Lemma 10, for any value k , the suffixes used to obtain i -th k in L_{pal} and in F_{pal} are the same, and hence, one can observe visually that the arrows starting from the same L_{pal} -value are not crossed.

- When $\pi(cw) = k \neq \infty$. Using Lemma 9 in a symmetric way, $[b'..e']$ is obtained by mapping the positions of $\pi(cw)$ in $L_{\text{pal}}[b..e]$ by the LF_{pal} function. More specifically, $b' = \text{select}_{F_{\text{pal}}}(\text{rank}_{L_{\text{pal}}}(b-1, k) + 1, k)$ and $e' = \text{select}_{F_{\text{pal}}}(\text{rank}_{L_{\text{pal}}}(e, k), k)$, which can be computed in $O(1)$ time.
- When $\pi(cw) = \infty$. We note that $[b'..e']$ is the maximal interval such that $T[\text{SA}_{\text{pal}}[i]..]$ does not have non-trivial prefix-palindrome (i.e. $\pi(T[\text{SA}_{\text{pal}}[i]..]) = \infty$) or $T[\text{SA}_{\text{pal}}[i]..]$ has the non-trivial shortest prefix-palindrome of length longer than $|cw|$ (i.e. $\pi(T[\text{SA}_{\text{pal}}[i]..]) > g$). Thus, $e' - b' + 1$ is equivalent to the number of occurrences of values larger than g in $L_{\text{pal}}[b..e]$, which can be computed in $\text{rangeCount}_{L_{\text{pal}}}(b, e, g, \infty)$ in $O(1)$ time. Moreover, it holds that $e' = LF_{\text{pal}}(\text{RMQ}_V(b, e))$ because $\text{ssp}(T[\text{SA}_{\text{pal}}[i]-1..])$ with $\pi(T[\text{SA}_{\text{pal}}[i]-1..]) = L_{\text{pal}}[i] > g$ is always lexicographically larger than $\text{ssp}(T[\text{SA}_{\text{pal}}[j]-1..])$ with $\pi(T[\text{SA}_{\text{pal}}[j]-1..]) = L_{\text{pal}}[j] \leq g$. Thus, we can compute $[b'..e']$ in $O(1)$ time.

Backward search for P requires $\pi(P[i..])$ and the number g of prefix-pal-groups of $P[i..]$ for all $1 \leq i \leq m$, which can be computed by ssp_{PR} and G_{PR} in $O(m)$ time using Lemmas 15 and 14.

Putting all together, we get the theorem. ◀

▶ **Theorem 17.** *Let T be a string of length n over an alphabet of size σ and Δ be an integer in $[1..n]$. There is a data structure of $2n \lg \min(\sigma, \lg n) + \frac{n}{\Delta} \lg n + 3n + o(n)$ bits of space to support the locating queries for the pal-matching problem in $O(m + \Delta \text{occ})$ time, where m is the length of a given pattern P and occ is the number of occurrences to report.*

Proof. We use the data structure and the algorithm of Theorem 16 to compute P -interval in $2n(1 + \lg \min(\sigma, \lg n)) + o(n)$ bits of space and $O(m)$ time. The occurrences of P (in the sense of pal-matching) can be answered by the SA_{pal} -values in P -interval. We employ exactly the same sampling technique used in the FM-index to retrieve SA-values (e.g., see [7]): We make a bit vector B of length $n + 1$ marking the positions i in SA_{pal} such that $\text{SA}_{\text{pal}}[i] = \Delta k + 1$ for some integer k , and the sparse suffix array S holding only the marked SA_{pal} -values in the order. B is equipped with a data structure to support the rank queries and the additional space to Theorem 16 is $\frac{n}{\Delta} \lg n + n + o(n)$ bits in total.

If position i is marked, $\text{SA}_{\text{pal}}[i]$ is retrieved by $S[\text{rank}_B(i, 1)]$ in $O(1)$ time. If position i is not marked, we apply LF-mapping k times from i until we reach a marked position j and retrieve $\text{SA}_{\text{pal}}[i]$ by $S[\text{rank}_B(j, 1)] + k$. Since text positions are marked every Δ positions, the number k of LF-mapping steps is at most Δ , and hence, $\text{SA}_{\text{pal}}[i]$ can be retrieved in $O(\Delta)$ time. Therefore we can report each occurrence of P in $O(\Delta)$ time, and the theorem follows. ◀

6 Conclusions and future work

In this paper, we developed new encoding schemes for pal-matching and proposed the PalFM-index, a space-efficient index for pal-matching based on the FM-index. Future work includes to present an efficient construction algorithm of the PalFM-index, and to reduce the space requirement (e.g. by incorporating with the idea of [13]). Another interesting research direction would be to develop a general framework to design FM-index type indexes in generalized pattern matching. We believe that switching encoding from `lpal` to `ssp` to design the PalFM-indexes gives a good hint to pursue this direction, and conjecture that any generalized pattern matching under a substring consistent equivalent relation [27] admits such shortest positional encodings to design FM-index type indexes.

References

- 1 Jean-Paul Allouche, Michael Baake, Julien Cassaigne, and David Damanik. Palindrome complexity. *Theor. Comput. Sci.*, 292(1):9–31, 2003.
- 2 Mira-Cristiana Anisiu, Valeriu Anisiu, and Zoltán Kása. Total palindrome complexity of finite words. *Discrete Mathematics*, 310(1):109–114, 2010. doi:10.1016/j.disc.2009.08.002.
- 3 Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM) 2017*, pages 23:1–23:12, 2017. doi:10.4230/LIPIcs.CPM.2017.23.
- 4 Srečko Brlek, Sylvie Hamel, Maurice Nivat, and Christophe Reutenauer. On the palindromic complexity of infinite words. *Int. J. Found. Comput. Sci.*, 15(2):293–306, 2004. doi:10.1142/S012905410400242X.
- 5 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994.
- 6 Xavier Droubay, Jacques Justin, and Giuseppe Pirillo. Episturmian words and some constructions of de luca and rauzy. *Theor. Comput. Sci.*, 255(1-2):539–553, 2001. doi:10.1016/S0304-3975(99)00320-5.
- 7 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398, 2000.
- 8 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.
- 9 Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48, 2014. StringMasters 2012 & 2013 Special Issue (Volume 1). doi:10.1016/j.jda.2014.08.001.

- 10 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
- 11 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In *Proc. 25th Annual European Symposium on Algorithms (ESA) 2017*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.
- 12 Zvi Galil and Joel I. Seiferas. A linear-time on-line recognition algorithm for “palstar”. *J. ACM*, 25(1):102–111, 1978. doi:10.1145/322047.322056.
- 13 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2017*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.
- 14 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In *Proc. 28th International Symposium on Algorithms and Computation (ISAAC) 2017*, pages 35:1–35:13, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.
- 15 Amy Glen, Jacques Justin, Steve Widmer, and Luca Q. Zamboni. Palindromic richness. *Eur. J. Comb.*, 30(2):510–531, 2009. doi:10.1016/j.ejc.2008.04.006.
- 16 Alexander Golynski, Rajeev Raman, and S. Srinivasa Rao. On the redundancy of succinct data structures. In Joachim Gudmundsson, editor, *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT) 2008*, volume 5124 of *Lecture Notes in Computer Science*, pages 148–159. Springer, 2008.
- 17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2003*, pages 841–850. ACM/SIAM, 2003.
- 18 Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Counting and verifying maximal palindromes. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE) 2010*, pages 135–146, 2010.
- 19 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome pattern matching. *Theor. Comput. Sci.*, 483:162–170, 2013. doi:10.1016/j.tcs.2012.01.047.
- 20 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM) 2014*, volume 8486 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2014.
- 21 Ignacio Tinoco Jr., Olke C. Uhlenbeck, and Mark D. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971.
- 22 Sung-Hwan Kim and Hwan-Gue Cho. A compact index for cartesian tree matching. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM) 2021*, volume 191 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- 23 Sung-Hwan Kim and Hwan-Gue Cho. Simpler FM-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. doi:10.1016/j.ipl.2020.106026.
- 24 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
- 25 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Pal k is linear recognizable online. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, pages 289–301, 2015. doi:10.1007/978-3-662-46078-8_24.
- 26 Glenn K. Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.
- 27 Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theor. Comput. Sci.*, 656:225–233, 2016.

- 28 Antonio Restivo and Giovanna Rosone. Burrows-wheeler transform and palindromic richness. *Theor. Comput. Sci.*, 410(30-32):3018–3026, 2009. doi:10.1016/j.tcs.2009.03.008.
- 29 Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018. doi:10.1016/j.ejc.2017.07.021.

Computing MEMs on Repetitive Text Collections

Gonzalo Navarro 

Center for Biotechnology and Bioengineering (CeBiB), Santiago, Chile
Department of Computer Science, University of Chile, Santiago, Chile

Abstract

We consider the problem of computing the Maximal Exact Matches (MEMs) of a given pattern $P[1..m]$ on a large repetitive text collection $T[1..n]$, which is represented as a (hopefully much smaller) run-length context-free grammar of size g_{ri} . We show that the problem can be solved in time $O(m^2 \log^\epsilon n)$, for any constant $\epsilon > 0$, on a data structure of size $O(g_{ri})$. Further, on a locally consistent grammar of size $O(\delta \log \frac{n}{\delta})$, the time decreases to $O(m \log m (\log m + \log^\epsilon n))$. The value δ is a function of the substring complexity of T and $\Omega(\delta \log \frac{n}{\delta})$ is a tight lower bound on the compressibility of repetitive texts T , so our structure has optimal size in terms of n and δ .

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases grammar-based indices, maximal exact matches, locally consistent grammars, substring complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.24

Funding Funded by Basal Funds FB0001, Mideplan, Chile, and Fondecyt Grant 1-200038, Chile.

1 Introduction and Related Work

Mutations and experimental sequencing errors make exact pattern matching seldom used in Bioinformatic applications, except possibly for very short patterns and some niche applications [19, 37, 28]. A much more interesting problem is that of finding the Maximal Exact Matches (MEMs) of a given pattern $P[1..m]$ in a text $T[1..n]$. A MEM is a maximal substring $P[i..j]$ that appears in T (i.e., $P[i-1..j]$ and $P[i..j+1]$ are out of bounds or do not occur in T). This is useful, for example, to find long conserved areas of a gene or to best align a read (where m is typically in the hundreds or thousands) on a reference genome (where n can be in the billions), and even to find similarities between two genomes. In this paper we are interested in the case where T is known in advance and can be indexed.

Finding MEMs is a classic problem in stringology and can be solved in optimal $O(m)$ time using a suffix tree of T [41, 31] (see, e.g., the similar problem of computing matching statistics [19, Sec. 7.8]). Suffix trees, even if using linear space, are too large to maintain in main memory for current text collection sizes, however. The suffix tree of a single human genome, for example, with $n \approx 3 \cdot 10^9$, may take 60GB with a decent implementation. This makes suffix trees hard to use directly on current bioinformatic collections. Even if lower-space alternatives can replace suffix trees for most tasks [28, MEMs in Sec. 11.1.3], this space reduction is still insufficient to face current projects for sequencing millions of human genomes (see <https://b1mg-project.eu>).

A fortunate situation is that many of the fastest growing text collections are highly repetitive [33]. For example, collections of genomes of the same species feature a small percentage of differences between any pair of genomes. Several text indices exploiting repetitiveness to reduce space have appeared [34]. Those indices may take orders of magnitude less space than the raw data, and even more orders less space than a suffix tree on the data.

Those compressed indices support exact pattern matching, that is, they can list all the positions where P occurs in T . While useful, this is less than the full suffix tree functionality, and insufficient to efficiently implement the classic $O(m)$ -time MEM finding algorithm.



© Gonzalo Navarro;

licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 24; pp. 24:1–24:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Compressed suffix trees for highly repetitive text collections do exist, but do not compress that much. Gagie et al. [17] show how to simulate a suffix tree within space $O(r \log \frac{n}{r})$, where r is the number of equal-letter runs in the BWT [7] of T . It could find the MEMs in time $O(m \log \frac{n}{r})$ if we run the algorithm backwards on P , using operations *parent* and *Weiner link* instead of *child* and *suffix link*. The problem is the space: while r is an accepted measure of repetitiveness [21], it is a weak one [33, 21], and multiplying it by $\log \frac{n}{r}$ makes it grow by an order of magnitude. Current implementations of compressed suffix trees for repetitive texts achieve remarkable space, but still use at least 2–4 bits per symbol [39, 15, 8, 5].

Another trend has been to expand the functionality of a more basic compressed text index for repetitive texts so as to support specific operations, MEMs in our case. Bannai et al. [1] show how to compute matching statistics (from where MEMs are easily extracted in $O(m)$ time) by extending the RLBWT-index [29], in $O(m(s + \log \log n))$ time and $O(r)$ space, with the help of a data structure that provides access to a symbol of T in time $O(s)$. This can be, for example, the samples of the RLBWT-index, which add $O(n/s)$ space to the index, or a context-free grammar of T , which provides access in time $s = O(\log n)$ [4]. Various implementations of this idea [38, 6, 40] showed its practicality on large genome collections, with indices that are an order of magnitude smaller than the text.

All those results have been obtained on the so-called *suffix-based* compressed indices for repetitive collections [34]. This is natural because those emulate variants of suffix trees or arrays [30], which simplifies the problem of simulating the suffix tree traversal of the classic MEM-finding algorithm. Even the naive algorithm of searching for all the $O(m^2)$ substrings of P can be run in $O(m^2 \log \log n)$ time on those $O(r)$ -sized indices.

The problem is much harder on the so-called *parsing-based* indices [34]. Those are potentially smaller than the suffix-based indices because they build on stronger measures of repetitiveness. For example, the size g of the smallest context-free grammar that generates T is usually considerably smaller than r [33]. Because these indices cut T into phrases, even exact pattern matching is complicated because the occurrences of P can appear in many different forms, and many possible cuts of P must be tried out ($m - 1$ in the general case) [12]. This makes the problem of finding MEMs considerably harder. We are only aware of the results of Gao [18], who computes matching statistics in time $O(m^2 \log^\epsilon \gamma + m \log n)$ using $O(\delta \log \frac{n}{\delta})$ space (for any constant $\epsilon > 0$), or $O(m^2 + m \log \gamma \log \log \gamma + m \log n)$ using $O(\delta \log \frac{n}{\delta} + \gamma \log \gamma)$ space. Here $\delta \leq \gamma$ are lower-bounding measures of repetitiveness [22, 11]. The size $O(\delta \log \frac{n}{\delta})$ matches a tight lower bound on the size of compressed representations of T [25], so a structure of this size uses asymptotically optimal space for every n and δ .

Let g_{rl} be the size of any run-length context-free grammar generating T (those include and extend classic context-free grammars). The smallest such grammar is of size $g_{rl} = O(\delta \log \frac{n}{\delta})$ [25]. We first show that, on an index of size $O(g_{rl})$, one can compute the MEMs in time $O(m^2 \log^\epsilon g_{rl})$, for any constant $\epsilon > 0$. This is done by sliding the window $P[i..j]$ of the classic algorithm while we simulate the process of searching for that window with the grammar. The simulation is carefully crafted to avoid expensive operations, so the time stays proportional to the number of cuts tried out on a single search for P . The space $O(g_{rl})$ is the least known to support direct access to T with logarithmic time guarantees [33]. The result essentially matches the first one of Gao, which could also run within $O(g_{rl})$ space.

We further show that, on a particular grammar featuring local consistency properties [24], we can reduce the time to $O(m \log m (\log m + \log^\epsilon n))$ by exploiting the fact that only $O(\log(j - i + 1))$ cuts need to be tried out for $P[i..j]$, and using much more sophisticated techniques to amortize the costs. This grammar is of size $O(\delta \log \frac{n}{\delta})$, optimal for every n and δ , and within this space we sharply break the quadratic time of previous solutions.

2 Maximal Exact Matches (MEMs) and How to Find Them

We assume the usual notation on strings $S[1..n]$ and that the reader is familiar with the concepts related to suffix trees [41, 31, 13]. We start by defining MEMs.

► **Definition 1.** A Maximal Exact Match (MEM) of a pattern $P[1..m]$ in a string T is a substring $P[i..j]$ that occurs in T , but in addition

- $i = 1$ or $P[i-1..j]$ does not occur in T , and
- $j = m$ or $P[i..j+1]$ does not occur in T .

► **Definition 2.** Given a text $T[1..n]$ that can be preprocessed, the MEM-finding problem is that of, given a pattern $P[1..m]$, return the range (i, j) of each of its MEMs $P[i..j]$ in T , in increasing order of i (or j). A position where each MEM occurs in T must also be returned.

The MEM finding problem can be solved in $O(m)$ time with a suffix tree. Algorithm 1 shows how, abstracting away some complications of implementing it on the long edges of suffix trees. The next problem is strongly related to the MEM finding problem.

► **Definition 3.** Given a text $T[1..n]$ that can be preprocessed, the matching statistics problem is that of, given a pattern $P[1..m]$, return the length $M[k]$ of the longest prefix of $P[k..]$ that occurs in T , for every $1 \leq k \leq m$. A position where each such longest prefix occurs must be given for each k .

Given a solution to the MEM finding problem, $(i_1, j_1), \dots, (i_s, j_s)$, we compute the matching statistics as follows. Set all $M[k]$ to zero and then traverse the tuples (i_r, j_r) in order. Set $M[k] = j_r - k + 1$ for all $i_r \leq k \leq \min(j_r, i_{r+1} - 1)$, assuming $i_{s+1} = m + 1$. The occurrence of each $M[k] > 0$ is that of its (i_r, j_r) shifted by $k - i_r$. Conversely, given the matching statistics $M[k]$ for $1 \leq k \leq m$, we obtain the MEMs by reporting, for increasing i , every pair $(i, i + M[i] - 1)$ such that $i = 1$ or $M[i] \geq M[i - 1]$, and $M[i] > 0$. Therefore, both problems are interchangeable as one can convert one output to the other in optimal $O(m)$ time. Gusfield [19, Sec. 7.8] shows how to compute matching statistics with the suffix tree.

■ **Algorithm 1** Finding the MEMs of $P[1..m]$ in T using the suffix tree of T .

```

1  $i \leftarrow 1; j \leftarrow 0;$ 
2  $v \leftarrow$  suffix tree root;
3 while  $j < m$  do
4   if  $v$  has no child labeled  $P[j + 1]$  then
5      $i \leftarrow i + 1; j \leftarrow j + 1;$ 
6   end
7   else
8     while  $j < m$  and  $v$  has a child labeled  $P[j + 1]$  do
9        $j \leftarrow j + 1; v \leftarrow$  the child of  $v$  by  $P[j + 1];$ 
10    end
11    report  $(i, j)$  with some occurrence of  $v;$ 
12    while  $i \leq j < m$  and  $v$  has no child labeled  $P[j + 1]$  do
13       $i \leftarrow i + 1; v \leftarrow$  the suffix link of  $v;$ 
14    end
15  end
16 end

```

3 Grammar based Indices

Let $T[1..n]$ be a text. Grammar-based compression of T consists in replacing it by a context-free grammar (CFG) that generates only T [23]. The compression ratio is then the size of the grammar divided by the text size.

We consider a slightly more powerful type of grammar called run-length context-free grammar (RLCFG), which includes run-length rules of constant size. To simplify, we disallow rules of the form $A \rightarrow \varepsilon$, which are easily removed without increasing the grammar size.

► **Definition 4.** A Run-Length Context-Free Grammar (RLCFG) for T is a context-free grammar that generates (only) T , having exactly one rule per nonterminal A . The rules are of the form $A \rightarrow B_1 \cdots B_k$ for $k > 0$ and terminals or nonterminals B_i (this rule is said to be of size k), and of the form $A \rightarrow B^k$ for $k > 1$ and a terminal or nonterminal B , which is identical to $A \rightarrow B \cdots B$ with k copies of B , but is said to be of size 2. The size of the RLCFG is the sum of the sizes of all of its rules. A Context-Free Grammar (CFG) for T is a RLCFG for T that does not use rules of the form $A \rightarrow B^k$.

Clearly, the size g_{rl} of the smallest RLCFG for T is always less than or equal to the size g of the smallest CFG for T . Grammar-based compression (with or without run-length rules) has proved to be particularly effective on highly repetitive texts [34]. While finding the smallest grammar is NP-hard [10], heuristics like RePair obtain very good results [27].

Note that our RLCFGs have a unique parse tree, defined as follows [11, Sec. 4].

► **Definition 5.** The parse tree of a RLCFG for T has a root labeled with the initial symbol. If a node is labeled A and its rule is $A \rightarrow B_1 \cdots B_k$, then the node has k children labeled B_1, \dots, B_k left to right. If its rule is $A \rightarrow B^k$, then the node has k children labeled B . It follows that the i th left-to-right leaf of the parse tree is labeled $T[i]$.

While the parse tree has size $\Theta(n)$, a convenient representation of a RLCFG is the so-called grammar tree, which is of size $O(g_{rl})$ [11, Sec. 6].

► **Definition 6.** The grammar tree of a RLCFG is obtained by pruning its parse tree, preserving the leftmost internal node labeled A for each nonterminal A , and converting the others to leaves. Further, for the remaining internal nodes labeled A with rules $A \rightarrow B^k$ we preserve their first child only, replacing the other $k - 1$ children (which are leaves) with a single special leaf labeled $B^{[k-1]}$. If the RLCFG size is g_{rl} , its grammar tree has $g_{rl} + 1$ nodes.

We will sometimes identify a nonterminal with its (only) internal node in the grammar tree. We call $exp(A)$ the string of terminals to which symbol A expands, and $exp(a) = a$ for terminals a . The grammar tree defines a parse of T , as follows.

► **Definition 7.** The grammar tree, with leaves v_1, \dots, v_k , induces the parse $T = exp(v_1) \cdot exp(v_2) \cdots exp(v_k)$ into phrases $exp(v_i)$.

