

33rd Annual Symposium on Combinatorial Pattern Matching

CPM 2022, June 27–29, 2022, Prague, Czech Republic

Edited by

Hideo Bannai

Jan Holub



Editors

Hideo Bannai 

M&D Data Science Center, Tokyo Medical and Dental University, Tokyo, Japan
hdbn.dsc@tmd.ac.jp

Jan Holub 

Department of Theoretical Computer Science, Czech Technical University in Prague, Czech Republic
Jan.Holub@fit.cvut.cz

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 978-3-95977-234-1

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/978-3-95977-234-1>.

Publication date

June, 2022

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.2022.0

ISBN 978-3-95977-234-1

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)
- 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 algorithmic stringologists in the world

■ Contents

Preface	
<i>Hideo Bannai and Jan Holub</i>	0:xi
Programme Committee	
.....	0:xiii
External Subreviewers	
.....	0:xv
Authors of Selected Papers	
.....	0:xvii

Invited Talks

Invitation to Combinatorial Reconfiguration	
<i>Takehiro Ito</i>	1:1–1:1
Using Automata and a Decision Procedure to Prove Results in Pattern Matching	
<i>Jeffrey Shallit</i>	2:1–2:3
Compact Text Indexing for Advanced Pattern Matching Problems: Parameterized, Order-Isomorphic, 2D, etc.	
<i>Sharma V. Thankachan</i>	3:1–3:3

Regular Papers

The Fine-Grained Complexity of Episode Matching	
<i>Philip Bille, Inge Li Gørtz, Shay Mozes, Teresa Anna Steiner, and Oren Weimann</i>	4:1–4:12
Mechanical Proving with Walnut for Squares and Cubes in Partial Words	
<i>John Machacek</i>	5:1–5:11
An FPT-Algorithm for Longest Common Subsequence Parameterized by the Maximum Number of Deletions	
<i>Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau</i>	6:1–6:11
Beyond the Longest Letter-Duplicated Subsequence Problem	
<i>Wenfeng Lai, Adiesha Liyanage, Binhai Zhu, and Peng Zou</i>	7:1–7:12
Reduction Ratio of the IS-Algorithm: Worst and Random Cases	
<i>Vincent Jugé</i>	8:1–8:23
Arbitrary-Length Analogs to de Bruijn Sequences	
<i>Abhinav Nellore and Rachel Ward</i>	9:1–9:20
Partial Permutations Comparison, Maintenance and Applications	
<i>Avivit Levy, Ely Porat, and B. Riva Shalom</i>	10:1–10:17
Bi-Directional r-Indexes	
<i>Yuma Arakawa, Gonzalo Navarro, and Kunihiko Sadakane</i>	11:1–11:14

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub



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

Making de Bruijn Graphs Eulerian <i>Giulia Bernardini, Huiping Chen, Grigorios Loukides, Solon P. Pissis, Leen Stougie, and Michelle Sweering</i>	12:1–12:18
Back-To-Front Online Lyndon Forest Construction <i>Golnaz Badkobeh, Maxime Crochemore, Jonas Ellert, and Cyril Nicaud</i>	13:1–13:23
Cartesian Tree Subsequence Matching <i>Tsubasa Oizumi, Takeshi Kai, Takuya Mieno, Shunsuke Inenaga, and Hiroki Arimura</i>	14:1–14:18
Polynomial-Time Equivalences and Refined Algorithms for Longest Common Subsequence Variants <i>Yuichi Asahiro, Jesper Jansson, Guohui Lin, Eiji Miyano, Hirotaka Ono, and Tadatashi Utashima</i>	15:1–15:17
On Strings Having the Same Length- k Substrings <i>Giulia Bernardini, Alessio Conte, Esteban Gabory, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, Giulia Punzi, and Michelle Sweering</i>	16:1–16:17
The Normalized Edit Distance with Uniform Operation Costs Is a Metric <i>Dana Fisman, Joshua Grogin, Oded Margalit, and Gera Weiss</i>	17:1–17:17
The Dynamic k -Mismatch Problem <i>Raphaël Clifford, Paweł Gawrychowski, Tomasz Kociumaka, Daniel P. Martin, and Przemysław Uznański</i>	18:1–18:15
Indexable Elastic Founder Graphs of Minimum Height <i>Nicola Rizzo and Veli Mäkinen</i>	19:1–19:19
Longest Palindromic Substring in Sublinear Time <i>Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski</i>	20:1–20:9
Permutation Pattern Matching for Doubly Partially Ordered Patterns <i>Laurent Bulteau, Guillaume Fertin, Vincent Jugé, and Stéphane Vialette</i>	21:1–21:17
Linear-Time Computation of Shortest Covers of All Rotations of a String <i>Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Waleń, and Wiktor Zuba</i>	22:1–22:15
Rectangular Tile Covers of 2D-Strings <i>Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Waleń, and Wiktor Zuba</i>	23:1–23:14
Reordering a Tree According to an Order on Its Leaves <i>Laurent Bulteau, Philippe Gambette, and Olga Seminck</i>	24:1–24:15
A Theoretical and Experimental Analysis of BWT Variants for String Collections <i>Davide Cenzato and Zsuzsanna Lipták</i>	25:1–25:18
RePair Grammars Are the Smallest Grammars for Fibonacci Words <i>Takuya Mieno, Shunsuke Inenaga, and Takashi Horiyama</i>	26:1–26:17
Minimal Absent Words on Run-Length Encoded Strings <i>Tooru Akagi, Kouta Okabe, Takuya Mieno, Yuto Nakashima, and Shunsuke Inenaga</i>	27:1–27:17