A classic grammar-based index [12] divides the occurrences of a pattern $P[1..m]$ into *primary* and *secondary*, depending on whether they cross a phrase boundary or lie within a phrase, respectively (if $m = 1$, its occurrences ending a phrase boundary are taken as primary). It uses the fact that every occurrence has primary occurrences and that all the secondary ones can be found inside pruned leaves of nonterminals that contain other occurrences. In this paper we will be interested in the mechanism to find the primary occurrences. This is based on the parsing, but defined in a particular way to avoid reporting multiple times the primary occurrences that cross several phrase boundaries. The mechanism was extended to RLCFGs [11, Sec. 6 and App. A].

► **Definition 8.** Let \mathcal{X} and \mathcal{Y} be multisets of strings defined as follows. For each rule $A \rightarrow B_1 \cdots B_t$, for each $1 < s \leq t$, the string $\text{exp}(B_{s-1})^{\text{rev}}$ (i.e., $\text{exp}(B_{s-1})$ read backwards) is inserted in \mathcal{X} and the string $\text{exp}(B_s) \cdots \text{exp}(B_t)$ is inserted in \mathcal{Y} ; we say those two are corresponding strings. Similarly, for each rule $A \rightarrow B^t$, $\text{exp}(B)^{\text{rev}}$ is inserted in \mathcal{X} and $\text{exp}(B)^{t-1}$ (i.e., $t-1$ concatenations of $\text{exp}(B)$) is inserted in \mathcal{Y} . A grid \mathcal{G} has one row per string in \mathcal{Y} and one column per string in \mathcal{X} . After lexicographically sorting \mathcal{X} and \mathcal{Y} , a point (x, y) is set in \mathcal{G} if the x th string of \mathcal{X} corresponds to the y th string of \mathcal{Y} .

The grammar-based index includes a Patricia tree $P_{\mathcal{X}}$ storing the strings of \mathcal{X} and another Patricia tree $P_{\mathcal{Y}}$ storing the strings of \mathcal{Y} [32]. Let us add some data to nodes for our convenience. Each Patricia tree node v stores its range $[v^1, v^2]$ of the left-to-right ranks of the leaves descending from v . The edges of the Patricia tree nodes can represent strings, so prefixes that end in the middle of an edge that leads to a node v correspond to *virtual* nodes u ; the range $[u^1, u^2]$ is the same $[v^1, v^2]$. The nodes v also store their string depth $|v|$, which is also easily computed for virtual nodes as we descend or ascend in the Patricia tree.

Each primary occurrence consists of a suffix of some string $X \in \mathcal{X}$ matching $P[1..i]$ corresponding to some string $Y \in \mathcal{Y}$ whose prefix matches $P[i+1..m]$, for some $1 \leq i < m$ (if $m = 1$, it is just a suffix of X matching P) [11, Sec. A.4]. Therefore, to find the primary occurrences of P , the index tries out every cutting point i , and searches $P_{\mathcal{X}}$ for $P[1..i]^{\text{rev}}$ and $P_{\mathcal{Y}}$ for $P[i+1..m]$. If both nodes $x \in P_{\mathcal{X}}$ and $y \in P_{\mathcal{Y}}$ exist, then the points in the orthogonal range $[x^1, x^2] \times [y^1, y^2]$ of \mathcal{G} represent the primary occurrences of P cut at position i , and are efficiently found with a geometric data structure on \mathcal{G} . By storing the position t of T where $\text{exp}(B_{s-1})$ ends for such point, we know that P occurs in $T[t-i+1..t-i+m]$ (the actual index stores pointers to the grammar tree, but this suffices for us).

Both the Patricia trees and the grid take $O(g_{rl})$ space. The index also needs to verify the matches of the Patricia trees. It uses an $O(g_{rl})$ -space data structure \mathcal{A} that can read, in $O(\ell)$ time, any length- ℓ prefix or suffix of $\text{exp}(A)$, for any nonterminal A [11, Lem 6.6]. If x is a node of $P_{\mathcal{X}}$, its corresponding string is the $|x|$ -length reversed suffix of any string between the x^1 th and the x^2 th in \mathcal{X} . Let $X = \text{exp}(B_{s-1})^{\text{rev}}$ be one such string, then we store $\langle v \rangle = B_{s-1}$ associated with v . Similarly, a node $v \in P_{\mathcal{Y}}$ that prefixes $\text{exp}(B_s) \cdots \text{exp}(B_t)$ stores $\langle v \rangle = B_s$ (from where we can obtain the subsequent siblings). We can then obtain the string represented by any v using \mathcal{A} on $\langle v \rangle$.

4 A Quadratic-Time Solution

We now present a quadratic-time solution that works with any RLCFG of size g_{rl} for T ; we use the $O(g_{rl})$ -space data structures described in the previous section. Since any CFG is a particular case of RLCFG, our algorithm also runs with any CFG.

The generic idea follows that of Algorithm 1, sliding a window $P[i..j]$ along the pattern. We maintain a set of so-called *active positions* $r \in [i..j]$.

► **Definition 9.** A position $r \in [i..j]$ is active if $P[r+1..j]$ prefixes some string in $P_{\mathcal{Y}}$.

Note that, since we slide the window $P[i..j]$ forwards, once a position r becomes inactive, it will not become active again.

4.1 Algorithm

The algorithm maintains the invariant that, when the window is $P[i..j]$, (i, j) is the last MEM of $P[1..j]$ (if $i \leq j$) and all the MEMs ending before j have already been reported. It maintains the set $R \subseteq [i..j]$ of active positions, and for each such active position $r \in R$:

- The node $y_r \in P_{\mathcal{Y}}$ corresponding to $P[r + 1..j]$; this node can be virtual. Note that $[y_r^1, y_r^2]$ is the same range of rows in \mathcal{G} of the strings of \mathcal{Y} that start with $P[r + 1..j]$.
- The length ℓ_r of the maximum prefix of $P[r + 1..j]$ that prefixes a string in $P_{\mathcal{Y}}$; note that r is active iff $r + \ell_r \geq j$.
- The node $x_r \in P_{\mathcal{X}}$ corresponding to the longest prefix of $P[i..r]^{rev}$ that exists in $P_{\mathcal{X}}$, and such that there are points in \mathcal{G} in the range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$. Note again that x_r can be virtual and that $[x_r^1, x_r^2]$ is the same range of columns in \mathcal{G} of the strings of \mathcal{X} that start with $P[r - |x_r| + 1..r]^{rev}$.

Our algorithm, depicted in Algorithm 2, iterates over j , from 0 to $m - 1$, and at each cycle it extends the current window to end in $j + 1$. When $i = j + 1$ (including when we start with $i = 1$ and $j = 0$), the window is empty and there are no active positions. Line 3 first sees, in this case, if we can descend from the root of $P_{\mathcal{X}}$ by $P[j + 1]$, to start a new nonempty substring $P[j + 1, j + 1]$. If this is not possible, it just increases i and goes for the next value of j . Otherwise, there will be active positions for the window ending at $j + 1$ and we enter into the main process.

Lines 5–7 first create the new active position $r = j + 1$, with corresponding y_r set at the root of $P_{\mathcal{Y}}$. To compute ℓ_r , we descend in $P_{\mathcal{Y}}$ as much as possible by $P[r + 1..j]$. To compute x_r , we also descend in $P_{\mathcal{X}}$ as much as possible by $P[i..r]^{rev}$. Those are classic Patricia tree searches, first reaching a candidate node v by comparing only the branching characters in the trie, and then verifying which ancestor of v is the correct answer. The verification proceeds by extracting the needed prefix from $\langle v \rangle$ in $P_{\mathcal{Y}}$ (at most $\ell_r + 1$ characters) or the needed suffix in $P_{\mathcal{X}}$ (at most $|x_r| + 1$ characters).

Lines 8–16 then remove the active positions that do not reach $j + 1$ and updates the variables for the surviving ones. Line 10 first removes the active positions r where $r + \ell_r = j$. On the remaining ones, each y_r moves to its child by $P[j + 1]$ in $P_{\mathcal{Y}}$ in line 12 (this shrinks the range $[y_r^1, y_r^2]$). Note that, once we know that we can descend from y_r by $P[j + 1]$ (because $r + \ell_r \geq j + 1$), we can compute the child node on the Patricia tree without accessing the text, both for explicit and virtual nodes y_r . Thus, by computing ℓ_r once when the active position r is created, in time $O(\ell_r)$, we save all the accesses to T that would have been needed to descend from virtual nodes $y_r \in P_{\mathcal{Y}}$: when y_r is not the root, its text position is not phrase-aligned, so we cannot access its first symbols in constant time using \mathcal{A} .

Line 13 updates the nodes x_r of the surviving active positions, because some ranges $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ could be empty after we reduce $[y_r^1, y_r^2]$. For every active position r , as long as there are no points in $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$, we move x_r to its parent in $P_{\mathcal{X}}$. This process eventually terminates because, when x_r is the root and $[x_r^1, x_r^2]$ is the whole range of columns, we know that there are points in the band $[y_r^1, y_r^2]$ because it corresponds to the node y_r .

Lines 8, 14, and 17 recompute the value $p = \min\{r - |x_r| + 1, r \text{ is active}\}$. This is necessary to make i grow as needed so that $P[i..j + 1]$ occurs in T , then reestablishing the invariant that $P[i..j + 1]$ is the last MEM of $P[..j + 1]$. If $p = i$, then $P[i..j + 1]$ occurs in T (as it has a primary occurrence in some $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$), so we can retain the current value of i ; line 18 collects some text position t to be reported in case $(i, j + 1)$ turns out to be a MEM of the whole P . If, on the other hand, $p > i$, this means $P[i..j + 1]$ does not occur in T and thus (i, j) was a MEM. Lines 20–21 then report MEM (i, j) with its text position t (collected in the previous cycle of j) and increase i to p , since only $P[p..j + 1]$ occurs in T . This could make i exceed $j + 1$ when the window becomes empty; otherwise line 23 finally inserts $j + 1$ as an active position. Line 26 reports the final MEM when j reaches m .

■ **Algorithm 2** Finding the MEMs of $P[1..m]$ in T using a grammar-based index.

```

1  $i \leftarrow 1; R \leftarrow \emptyset;$ 
2 for  $j \leftarrow 0, \dots, m - 1$  do
3   if  $i = j + 1$  and the root of  $P_{\mathcal{X}}$  has no child labeled  $P[j + 1]$  then  $i \leftarrow i + 1$  ;
4   else
5      $y_{j+1} \leftarrow$  root of  $P_{\mathcal{Y}}$ ;
6      $v \leftarrow$  descend in  $P_{\mathcal{Y}}$  as much as possible with  $P[j + 2..]$ ;  $\ell_{j+1} \leftarrow |v|$ ;
7      $x_{j+1} \leftarrow$  descend in  $P_{\mathcal{X}}$  as much as possible with  $P[i..j + 1]^{rev}$ ;
8      $r_{\min} \leftarrow j + 1$ ;
9     for  $r \in R$  do
10      if  $r + \ell_r = j$  then  $R \leftarrow R \setminus \{r\}$  ;
11      else
12         $y_r \leftarrow$  child of  $y_r$  by  $P[j + 1]$ ;
13        while the range  $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$  is empty do  $x_r \leftarrow$  parent of  $x_r$  ;
14        if  $r - |x_r| < r_{\min} - |x_{r_{\min}}|$  then  $r_{\min} \leftarrow r$  ;
15      end
16    end
17     $p \leftarrow r_{\min} - |x_{r_{\min}}| + 1$ ;
18    if  $p = i$  then  $t \leftarrow$  text position of some point in  $[x_{r_{\min}}^1, x_{r_{\min}}^2] \times [y_{r_{\min}}^1, y_{r_{\min}}^2]$  ;
19    else
20      report  $(i, j)$  with position  $T[t - j + i..t]$ ;
21       $i \leftarrow p$ 
22    end
23    if  $i \leq j + 1$  then  $R \leftarrow R \cup \{j + 1\}$  ;
24  end
25 end
26 if  $i \leq m$  then report  $(i, m)$  with position  $T[t - m + i..t]$ ;

```

4.2 Analysis

For each value of j , we spend $O(1)$ time per active position. Since there are $O(m)$ active positions at any time, this amounts to $O(m^2)$ time.

The costs of lines 6, 7, and 13, are better charged to each active position r , from its creation to its inactivation. When r is created, we spend $O(m)$ time to compute $\ell_r \leq m$ and x_r (since $|x_r| \leq m$). Later, we can decrease $|x_r|$ several times, performing one range emptiness query in $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$ per decrement of $|x_r|$ (in fact we can go directly to the lowest physical ancestor of x_r rather than to its possibly virtual parent node, since otherwise the range $[x_r^1, x_r^2]$ will not change). Thus, we perform overall $O(m^2)$ emptiness queries, up to m per position r along its life. Maintaining the variables associated with active positions allows us amortizing these costs along the process.

Emptiness queries on \mathcal{G} can be solved in $O(\log^\epsilon g_{rl})$ time and $O(g_{rl})$ space for any constant $\epsilon > 0$ [9]; a recent construction takes $O(g\sqrt{\log g})$ time [2]. The same complexity holds for returning one point in nonempty ranges. The $O(m^2)$ cost charged to positions r is then multiplied by this factor. The rest of the construction time is inherited from the CFG-based index [12]; extending it to RLCFGs does not increase it.

► **Theorem 10.** *Let g_{rl} be the size of a RLCFG generating only $T[1..n]$. Then, for any constant $\epsilon > 0$, we can build in $O(g_{rl} \log^2 n)$ time a data structure of size $O(g_{rl})$ that finds the MEMs of any given pattern $P[1..m]$ in time $O(m^2 \log^\epsilon g_{rl}) \subseteq O(m^2 (\log^\epsilon \delta + \log \log n))$, with an occurrence of each. The query process uses $O(m)$ additional space.*

As mentioned, any CFG can also be used in the theorem. By using an emptiness structure of size $O(g_{rl} \log \log g_{rl})$ [9], we find the MEMs in time $O(m^2 \log \log g_{rl})$.

5 Indexing Locally Consistent Grammars

Before entering into the details of our more sophisticated solution, we must introduce some new concepts. A *locally consistent grammar* is a kind of RLCFG that guarantees that equal substrings of T are covered by similar subtrees of the parse tree, differing in $O(1)$ nonterminals at each level of both subtrees. This has been used to produce grammar-based indices that find all the primary occurrences with only a logarithmic number of cuts in P , thereby obtaining exact pattern searches in time that grows only linearly with m [11, 25, 24]. In this paper we make use of the latest result [24]. We present a lighter informal description; see the original paper for full details.

5.1 The Grammar

We first define the grammar [24, Sec. 3], which is produced level by level, for $O(\log n)$ levels. Let S_k be the sequence of terminals and nonterminals forming level k of the grammar. Let $\ell_k = (4/3)^{\lceil k/2 \rceil - 1}$, and let \mathcal{A}_k be the set of symbols A such that $|\text{exp}(A)| \leq \ell_k$. Those are the symbols that can be grouped to form new nonterminals in level k .

Our string at level 0 is $S_0 = T$. To form the string S_1 , we detect the maximal *runs* of (at least 2) equal consecutive symbols in S_0 that are in $\mathcal{A}_1 = \Sigma$ (Σ is the alphabet of T and also the set of terminals of the RLCFG). For each such run, say of t symbols $a \in \mathcal{A}_1$, we create the rule $A \rightarrow a^t$ and replace the run by the nonterminal A . The resulting sequence after all the runs have been replaced is $S_1 = \text{rle}_{\mathcal{A}_1}(S_0)$, which contains terminals and nonterminals. To form level 2, we define a function π_2 that reorders at random the distinct symbols of S_1 , and use it to define *blocks* in S_1 . Each position $0 < i < |S_1|$ such that

$$\pi_2(S_1[i-1]) > \pi_2(S_1[i]) < \pi_2(S_1[i+1])$$

is the end of a block. We also set ends of blocks at $|S_1|$ and before and after every symbol not in \mathcal{A}_2 (which is still Σ per the formula of ℓ_k , so the runs introduced in S_1 cannot yet be grouped). For each distinct resulting block $S_1[i..j]$ we create a new rule $A \rightarrow S_1[i..j]$ and replace every occurrence of the same block in S_1 by A . The resulting string is called $S_2 = \text{bc}_{\pi_2, \mathcal{A}_2}(S_1)$. The process continues in the same way for odd and even levels:

$$\begin{aligned} S_k &= \text{rle}_{\mathcal{A}_k}(S_{k-1}) && \text{if } k \text{ is odd,} \\ S_k &= \text{bc}_{\pi_k, \mathcal{A}_k}(S_{k-1}) && \text{if } k \text{ is even,} \end{aligned}$$

until we reach $|S_k| = 1$ for some $k = O(\log n)$. The algorithm is Las Vegas type, trying out functions π_k to obtain some desired grammar size, but otherwise any functions π_k yield a correct index. They [24] prove that, in $O(n)$ expected time, a RLCFG of size $O(\delta \log \frac{n}{\delta})$ is obtained, where δ is a lower bound measure based on the substring complexity of T [11]: let T_ℓ be the number of distinct length- ℓ substrings in T , then $\delta = \max\{T_\ell/\ell, \ell > 0\}$. Interestingly,

for every n and δ , there exists a string family that *requires* $\Omega(\delta \log \frac{n}{\delta})$ space (i.e., $\log(n)$ -bit words) to be represented [25]; therefore using space $O(\delta \log \frac{n}{\delta})$ for a grammar (and for an index) is asymptotically optimal for any specific n and δ .

A key property of this grammar is *local consistency*. Let \mathcal{B}_k be the set of all the ends of level- k blocks:

$$\mathcal{B}_k = \{|\text{exp}(S_k[. . j])|, 1 \leq j \leq |S_k|\},$$

where we are extending $\text{exp}(\cdot)$ homomorphically to strings. The cuts of level k that fall inside the substring at $T[i . . j]$ have the following positions inside $T[i . . j]$:

$$\mathcal{B}_k(i, j) = \{p - i + 1, p \in \mathcal{B}_k \cap [i . . j - 1]\}.$$

Local consistency makes the sets $\mathcal{B}_k(i, j)$ and $\mathcal{B}_k(i', j')$ similar if $T[i . . j] = T[i' . . j']$, except at the extremes. Concretely, let $\alpha_k = \lceil 8\ell_k \rceil$, then $\mathcal{B}_k(i + 2\alpha_k, j - \alpha_k) = \mathcal{B}_k(i' + 2\alpha_k, j' - \alpha_k)$.

An additional property of the resulting grammar is that it is *locally balanced*: the subtree of the parse tree rooted at nonterminal A is of height $O(\log |\text{exp}(A)|)$. This is a consequence of the fact that in S_k there are fewer than $1 + 4(j - i + 1)/\ell_{k+1}$ blocks ending inside $T[i . . j]$, and the height of A is never more than the level k of the string S_k where it was created.

5.2 Pattern Searching

Let us now define which cuts of P we need to try out in order to capture all the primary occurrences with this grammar [24, Sec. 4]. Since ends of blocks in $\mathcal{B}_k(i, j)$ correspond to the phrase endings where a primary occurrence $T[i . . j] = P$ can be cut, our set of cutting positions must suffice to capture those possible block endings for all k and for every possible $T[i . . j]$ that matches P . We define

$$\begin{aligned} M_k(i, j) &= \mathcal{B}_k(i, j) \setminus [2\alpha_{k+1} + 1 . . j - i - \alpha_{k+1}] \\ &\cup \{\min(\mathcal{B}_k(i, j) \cap [2\alpha_{k+1} + 1 . . j - i - \alpha_{k+1}])\}, \end{aligned}$$

that is, all the cutting points in the extremes, where the different occurrences of $T[i . . j]$ may differ, and just the first one in the part that is guaranteed to be equal. Over all the levels,

$$M(i, j) = \bigcup_{k \geq 0} M_k(i, j).$$

The key point [24] is that $M(i, j)$ depends only on the content of $T[i . . j]$ (not on its position in T), so we can define $M(P) = M(i, j)$ if $P = T[i . . j]$, and this is the same set for every possible occurrence of P in T . Further, $|M(P)| = O(\log m)$. In operational terms, this means that, at query time, we parse P in $O(m)$ time using the same rules we defined for T , producing a parse tree of height $O(\log m)$ and finding the $O(\log m)$ cutting points $M(P)$.

6 A Faster Solution using Locally Consistent Grammars

The idea to use the index of the preceding section is to exploit the fact that $O(\log(j - i + 1))$ cutting points suffice to find all the primary occurrences of any window $P[i . . j]$. We will then maintain the parse tree of $P[i . . j]$, and the set $M(P[i . . j])$, as we slide it through P , and use it to maintain the number $|R|$ of active positions within $O(\log m)$. We also need more sophisticated mechanisms to avoid the quadratic costs in lines 6, 7, and 13 of Algorithm 2.

6.1 Parsing the Pattern

In this section we show how we maintain the parse tree of $P[i..j]$, or more precisely, the corresponding strings S_0, S_1, \dots , as well as $M(P[i..j])$ and R , as we slide $P[i..j]$ along P . Recall that the height of the parse tree of P is $O(\log m)$ in our locally-balanced grammar.

Maintaining the parse

Assume the parse tree is built for $P[i..j]$ and now we have to increment j . At level $k = 0$, we simply extend S_0 by the symbol $e_0 = P[j + 1]$. This propagates upwards as follows, where l_k is the last symbol of S_k and e_{k-1} has just been added at the end of S_{k-1} .

1. If k is odd (a run-formation level), $l_k = l_{k-1}$ or $l_k \rightarrow l_{k-1}^t$, $|exp(e_{k-1})| \leq \ell_k$, and $l_{k-1} = e_{k-1}$, we find or create a rule $e_k \rightarrow l_{k-1}^{t+1}$ ($t = 1$ if $l_k = l_{k-1}$) and replace l_k by e_k .
2. If k is even (a block-formation level), $l_k = l_{k-1}$ or $l_k \rightarrow \beta l_{k-1}$, $|exp(e_{k-1})| \leq \ell_k$, and $\pi_k(l_{k-1}) > \pi_k(e_{k-1})$, we find or create a rule $e_k \rightarrow \beta l_{k-1} e_{k-1}$ ($\beta = \varepsilon$ if $l_k = l_{k-1}$) and replace l_k by e_k .
3. In any other case, we just append $e_k = e_{k-1}$ at the end of S_k .

We see that insertions at the end of S_{k-1} propagate as new insertions or replacements at the end of S_k . We process those replacements as the deletion of l_k followed by the insertion of e_k at the end. The following are the rules to propagate to S_k the deletion of l_{k-1} .

1. If $l_k = l_{k-1}$, we delete l_k .
2. If $l_k \rightarrow l_{k-1}^t$ is a run, we find or create the rule $e_k \rightarrow l_{k-1}^{t-1}$ (just $e_k = l_{k-1}$ if $t - 1 = 1$) and replace l_k by e_k .
3. If $l_k \rightarrow \beta l_{k-1}$ is a block, we find or create the rule $e_k \rightarrow \beta$ (just $e_k = \beta$ if $|\beta| = 1$) and replace l_k by e_k .

Each of those updates can be carried out in constant time by just maintaining linked lists with the sequence of symbols in each string S_k , perfect hash tables with the existing right-hand sides of the run-formation rules $l_k \rightarrow l_{k-1}^t$, and tries with the right-hand sides of the block-formation rules $l_k \rightarrow \beta l_{k-1}$. In particular, each block-formation nonterminal l_k points to the node in the trie that represents its string βl_{k-1} , and the trie node representing βl_{k-1} stores l_k . With parent pointers in the trie, we have constant-time access to the node of β from the node of βl_{k-1} , and with child pointers we move from βl_{k-1} to $\beta l_{k-1} e_{k-1}$. The children of trie nodes are stored in perfect hash tables to enable downward traversals in constant time. All this can be precomputed in expected time linear in the grammar size.

The case when i increases is symmetric. We start by deleting the first symbol of S_0 , and propagate the update upwards acting on the first symbols f_k at each level k . To handle those operations we need the lists for S_k be doubly-linked, and also to store tries for all the right-hand sides read as $f_k \rightarrow f_{k-1} \beta'$.

The number of updates are actually bounded to $O(1)$ updates per level, and thus to $O(\log m)$ per increase of j or of i . Consider the string $P[i..j] \cdot \$$, where $\$$ is a special symbol for which we assume $\pi_k(\$) = +\infty$ for all k . The parse tree of $P[i..j] \cdot \$$ is then identical to that of $P[i..j]$, just adding $\$$ at the end of every S_k . The strings S_k formed for $P[i..j]$ followed by $\$$ are the same formed for $P[i..j]$ followed by $P[j + 1]$, except for the first $2\alpha_k$ and the last α_k positions of $P[i..j]$ [24, Lem. 3.7]. Therefore, the addition of $P[j + 1]$ can only alter the last $1 + \alpha_k$ positions of $P[i..j + 1]$ in each S_k . Analogously, removing $P[i]$ can only alter the first $2\alpha_k - 1$ positions of $P[i + 1..j]$ in its strings S_k . On the other hand, those $O(\alpha_k)$ positions correspond to only $O(1)$ symbols in S_k [24, Lem. 3.8]. The total amount of work is proportional to the number of updated symbols if we perform them levelwise, on S_0 , then on S_1 , and so on. All the changes then add up to $O(m \log m)$ along the processing of P .

Dealing with unknown symbols

We analyzed the parsing process as if we would always find a known nonterminal for the right-hand sides we modify, but it could be that we have to create new nonterminals that were never formed during the parsing of T .

To handle those cases, we create fresh nonterminals and continue the process normally, removing them when they are no longer needed. An easy way to handle this would be to make the hash tables and tries of right-hand sides dynamic, so we can add and remove elements in the tables and nodes; we will soon sharpen this solution.

We must assign values $\pi_k(e_k)$ to the fresh symbols e_k we create in S_k . We can assign arbitrary values (different from the other π_k values) without affecting correctness: the index works correctly for arbitrary functions π_k , as explained. No matter how we choose the values $\pi_k(e_k)$, we can add $O(m \log m)$ fresh symbols along the whole process, but we can do better.

New symbols e_k may appear in the parsing of $P[i..j]$ even if $P[i..j]$ appears in T , because the parsing of $P[i..j]$ can be different from that of its occurrences in T . However, this can happen only in the first $2\alpha_k$ and the last α_k positions, in S_k . Once the end of e_k falls before position $j - \alpha_k$, and it is after the position $2\alpha_k$, then e_k should have appeared in the parsing of every occurrence of $P[i..j]$ in T [24, Lem 3.7]. Therefore, when we completely incorporate $P[j+1]$ and as a result the end of a fresh symbol e_k of S_k falls behind position $j+1 - \alpha_k$ of $P[i..j+1]$, we know $P[i..j+1]$ cannot appear in T until the position falls behind $i+2\alpha_k$. At this point, then, we can suspend the search (very much as Algorithm 2 does in line 3) and increase i until e_k ends within the leftmost $2\alpha_k$ symbols of $P[i..j+1]$.

This has as a consequence that we can have only $O(1)$ fresh symbols per level, just as $M_{i,j}(P)$, and $O(\log m)$ in total. Instead of making the tries and hash tables dynamic, we can have one extra atomic heap per hash table (the one for the run-length symbols and the one in each trie node) where we can insert/delete the necessary fresh symbols, and they will be processed in constant time. We then retain the $O(m \log m)$ total processing cost.

Maintaining $M(P)$ and R

After we finish updating the parse tree, we collect the first $2\alpha_{k+1}$ positions, the position of the following end of block, and the last α_{k+1} positions, in each list S_k , to form the sets $M_k(P)$. Those are then merged into $M(P)$ and sorted by increasing value. Since $|M(P)| = O(\log m)$, and its values are integers in $[1..m]$, $M(P)$ can be sorted in $O(\log m)$ time with atomic heaps. We then traverse $M(P)$ and the current set R in synchronization, so as to (1) remove the positions of R that are not anymore in $M(P)$, and (2) insert in R the positions that are now in $M(P)$, as long as the position had not been in R before and had been inactivated (this is easily marked in an array of size m). At the end of this process, it always holds that $R \subseteq M(P[i..j])$, and thus $|R| = O(\log m)$. Due to the parsing, an active position r may enter and leave R several times along the process, but this time that will not be an issue.

Overall, we maintain the parsing, $M(P[i..j])$, and R in time $O(m \log m)$. Since all the lines in Algorithm 2 other than 6, 7, and 13, take $O(1)$ time per element in R , the total time spent in those lines adds up to $O(m \log m)$.

6.2 Patricia Tree Searches

Lines 6 and 7 of Algorithm 2 perform $\Theta(m)$ -time searches in P_x and P_y . Since each of the m positions becomes active when $j+1$ reaches it, this amortizes to no less than $\Theta(m^2)$, which is now too high for us. We then resort to a different technique.

Instead of computing ℓ_{j+1} and x_{j+1} inside the main cycle, we will compute them all beforehand, in batch form. We make use of the following result, which was key to obtain subquadratic times in grammar-based indexing.

► **Lemma 11** ([11, Lem 6.5]). *Let \mathcal{S} be a set of strings and assume we have a data structure supporting extraction of any length- ℓ prefix of strings in \mathcal{S} in time $f_e(\ell)$ and computation of a given Karp–Rabin signature κ of any length- ℓ prefix of strings in \mathcal{S} in time $f_h(\ell)$. We can then build a data structure of $O(|\mathcal{S}|)$ words such that, later, given a pattern $P[1..m]$ and τ suffixes Q_1, \dots, Q_τ of P , we find the ranges of strings in (the lexicographically-sorted) set \mathcal{S} prefixed by each Q_i , in $O(m + \tau(f_h(m) + \log m) + f_e(m))$ total time.*

When $\mathcal{S} = \mathcal{X}$ or $\mathcal{S} = \mathcal{Y}$, our access data structure \mathcal{A} provides the required prefix/suffix extraction in time $f_e(\ell) = O(\ell)$. As for Karp–Rabin signatures [20], a result of independent interest is that we can obtain $f_h(\ell) = O(\log \ell)$ time on our grammar, as proved next. We consider the more complicated case of $Y \in \mathcal{Y}$; the case of $X \in \mathcal{X}$ is analogous. Recall we can compute in $O(1)$ time one of $\kappa(S \cdot S')$, $\kappa(S)$, and $\kappa(S')$, from the other two [11, Sec. A.3].

► **Lemma 12.** *The Karp–Rabin signature $\kappa(Y[..\ell])$ of any $Y \in \mathcal{Y}$ can be computed in time $O(\log \ell)$ with our grammar.*

Proof. We build on the same structure \mathcal{A} used for extraction from the root of P_Y . The strings in \mathcal{Y} are concatenations $Y = \text{exp}(B_s) \cdots \text{exp}(B_t)$ of siblings in rules $A \rightarrow B_1 \cdots B_t$ in the grammar tree. The node $v \in P_Y$ of Y stores $\langle v \rangle = B_s$. Let us first assume that $|\text{exp}(B_s)| \geq \ell$, so the signature can be computed on $\text{exp}(B_s)[..\ell]$.

Structure \mathcal{A} is a set of tries on the grammar symbols. The terminals Σ form the trie roots. If $A \rightarrow B_1 \cdots B_t$, then B_1 is the parent of A . If $A \rightarrow B^t$, then B is the parent of A . Any ancestor C of B_s in the tries is a node that descends from B_s by the leftmost path in the parse tree. The structure \mathcal{A} can jump from B_s to any such C in constant time in the tries. Our grammar is locally balanced: there can be only one block ending inside $\text{exp}(B_s[..\ell])$ at level $k = 1 + 2 \log_{4/3}(4\ell)$ [24, Lem. 3.8], and thus the lowest C such that $|\text{exp}(C)| \geq \ell$ has level at most $k + 1$; its height is $d \leq k + 1 = O(\log \ell)$. It can then be found in $O(\log \log \ell)$ time with exponential search on the ancestors of B_s . We then have that $\text{exp}(B_s)[..\ell] = \text{exp}(C)[..\ell]$ and can compute the signature on C instead.

The basic algorithm to compute signatures takes time $O(\log^2 \ell)$ [11, Lem 6.7]. It moves from C towards the leaf L of the parse tree that corresponds to $\text{exp}(C)[\ell]$. Let $C \rightarrow C_1 \cdots C_t$, then it stores every $w_i(C) = |\text{exp}(C_1 \cdots C_i)|$ and every $\kappa_i(C) = \kappa(\text{exp}(C_1 \cdots C_i))$. The algorithm finds, in $O(\log i) \subseteq O(\log \ell)$ time, using exponential search, the C_i that is in the path to L (i.e., $w_{i-1} < \ell \leq w_i$), sets $\ell \leftarrow \ell - w_{i-1}$, collects $\kappa_{i-1}(C)$, and continues by C_i . It composes all those κ values towards L to obtain $\kappa(Y[..\ell])$. In rules $C \rightarrow C_1^t$ it obtains i in constant time but spends $O(\log i)$ time to compute $\kappa_{i-1}(C)$ from the stored $\kappa(\text{exp}(C_1))$.

Instead, an $O(\log n)$ time algorithm [11, Thm. A.3] replaces the exponential searches by a more sophisticated scheme, whose cost is the telescoping sum $\sum_{h=1}^p \log(t_h/t_{h-1}) \leq \log t_p$, where t_h is the number of children (counting $C \rightarrow C_1^t$ as having t children) of the ancestor at distance h of leaf L . In their case, they start from the root, which could have $t_p = n$, but if we start it from a node C , its time is $\log t_p \leq \log |\text{exp}(C)|$. Another component of the cost is the number of times one leaves from heavy paths; this is again $O(\log n)$ in general but just $O(d) = O(\log \ell)$ if we start from the position of C in its heavy path.

It could be, however, that $\text{exp}(C)$ is as long as n . Because it was formed in S_k , however, the children C_i of C belonged to \mathcal{A}_k (only those nonterminals are allowed to form rules in S_k), and thus by definition $|\text{exp}(C_i)| \leq \ell_k$ and $\log |\text{exp}(C_i)| = O(k) = O(\log \ell)$. We can then

find i and compute $\kappa_{i-1}(C)$ in time $O(\log i) \subseteq O(\log \ell)$ with the basic method [11, Lem 6.7] and then continue from C_i , where the more sophisticated technique [11, Thm. A.3] completes the computation in another $O(\log |\text{exp}(C_i)|) \subseteq O(\log \ell)$ time.

In case $|\text{exp}(B_s)| < \ell$, we find the first $s < i \leq t$ such that $w_i(A) \geq \ell$, and compute instead the signature of $\text{exp}(B_i)[\cdot \ell - w_{i-1}(A)]$, to then compose it with the stored values $\kappa_{s-1}(A)$ and $\kappa_{i-1}(A)$ to obtain the final signature $\kappa(Y[\cdot \ell]) = \kappa(\text{exp}(A)[w_{s-1}(A) + 1 \dots w_{s-1}(A) + \ell])$. ◀

Batched searches

The m searches for all the values ℓ_r , $1 \leq r \leq m$, correspond to searching $P_{\mathcal{Y}}$ for every suffix $P[r+1..]$. Note that Lemma 11 does not yield the node v of line 6, but rather its corresponding range $[v^1, v^2]$. By performing a lowest common ancestor (LCA) query on $P_{\mathcal{Y}}$ from the v^1 th and v^2 th leaves, we obtain $v = \text{lca}(v^1, v^2)$ (identifying leaves with their ranks). The answer is indeed v if $|v| = m - r$; if $m - r < |v|$ the answer is the virtual node of string length $m - r$ on the edge of $P_{\mathcal{Y}}$ that leads to v . Linear-space LCA data structures that are built in linear time and answer lca in $O(1)$ time are well known [3].

The problem is that Lemma 11 works only if $P[r+1..]$ actually prefixes some string in \mathcal{Y} . Otherwise, unlike classical trie searching, it does not even yield the maximum prefix of $P[r+1..]$ that prefixes some string in \mathcal{Y} . We will resort to, essentially, binary searching for those longest prefixes using Lemma 11 as an internal tool.

Assume m is a power of 2 for simplicity; the general case is easily deduced. We define sets $\mathcal{Q}_{a,b}$ of positions, containing those values r such that $P[r+1..a]$ is known to be a prefix in \mathcal{Y} and $P[r+1..b+1]$ is known not to be a prefix in \mathcal{Y} (the first condition is assumed to hold if $r+1 > a$). We start with the set $\mathcal{Q}_{1,m} = \{1, \dots, m\}$. To process a set $\mathcal{Q}_{a,b}$, we search for all the $\tau = |\mathcal{Q}_{a,b}|$ suffixes $\{P[r+1..c], r \in \mathcal{Q}_{a,b}\}$ of $P[\cdot c]$ using Lemma 11, with $c = (a+b+1)/2$. The values r where $P[r+1..c]$ is found are moved to $\mathcal{Q}_{c,b}$, and the others to $\mathcal{Q}_{a,c-1}$ (if $r+1 > c$, then $P[r+1..c] = \varepsilon$, so we can directly move r to $\mathcal{Q}_{c,b}$ without searching for it). We will associate the node $v_{r,c} \in P_{\mathcal{Y}}$ to those values r for which $P[r+1..c]$ is found in \mathcal{Y} ; those not found retain their previous node $v_{r,*}$ (in the beginning all such nodes are $v_{r,r}$ and equal to the root of $P_{\mathcal{Y}}$).

Note that the values $b - a + 1$ halve as the elements in $\mathcal{Q}_{a,b}$ are separated into two sets. Any value r is then moved $O(\log m)$ times until it ends up in a set of the form $\mathcal{Q}_{c,c}$; at this point we know that the longest prefix of $P[r+1..]$ that is also a prefix in \mathcal{Y} is $P[r+1..c]$, and also know its node $v_{r,c}$.

The cost of using Lemma 11 has two parts. The cost $f_h(m) + \log m = O(\log m)$ can be charged to each of the τ suffixes sought, and there is an additional global cost of $m + f_e(m) = O(m)$. Since every suffix $P[r+1..]$ participates $O(\log m)$ times in the lemma, the first cost adds up to $O(m \log^2 m)$ over all the m positions r . The second part is potentially very large, however: the suffixes in $\mathcal{Q}_{a,b}$ may start well ahead of a , thus the pattern is $P[\cdot c]$, not $P[a..c]$; a simple application of the lemma would lead to a quadratic cost again.

Smarter substring extractions

To reduce this time, we consider where the $O(m)$ cost in Lemma 11 comes from. A first part refers to the time needed to compute the Karp-Rabin signatures for all the suffixes in $\mathcal{Q}_{a,b}$. This cost is easily maintained within $O(m)$ overall because we can compute the signatures $\kappa(P[r+1..])$, for all $1 \leq r \leq m$, in a single pass over P , and then any $\kappa(P[r+1..j])$ is obtained in constant time from $\kappa(P[r+1..])$ and $\kappa(P[j+1..])$.

The second part of the $O(m)$ cost corresponds to the time $f_e(m)$ to verify the longest suffix among those that passed some previous filters; the rest of the verification is built on that extracted suffix. Let $P[r+1..c]$ be the longest candidate suffix. If $r+1 > a$, we extract the actual suffix $P'[..c-r]$ regularly in time $f_e(c-r) = O(b-a)$ with \mathcal{A} , because P' starts at the root of $P_{\mathcal{Y}}$ and thus it belongs to \mathcal{Y} .

Otherwise, $r+1 \leq a$ and thus $P[r+1..a]$ had been successfully matched before and we have its node $v_{r,a} \in P_{\mathcal{Y}}$. As mentioned, the process of Lemma 11 performs several checks before performing the final extraction of the longest suffix surviving the checks. We will add a new check to those, which can only speed up the process: the candidate node v for $P[r+1..c]$ must now descend from $v_{r,a}$ in order to be further considered. The descent check is performed in constant time by comparing the leaf range $[v^1, v^2]$ of v with that of $v_{r,a}$. If v passes the test, we know that $P'[..c-r]$ does start with $P[r+1..a]$, and then only need to extract $P'[a-r+1..c-r]$ from the text, which is of length $O(b-a)$.

This time, however, the string to extract does not start at the root of $P_{\mathcal{Y}}$, and thus it requires a random access to T .¹ Recall, as in Lemma 12, that the strings in \mathcal{Y} are concatenations $Y = \text{exp}(B_s) \cdots \text{exp}(B_t)$ of consecutive siblings in rules $A \rightarrow B_1 \cdots B_t$ in the grammar tree (if $A \rightarrow B^t$, then the node stores $\langle v \rangle = B^{[t-1]}$ and we have $B_s = B$). Let us first assume that $|\text{exp}(B_s)| \geq c$, so the substring to extract is within $\text{exp}(B_s)[..c]$. We use again the structure \mathcal{A} , now to extract the string in time $O(b-a + \log c)$.

Once again, we can search in $O(\log \log c)$ time for the lowest descendant C of B_s such that $|\text{exp}(C)| \geq c$; its height is $d = O(\log c)$ because the grammar is locally balanced. Since $\text{exp}(B_s)[..c] = \text{exp}(C)[..c]$, we descend from C to the leaf L in the parse tree representing $\text{exp}(C)[a-r+1..c]$. Using the same techniques as in Lemma 12, the time is $O(d) = O(\log c)$. From L , $\text{exp}(C)[a-r+1..c-r] = P'[a-r+1..c-r]$ is extracted in time $O(c-a) = O(b-a)$.

In case $|\text{exp}(B_s)| < c$, the node C is not a descendant of B_s but we use $C = A$ instead. Given the limitation on $|\text{exp}(B_s)|$, the height of B_s is $O(\log c)$, and so is the height of A .

The $O(\log c) = O(\log m)$ cost can be charged to the suffix sought, which adds up to $O(m \log^2 m)$ over the $O(\log m)$ times each suffix may use the lemma. The $O(b-a)$ terms add up to $O(m)$ per level of sets $Q_{a,b}$ (a level corresponding to a difference $b-a+1$). Since all the ranges (a,b) of a level are disjoint (level ℓ partitions $(1,m)$ into 2^ℓ ranges of size $m/2^\ell$), the $b-a$ values add up to $O(m)$ per level. Since there are $O(\log m)$ levels, that part of the cost adds up to $O(m \log m)$.

We similarly compute the nodes x_r for every $P[..r]^{rev}$ on $P_{\mathcal{X}}$. While in line 7 of Algorithm 2 we search only for $P[i..r]^{rev}$ because we are not interested in positions before i , this time we precompute all the values in advance for the smallest $i = 1$. Later, when i increases, we will move to the required ancestors of x_r in line 13.

Overall, the Patricia trie searches execute lines 6 and 7 in $O(m \log^2 m)$ total time.

6.3 Emptiness Queries

In line 13, we perform one range emptiness query every time we decrement $|x_r|$ for some r ; this amounts to $O(m^2)$ emptiness queries, which we cannot afford now. We will instead use a faster method based on orthogonal range successor queries: given a range $[x_r^1, x_r^2] \times [y_r^1, y_r^2]$,

¹ It is tempting to say that, since we had already matched $P[r+1..a]$ from the root of $P_{\mathcal{Y}}$, we could somehow save the state of that extraction so as to continue without paying the overhead of the random access. However, we might have never extracted the text of the node $v_{r,a}$ explicitly; its verification may have been carried out as a subproduct of reading longer suffix, starting before $r+1$.

we can find the largest value $x_< \leq x_r^1$ such that $[x_<, x_r^2] \times [y_r^1, y_r^2]$ contains a point, and the smallest value $x_> \geq x_r^2$ such that $[x_r^1, x_>] \times [y_r^1, y_r^2]$ contains a point. Those queries can run in $O(\log^\epsilon g)$ time on a grid with g points, using an $O(g)$ -space data structure, for any constant $\epsilon > 0$ defined at construction [35]; construction time can be made $O(g\sqrt{\log g})$ [2].