Parallel Algorithm for Pattern Matching Problems Under Substring Consistent Equivalence Relations <i>Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara</i>	28:1–28:21
Efficient Construction of the BWT for Repetitive Text Using String Compression <i>Diego Díaz-Domínguez and Gonzalo Navarro</i>	29:1–29:18

■ 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 33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022) held on June 27–29, 2022 in Prague, Czech Republic (in a hybrid mode due to the continuing Covid-19 pandemic). The conference program includes 26 contributed papers and three invited talks by Takehiro Ito (Tohoku University, Japan), Jeffrey Shallit (University of Waterloo, Canada), and Sharma V. Thankachan (University of Central Florida, USA). For the fourth time, CPM includes the “Highlights of CPM” special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this fourth edition we invited Moses Ganardi (Max Planck Institute for Software Systems (MPI-SWS), Germany) to present his ESA 2021 paper “Compression by Contracting Straight-Line Programs” and Tomasz Kociumaka (University of California, Berkeley) to present a FOCS 2021 paper by T. Kociumaka, E. Porat and T. Starikovskaya “Small space and streaming pattern matching with k edits”.

The contributed papers were selected out of 43 submissions, corresponding to an acceptance ratio of about 60%. 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, and Wrocław. From 1992 to the 2015 meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, the CPM proceedings appear in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM 2016), 78 (CPM 2017), 105 (CPM 2018), 128 (CPM 2019), 161 (CPM 2020), and 191 (CPM 2021).

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.