The lowest ancestor x of x_r containing some point in $[x^1, x^2] \times [y_r^1, y_r^2]$ must then hold $x^1 \leq x_<$ or $x^2 \geq x_>$. In the first case, it is $v_1 = \text{lca}(x_<, x_2)$; in the second, it is $v_2 = \text{lca}(x_1, x_>)$. Both v_1 and v_2 are ancestors of x_r , and thus of each other. The correct node x is then the lowest of v_1 and v_2 , which is known from the leaf ranges stored at the nodes.

With this query, line 17 of Algorithm 2 does not cycle; it just performs one $O(\log^\epsilon g)$ -time step. It can then be counted as one of the $O(|R|)$ operations performed in each cycle j . Since there are $O(m \log m)$ such operations, this one adds $O(m \log m \log^\epsilon g)$ to the total cost.

6.4 The Final Result

Our time complexities then add up to $O(m \log m (\log m + \log^\epsilon g))$ for a grammar of size g . Since in our case $g = O(\delta \log \frac{n}{\delta})$, we can write the time as $O(m \log m (\log m + \log^\epsilon \delta + \log \log n))$. The construction time of all the data structures we use is dominated by the $O(n \log n)$ expected time needed to build the Karp-Rabin hashes of Lemma 11 [11, Sec. 6.6] (the grammar is built in $O(n)$ expected time, see [24, Cor. 3.15]).

► **Theorem 13.** *Let $T[1..n]$ have substring complexity δ . Then, for any constant $\epsilon > 0$, we can build in $O(n \log n)$ expected time a data structure of size $O(\delta \log \frac{n}{\delta})$ that finds the MEMs of any given pattern $P[1..m]$ in time $O(m \log m (\log m + \log^\epsilon \delta + \log \log n)) \subseteq O(m \log m (\log m + \log^\epsilon n))$, with an occurrence of each. The query process uses $O(m)$ additional space.*

We have assumed $m \leq n$, but it could be the other way in some applications. Since in this case no substring longer than n will be matched inside P , we can run $O(m/n)$ iterations finding the MEMs of $P[1..2n]$, $P[n..3n]$, $P[2n..4n]$, and so on, avoiding repeated MEMs in the output. The total cost would then be $O(m \log^2 n)$ and the query space would be $O(n)$.

7 Conclusions

We have obtained improved results, including the first subquadratic algorithm, to find MEMs on parsing-based indices, which are the most promising in terms of space for highly repetitive text collections. While suffix-based indices can preprocess $T[1..n]$ to find the MEMs of $P[1..m]$ in T in time $O(m \log \log n)$, their space is $\Omega(r)$, where r (the number of runs in the BWT of T) is not such a strong measure of repetitiveness [34]. Our first result is a data structure of size $O(g_{rl})$, where g_{rl} is the size of the smallest RLCFG that generates T . This is currently the best possible space for any structure able to access T with relevant time guarantees [34]. Our structure finds the MEMs in $O(m^2 \log^\epsilon n)$ time for any constant $\epsilon > 0$. This is very similar to the time of previous work [18], which could also run in $O(g_{rl})$ space. Within $O(\delta \log \frac{n}{\delta})$ space, we obtain the first subquadratic time, $O(m \log m (\log m + \log^\epsilon n))$, on a particular RLCFG that has local consistency properties. This space is optimal for every n and δ , though g_{rl} is always $O(\delta \log \frac{n}{\delta})$ and can be $o(\delta \log \frac{n}{\delta})$ in some text families [25].

A challenge for future work is to extend our results to finding k -MEMs, which are the maximal substrings of P that appear at least k times in T , for k given at query time. The basic Algorithm 1 is easily modified to find the k -MEMs in $O(m)$ time, but running in compressed space is more costly. In the extended version we will show how find k -MEMs, changing the $\log^\epsilon n$ terms to $\log^{2+\epsilon} n$, by building on grammar-based indices that can count

the number of occurrences associated with a set of primary occurrences [11]. The space also increases: the $O(g_{rl})$ space becomes $O(g)$ ($g \geq g_{rl}$ being size of the smallest CFG) and the $O(\delta \log \frac{n}{\delta})$ space becomes $O(\gamma \log \frac{n}{\gamma})$ ($\gamma \geq \delta$ being the size of a string attractor of T [22]).




Our techniques are presented on a particular locally consistent grammar [24] that yields the best complexities, but they would work on others too. We plan to implement them on practical constructions of CFGs [12] built with RePair [27] or of locally consistent grammars based on induced suffix sorting [14, 36]. Further, even without having theoretical guarantees, the algorithm for arbitrary RLCFGs will probably be competitive if implemented on Lempel-Ziv based indices [26, 16], which are considerably smaller than those based on grammars.

References

- 1 H. Bannai, T. Gagie, and T. I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020.
- 2 D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.
- 3 M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
- 4 P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. S. Rao, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.
- 5 C. Boucher, O. Cvacho, T. Gagie, J. Holub G. Manzini, G. Navarro, and M. Rossi. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–72, 2021.
- 6 C. Boucher, T. Gagie, T. I., D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. 31th Data Compression Conference (DCC)*, pages 193–202, 2021.
- 7 M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 8 M. Cáceres and G. Navarro. Faster repetition-aware compressed suffix trees based on block trees. *Information and Computation*, 285B:article 104749, 2022.
- 9 T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.
- 10 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
- 11 A. R. Christiansen, M. B. Ettiienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):article 8, 2020.
- 12 F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.
- 13 M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.
- 14 D. Díaz-Domínguez, G. Navarro, and A. Pacheco. An LMS-based grammar self-index with local consistency properties. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 100–113, 2021.
- 15 A. Farruggia, T. Gagie, G. Navarro, S. J. Puglisi, and J. Sirén. Relative suffix trees. *The Computer Journal*, 61(5):773–788, 2018.
- 16 H. Ferrada, D. Kempa, and S. J. Puglisi. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8, 2018.
- 17 T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.
- 18 Y. Gao. Computing matching statistics on repetitive texts. In *Proc. 32nd Data Compression Conference (DCC)*, pages 73–82, 2022.

- 19 D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 20 R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 2:249–260, 1987.
- 21 D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013, 2020.
- 22 D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 827–840, 2018.
- 23 J. C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.
- 24 T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ -optimal space. In *Proc. 15th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–103, 2022.
- 25 T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023.
- 26 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
- 27 J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
- 28 V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
- 29 V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
- 30 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 31 E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- 32 D. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
- 33 G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.
- 34 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.
- 35 Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*, pages 271–282, 2012.
- 36 D. Nunes, F. Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. Grammar compression by induced suffix sorting. *ACM Journal of Experimental Algorithmics*, 27:article 1.1, 2022.
- 37 E. Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
- 38 M. Rossi, M. Oliva, B. Langmead, T. Gagie, and C. Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.
- 39 L. M. S. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):article 53, 2011.
- 40 I. Tatarnikov, A. S. Farahani, S. Kashgouli, and T. Gagie. MONI can find k-MEMs. *CoRR*, 2202.05085, 2022. [arXiv:2202.05085](https://arxiv.org/abs/2202.05085).
- 41 P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.

L-Systems for Measuring Repetitiveness

Gonzalo Navarro   

Department of Computer Science, University of Chile, Santiago, Chile
Centre for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Cristian Urbina   

Department of Computer Science, University of Chile, Santiago, Chile
Centre for Biotechnology and Bioengineering (CeBiB), Santiago, Chile

Abstract

In order to use them for compression, we extend L-systems (without ε -rules) with two parameters d and n , and also a coding τ , which determines unambiguously a string $w = \tau(\varphi^d(s))[1 : n]$, where φ is the morphism of the system, and s is its axiom. The length of the shortest description of an L-system generating w is known as ℓ , and it is arguably a relevant measure of repetitiveness that builds on the self-similarities that arise in the sequence.

In this paper, we deepen the study of the measure ℓ and its relation with a better-established measure called δ , which builds on substring complexity. Our results show that ℓ and δ are largely orthogonal, in the sense that one can be much larger than the other, depending on the case. This suggests that both mechanisms capture different kinds of regularities related to repetitiveness.

We then show that the recently introduced NU-systems, which combine the capabilities of L-systems with bidirectional macro schemes, can be asymptotically strictly smaller than both mechanisms for the same fixed string family, which makes the size ν of the smallest NU-system the unique smallest reachable repetitiveness measure to date. We conclude that in order to achieve better compression, we should combine morphism substitution with copy-paste mechanisms.

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

Keywords and phrases L-systems, String morphisms, Repetitiveness measures, Text compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.25

Funding *Gonzalo Navarro*: Funded by Basal Funds FB0001 and Fondecyt Grant 1-200038, Chile.
Cristian Urbina: Funded by Basal Funds FB0001 and ANID National Doctoral Scholarship – 21210580, Chile.

1 Introduction

In areas like Bioinformatics, it is often necessary to handle big collections of highly repetitive data. For example, two human genomes share 99.9% of their content [23]. In another scenario, for sequencing a genome, one extracts so-called *reads* (short substrings) from it, with a “coverage” of up to 100X, which means that each position appears on average in 100 reads.¹ There is a need in science and industry to maintain those huge string collections in compressed form. Traditional compressors based exclusively on *Shannon’s entropy* are not good for handling repetitive data, as they only exploit symbol frequencies when compressing. Finding good measures of repetitiveness and also compressors exploiting this repetitiveness has then become a relevant research problem.

¹ <https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html>



A strong theoretical measure of string repetitiveness introduced by Kociumaka et al. [12] is δ , based on the substring complexity function. This measure has several nice properties: it is computable in linear time, monotone, resistant to string edits, insensitive to simple string transformations, and it lower-bounds almost every other theoretical or *ad-hoc* repetitiveness measure considered in the literature. Further, although $O(\delta)$ space is unreachable, there exist $O(\delta \log(n/\delta))$ -space representations supporting efficient pattern matching queries [12, 11], and this space is tight: no $o(\delta \log(n/\delta))$ -space representation can exist [12].

The idea that δ is a sound lower bound for repetitiveness is reinforced by the fact that it is always $O(b)$, where b is the size of the smallest *bidirectional macro scheme* generating a string [26]. Those macro schemes arguably capture every possible way of exploiting copy-paste regularities in the sequences. Some very recent works [19], however, explore other sources of repetitiveness, in particular *self-similarity*, and are shown to break the lower bound of δ .

The simplest of those schemes, which reuse the name *L-system* for simplicity [19], builds upon Lindenmayer systems [15, 16], in particular on the variant called CPD0L-systems. A CPD0L-system describes the language of the images, under a coding τ , of the powers of a non-erasing morphism φ starting from an string s (called the *axiom*), that is, the set $\{\tau(\varphi^i(s)) \mid i \geq 0\}$. The L-systems extend CPD0L-systems with two parameters d and n , so as to unambiguously determine the string $w = \tau(\varphi^d(s))[1 : n]$. The size of the shortest description of an L-system generating w in this fashion is called ℓ . Intuitively, ℓ works as a repetitiveness measure because any occurrence of the symbol a at level i expands to the same string at level $i + j$ for every j .

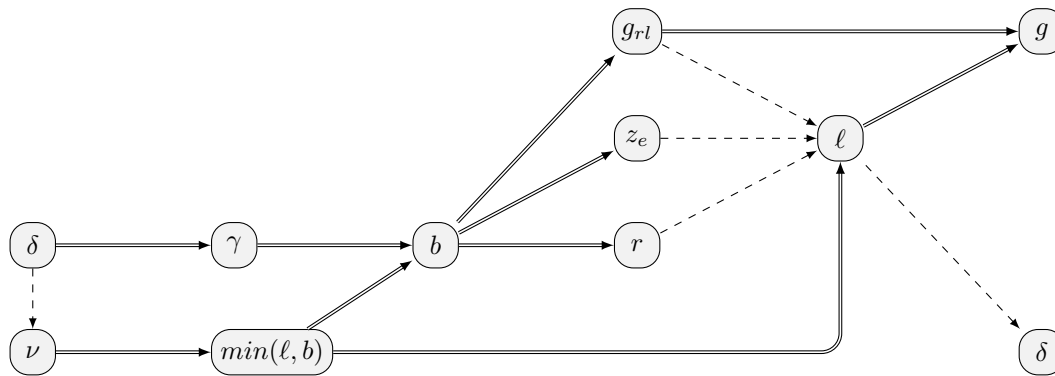
Since ℓ is a reachable measure of repetitiveness (because the L-system is a representation of w of size $O(\ell)$), there are string families where $\delta = o(\ell)$. Intriguingly, it has been shown [19] that there are other string families where $\ell = o(\delta)$, so (1) both measures are not comparable and (2) the lower bound δ does not capture this kind of repetitiveness. On the other hand, it is shown that $\ell = O(g)$, where g is the size of the smallest deterministic context-free grammar generating only w . This comparison is relevant because L-systems are similar to grammars, differing in that they have no terminal symbols, so their expansion must be explicitly stopped at level d and then possibly converted to terminals with τ .

Grammars provide an upper bound to repetitiveness that is associated with well-known compressors, so this upper bound makes ℓ a good measure of repetitiveness.

A more complex scheme that was also introduced [19] are NU-systems, which combine the power of L-systems with bidirectional macro schemes. The measure ν , defined as the size of the smallest NU-system generating w , naturally lower bounds both ℓ and b . The authors could not, however, find string families where ν is asymptotically better than both ℓ and b , so it was unclear if NU-systems are actually better than just the union of both underlying schemes.

In this paper we deepen the study of the relations between these new intriguing measures and more established ones like δ and g . Our results are as follows:

1. We show that ℓ can be much smaller than δ , by up to a \sqrt{n} factor, improving a previous result [19] and refuting their conjecture that $\ell = \Omega(\delta/\log n)$.
2. On the other hand, we expose string families where ℓ is larger than the output of several repetitiveness-aware compressors like the size g_{rl} of the smallest run-length context-free grammar, the size z_e of the greedy LZ-End parse [13], and the number of runs r in the Burrows-Wheeler Transform of the string [2]. We then conclude that ℓ is uncomparable to almost all measures other than g , which suggests that the source of repetitiveness it captures is largely orthogonal to the typical cut-and-paste of macro schemes.



■ **Figure 1** Asymptotic relations between ℓ , ν , and other repetitiveness measures. A double black arrow from v_1 to v_2 means that it always holds that $v_1 = O(v_2)$, and there exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$.

3. We introduce a string family where ν is asymptotically strictly smaller than both ℓ and b , which shows that NU-systems are indeed relevant and positions ν as the unique smallest reachable repetitiveness measure to date that captures both kinds of repetitiveness in non-trivial ways.
4. We study various ways of simplifying L-systems and show that, in most cases, we end up with a weaker repetitiveness measure. We also study some of those weaker variants of ℓ , which can be of independent interest.

Overall, our results contribute to understanding how to measure repetitiveness and how to exploit it in order to build better compressors. We summarize the state of ℓ and ν after this work in Figure 1.

2 Basic concepts

In this section we explain the basic concepts needed to understand the rest of the paper, from strings and morphisms to relevant repetitiveness measures.

2.1 Strings

An *alphabet* is a finite set of *symbols* and is usually denoted by Σ . A (finite) *string* w is a finite sequence $w[1]w[2] \dots w[n]$ of symbols where $w[i] \in \Sigma$ for $i \in [1, n]$, and its *length* is denoted by $|w| = n$. The *empty string*, whose length is 0, is denoted by ε . The set of all possible finite strings over Σ is denoted by Σ^* . If $x = x[1] \dots x[n]$ and $y = y[1] \dots y[m]$, the concatenation operation $x \cdot y$ (or just xy) yields the string $x[1] \dots x[n]y[1] \dots y[m]$. Let $w = xyz$. Then y (resp. x, z) is a *substring* (resp. *prefix, suffix*) of w . It is *proper* if it is not equal to w , and *non-trivial* if it is distinct from ε and w . The notation $w[i : j]$ stands for the substring $w[i] \dots w[j]$ if $i \leq j$, and ε otherwise. We also use the conventions $w[i : j] = w[1 : j]$ if $i < 1$, $w[i : j] = w[i : n]$ if $j > n$, and $w[i : j] = \varepsilon$ if $i > n$ or $j < 1$. Other convenient notations are $w[: i] = w[1 : i]$ and $w[i : :] = w[i : |w|]$ for prefixes and suffixes, respectively.

A (right) *infinite string* \mathbf{w} (we use boldface to emphasize them) over an alphabet Σ is a mapping from \mathbb{Z}^+ to Σ , and its length is called ω , which is greater than n for any $n \in \mathbb{Z}^+$. It is possible to define the concatenation $x \cdot \mathbf{y}$ if x is finite and \mathbf{y} infinite. The concepts of substring, prefix, and suffix carry over to infinite strings, with proper prefixes always being finite and suffixes always being infinite. The notations $\mathbf{w}[i]$, $\mathbf{w}[i : j]$, $\mathbf{w}[: i]$, and $\mathbf{w}[i : :] = \mathbf{w}[i]\mathbf{w}[i + 1] \dots$ also carry over to infinite strings.

2.2 Morphisms

The set Σ^* together with the (associative) concatenation operator and the (identity) string ε form a *monoid* structure $(\Sigma^*, \cdot, \varepsilon)$. A *morphism* on strings is a function $\varphi : \Sigma_1^* \rightarrow \Sigma_2^*$ satisfying $\varphi(x \cdot y) = \varphi(x) \cdot \varphi(y)$ for all x, y (i.e., a function preserving the monoid structure), where Σ_1 and Σ_2 are alphabets. To define a morphism on strings, it is sufficient to define how it acts over the symbols in its domain. The pairs $(a, \varphi(a))$ for $a \in \Sigma_1$, usually denoted $a \rightarrow \varphi(a)$, are called the *rules* of the morphism, and there are $|\Sigma_1|$ of them. If $\Sigma_1 = \Sigma_2$, then the morphism is called an *endomorphism*.

Let $\varphi : \Sigma_1^* \rightarrow \Sigma_2^*$ be a morphism on strings. Some useful definitions are *width*(φ) = $\max_{a \in \Sigma_1} |\varphi(a)|$ and *size*(φ) = $\sum_{a \in \Sigma_1} |\varphi(a)|$. A morphism is *non-erasing* if $\forall a \in \Sigma_1, |\varphi(a)| > 0$, *expanding* if $\forall a \in \Sigma_1, |\varphi(a)| > 1$, *k-uniform* if $\forall a \in \Sigma_1, |\varphi(a)| = k > 1$, and it is a *coding* if $\forall a \in \Sigma_1, |\varphi(a)| = 1$ (sometimes called a 1-uniform morphism).

Let $\varphi : \Sigma^* \rightarrow \Sigma^*$ be an endomorphism. Then φ is *prolongable* on a symbol a if $\varphi(a) = ax$ for some string $x \neq \varepsilon$. If this is the case, then for each i, j with $0 \leq i \leq j$, it holds that $\varphi^i(a)$ is a prefix of $\varphi^j(a)$, and $\mathbf{x} = \varphi^\omega(a) = ax\varphi(x)\varphi^2(x)\dots$ is the unique infinite fixed-point of φ starting with the symbol a . An infinite string $\mathbf{w} = \varphi^\omega(a)$ that is the fixed-point of a morphism is called a *purely morphic word*, its image under a coding $\mathbf{x} = \tau(\mathbf{w})$ is called a *morphic word*, and if the morphism φ is *k-uniform*, then \mathbf{x} is said to be *k-automatic*.

2.3 Repetitiveness measures

A *repetitiveness measure* μ is a function that arguably captures the degree of *repetitiveness* of strings. Repetitiveness is an intuitive and elemental concept, yet is still subject to debate. In general, a repetitive string is understood as one containing many copies of the same substrings. The more repetitive is a string w , the smaller the value $\mu(w)$ should be.

If we can represent every string $w[1 : n]$ within space $O(\mu(w))$ (where the asymptotics refer to n), then we say the measure μ is *reachable*. Space is usually measured in $\Theta(\log n)$ -bit words following the conventions of the *transdichotomous RAM model of computation*. Hence, $O(\mu(w))$ space means $O(\mu(w) \log n)$ bits. We can represent any symbol in the alphabet of $w[1 : n]$ using a constant number of words as long as $|\Sigma| = O(n^d)$ for some $d \geq 0$.

A repetitiveness measure u_1 is *smaller* or *lower-bounds* another measure u_2 if $u_1(w) = O(u_2(w))$ for every $w[1 : n] \in \Sigma^*$. If, in addition, there is an infinite string family $\mathcal{F} \subseteq \Sigma^*$ where $u_1(w) = o(u_2(w))$ for every $w[1 : n] \in \mathcal{F}$, we say that u_1 is *strictly smaller* or *strictly lower-bounds* u_2 . Two repetitiveness measures u_1 and u_2 are *equivalent* if each one is smaller than the other, and *uncomparable* if none is (i.e., $u_1 = o(u_2)$ on a string family \mathcal{F}_1 and $u_2 = o(u_1)$ on another string family \mathcal{F}_2).

In the following, we explain the most relevant repetitiveness measures to be considered in the rest of the paper. For a more in-depth review, see a recent survey [17].

Grammar-based measures

There exist several compressors, and measures of repetitiveness associated with their size, that build on context-free grammars.

A *straight-line program* (SLP) is a deterministic context-free grammar G in Chomsky Normal Form whose language is a singleton $\{w\}$. We denote the string generated by the SLP as $\text{exp}(G) = w$, and extend this notation to the unique strings generated by the non-terminals of the grammar. The measure g is defined as the size of the smallest SLP G generating w . Finding the smallest SLP is an NP-complete problem [3], although there exist algorithms providing log-approximations [7, 24].

A measure based on context-free grammars that strictly lower-bounds g is g_{rl} , the size of the smallest *run-length* SLP (RLSLP) generating w [20]. RLSLPs allow constant-size rules of the form $A \rightarrow B^n$ for $n > 1$, which can make a noticeable difference in some string families like $\{\mathbf{a}^n \mid n \geq 0\}$, where $g = \Theta(\log n)$, but $g_{rl} = O(1)$.

A *collage system* is an RLSLP that, in addition, supports rules of the form $A \rightarrow B[i : j]$ for some $i, j \in [1, |\exp(B)|]$. These mean that $\exp(A) = \exp(B)[i : j]$. The size c of the smallest *collage system* [10] strictly lower-bounds g_{rl} .

Parsing-based measures

A *parsing* of size k produces a factorization of a string w into non-empty *phrases*, i.e., $w = w_1 w_2 \dots w_k$ where $w_i \in \Sigma^+$ for $i \in [1, k]$. Several compressors work by parsing w in a way that storing summary information about the phrases enables recovering w .

The *Lempel-Ziv* (LZ) parsing processes a string greedily from left to right, always forming the longest phrase that has a copy (called a source) starting inside some previous phrase or forming an *explicit* phrase of length 1 otherwise [14]. The source can overlap the new phrase. The LZ-no parsing, instead, does not allow the source overlap the new phrase. The LZ-end parsing [13] requires, in addition, that the source ends at a previous phrase boundary. All of these parsings can be constructed in linear time, and their number of phrases are denoted by z , z_{no} , and z_e , respectively. While z and z_{no} are optimal among the parsings satisfying their respective conditions, this is not always the case for z_e . The optimal factorization where each phrase w_{i+1} appears as a suffix of $w_1 \dots w_j$ for some $j \leq i$ is denoted by z_{end} . Because of the optimality of z , z_{no} , and z_{end} , it holds that $z \leq z_{no} \leq z_{end} \leq z_e$ for every string.

A *bidirectional macro scheme* (BMS) [26] is any parsing where each phrase of length greater than 1 has a copy starting at a different position in such a way that the original string can be recovered following these pointers (assuming that the phrases of length 1 store their symbol explicitly). The measure $b(w)$ is defined as the size of the smallest BMS for w . It strictly lower-bounds all the other reachable repetitiveness measures [18], except for the ones we focus on in this paper, ℓ and ν [19]. On the other hand, b is NP-hard to compute [5].

Another interesting parsing-based measure is the size of the greedy *lexicographic parsing* of w , denoted as $v(w)$ [18]. This parsing processes w from left to right, taking as the next phrase the longest common prefix between the unprocessed part of the string and a lexicographically smaller suffix of the processed part (a unique symbol $\$$, smaller than the others, is assumed to exist at the end of w). It forms an explicit phrase of length one if the longest common prefix is empty or no predecessor exists.

Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) [2] is a reversible transformation that usually makes a string more compressible. It is obtained by concatenating the last symbols of the sorted suffixes of $w\$$, where $\$$ is a symbol smaller than any symbol appearing in w . The BWT tends to produce long runs of the same symbol when a string is repetitive, and these (maximal) runs can be compressed into one symbol in the alphabet $\{(a, k) \mid a \in \Sigma, k \in [1, n]\}$ using *run-length encoding* (*rle*). A repetitiveness measure based on this idea is defined as $r(w) = |\text{rle}(\text{BWT}(w))|$. Although r is not ideal as a repetitiveness measure [6], its size can be bound in terms of z [8]. It has many practical applications representing repetitive sequences in Bioinformatics because of its indexing power [4].

String attractors

Kempa and Prezza [9] introduced the notion of *string attractor* as a unifying framework and lower bound for dictionary-based compressors.

Let w be a string of length n . A string attractor for w is a set of positions $\Gamma \subseteq [1, n]$ such that for each substring $w[i : j]$ of w , there exist integers $i', j' \in [1, n]$ and $k \in \Gamma$, such that $w[i : j] = w[i' : j']$ and $i' \leq k \leq j'$. That is, every substring of w has a copy covering a position in Γ . The measure $\gamma(w)$ is defined as the size of the smallest string attractor for w .

Computing γ is an NP-complete problem. The measure γ is a lower bound to b and is also strictly smaller than b when considering the infinite family of Thue-Morse words [1]. On the other hand, it is unknown if γ space or even $o(\gamma \log(n/\gamma))$ space is reachable.

Substring complexity

Let $F_w(k)$ be the set of distinct substrings of w of length k . The *complexity function* of w is defined as $P_w(k) = |F_w(k)|$. Kociumaka et al. [12] introduced a repetitiveness measure based on the complexity function, defined as $\delta(w) = \max\{P_w(k)/k \mid k \in [1..|w|]\}$.

The measure δ has several nice properties: it is computable in linear time, monotone, insensitive to reversals, resistant to small edits on w , can be used to construct $O(\delta \log(n/\delta))$ -space representations supporting efficient access and pattern matching queries [12, 11], and is a lower bound to almost every other theoretical or *ad-hoc* repetitiveness measure considered in the literature, including γ . On the other hand, $o(\delta \log(n/\delta))$ space is unreachable [12].

3 The measure ℓ

The class of *CPD0L-systems* is a variant of the original *L-systems*, the parallel grammars without terminals defined by Aristid Lindenmayer to model cell divisions in the growth of plants and algae [15, 16].

Formally, a CPD0L-system is a 4-tuple $L = (\Sigma, \varphi, \tau, s)$, where Σ is the *alphabet*, φ is the set of *rules* (a non-erasing endomorphism on Σ^*), τ is a coding on Σ^* , and $s \in \Sigma^+$ is the *axiom*. The system generates the sequence $(\tau(\varphi^d(s)))_{d \in \mathbb{N}}$. The “D0L” stands for *deterministic L-system with 0 interactions*. The “P” stands for *propagating*, which means that it has no ε -rules. The “C” stands for *coding*, which means that the system is extended with a coding. To define a compressor based on CPD0L-system, we extend them to 6-tuples by fixing d and using another parameter n , so we can uniquely determine a string of the sequence generated by a system and then extract a prefix from it. For simplicity, in the rest of this paper, we refer to these extended CPD0L-systems just as L-systems.

► **Definition 1 (L-systems).** *An L-system is a 6-tuple $L = (\Sigma, \varphi, \tau, s, d, n)$ where Σ is the alphabet, φ is the set of rules (an endomorphism on Σ^*), τ is a coding on Σ^* , $s \in \Sigma^+$ is the axiom, and d and n are two non-negative integers. The string generated by L is $w = \tau(\varphi^d(s))[1 : n]$.*

We now define the size of an L-system and the measure ℓ .

► **Definition 2 (Measure ℓ).** *The size of an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ is $\text{size}(L) = \text{size}(\varphi) + |s| + |\Sigma| + 2$. The measure $\ell(w)$ is defined as the size of the smallest L-system generating w .*

The size of an L-system accounts for the lengths of the right-hand sides of its rules, the length of the axiom, the coding τ , and the values d and n , so we can effectively represent the system using $O(\text{size}(L))$ space. Hence, the measure ℓ is reachable.

As a convention, we always assume that $d = n^{O(1)}$ and that $\Sigma = n^{O(1)}$. Otherwise, we could need too many words to represent the integer d or the symbols of the alphabet.

A finer-grained analysis about the number of bits needed to represent an L-system of size ℓ yields $O(\ell \log |\Sigma| + \log n)$ bits, the second term corresponding to d and n ; note that Σ contains the alphabet of w .

An important result about ℓ is that it always holds that $\ell = O(g)$ [19]. More importantly, sometimes $\ell = o(\delta)$, which implies that δ is not a lower bound for ℓ , and questions δ as a definitive measure of repetitiveness.

To understand the particularities of ℓ , we study several classes of L-systems with different restrictions and define measures based on them. We define the measure ℓ_e that restricts the morphism to be expanding. The measure ℓ_u restricts the morphism to be k -uniform for some $k > 1$. The variant ℓ_m forces the morphism of the system to be a -prolongable for some symbol a and the axiom to be $s = a$. The variant ℓ_d essentially removes the coding by forcing it to be the identity function. Finally, ℓ_p refers to the intersection of ℓ_m and ℓ_d , and ℓ_a refers to the intersection of ℓ_m and ℓ_u , which intuitively perform well in prefixes of purely morphic words and prefixes of automatic sequences, respectively.

► **Definition 3.** *An L-system $(\Sigma, \varphi, \tau, s)$ is a -prolongable if there exists a symbol a such that $s = a$ and $a \rightarrow ax$ with $x \neq \varepsilon$. An L-system is prolongable if it is a -prolongable for some symbol a .*

► **Definition 4 (ℓ -variants).** *We define the following ℓ -variants*

1. *The variant ℓ_e denotes the size of the smallest L-system generating w , satisfying that all its rules have a size at least 2.*
2. *The variant ℓ_m denotes the size of the smallest prolongable L-system generating w .*
3. *The variant ℓ_d denotes the size of the smallest L-system generating w , satisfying that τ is the identity function.*
4. *The variant ℓ_u denotes the size of the smallest L-system generating w , satisfying that all its rules have the same size, at least 2.*
5. *The variant ℓ_p denotes the size of the smallest prolongable L-system generating w , satisfying that τ is the identity function.*
6. *The variant ℓ_a denotes the size of the smallest prolongable L-system generating w , satisfying that all its rules have the same size, at least 2.*

It is known that different classes of L-systems produce different classes of languages [21]. Some of these classes also differ in the factor complexity of the sequences they can generate [22]. It is interesting to understand how these differences in terms of expressive power and factor complexity translate into compression power.

We defer the study of ℓ -variants to Section 7. We define them early, as some of our results relating ℓ to better-established measures in Sections 4 and 5 also apply for some of the ℓ -variants.

4 Breaking the δ -lower-bound for repetitiveness

It is known that the repetitiveness measure δ is a (strict) lower bound to all the other repetitiveness measures [17]. It is also known that δ is a lower bound to k -th order empirical entropy, which plays a role in several compressors [17]. This makes δ an asymptotic lower bound to the size of almost every existing compressor and compressibility measure.

25:8 L-Systems for Measuring Repetitiveness

Certainly, we cannot expect to find a reachable measure upper-bounded by $O(\delta)$ because δ -space is unreachable, as shown by Kociumaka et al. [12]. Still, it could be possible to design measures capturing repetitiveness and going below δ in some restricted but relevant scenarios. In this context, we raise the following question:

Can we find a competitive and reachable repetitiveness measure achieving space lower than δ on some restricted but relevant cases?

It is not difficult to design a trivial measure breaking the δ -lower-bound for some specific string families. We also require, however, this measure to make sense, be reachable, and be *competitive*, the latter meaning that it is at least as good as z , g , or r (i.e., the most popular reachable measures in practice) in terms of space.

The repetitiveness measure ℓ was designed with the conditions above in mind. As we already mentioned, ℓ cannot lower-bound δ because ℓ is a reachable measure.

► **Lemma 5** ([19, Theorem 4]). *There exist string families where $\ell = \Omega(\delta \log n)$.*

It was shown that ℓ is a competitive repetitiveness measure: the smallest L-system for a string is always asymptotically smaller than the smallest grammar (their proof applies to the variant ℓ_d as well). This shows that the measure ℓ is always reasonable for repetitive strings:

► **Lemma 6** ([19, Theorem 6]). *It always holds that $\ell = O(g)$.*

On the other hand, they [19] showed a string family satisfying that $\delta = \Omega(\ell \log n)$, and conjectured that this gap was the maximum possible, that is, that the lower bound $\ell = \Omega(\delta / \log n)$ holds for any string family. We now disprove this conjecture. We show a string family where δ is $\Theta(\sqrt{n})$ times bigger than the size ℓ of the smallest L-system.

► **Lemma 7.** *There exists a string family where $\delta = \Theta(\ell \sqrt{n})$.*

Proof. Consider a c-prolongable L-system $L_d = (\Sigma, \varphi, \tau, s, d, n)$, where

$$\begin{aligned}\Sigma &= \{a, b, c\} \\ \varphi &= \{a \rightarrow a, b \rightarrow ab, c \rightarrow cb\} \\ \tau &= \{a \rightarrow a, b \rightarrow b, c \rightarrow c\} \\ s &= c \\ n &= 1 + \frac{(d-1)d}{2} + d\end{aligned}$$

for any $d \geq 0$. By iterating the morphism φ we obtain the words

$$\begin{aligned}\varphi^0(c) &= c \\ \varphi^1(c) &= cb \\ \varphi^2(c) &= cbab \\ \varphi^3(c) &= cbabaab \\ \varphi^4(c) &= cbabaabaaab \\ \varphi^5(c) &= cbabaabaaabaaaab\end{aligned}$$

and so on, from which we extract as a prefix the whole word (depending on the value of d chosen). It is easy to check by induction that for each $d \geq 0$, the string generated by the system L_d is $s_d = c \prod_{i=0}^{d-1} a^i b$ and it has length $1 + \frac{(d-1)d}{2} + d$.

$$s_3 = c \ b \ a \ b \ a \ a \ b$$

$$s_6 = c \ b \ a \ b \ a \ a \ \underline{b \ a \ a \ a \ b \ a \ a \ a \ b \ a \ a \ a \ a \ b}$$

■ **Figure 2** All the substrings of length 6 of the string s_6 of Lemma 7 starting inside some position $i \leq |s_3| = 7$ are distinct, because the runs of **a**'s considered have all different and increasing lengths, and d is big enough. The last of the substrings considered is underlined. Extending these substrings one position to the left yields $|s_3|$ different strings of length 7, so the claim holds for even and odd values of $d \geq 2$.

It holds that ℓ is $\Theta(1)$ in this family. The system is essentially the same for every string in the family. The only changes are the integers d and n , which always fit in constant space.

On the other hand, the first $|s_{\lfloor d/2 \rfloor}| = 1 + (\lfloor d/2 \rfloor - 1)(\lfloor d/2 \rfloor)/2 + \lfloor d/2 \rfloor$ substrings of length d of s_d (for $d \geq 2$) are completely determined by the **b**'s they cross, and the number of **a**'s at their extremes, so they are all distinct. An example can be seen in Figure 2.

This gives the lower bound $\delta = \Omega(d) = \Omega(\sqrt{n})$. The upper bound $O(\sqrt{n})$ holds trivially for run-length grammars as the strings considered have $\Theta(\sqrt{n})$ runs of **a**'s followed by **b**'s, so $\delta = \Theta(\sqrt{n})$. Thus $\delta = \Theta(\ell\sqrt{n})$ in this string family. ◀

This string of Lemma 7 is easy to describe yet hard to represent with copy-paste mechanisms. Intuitively, the simplicity of the sequence relies on the fact that many substrings can be described in terms of previous ones, so it is arguably highly repetitive, though not via copy-paste. The repetitiveness in this family is better captured by an L-system, instead.

5 Uncomparability of ℓ with other repetitiveness measures

As a corollary of Lemmas 5 and 7 (and also mentioned in previous work [19]), we obtain that ℓ and δ are uncomparable as repetitiveness measures.

► **Corollary 8.** *The measures ℓ and δ are uncomparable.*

Moreover, this is also true for the variant ℓ_p because, in Lemma 7, we considered a prolongable L-system with the identity function as the coding. As we prove later in Section 7, the variant ℓ_p is, in general, far from ideal for measuring repetitiveness, so the fact that δ is uncomparable to this weak variant is even more surprising.

A natural question is then to identify which other measures are also uncomparable to ℓ (and ℓ_p). We show in this section that this holds for almost any other repetitiveness measure. To do so, we first recall the string family defined by Kociumaka et al. [12], satisfying that it needs $\Omega(\log^2 n)$ bits to be represented with any method. This string family will be crucial in the following proofs.

► **Definition 9** ([12]). *The string family \mathcal{K} is formed by all the infinite strings s over $\{a, b\}$ constructed as follows:*

1. Let $s[1] = b$.
2. For any $i \geq 2$, choose a position j_i in $[2 \cdot 4^{i-2} + 1, 4^{i-1}]$. Then, $s[j_i] = b$.
3. If $j > 1$ and $j \neq j_i$ for any $i \geq 2$, then $s[j] = a$.

The family \mathcal{K}_n for $n \geq 0$ is formed by all the prefixes of length n of some string in \mathcal{K} .

It is easy to see that the strings in the family \mathcal{K}_n have $\Theta(\log n)$ symbol **b**'s. Also, note that with the possible exception of the first two positions, there are no consecutive **b**'s.

25:10 L-Systems for Measuring Repetitiveness

Now we are ready to prove that, in general, it does not hold that $\ell = O(g_{rl})$, making L-systems uncomparable to RLSLPs.

► **Lemma 10.** *There exists a string family where $\ell = \Omega(g_{rl} \log n / \log \log n)$.*

Proof. Consider the string family \mathcal{K}_n needing $\Omega(\log^2 n)$ bits (or $\Omega(\log n)$ space) to be represented with any method [12]. Strings in \mathcal{K}_n have $O(\log n)$ runs of **a**'s separated by **b**'s, so it is easy to see that $g_{rl} = O(\log n)$ in this family. Because of this, and because g_{rl} is a reachable measure, it holds that $g_{rl} = \Theta(\log n)$ in \mathcal{K}_n . On the other hand, the minimal L-system for a string in this family can be represented with $O(\ell \log |\Sigma| + \log n) \subseteq O(\ell \log \ell + \log n)$ bits, which must be in $\Omega(\log^2 n)$ bits because the L-system is also reachable. It follows that $\ell = \Omega(\log^2 n / \log \log n)$; otherwise,

$$\begin{aligned} \ell \log \ell &= o((\log^2 n / \log \log n) \log(\log^2 n / \log \log n)) \\ &= o(\log^2 n), \end{aligned}$$

which contradicts ℓ being reachable. Thus, $\ell = \Omega(g_{rl} \log n / \log \log n)$ in this string family. ◀

The same result holds for LZ-like parsings. Even the greedy LZ-End parsing (the largest of them) can be asymptotically smaller than ℓ in some string families.

► **Lemma 11.** *There exists a string family where $\ell = \Omega(z_e \log n / \log \log n)$.*

Proof. Take each string in \mathcal{K}_n and prepend \mathbf{a}^n to it. This new family of strings still needs $\Omega(\log^2 n)$ bits to be represented with any method because the size of the family is the same, and n just doubled. Just as in Lemma 10, it holds that $\ell = \Omega(\log^2 n / \log \log n)$ in this family. On the other hand, the LZ-End parsing needs $\Theta(\log n)$ phrases only to represent the prefix $\mathbf{a}^n \mathbf{b}$, and then for each run of **a**'s followed by **b**, its source is aligned with $\mathbf{a}^n \mathbf{b}$, so $z_e = \Theta(\log n)$. Thus, $\ell = \Omega(z_e \log n / \log \log n)$. ◀

The same result also holds for the number of equal-letter runs of the Burrows-Wheeler transform of a string.

► **Lemma 12.** *There exists a string family where $\ell = \Omega(r \log n / \log \log n)$.*

Proof. Consider the family \mathcal{K}_n again. Clearly $r = \Omega(\log n)$, because r is reachable. Because a string in this family has $O(\log n)$ **b**'s, its BWT has also $O(\log n)$ runs of **a**'s separated by **b**'s (or the unique \$). Thus, $r = \Theta(\log n)$ and $\ell = \Omega(r \log n / \log \log n)$ in this string family. ◀

We conclude that the measure ℓ is uncomparable to almost every other repetitiveness measure. We summarize these results in the following theorem.

► **Theorem 13.** *The measure ℓ is uncomparable to the repetitiveness measures $\delta, \gamma, b, v, c, g_{rl}, z, z_{no}, z_{end}, z_e$, and r .*

Proof. There exist string families where $\ell = o(\delta)$. In these families, it holds $\ell = o(\mu)$ where μ is any of the measures considered above, because δ is a lower bound to all of them. On the other hand, all the measures above are upper-bounded by at least one of z_e, g_{rl} , or r , which by Lemmas 10, 11, and 12, respectively, can be asymptotically smaller than ℓ for some string families. ◀

This shows that ℓ , although reachable and competitive as a repetitiveness measure, captures the regularities in strings in a form that is largely orthogonal to other repetitiveness measures. As the underlying regularities captured by ℓ and the other measures are apparently different, we try to combine them to obtain more powerful measures/compressors.

6 NU-systems and the measure ν

A *NU-system* [19] is a tuple $N = (V, R, \Gamma, s, d, n)$ that generates a unique string in a similar way to an L-system. The key difference is that on the right-hand side of its rules, a NU-system is permitted to have special symbols of the form $a(k)[i : j]$, whose meaning is to generate the k -th level from a , then extract the substring starting at position i and ending at position j , and finally apply the coding to the resulting substring.

The indices in a NU-system (e.g., levels, intervals) must be less or equal to n to fit in a $\Theta(\log n)$ -bits word. Also, the NU-system must not produce any loops when extracting a prefix from some level, which is decidable to detect. The size of a NU-system is defined analogously to the size of L-systems, with the extraction symbols $a(k)[i : j]$ being symbols of length 4. The measure ν is defined as the size of the smallest NU-system generating w .

It holds that $\nu = O(\ell)$ and $\nu = O(b)$ [19]. Moreover, there exist families where both asymptotic bounds are strict. We now show that NU-systems exploit the features of L-systems and macro schemes in a way that, for some string families, can reach sizes that are unreachable for both L-systems and macro schemes independently.

► **Theorem 14.** *There exists a family of strings where $\nu = o(\min(\ell, b))$.*

Proof. Let \mathcal{K}_m be the family of strings defined by Kociumaka et al. [12], needing $\Omega(\log^2 m)$ bits to be represented with any method, but over the alphabet $\{0, 1\}$. We construct a new family $\mathcal{F} = \{x \cdot \mathbf{y}[: m] \mid x \in \mathcal{K}_m\}$, where \mathbf{y} is the infinite fixed point generated by the c -prolongable L-system with the identity function as the coding, utilized in Lemma 7.

Let $n = 2m$. As shown in Lemma 10, it holds that $\ell = \Omega(\log^2 n / \log \log n)$ in this family. On the other hand, $b = \Omega(\sqrt{n})$, because $\delta = \Omega(\sqrt{n})$ on prefixes of \mathbf{y} , and the alphabets between the prefix in \mathcal{K}_m and $\mathbf{y}[: m]$ are disjoint.

Let x be a string in \mathcal{K}_m with k symbols 1. Let i_j be the number of 0's in x between the $(j-1)$ -th 1 and the j -th 1, for $j \in [2, k]$. Also, let i_1 and i_{k+1} be the number of 0's at the left and right extremes of x . We construct a NU-system $N = (V, R, \Gamma, s, d, n)$ as follows:

$$\begin{aligned} V &= \{0, 1, \mathbf{a}, \mathbf{b}, \mathbf{c}\} \\ R &= \{0 \rightarrow 00, 1 \rightarrow 1, \mathbf{a} \rightarrow \mathbf{a}, \mathbf{b} \rightarrow \mathbf{ab}, \mathbf{c} \rightarrow \mathbf{cb}\} \\ \Gamma &= \{0 \rightarrow 0, 1 \rightarrow 1, \mathbf{a} \rightarrow \mathbf{a}, \mathbf{b} \rightarrow \mathbf{b}, \mathbf{c} \rightarrow \mathbf{c}\} \\ s &= 0(m)[: i_1]10(m)[: i_2]1 \dots 0(m)[: i_k]10(m)[: i_{k+1}]\mathbf{c}(m)[: m] \\ d &= 0 \\ n &= 2m \end{aligned}$$

By construction, this NU-system generates the string $x \cdot \mathbf{y}[: m]$ of length n , and its axiom has size $4(k+2) + k$, where $k = \Theta(\log n)$. Hence, it holds that ν is $O(\log n)$ for these strings. Thus, $\nu = o(\min(\ell, b))$ in the family \mathcal{F} we constructed. ◀

NU-systems can then be smaller representations than those produced by any other compression method exploiting repetitiveness. This shows that combining copy-paste mechanisms with iterated morphisms is an effective way of improving compression from a theoretical point of view.

On the other hand, though computable, no efficient decompression scheme has been devised for NU-systems. In turn, finding the smallest NU-system is very likely an NP-complete problem, and probably very difficult to even approximate.

7 ℓ -variants are weaker than ℓ

In this section we show that the features we include in the L-systems used in the definition of ℓ are necessary to obtain a competitive repetitiveness measure; removing any of them yields an inherent loss in compression power.

We start by showing that ℓ can be asymptotically strictly smaller than ℓ_m . That is, restricting L-systems to be prolongable yields a weaker measure.

► **Lemma 15.** *There exists a string family where $\ell = o(\ell_m)$.*

Proof. Let $\mathcal{F} = \{\mathbf{a}^{n-1}\mathbf{b} \mid n \geq 1\}$. Clearly, ℓ is constant in this string family: the L-system $L_n = (V, \varphi, \tau, s, d, n)$ where $V = \{\mathbf{a}, \mathbf{b}\}$, $\varphi = \{\mathbf{a} \rightarrow \mathbf{a}, \mathbf{b} \rightarrow \mathbf{ab}\}$, $\tau = \{\mathbf{a} \rightarrow \mathbf{a}, \mathbf{b} \rightarrow \mathbf{b}\}$, $s = \mathbf{b}$, and $d = n - 1$ produces each string in \mathcal{F} by changing only the value of n and d accordingly. Note that these L-systems are not prolongable on the axiom.

For the sake of contradiction, suppose that $\ell_m = O(1)$ in \mathcal{F} . Let $L_n = (\Sigma_n, \varphi_n, \tau_n, c, d_n, n)$ be the smallest c -prolongable system generating $\mathbf{a}^{n-1}\mathbf{b}$. Because $\ell_m = O(1)$, there exists a constant C satisfying that $|\Sigma_n| < C$ and $\text{width}(\varphi_n) < C$ for every n . Observe that it is only necessary to have one symbol $c' \in \Sigma_n$ with $\tau_n(c') = \mathbf{b}$ because there is only one \mathbf{b} in $\mathbf{a}^{n-1}\mathbf{b}$, so w.l.o.g. assume that $\mathbf{b} \in \Sigma_n$ and $\tau_n(\mathbf{b}) = \mathbf{b}$. As the system is c -prolongable, each level is a prefix of the next one. This implies that the morphism should be iterated until \mathbf{b} appears for the first time, and then we can safely extract the prefix. This must happen in the first C iterations of the morphism; otherwise, \mathbf{b} is not reachable from C (i.e., if an iteration does not yield a new symbol, then no new symbols will appear since then, and there are less than C symbols). But in the first C iterations, we cannot produce a string longer than the constant C^C . For sufficiently large n , this implies that the symbol \mathbf{b} , if it is reachable, will appear for the first time before the n -th position, which is a contradiction. ◀

Because we used the identity coding in the proof above, we can obtain the following corollary.

► **Corollary 16.** *There exists a string family where $\ell_d = o(\ell_m)$.*

We can prove a similar result for uniform morphisms.

► **Proposition 17.** *There exists a string family where $\ell_u = o(\ell_m)$.*

Proof. It is not difficult to see that ℓ_u is constant in the family $\{\mathbf{a}^{2^k}\mathbf{b} \mid k \geq 0\}$: consider the axiom $s = \mathbf{ab}$ and the rules $\mathbf{a} \rightarrow \mathbf{aa}$, $\mathbf{b} \rightarrow \mathbf{bb}$, the level $d = k$ and the prefix length $n = 2^k + 1$. A similar argument to the one of Lemma 15 yields that $\ell_u = o(\ell_m)$ for this other string family. ◀

Further, we can find a concrete asymptotic gap between ℓ and ℓ_m in the string family of the proof of the previous lemma.

► **Lemma 18.** *There exists a string family where $\ell_m = \Omega(\ell \log n / \log \log n)$.*

Proof. Let $\mathcal{F} = \{\mathbf{a}^{n-1}\mathbf{b} \mid n \geq 1\}$. Recall that $\ell = O(1)$ in this family. Let $k = |\Sigma|$ and $t = \text{width}(\varphi)$ obtained from the morphism of the smallest c -prolongable system generating $\mathbf{a}^{n-1}\mathbf{b}$ (we assume again that the only symbol mapped to \mathbf{b} by the coding is \mathbf{b}). In the first k iterations, \mathbf{b} must appear (as in the previous proof) and cannot be deleted in the following levels, so it cannot appear before position n . Hence, $t^k \geq n$, which implies

$k \geq \log_t n$. By definition, $\ell_m \geq k \geq \log_t n$ and $\ell_m \geq t$, so $\ell_m \geq \max(t, \log_t n)$. The solution to the equation $t = \log_t n$ is the smallest value that $\max(t, \log_t n)$ can take for $t \in [2..n]$. This value is $\Omega(\log n / W(\log n))$ where $W(x)$ is the Lambert W function, and it holds that $W(\log n) = \Theta(\log \log n)$. Therefore, $\ell_m = \Omega(\ell \log n / \log \log n)$ in this string family. ◀

As a corollary, we obtain the following result.

► **Corollary 19.** *There exists a string family where $\ell_m = \Omega(\ell_d \log n / \log \log n)$.*

We now show that if we remove the coding from prolongable L-systems, which corresponds to the variant ℓ_p , we end with a much worse measure. We change the usual alphabet for clarity of presentation.

► **Lemma 20.** *There exists a string family where $\ell_p = \Omega(\ell_m \sqrt{n})$.*

Proof. We prove that $\ell_p = \Theta(n)$ whereas $\ell_m = O(\sqrt{n})$ on $\mathcal{F} = \{0^{n-1}1 \mid n \geq 2\}$. Any prolongable morphism with an identity coding generating $0^{n-1}1$ must have the rule $0 \rightarrow 0^{n-1}1$, which implies $\ell_p = \Theta(n)$. The reason is that if the system is prolongable, but it has no coding, then the axiom must be 0, and in the prolongable rule $0 \rightarrow 0w$, if $|\varphi(0)| \leq n$, then the non-empty string w could only contain 0's and 1's, otherwise undesired symbols would appear in the final string because the starting level is a prefix of the final level. If w does not contain 1's, then 1 is unreachable from 0. If w contains a 1, then the first of them should be at position n .

On the other hand, we can construct an a-prolongable morphism, with $\tau(1) = 1$ and $\tau(a) = 0$ for every other symbol $a \neq 1$ as follows: Let $n - 1 = k \lfloor \sqrt{n-1} \rfloor + j$ with $\lfloor \sqrt{n-1} \rfloor > 3, k > 1, 0 \leq j < \lfloor \sqrt{n-1} \rfloor$ (k and j integers). We can assume n is sufficiently big so the requirements are satisfied. Then, define the following rules

$$\begin{aligned} a &\rightarrow ab \\ b &\rightarrow c^{k-1}d \\ c &\rightarrow 0^{\lfloor \sqrt{n-1} \rfloor - 1} \\ d &\rightarrow 0^{\lfloor \sqrt{n-1} \rfloor - 3 + j} 1. \end{aligned}$$

The first four levels are

$$\begin{aligned} \varphi^0(a) &= a \\ \varphi^1(a) &= ab \\ \varphi^2(a) &= abc^{k-1}d \\ \varphi^3(a) &= abc^{k-1}d0^{(\lfloor \sqrt{n-1} \rfloor - 1)(k-1)}0^{\lfloor \sqrt{n-1} \rfloor - 3 + j}1, \end{aligned}$$

and it holds that

$$|\varphi^3(a)| = 3 + (k-1) + (\lfloor \sqrt{n-1} \rfloor - 1)(k-1) + (\lfloor \sqrt{n-1} \rfloor - 3 + j) + 1 = n.$$

Hence, $\tau(\varphi^3(a)) = 0^{n-1}1$. The system $L = (\{a, b, c, d, 0, 1\}, \varphi, \tau, a, 3, n)$ generates $0^{n-1}1$ as required for n bigger than some constant. The size of the system is clearly $O(\sqrt{n})$. Thus, the claim holds. ◀

By using the same family above, the following corollary holds.

► **Corollary 21.** *There exists a string family where $\ell_p = \Omega(\ell_d n)$.*

It is surprising that this weak measure ℓ_p can be much smaller than δ for some string families. This can be deduced from Lemma 7. On the other hand, it does not hold that $\ell_p = O(g)$ for any string family, because $g = \Theta(\log n)$ on $\{0^{n-1}1 \mid n \geq 1\}$.

► **Corollary 22.** *The variant ℓ_p is uncomparable to the measures δ and g .*

If we restrict L-systems to be expanding, that is, with all its rules having a length of at least 2, we also end with a weaker measure. This shows that, in general, it is not possible to transform L-systems into expanding ones without incurring an increase in size.

► **Lemma 23.** *There exists a string family where $\ell = o(\ell_e)$.*

Proof. Let $\mathcal{F} = \{\mathbf{a}^k \mathbf{b} \mathbf{a}^{2^k} \mid k \geq 0\}$. Clearly ℓ is constant in \mathcal{F} : the L-system $(\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}, \{\mathbf{a} \rightarrow \mathbf{a}\mathbf{a}, \mathbf{b} \rightarrow \mathbf{c}\mathbf{b}, \mathbf{c} \rightarrow \mathbf{c}\}, \{\mathbf{a} \rightarrow \mathbf{a}, \mathbf{b} \rightarrow \mathbf{b}, \mathbf{c} \rightarrow \mathbf{a}\}, \mathbf{b}\mathbf{a}, d = k, n = 2^k + k + 1)$ produces $\mathbf{a}^k \mathbf{b} \mathbf{a}^{2^k}$ and stays constant-size as k (and d and n) grows.

Suppose that ℓ_e is also constant in \mathcal{F} . Then there is a constant C such that the minimal expanding L-systems generating the strings in this family have at most C rules, each one of length at most C . Without loss of generality, assume that for each of these systems, the only symbol mapped to \mathbf{b} by the coding is \mathbf{b} . Also, assume that the axiom is a single symbol a_0 . Note that because the systems are expanding with rules of size at most C , their level must be $d \geq \log_C 2^k = \frac{k}{\log_2 C}$.

Let a_0, a_1, \dots, a_d be the sequence of the first symbols of $\varphi^i(a_0)$ for $i \leq d$. By the pigeonhole principle, for sufficiently big values of k (and consequently big values of d), this sequence has a period of length q starting from a_p , with $p + q \leq C \leq d$. Then there exist indexes t and j such that $t = d - jq$ and $p \leq t < p + q$. By the q -periodicity of the sequence starting at a_t , it is clear that $\varphi^q(a_t) = a_t w$ for some $w \neq \varepsilon$ (because the morphism is expanding), so φ^q is prolongable on a_t . This implies that $\varphi^{iq}(a_t)$ is a prefix of $\varphi^{jq}(a_t)$ for $i \leq j$. As before, if \mathbf{b} is reachable from a_t via φ^q , that must happen in the first C iterations, so $\varphi^{Cq}(a_t)$ contains a \mathbf{b} , and so does $\varphi^{jq}(a_t)$, which is a prefix of $\varphi^d(a_0)$. This implies that $\varphi^d(a_0)$ contains a \mathbf{b} before position C^{Cq} , which is bounded by C^{C^2} , a contradiction for sufficiently long strings in the family. So it has to be that \mathbf{b} is not reachable via φ^q from a_t , but this is also a contradiction for sufficiently long strings because $\varphi^{jq}(a_t)$ is a prefix of $\varphi^d(a_0)$ of length at least $2^{d-t} = \omega(k)$, yielding too many symbols not mapped to \mathbf{b} before the first \mathbf{b} at level d . Thus, ℓ_e cannot be $O(1)$ in \mathcal{F} . ◀

We summarize the results of this section in Figure 3. Overall, we have shown that imposing restrictions on the length of the rules of an L-system or forcing them to be prolongable wildly impacts their compression power. We have not yet found an example where ℓ_d could be asymptotically smaller than ℓ , which would prove that the coding contributes to the measure ℓ in a fundamental way (the purpose of the coding is to make ℓ constant in the case of prefixes of general morphic words, but it is unknown if it is really needed). We conjecture that such a family exists and the coding is necessary.

8 Conclusions and open questions

The measure ℓ is arguably a strong reachable repetitiveness measure, which can break the limits of δ (a measure considered a stable lower bound for repetitiveness) by a wide margin (a factor of \sqrt{n}). On the other hand, however, ℓ can be asymptotically weaker than the space reached by several compressors based on run-length context-free grammars, many Lempel-Ziv variants, and the Burrows-Wheeler transform. Only the size of context-free grammars is an upper bound to ℓ . This suggests that the self-similarity exploited by L-systems is mostly

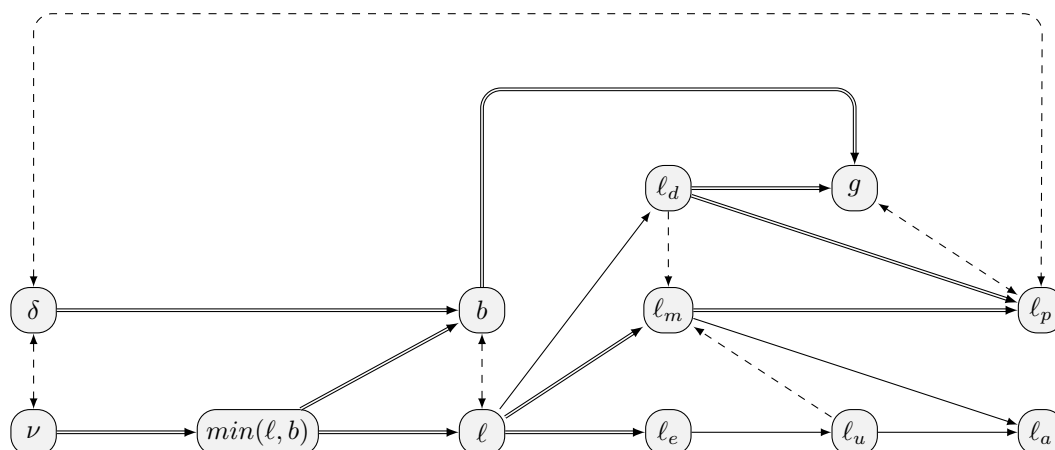


Figure 3 Asymptotic relations between ℓ -variants and other relevant measures. A black arrow from v_1 to v_2 means that it always holds that $v_1 = O(v_2)$. A double black arrow from v_1 to v_2 means that it also exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$.

independent of the source of repetitiveness exploited by other compressors and measures, which build on copy-paste mechanisms. We also show that several attempts to simplify or restrict L-systems lead to weaker measures.

A relevant question about L-systems is whether they can be useful for building compressed sequence representations that support direct access. More formally, can we build an $O(\ell)$ -space representation of a string $w[1 : n]$ providing random access to any position of the string in $O(\text{polylog } n)$ time? The closest result (as far as we know) is an algorithm designed by Shallit and Swart [25], which computes $\varphi^d(a)[i]$ in time bounded by a polynomial in $|\Sigma|, \text{width}(\varphi), \log d$ and $\log i$. It uses more space and takes more time than our aim. The main bottleneck is having to store the incidence matrix of the morphism and compute its powers. As suggested by Shallit and Swart, this could be solved by finding closed forms for the growth functions (recurrences) of each symbol. If this approach were taken, these formulas should be easily described within $O(\ell)$ space.

In terms of improving compression, on the other hand, the recent measure ν [19] aims to unify the repetitiveness induced by self-similarity and by explicit copies. This measure is the smallest size of a NU-system, a natural way to combine L-systems (with minimum size ℓ) with macro schemes (with minimum size $b \geq \delta$). In line with our finding that ℓ and δ are mostly orthogonal, we prove in this paper that ν is strictly more powerful than both ℓ and b , which makes ν the unique smallest reachable measure of repetitiveness to date.

There are several open questions related to NU-systems and ν . For example, does it hold that $\nu = \Omega(\ell \log \log n / \log n)$, or $\nu = \Omega(\delta / \sqrt{n})$, for every string family? Is $\nu = O(\gamma)$, or at least $o(\gamma \log(n/\gamma))$, for every string family? (recall that γ and $o(\gamma \log(n/\gamma))$ space is unknown to be reachable [9]). And towards having a practical compressor based on ν , can we decompress a NU-system efficiently?

In a more general perspective, this paper pushes a little further the discussion of what we understand by a repetitive string. Intuitively, repetitiveness is about copies, and macro schemes capture those copies pretty well, but there are other aspects in a text that could be repeated besides explicit copies, such as general patterns and the relative ordering of symbols. Macro schemes capture explicit copies, L-systems capture self-similarity, and NU-systems

capture both. What other regularities could we exploit when compressing strings, keeping the representation (more or less) simple and the associated repetitiveness measure (hopefully efficiently) computable?

References

- 1 H. Bannai, M. Funakoshi, T. I. D. Köppl, T. Mieno, and T. Nishimoto. A separation of γ and b via Thue–Morse words. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science (LNCS)*, pages 167–178, 2021.
- 2 M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 3 M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
- 4 T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.
- 5 J. K. Gallant. *String Compression Algorithms*. PhD thesis, Princeton University, 1982.
- 6 S. Giuliani, S. Inenaga, Z. Lipták, N. Prezza, M. Sciortino, and A. Toffanello. Novel results on the number of runs of the Burrows-Wheeler-Transform. In *Proc. Theory and Practice of Computer Science (SOFSEM)*, pages 249–262, 2021.
- 7 A. Jež. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.
- 8 D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. *Communications of the ACM*, 65(6):91–98, 2022.
- 9 D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 827–840, 2018.
- 10 T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.
- 11 T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ -optimal space. In *Proc. 15th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 13568 of *Lecture Notes in Computer Science (LNCS)*, pages 88–103, 2022.
- 12 T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023.
- 13 S. Krefl and G. Navarro. LZ77-like compression with fast random access. In *2010 Data Compression Conference (DCC)*, pages 239–248, 2010.
- 14 A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
- 15 A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.
- 16 A. Lindenmayer. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968.
- 17 G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.
- 18 G. Navarro, C. Ochoa, and N. Prezza. On the approximation ratio of ordered parsings. *IEEE Transactions on Information Theory*, 67(2):1008–1026, 2021.

- 19 G. Navarro and C. Urbina. On stricter reachable repetitiveness measures. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 12944 of *Lecture Notes in Computer Science (LNCS)*, pages 193–206, 2021.
- 20 T. Nishimoto, T. I. S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72:1–72:15, 2016.
- 21 J.-J. Pansiot. On various classes of infinite words obtained by iterated mappings. In *Automata on Infinite Words*, volume 192 of *Lecture Notes in Computer Science*, pages 188–197, 1984.
- 22 J.-J. Pansiot. Subword complexities and iteration. *Bulletin of the European Association for Theoretical Computer Science*, 26:55–62, 1985.
- 23 M Przeworski, RR Hudson, and A Di Rienzo. Adjusting the focus on human variation. *Trends in Genetics*, 16(7):296–302, 2000.
- 24 W. Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.
- 25 J. Shallit and D. Swart. An efficient algorithm for computing the i th letter of $\varphi^n(a)$. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 768–775, 1999.
- 26 J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

MONI Can Find k -MEMs

Igor Tatarnikov ✉ 

Dalhousie University, Halifax, Canada

Ardavan Shahrabi Farahani ✉

Dalhousie University, Halifax, Canada

Sana Kashgouli ✉

Dalhousie University, Halifax, Canada

Travis Gagie ✉ 

Dalhousie University, Halifax, Canada

Abstract

Suppose we are asked to index a text $T[0..n-1]$ such that, given a pattern $P[0..m-1]$, we can quickly report the maximal substrings of P that each occur in T at least k times. We first show how we can add $O(r \log n)$ bits to Rossi et al.'s recent MONI index, where r is the number of runs in the Burrows-Wheeler Transform of T , such that it supports such queries in $O(km \log n)$ time. We then show how, if we are given k at construction time, we can reduce the query time to $O(m \log n)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compact data structures, Burrows-Wheeler Transform, run-length compression, maximal exact matches

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.26

Funding This research was supported by National Institutes of Health (NIH) NIAID (grant no. HG011392), the National Science Foundation NSF IIBR (grant no. 2029552), a Natural Science and Engineering Research Council (NSERC) Discovery Grant (grant no. RGPIN-07185-2020) and the Comisión Nacional de Investigación Científica y Tecnológica (CONICYT) through the Centre for Biotechnology and Bioengineering (grant CeBiB FB0001).

Acknowledgements The authors thank Christina Boucher, Ben Langmead, Manuel Mattheisen and Massimiliano Rossi for helpful discussions.

1 Introduction

In his foundational text *Compact Data Structures: A Practical Approach* [9, Section 11.6.1], Navarro posed the following problem:

“Assume we have the suffix tree of a collection of genomes $T[0..n-1]$. We then receive a short DNA sequence $P[0..m-1]$ and want to output all the maximal substrings of P that appear at least k times in T . . . Those substrings of P are likely to have biological significance.”¹

He described how to solve the problem with a suffix tree for T in $O(m \text{polylog}(n))$ time. Since T is a collection of genomes, it is likely to be highly repetitive and the theoretically best suffix-tree implementation is likely to be the $O(r \log(n/r))$ -space one by Gagie, Navarro and Prezza [5], where r is the number of runs in the Burrows-Wheeler Transform (BWT) of T . That full data structure is complicated, however, and has never been implemented.

¹ We have changed T to $T[0..n-1]$ and $P[1..m]$ to $P[0..m-1]$ for consistency with the rest of this paper, and omitted a parameter bounding from below the length of the substrings (since we can filter them afterwards).



Very recently, Navarro [10] also gave solutions not based on a suffix tree, with the following bounds:

- $O(g_{rl})$ space and $O(km^2 \log^\epsilon n)$ query time;
- $O(\delta \log(n/\delta))$ space and $O(m \log m (\log m + k \log^\epsilon n))$ query time;
- $O(g)$ space and $O(m^2 \log^{2+\epsilon} n)$ query time when $k = \omega(\log^2 n)$;
- $O(\gamma \log(n/\gamma))$ space and $O(m \log m \log^{2+\epsilon} n)$ query time when $k = \omega(\log^2 n)$.

We refer readers to Navarro’s paper and the references therein for definitions of g_{rl} , δ , g and γ .

In this paper we first show how we can add $O(r \log n)$ bits to Rossi et al.’s [13] recent MONI index to obtain a solution with $O(km \log n)$ query time. We then show how, if we are given k at construction time, we can reduce the query time to $O(m \log n)$, simultaneously using less space than Gagie et al.’s compressed suffix tree and less time than Navarro’s solutions. The rest of the paper is laid out as follows: in Section 2 we review MONI in enough depth to build on it; in Section 3 we show how we can extend ϕ queries to support sequential access to the LCP array; in Section 4 we show how to use ϕ and ϕ^{-1} queries and LCP access to obtain a solution with $O(km \log n)$ query time; in Section 5 we show how, if we are given k at construction time, we can precompute some answers, reducing the query time to $O(m \log n)$; and we conclude in Section 6. For the sake of brevity we assume readers are familiar with the concepts in Navarro’s text.

2 MONI

Bannai, Gagie and I [1] designed an index for T that takes $O(r \log n)$ bits plus the space needed to support fast random access to T , and lists all the maximal exact matches (MEMs) of P with respect to T – that is, all the substrings $P[i..j]$ of P occurring in T such that $i = 0$ or $P[i-1..j]$ does not occur in T , or $j = m-1$ or $P[i..j+1]$ does not occur in T – in $O(m \log \log n)$ time plus the time needed for $O(m)$ random accesses to T . MEMs are widely used in DNA alignment [8] and they are the substrings of P Navarro asks for when $k = 1$. Generalizing to arbitrary k , we refer to the substrings he asks for as k -MEMs.

Bannai et al. did not give an efficient construction algorithm or an implementation, but Rossi et al. later did. They called their implementation MONI, the Finnish word for “multi”, since it is intended to store a multi-genome reference. Boucher et al. [2] then gave a version of MONI that processes P online using longest common extension (LCE) queries on T instead of random access. We can support those LCE queries in $O(\log n)$ time with a balanced straight-line program for T , which in practice takes significantly less space than the rest of MONI.

We now sketch how Boucher et al.’s version of MONI works, incorporating ideas from Nishimoto and Tabei [12] and Brown, Gagie and Rossi [4] about replacing rank queries by table lookup and assuming we have an LCE data structure. Suppose

$$T = \text{GATTACAT}\#\text{AGATACAT}\#\text{GATACAT}\#\text{GATTAGAT}\#\text{GATTAGATAS}\$$$

with $\$ \prec \# \prec \text{A} \prec \dots \prec \text{T}$, and consider Table 1, in which the permutation FL is just the inverse of the more familiar permutation LF.

For each of the value j between 0 and $r-1 = 13$, we conceptually extract from this table the starting and ending positions $\text{head}(j)$ and $\text{tail}(j)$ of run j in the BWT, $\text{SA}[\text{head}(j)]$, $\text{SA}[\text{tail}(j)]$, $\text{BWT}[\text{head}(j)]$, $\text{LF}[\text{head}(j)]$ and the rank of the predecessor of $\text{LF}[\text{head}(j)]$ in the set

$$\{\text{head}[0], \dots, \text{head}[r-1]\}.$$

■ **Table 1** The full table from which we conceptually start when building MONI for our example text $T = \text{GATTACAT}\#\text{AGATACAT}\#\text{GATACAT}\#\text{GATTAGAT}\#\text{GATTAGATA}\$$.

i	$SA[i]$	$LCP[i]$	lexicographically i th cyclic shift of T	BWT[i]	LF(i)	FL(i)
0	44	0	\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA	A	5	29
1	8	0	#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT	T	32	11
2	17	1	#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT	T	33	28
3	25	4	#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT	T	34	30
4	34	9	#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT	T	35	31
5	43	0	A\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGAT	T	36	0
6	4	1	ACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATT	T	37	22
7	13	5	ACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGAT	T	38	23
8	21	8	ACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GAT	T	39	24
9	30	1	AGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATT	T	40	25
10	39	4	AGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATT	T	41	26
11	9	5	AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#	#	1	27
12	6	1	AT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTAC	C	22	32
13	15	3	AT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATAC	C	23	33
14	23	6	AT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATAC	C	24	34
15	32	11	AT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAG	G	25	35
16	41	2	ATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAG	G	26	36
17	11	3	ATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AG	G	27	38
18	19	10	ATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#G	G	28	39
19	1	2	ATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$G	G	29	42
20	27	4	ATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#G	G	30	43
21	36	7	ATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#G	G	31	44
22	5	0	CAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTA	A	6	12
23	14	4	CAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATA	A	7	13
24	22	7	CAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATA	A	8	14
25	31	0	GAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTA	A	9	15
26	40	3	GATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTA	A	10	16
27	10	4	GATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#A	A	11	17
28	18	11	GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#	#	2	18
29	0	3	GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$	\$	0	19
30	26	5	GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#	#	3	20
31	35	8	GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#	#	4	21
32	7	0	T#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACA	A	12	1
33	16	2	T#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACA	A	13	2
34	24	5	T#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACA	A	14	3
35	33	10	T#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGA	A	15	4
36	42	1	TA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGA	A	16	5
37	3	2	TACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GAT	T	42	6
38	12	6	TACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGA	A	17	7
39	20	9	TACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GA	A	18	8
40	29	2	TAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GAT	T	43	9
41	38	5	TAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GAT	T	44	10
42	2	1	TTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GA	A	19	37
43	28	3	TTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GA	A	20	40
44	37	6	TTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GA	A	21	41

■ **Table 2** The values we extract from Table 1, with the last two columns sorted.

j	head(j)	SA[head(j)]	tail(j)	SA[tail(j)]	BWT[head(j)]	$\mu(j)$	finger(j)
0	0	44	0	44	A	0	0
1	1	8	10	39	T	1	1
2	11	9	11	9	#	2	1
3	12	6	14	23	C	3	1
4	15	32	21	36	G	5	1
5	22	5	27	10	A	6	1
6	28	18	28	18	#	12	3
7	29	0	29	0	\$	17	4
8	30	26	31	35	#	19	4
9	32	7	36	42	A	22	5
10	37	3	37	3	T	25	5
11	38	12	39	20	A	32	9
12	40	29	41	38	T	42	13
13	42	2	44	37	A	43	13

In practice we can compute the values directly without building Table 1, using prefix-free parsing [3].

We build Table 2 with these values but we sort the last two columns, which we refer to as $\mu(j)$ and finger(j). An equivalent way to define $\mu(j)$ and finger(j), illustrated in Table 1, is to draw boxes corresponding to the runs in the BWT, permute those boxes according to LF, and write their starting positions in order as the $\mu(j)$ values and the numbers of the runs in the BWT covering their starting positions as the finger(j) values. Storing Table 2 takes about

$$2r \lg(n/r) + 2r \lg n + r \lg \sigma + 2r$$

bits, where σ is the size of the alphabet. (We do not actually need to store tail(j) = head($j + 1$) - 1, of course, but we include it in Table 2 to simplify our explanation.) It is within a reasonable constant factor of the most space-efficient implementation and simple to build.

For each suffix $P[i..m - 1]$ of P from shortest to longest, MONI finds the length ℓ_i of the longest prefix $P[i..i + \ell_i - 1]$ of $P[i..m - 1]$ that occurs in T , the lexicographic rank q_i of a suffix of T starting with $P[i..i + \ell_i - 1]$, the starting position SA[q_i] of that suffix in T , and the row j_i of Table 2 such that head(j_i) is the predecessor of q_i in that column. We note that the (pos, len) pairs (SA[q_0], ℓ_0), ..., (SA[q_{m-1}], ℓ_{m-1}) are the matching statistics MS[$0..m - 1$] of P with respect to T .

Suppose we know i , ℓ_i , q_i , SA[q_i] and j_i , and we want to find ℓ_{i-1} , q_{i-1} , SA[q_{i-1}] and j_{i-1} . If BWT[head(j_i)] = $P[i - 1]$ then we perform an LF step, as we describe in a moment. If BWT[head(j_i)] \neq $P[i - 1]$ then we find the last row j'_i above row j_i with BWT[head(j'_i)] = $P[i - 1]$, and the first row j''_i below row j_i with BWT[head(j''_i)] = $P[i - 1]$, using rank and select queries on column BWT[head(j)] in Table 2. We use LCE queries to check whether T [SA[q_i]. $n - 1$] has a longer common suffix with T [SA[tail(j'_i)]. $n - 1$] or with T [SA[head(j''_i)]. $n - 1$] and, depending on that comparison, either reset

$$\begin{aligned} \ell_i &= \text{LCE}(\text{SA}[q_i], \text{SA}[\text{tail}(j'_i)]) \\ q_i &= \text{tail}(j'_i) \\ \text{SA}[q_i] &= \text{SA}[\text{tail}(j'_i)] \\ j_i &= j'_i \end{aligned}$$

or reset

$$\begin{aligned}\ell_i &= \text{LCE}(\text{SA}[q_i], \text{SA}[\text{head}(j_i'')]) \\ q_i &= \text{head}(j_i'') \\ \text{SA}[q_i] &= \text{SA}[\text{head}(j_i'')] \\ j_i &= j_i''.\end{aligned}$$

Now $\text{BWT}[\text{head}(j_i)] = P[i - 1]$, so we can proceed with the LF step.

For example, suppose $P[0..11] = \text{TAGATTACATTA}$, $i = 2$ and we have already found $\ell_2 = 8$ (because GATTACAT occurs in T but GATTACATT does not), $q_2 = 29$, $\text{SA}[q_2] = 0$ and $j_2 = 7$. Since $\text{BWT}[\text{head}(7)] = \$ \neq P[1] = \text{A}$, we find $j_2' = 5$ and $j_2'' = 9$ and compare

$$\text{LCE}(0, \text{SA}[\text{tail}(5)]) = \text{LCE}(0, 10) = 3$$

against

$$\text{LCE}(0, \text{SA}[\text{head}(9)]) = \text{LCE}(0, 7) = 0.$$

Since the former LCE is longer, we set $\ell_2 = 3$, $q_2 = 27$, $\text{SA}[q_2] = 10$ and $j_2 = 5$.

To perform an LF step with Table 2 when we know i , ℓ_i , q_i , $\text{SA}[q_i]$ and j_i , we first set

$$\begin{aligned}\ell_{i-1} &= \ell_i + 1 \\ q_{i-1} &= \mu(\pi(j_i)) + q_i - \text{head}(j_i) \\ \text{SA}[q_{i-1}] &= \text{SA}[q_i] - 1,\end{aligned}$$

where π is the permutation on $\{0, \dots, r - 1\}$ that stably sorts the column $\text{BWT}[\text{head}(j)]$. If we keep $\text{BWT}[\text{head}(j)]$ in a wavelet tree then we have fast access to π .

For our example, consider $i = 2$, $\ell_2 = 3$, $q_2 = 27$, $\text{SA}[q_2] = 10$ and $j_2 = 5$. Since $\text{BWT}[\text{head}(5)]$ is the second **A** in the column $\text{BWT}[\text{head}(j)]$ and there are 4 characters in the column lexicographically strictly less than **A**, $\pi(5) = 5$ and $\mu(5) = 6$, so we set $\ell_1 = 4$, $q_1 = 6 + 27 - 22 = 11$ and $\text{SA}[11] = 9$. Notice π is similar to an LF mapping for the sequence obtained by sampling one character from each run of the BWT (but in our example π has a fixed point at 5); in fact, it permutes the coloured boxes in Table 1 according to LF. It follows that $\mu(\pi(j_i)) = \text{LF}(\text{head}(j_i))$. Since LF maintains the relationship between elements in the same box,

$$\text{LF}(q_i) - \text{LF}(\text{head}(j_i)) = q_i - \text{head}(j_i);$$

substituting and rearranging, we obtain our formula for q_{i-1} .

The last thing left for us to do during an LF step is find j_{i-1} . For this, we use the $\text{finger}(j)$ column. By construction, $\text{head}(\text{finger}(\pi(j_i)))$ is the predecessor of $\text{LF}(\text{head}(j_i))$ in the set

$$\{\text{head}[0], \dots, \text{head}[r - 1]\}.$$

Therefore, since $q_{i-1} = \text{LF}(q_i) \geq \text{LF}(\text{head}(j_i))$, we can find the row j_{i-1} of Table 2 such that $\text{head}(j_{i-1})$ is the predecessor of q_{i-1} in that column, by starting an exponential search at row $\text{finger}(\pi(j_i))$. This takes $O(\log r)$ time in the worst case and in practice it takes constant time. Nishimoto and Tabei showed how to guarantee it takes constant time at the cost of increasing the size of Table 2 slightly.

For more formal discussions, we refer readers to previous papers on MONI [13] and the r -index [5, 12, 4, 11].

■ **Table 3** The table we use for ϕ queries and access to the LCP.

j	SA[head(j)]	SA[tail(j)]	LCP[head(j)]	finger(j)
0	0	18	3	9
1	2	38	1	12
2	3	42	2	12
3	5	36	0	12
4	6	9	1	7
5	7	35	0	12
6	8	44	0	13
7	9	39	5	12
8	12	3	6	2
9	18	10	11	7
10	26	0	5	0
11	29	20	2	9
12	32	23	11	9
13	44	37	0	12

3 LCP access

We can support ϕ queries with table lookup as well: for each run $\text{BWT}[i..j]$ in the BWT, we store $\text{SA}[i]$ and $\text{SA}[(i-1) \bmod n]$ as a row; we sort the rows by their first components; and we add to each row the number of the row containing the predecessor of the second component in the first column. Abusing notation slightly, we refer to the columns of the resulting table as $\text{SA}[\text{head}(j)]$, $\text{SA}[\text{tail}(j)]$ and $\text{finger}(j)$. Table 3 is for our running example, augmented with a column $\text{LCP}[\text{head}(j)]$ that stores the length of the longest common prefix of $T[\text{SA}[\text{head}(j)..n-1]]$ and $T[\text{SA}[\text{tail}(j)..n-1]]$. Since we are storing the row containing the predecessor of each entry in $\text{SA}[\text{tail}(j)]$ in the column $\text{SA}[\text{head}(j)]$, we can encode each entry in $\text{SA}[\text{tail}(j)]$ as the difference between it and its predecessor in $\text{SA}[\text{head}(j)]$.

Analysis shows the table then takes about $3r \lg(n/r) + r \lg r$ bits: we essentially gap-code the interleaving of column $\text{SA}[\text{head}(j)]$ and the sorted column $\text{SA}[\text{tail}(j)]$, which consists of $2r$ sorted numbers between 0 and $n-1$ and thus takes about $2r \lg(n/r)$ bits; Kärkkäinen, Kempa and Piątkowski [6] showed that the entries in $\text{LCP}[\text{head}(j)]$ sum to $O(n \log r)$ so, by Jensen's Inequality, we can store them in a total of about $r \lg \frac{O(n \log n)}{r} = r \lg(n/r) + r \lg \lg r + O(r)$ bits; and $\text{finger}(j)$ takes about $r \lg r$ bits.

To see how we use Table 3 to answer ϕ queries, suppose we know that the predecessor of 24 in $\text{SA}[\text{head}(j)]$ is in row 9. Then we have

$$\phi(24) = \text{SA}[\text{tail}(9)] + 24 - \text{SA}[\text{head}(9)] = 10 + 24 - 18 = 16.$$

We know that the predecessor of 10 in $\text{SA}[\text{head}(j)]$ is in row $\text{finger}(9) = 7$, but the predecessor of 16 could be in a later row. Again, we perform an exponential search starting in row $\text{finger}(9) = 7$ and find the predecessor 12 of 16 in row 8. Then we have

$$\phi(16) = \text{SA}[\text{tail}(8)] + 16 - \text{SA}[\text{head}(8)] = 3 + 16 - 12 = 7.$$

Looking at rows 32 to 34 in Table 1, we see that indeed $\phi(24) = 16$ and $\phi(16) = 7$. This works because, similar to the equation for LF, if $\text{BWT}[j-1] = \text{BWT}[j]$ then $\phi(\text{SA}[j]-1) = \phi(\text{SA}[j]) - 1$. Again, for more formal discussions, we refer readers to previous papers on the r -index [5, 12, 4, 11].

We do not know how to support random access to the LCP array quickly in $O(r \log n)$ bits, but we can use Table 3 to provide a kind of sequential access to it. Specifically, as we use ϕ to enumerate the values in the SA – without necessarily knowing the positions of the cells of the SA those values appear in – we can use similar computations to enumerate the corresponding values in the LCP array. In our example, since the predecessor 18 of 24 in $\text{SA}[\text{head}(j)]$ is in row 9, we can compute the LCP value corresponding to the SA value 24 as

$$\text{LCP}[\text{SA}^{-1}[24]] = \text{LCP}[\text{head}(9)] + \text{SA}[\text{head}(9)] - 24 = 11 + 18 - 24 = 5.$$

Checking this, we see that $\text{LCP}[\text{SA}^{-1}[24]] = \text{LCP}[34] = 5$. Since the predecessor 12 of $\phi(24) = 16$ in $\text{SA}[\text{head}(j)]$ is in row 8 of Table 3,

$$\text{LCP}[\text{SA}^{-1}[16]] = \text{LCP}[\text{head}(8)] + \text{SA}[\text{head}(8)] - 16 = 6 + 12 - 16 = 2.$$

Checking this, we see that $\text{LCP}[\text{SA}^{-1}[16]] = \text{LCP}[33] = 2$.

Notice we do not use the SA row numbers 34 and 33 to compute the LCP value, as the SA value 24 is sufficient. We could avoid using the inverse suffix array SA^{-1} in our formula by writing $\text{LCP}[\text{SA}^{-1}[24]]$ as $\text{PLCP}[24]$, for example, where $\text{PLCP}[0..n-1]$ denotes the permuted LCP array [7] of T . The kind of sequential access we obtain to the LCP is actually random access to the PLCP array, and it is easier to explain why it works from that perspective – because if $\text{BWT}[j-1] = \text{BWT}[j]$ then $\text{PLCP}[\text{SA}[j]-1] = \text{PLCP}[\text{SA}[j]] + 1$.² Nevertheless, we present our results in terms of the LCP and SA^{-1} because we will use them later in conjunction with ϕ queries to enumerate the values in LCP intervals.

Symmetric to using Table 3 to support ϕ queries, we can use a table to support ϕ^{-1} queries. In fact, the $(\text{SA}[\text{head}(j)], \text{SA}[\text{tail}(j)])$ pairs in the table are the same, but sorted by their second components; now we add to each row the number of the row containing the predecessor in the second column of the first component. Since we are storing the row containing the predecessor of each entry in $\text{SA}[\text{head}(j)]$ in the column $\text{SA}[\text{tail}(j)]$, we can encode each entry in $\text{SA}[\text{head}(j)]$ as the difference between it and its predecessor in $\text{SA}[\text{tail}(j)]$. Analysis then shows the table takes about $2r \lg(n/r) + r \lg r$ bits. Table 4 is for supporting ϕ^{-1} queries on our running example. For example, if we know that the predecessor of 7 in $\text{SA}[\text{tail}(j)]$ is in row 1, then we can compute

$$\phi^{-1}(7) = \text{SA}[\text{head}(1)] + 7 - \text{SA}[\text{tail}(1)] = 12 + 7 - 3 = 16$$

and we can find the row containing the predecessor of 16 in $\text{SA}[\text{tail}(j)]$ with an exponential search starting at row $\text{finger}(1) = 3$ (and ending in the same row). We can then compute

$$\phi^{-1}(16) = \text{SA}[\text{head}(3)] + 16 - \text{SA}[\text{tail}(3)] = 18 + 16 - 10 = 24$$

and we can find the row 6 containing the predecessor of 24 in $\text{SA}[\text{tail}(j)]$ with an exponential search starting at row $\text{finger}(3) = 4$.

² The formula for PLCP has a +1 where the formula for ϕ has a -1,

$$\begin{aligned} \phi(\text{SA}[j]-1) &= \phi(\text{SA}[j]) - 1 \\ \text{PLCP}[\text{SA}[j]-1] &= \text{PLCP}[\text{SA}[j]] + 1, \end{aligned}$$

because if $\text{BWT}[j-1] = \text{BWT}[j]$ then moving from j to $LF(j)$ decrements the SA entry but increments the LCP entry.

■ **Table 4** The table we use for ϕ^{-1} queries.

j	SA[head(j)]	SA[tail(j)]	finger(j)
0	26	0	6
1	12	3	3
2	6	9	1
3	18	10	4
4	0	18	0
5	29	20	6
6	32	23	6
7	7	35	11
8	5	36	1
9	44	37	13
10	2	38	0
11	9	39	2
12	3	42	1
13	8	44	1

With these two $O(r \log n)$ -bit tables, given k , j and $\text{SA}[j]$, we can compute $\text{SA}[j - k + 1..j + k - 1]$ and $\text{LCP}[j - k + 1..j + k - 1]$ in $O(k \log r) \subseteq O(k \log n)$ time. (Actually, we can achieve that bound even without the $\text{finger}(j)$ columns in the tables, but Brown et al.'s results suggest those will provide a significant speedup in practice.) With Nishimoto and Tabei's modification, we can reduce that to $O(k)$ time while keeping the tables in $O(r \log n)$ bits; this would slightly improve the time bound we give in the next section to $O(m(k + \log n))$.

► **Lemma 1.** *We can store two $O(r \log n)$ -bit tables such that, given k , j and $\text{SA}[j]$, we can compute $\text{SA}[j - k + 1..j + k - 1]$ and $\text{LCP}[j - k + 1..j + k - 1]$ in $O(k \log n)$ time.*

4 Finding k -MEMs with Lemma 1

We store the tables described in Sections 2 and 3 for T , which add $O(r \log n)$ bits to MONI. Given P and k , we find the MEMs of P with respect to T as before but then, from each $\text{SA}[q_i]$, we use Lemma 1 to find $\text{LCP}[q_i - k + 2..q_i + k - 1]$ in $O(k \log n)$ time.

For example, suppose that $P[0..11] = \text{TAGATTACATTA}$, as in Section 2, and $k = 3$. Starting with $q_{12} = 22$, with MONI we compute the values shown in columns q_i , $\text{SA}[q_i]$, ℓ_i and $\text{BWT}[q_i]$ of Table 5. (It is important that we choose q_i to be one of the endpoints of a run, since we store SA entries only at those positions, but this is true also for MONI.) The crossed out values are the ones we replace because $\text{BWT}[q_i] \neq P[i]$. If we look at the original $\text{SA}[q_i]$ and ℓ_i values, before any replacements, we obtain the matching statistics

$$\text{MS}[0..11] = (38, 5), (9, 4), (0, 8), (1, 7), (2, 6), (20, 5), (21, 4), (22, 3), (1, 4), (2, 3), (3, 2), (4, 1)$$

of P with respect to T , with (pos, len) pair $\text{MS}[i]$ indicating the starting position $\text{MS}[i].\text{pos}$ in T of an occurrence of the longest prefix of $P[i..m - 1]$ that occurs in T , and the length $\text{MS}[i].\text{len}$ of that prefix.

From the matching statistics, it is easy to compute the MEMs $P[0..4] = \text{TAGAT}$, $P[2..9] = \text{GATTACAT}$ and $P[8..11] = \text{ATTA}$ of P with respect to T : a MEM starts at any position i such that $i = 0$ or $\text{MS}[i - 1].\text{len} \leq \text{MS}[i].\text{len}$. For each i , after we compute q_i , $\text{SA}[q_i]$ and ℓ_i (and before we replace them, if we do), we use Lemma 1 to compute the sub-interval $\text{LCP}[q_i - 1..q_i + 2]$ of length $4 = 2k - 2$.

■ **Table 5** With MONI we compute the values shown in columns q_i , $SA[q_i]$, ℓ_i and $BWT[q_i]$ on the left side of the table, and from those we can compute the matching statistics and MEMs of $P[0..11] = TAGATTACATTA$ with respect to $T[0..44] = GATTACAT\#AGATACAT\#GATACAT\#GATTAGAT\#GATTAGATA\$$. After we have computed the values on the left side of the table, we can also compute the values in columns $LCP[q_i - k + 2..q_i + k - 1]$, L_i and $\min(\ell_i, L_i)$ on the right side of the table, and from those we can compute the 3-MEMs of P with respect to T .

i	$P[i]$	q_i	$SA[q_i]$	ℓ_i	$BWT[q_i]$	$LCP[q_i - 1..q_i + 2]$	L_i	$\min(\ell_i, L_i)$
12		22	5	0	A			
11	A	6	4	1	T	[0, 1, 5, 8]	5	1
10	T	37	3	2	T	[1, 2, 6, 9]	6	2
9	T	42	2	3	A	[5, 1, 3, 6]	3	3
8	A	14 19	23 1	2 4	C G	[10, 2, 4, 7]	4	4
7	C	24	22	3	A	[4, 7, 0, 3]	4	3
6	A	8	21	4	T	[5, 8, 1, 4]	5	4
5	T	37 39	3 20	5	T A	[6, 9, 2, 5]	6	5
4	T	42	2	6	A	[5, 1, 3, 6]	3	3
3	A	19	1	7	G	[10, 2, 4, 7]	4	4
2	G	27 29	10 0	3 8	A S	[11, 3, 5, 8]	5	5
1	A	10 11	39 9	4	T #	[4, 5, 1, 3]	4	4
0	T	41	38	5	T	[2, 5, 1, 3]	2	2

We scan each interval $LCP[q_i - k + 2..q_i + k - 1]$ in $O(k)$ time and find a sub-interval of length $k - 1$ such that the minimum LCP value L_i in that sub-interval is maximized. This LCP sub-interval corresponds to a sub-interval of length k in $SA[q_i - k + 1..q_i + k - 1]$ containing the starting positions of k suffixes of T – including $T[SA[q_i]..n - 1]$ itself – whose common prefix with $T[SA[q_i]..n - 1]$ has the maximum possible length L_i .

In our example, we scan each interval in column $LCP[q_i - 1..q_i + 2]$ of Table 5 and find the sub-interval of length 2 such that the minimum LCP value L_i is maximized. If we check Table 1, we find that the longest prefix of $T[4..44] = ACAT\#AGATA\dots$ that occurs at least 3 times in T indeed has length $L_{11} = 5$, the longest prefix of $T[3..44] = TACAT\#AGATA\dots$ that occurs at least 3 times in T indeed has length $L_{10} = 6$, and so on.

Since the common prefix of $P[i..m - 1]$ and $T[SA[q_i]]$ has the maximum possible length ℓ_i , the longest prefix of $P[i..m - 1]$ that occurs at least k times in T has length $\min(\ell_i, L_i)$. Computing $\min(\ell_i, L_i)$ for each i takes a total of $O(km \log n)$ time. The values $\min(\ell_i, L_i)$ are something like a parameterized version of the lengths in the matching statistics: $\min(\ell_i, L_i)$ is the length of the longest prefix of $P[i..m - 1]$ that occurs at least k times in T .

We can compute the k -MEMs of P with respect to T from the $\min(\ell_i, L_i)$ values in the same way we compute MEMs from the lengths in the matching statistics: a k -MEM starts at any position i such that $i = 0$ or $\min(\ell_{i-1}, L_{i-1}) \leq \min(\ell_i, L_i)$. In our example,

$$\begin{aligned} \min(\ell_0, \ell'_0) &\leq \min(\ell_1, \ell'_1) = 4 \\ \min(\ell_1, \ell'_1) &\leq \min(\ell_2, \ell'_2) = 5 \\ \min(\ell_4, \ell'_4) &\leq \min(\ell_5, \ell'_5) = 5 \\ \min(\ell_7, \ell'_7) &\leq \min(\ell_8, \ell'_8) = 4 \end{aligned}$$

and so the k -MEMs are $P[0..1] = TA$, $P[1..4] = AGAT$, $P[2..6] = GATTA$, $P[5..9] = TACAT$ and $P[8..11] = ATTA$.

We can compute $\min(\ell_i, L_i)$ as soon as we have computed $SA[q_i]$ and ℓ_i , so we can compute the k -MEMs of P with respect to T online.

► **Theorem 2.** *Suppose we have MONI for a text $T[0..n-1]$ whose BWT consists of r runs. We can add $O(r \log n)$ bits to MONI such that, given $P[0..m-1]$ and k , we can find the k -MEMs of P with respect to T online in $O(k \log n)$ time per character of P .*

5 Finding k -MEMs with precomputed values

Suppose the interval of length k that we find in SA for $P[i..m-1]$, following the procedures in Section 4, is $\text{SA}[s_i..s_i+k-1]$ and $\text{BWT}[s_i] = \dots = \text{BWT}[s_i+k-1] = P[i-1]$. Then $\min(\ell_{i-1}, L_{i-1}) = \min(\ell_i, L_i) + 1$ and we can find the interval for $P[i-1..m-1]$ with an LF query for s_i , in $O(\log n)$ time. This means we need the results of Section 3 only when at least one character in $\text{BWT}[s_i..s_i+k-1]$ is not equal to $P[i-1]$.

First, suppose $\text{BWT}[q_i] \neq P[i-1]$. Following the procedures in Section 2, MONI resets q_i to the endpoint b of a run in the BWT, resets ℓ_i , and then computes $q_{i-1} = \text{LF}(b)$. Following the procedures in Section 4, we compute $\text{LCP}[q_{i-1}-k+2..q_{i-1}+k-1]$ and scan it to compute the interval $\text{SA}[s_{i-1}..s_{i-1}+k-1]$ for $P[i-1..m-1]$.

If we are given k at construction time, however, then for every endpoint b of a run in the BWT, we can precompute

- the sub-interval of length $k-1$ of $\text{LCP}[\text{LF}(b)-k+2.. \text{LF}(b)+k-1]$ that maximizes the minimum value $L(b)$ in the sub-interval,
- that value $L(b)$.

With this information, we do not need the results of Section 3 for this case either, and can handle it in $O(\log n)$ time as well. Since the sub-interval we store for b starts between $\text{LF}(b)-k+2$ and $\text{LF}(b)+k-1$, we can store it in $O(\log k)$ bits as an offset. This means we store $O(r \log k)$ bits on top of at most $2r$ LCP values, or $O(r \log n)$ bits in total.

The remaining case is when $\text{BWT}[q_i] = P[i-1]$ but some of the other characters in $\text{BWT}[s_i..s_i+k-1]$ are not equal to $P[i-1]$. If $\text{BWT}[q_i]$ is the end of a run, then we can proceed as in the previous case in $O(\log n)$ time, using our precomputed values for q_i (but without resetting q_i and ℓ_i). Otherwise, we claim we can choose such a character $\text{BWT}[b] = P[i-1]$ at the end of a run, set

$$\ell_i = \min(\text{LCE}(\text{SA}[q_i], \text{SA}(b)), \ell_i)$$

and $q_i = b$, and then proceed as in the previous case in $O(\log n)$ time, and still be sure of obtaining the correct k -MEMs of P with respect to T . (Continuing to run MONI with the new values of q_i and ℓ_i may not give us the correct MEMs, however.) To be able to change q_i and ℓ_i this way, it is important that we now work online, instead of running MONI on P and then using the results to find the k -MEMs.

To see why our claim holds, assume our query has worked correctly so far, so

$$T[\text{SA}[q_i].. \text{SA}[q_i] + \min(\ell_i, L_i) - 1] = T[\text{SA}[b].. \text{SA}[b] + \min(\ell_i, L_i) - 1]$$

is the longest prefix of $P[i..m-1]$ that occurs at least k times in T . Therefore, the k -MEMs starting in $P[0..i-1]$ are all completely contained in $P[0..i + \min(\ell_i, L_i) - 1]$. It follows that resetting

$$\ell_i = \min(\text{LCE}(\text{SA}[q_i], \text{SA}(b)), \ell_i)$$

and $q_i = b$ does not affect the set of k -MEMs we find that start in $P[0..i-1]$.

```

if BWT[si] = ... = BWT[si + k - 1] = P[i - 1] then
  qi-1 ← LF(qi)
  ℓi-1 ← ℓi + 1
  Li-1 ← Li + 1
  si-1 = LF(si)
else
  if BWT[qi] ≠ P[i - 1] then
    reset qi and ℓi as MONI does
  else if BWT[qi] is not at the end of a run
    choose b in [si..si + k - 1] with BWT[b] = P[i - 1] at the end of a run
    ℓi ← min(LCE(SA[qi], SA[b]), ℓi)
    qi ← b
  end if
  qi-1 ← LF(qi)
  ℓi-1 ← ℓi + 1
  Li-1 ← L(qi)
  si-1 ← LF(qi) - offset(qi)
end if

```

■ **Figure 1** Pseudo-code for how we find k -MEMs with precomputed values.

Figure 1 shows pseudo-code for how we find k -MEMs with precomputed values. Table 6 shows the offsets and $L(b)$ values for our example, surrounded by coloured boxes on the right, with each offset indicating how far above $\text{LF}(b)$ the sub-interval starts. The coloured boxes on the left indicate the sub-interval itself and the longest common prefix of the suffixes starting in the sub-interval of the SA.

For our example, suppose we again start with $q_{12} = 22$ and $\ell_{12} = 0$. Since $\text{BWT}[q_{12}] = P[11] = \text{A}$, we set $q_{11} = \text{LF}(22) = 6$ and $\ell_{11} = \ell_{12} + 1 = 1$. The values $\text{offset}(22) = 0$ and $L(22) = 5$ in the black rectangle in Table 6 tell us to set $s_{11} = \text{LF}(22) - 0 = 6$ and $L_{11} = 5$. This means the suffixes of T with starting points in

$$\text{SA}[6..8] = [4, 13, 21]$$

have a longest common prefix of length 5, which starts with the longest prefix of $P[11]$ that occurs at least 3 times in T . This longest prefix has length $\min(\ell_{11}, L_{11}) = 1$ – so it is just $P[11] = \text{A}$. After this initial setup, we can fill in Table 7 according to the pseudo-code in Figure 1, with crossed out values again indicating those that are replaced.

► **Theorem 3.** *Suppose we have MONI for a text $T[0..n - 1]$ whose BWT consists of r runs. Given k , we can add $O(r \log n)$ bits to MONI such that, given $P[0..m - 1]$, we can find the k -MEMs of P with respect to T online in $O(\log n)$ time per character of P .*

6 Conclusion

We have shown, first, how we can add $O(r \log n)$ bits to MONI for a text $T[0..n - 1]$, where r is the number of runs in the BWT of T , such that if we are given k at query time with $P[0..m - 1]$, then we can find the k -MEMs of P with respect to T online in $O(k \log n)$ time per character of P . We have then shown how, if we are given k at construction time, we can add $O(r \log k)$ bits and at most $2r$ LCP values – which are $O(r \log n)$ bits in total – such

26:12 MONI Can Find k -MEMs

■ **Table 6** The table showing the precomputed values we use to find 3-MEMs with respect to our example $T = \text{GATTACAT}\#\text{AGATACAT}\#\text{GATACAT}\#\text{GATTAGAT}\#\text{GATTAGATA}\$$.

i	$SA[i]$	$LCP[i]$	lexicographically i th cyclic shift of T	BWT[i]	LF(i)	offset(i)	$L(i)$
0	44	0	\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA	A	5	0	1
1	8	0	#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT	T	32	0	2
2	17	1	#GATTACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT	T	33		
3	25	4	#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT	T	34		
4	34	9	#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT	T	35		
5	43	0	\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGAT	T	36		
6	4	1	ACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATT	T	37		
7	13	5	ACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGAT	T	38		
8	21	8	ACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GAT	T	39		
9	30	1	AGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATT	T	40		
10	39	4	AGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATT	T	41	2	2
11	9	5	AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#	#	1	0	1
12	6	1	AT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTAC	C	22	0	4
13	15	3	AT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATAC	C	23		
14	23	6	AT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATAC	C	24	2	4
15	32	11	AT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAG	G	25	0	3
16	41	2	ATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAG	G	26		
17	11	3	ATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AG	G	27		
18	19	10	ATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#	G	28		
19	1	2	ATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$G	G	29		
20	27	4	ATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#G	G	30		
21	36	7	ATTAGATA\$GATTAGAT#AGATACAT#GATACAT#GATTAGAT#G	G	31	2	5
22	5	0	CAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTAC	A	6	0	5
23	14	4	CAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATA	A	7		
24	22	7	CAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATA	A	8		
25	31	0	GAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAG	A	9		
26	40	3	GATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAG	A	10		
27	10	4	GATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AG	A	11	2	4
28	18	11	GATACAT#GATTAGAT#GATTAGATA\$GATTAGAT#AGATACAT#	#	2	0	4
29	0	3	GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$	\$	0	0	0
30	26	5	GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#	#	3	1	4
31	35	8	GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#	#	4	2	4
32	7	0	T#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT	A	12	0	3
33	16	2	T#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACA	A	13		
34	24	5	T#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACA	A	14		
35	33	10	T#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT	A	15		
36	42	1	TA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGAT	A	16	0	3
37	3	2	TACAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GATT	T	42	0	3
38	12	6	TACAT#GATACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGAT	A	17	1	3
39	20	9	TACAT#GATTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GA	A	18	2	3
40	29	2	TACAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GAT	T	43	1	3
41	38	5	TACATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GATT	T	44	2	3
42	2	1	TTAGAT#AGATACAT#GATACAT#GATTAGAT#GATTAGATA\$GA	A	19	0	4
43	28	3	TTAGAT#GATTAGATA\$GATTACAT#AGATACAT#GATACAT#GA	A	20		
44	37	6	TTAGATA\$GATTACAT#AGATACAT#GATACAT#GATTAGAT#GA	A	21	2	4

■ **Table 7** The values we compute (except $\text{BWT}[s_i..s_i + 2]$, which we include here only for clarity) while finding the 3-MEMs of $P[0..11] = \text{TAGATTACATTA}$ with respect to our example $T = \text{GATTACAT}\#\text{AGATACAT}\#\text{GATACAT}\#\text{GATTAGAT}\#\text{GATTAGATA}\$$.

i	q_i	ℓ_i	L_i	$\min(\ell_i, L_i)$	s_i	$P[i - 1]$	$\text{BWT}[q_i]$	$\text{BWT}[s_i..s_i + 2]$
12	22	0				A	A	
11	6	1	5	1	6	T	T	TTT
10	37	2	6	2	37	T	T	TAA
9	42	3	3	3	42	A	A	AAA
8	14 19	2 4	4	4	19	C	C G	GGG
7	24	3	4	3	22	A	A	AAA
6	8	4	5	4	6	T	T	TTT
5	37 39	5	6	5	37	T	T A	TAA
4	42	6	3	3	42	A	A	AAA
3	19	7	4	4	19	G	G	GGG
2	27 29	3 8	5	5	29	A	A \$	\$\$\$
1	10 11	4	4	4	9	T	T #	TT#
0	41	5	2	2	39		T	ATT

that we can find the k -MEMs of P with respect to T online in $O(\log n)$ time per character of P . Along the way, we have also shown how to extend ϕ queries to support sequential access to the LCP, which may be of independent interest.

Although we have not discussed construction, we expect it will not be difficult to modify prefix-free parsing [2] to build our tables for ϕ , LCP and ϕ^{-1} queries. Once we can support those queries, we can use them to compute k -MEMs in $O(km \log n)$ time, or to build in $O(kr)$ time the table of precomputed values that we need to compute k -MEMs in $O(m \log n)$ time. In fact, once we have built the tables for ϕ , LCP and ϕ^{-1} queries – which take $O(r)$ space but may be significantly larger than our table of precomputed values – then we can store them in external memory and recover them only when we want to build a table of precomputed values for a different choice of k .

We believe our approach is a practical extension of MONI and we are currently implementing it. One possible application might be to index two genomic databases (possibly with two different values of k), one of haplotypes from people with symptoms of a genetic disease and one of haplotypes from people without; then, as the first step in a bioinformatics pipeline, we could use those indexes to mine for substrings that are common in one database and not in the other. We think this application is interesting because, except for a remark in Bannai et al.’s paper about potentially applying MEM-finding to rare-disease diagnosis, the r -index and MONI have so far been considered only as tools for pangenomic *alignment*, and this is an application to pangenomic *analysis*. If the disease is recessive or multifactorial then variations associated with it are likely to be present in both databases, so MEM-finding is unlikely to detect them; those variations could be more frequent in the first database, however, so k -MEM-finding may still be useful.

References

- 1 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r -index. *Theoretical Computer Science*, 812:96–108, 2020.
- 2 Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *2021 Data Compression Conference (DCC)*, pages 193–202. IEEE, 2021.

- 3 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):1–15, 2019.
- 4 Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. RLBWT tricks. In *20th Symposium on Experimental Algorithms (SEA 2022)*, pages 16:1–16:16, 2022.
- 5 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.
- 6 Juha Kärkkäinen, Dominik Kempa, and Marcin Piątkowski. Tighter bounds for the sum of irreducible LCP values. *Theoretical Computer Science*, 656:265–278, 2016.
- 7 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *20th Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192. Springer, 2009.
- 8 Heng Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint*, 2013. [arXiv:1303.3997](https://arxiv.org/abs/1303.3997).
- 9 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
- 10 Gonzalo Navarro. Computing MEMs on repetitive text collections. *arXiv preprint v3*, 2022. Accepted to this conference. [arXiv:2210.09914](https://arxiv.org/abs/2210.09914).
- 11 Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An optimal-time RLBWT construction in BWT-runs bounded space. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, pages 99:1–99:20, 2022.
- 12 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, 2021.
- 13 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.