Hideo Bannai and Jan Holub



■ Programme Committee

Golnaz Badkobeh
Goldsmiths University of London, UK

Hideo Bannai (co-chair)
Tokyo Medical and Dental University, Japan

Frédérique Bassino
University Paris 13, France

Djamal Belazzougui
DTISI-CERIST, Algeria

Philip Bille
Technical University of Denmark, Denmark

Christina Boucher
University of Florida, USA

Martin Farach-Colton
Rutgers University, USA

Gabriele Fici
University of Palermo, Italy

Johannes Fischer
TU Dortmund University, Germany

Travis Gagie
Dalhousie University, Canada

Paweł Gawrychowski
University of Wrocław, Poland

Jan Holub (co-chair)
Czech Technical University in Prague, Czech
Republic

Wing-Kai Hon
National Tsing Hua University, Taiwan

Tomohiro I
Kyushu Institute of Technology, Japan

Shunsuke Inenaga
Kyushu University, Japan

Dominik Kempa
Stony Brook University, USA

Tomasz Kociumaka
University of California, Berkeley, USA

Dmitry Kosolobov
Ural Federal University, Russia

Gad M. Landau
University of Haifa, Israel

Veli Mäkinen
University of Helsinki, Finland

Florin Manea
Göttingen University, Germany

Sebastian Maneth
University of Bremen, Germany

Robert Mercas
Loughborough University, UK

Gonzalo Navarro
University of Chile, Chile

Kunsoo Park
Seoul National University, South Korea

Nadia Pisanti
University of Pisa, Italy

Solon Pissis
CWI, Netherlands

Jakub Radoszewski
University of Warsaw, Poland



■ External Subreviewers

Alexandru Popa

Bartłomiej Dudek

Bastien Cazaux

Cristian Urbina

Daniel Gabric

Daniel Gibney

Daniel Saad

Danny Hermelin

Debarati Das

Diego Diaz

Dominik Köppl

Dominik Peters

Estéban Gabory

Florian Kurpicz

Francisco Olivares

Garance Gourdel

Giulia Bernardini

Grigorios Loukides

Henning Fernau

Itai Boneh

Jeffrey Shallit

Juliusz Straszyński

Karol Pokorski

Laszlo Kozma

Luis M. S. Russo

Marcin Piątkowski

Mark Daniel Ward

Markus L. Schmid

Marvin Künnemann

Massimiliano Rossi

Massimo Equi

Michelle Sweering

Moses Ganardi

Nathan Wallheimer

Oleg Merkurev

Olivier Carton

Panagiotis Charalampopoulos

Paweł Parys

Samah Ghazawi

Sharma V. Thankachan

Shoshana Marcus

Sven Rahmann

Takuya Mieno

Thierry Lecroq

Tomasz Walen

Tuukka Norri

Veronica Guerrini

Vincent Jugé

Vít Jelínek

Wiktoria Zuba

Wojciech Janczewski

Yuto Nakashima



■ Authors of Selected Papers

Abhinav Nellore	Jesper Jansson
Adiesha Liyanage	John Machacek
Alessio Conte	Jonas Ellert
Avivit Levy	Joshua Grogin
Ayumi Shinohara	Juliusz Straszyński
B. Riva Shalom	Kouta Okabe
Binhai Zhu	Kunihiko Sadakane
Costas Iliopoulos	Laurent Bulteau
Cyril Nicaud	Leen Stougie
Dana Fisman	Mark Jones
Daniel Martin	Maxime Crochemore
Davaajav Jargalsaikhan	Michelle Sweering
Davide Cenzato	Nicola Rizzo
Diego Diaz	Oded Margalit
Diptarama Hendrian	Olga Semincik
Eiji Miyano	Oren Weimann
Ely Porat	Panagiotis Charalampopoulos
Esteban Gabory	Pawel Gawrychowski
Gera Weiss	Peng Zou
Giulia Bernardini	Philip Bille
Giulia Punzi	Philippe Gambette
Golnaz Badkobeh	Przemysław Uznański
Gonzalo Navarro	Rachel Ward
Grigorios Loukides	Raphael Clifford
Guillaume Fertin	Roberto Grossi
Guohui Lin	Rolf Niedermeier
Hiroki Arimura	Ryo Yoshinaka
Hiroataka Ono	Shay Mozes
Huiping Chen	Shunsuke Inenaga
Inge Li Gørtz	Solon Pissis
Jakub Radoszewski	Stéphane Vialette

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub



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

0:xviii Authors of Selected Papers

Tadatoshi Utashima

Takashi Horiyama

Takeshi Kai

Takuya Mieno

Teresa Anna Steiner

Till Tantau

Tomasz Kociumaka

Tomasz Walen

Tooru Akagi

Tsubasa Oizumi

Veli Mäkinen

Vincent Jugé

Wenfeng Lai

Wiktor Zuba

Wojciech Rytter

Yuichi Asahiro

Yuma Arakawa

Yuto Nakashima

Zsuzsanna Liptak

Invitation to Combinatorial Reconfiguration

Takehiro Ito   

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Abstract

Combinatorial reconfiguration studies reachability and related questions over the solution space formed by feasible solutions of an instance of a combinatorial search problem. For example, as the solution space for the SATISFIABILITY problem, we may consider the subgraph of the hypercube induced by the satisfying truth assignments of a given CNF formula. Then, the reachability problem for SATISFIABILITY is the problem of asking whether two given satisfying truth assignments are contained in the same connected component of the solution space. The study of reconfiguration problems has motivation from a variety of fields such as puzzles, statistical physics, and industry. In this decade, reconfiguration problems have been studied intensively for many central combinatorial search problems, such as SATISFIABILITY, INDEPENDENT SET and COLORING, from the algorithmic viewpoints. Many reconfiguration problems are PSPACE-complete in general, although several efficiently solvable cases have been obtained. In this talk, I will give a broad introduction of combinatorial reconfiguration.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases Combinatorial reconfiguration, graph algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.1

Category Invited Talk

Funding Partially supported by JSPS KAKENHI Grant Numbers JP18H04091, JP19K11814 and JP20H05793, Japan.



© Takehiro Ito;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



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

Using Automata and a Decision Procedure to Prove Results in Pattern Matching

Jeffrey Shallit   

School of Computer Science, University of Waterloo, Canada

Abstract

The first-order theory of automatic sequences with addition is decidable, and this means that one can often prove combinatorial properties of these sequences “automatically”, using the free software **Walnut** written by Hamoon Mousavi. In this talk I will explain how this is done, using as an example the measure of minimize size string attractor, introduced by Kempa and Prezza in 2018.

Using the logic-based approach, we can also prove more general properties of string attractors for automatic sequences. This is joint work with Luke Schaeffer.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words; Theory of computation → Regular languages; Theory of computation → Logic and verification

Keywords and phrases finite automata, decision procedure, automatic sequence, Thue-Morse sequence, Fibonacci word, string attractor

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.2

Category Invited Talk

Related Version *Full Version:* <https://arxiv.org/abs/2012.06840>

Funding Research supported by NSERC 2018-04118.

1 Introduction

Many famous sequences, such as the Thue-Morse sequence $\mathbf{t} = 01101001\dots$ and the Fibonacci infinite word $\mathbf{f} = 01001010\dots$ appear as fundamental examples in combinatorial pattern matching.

As just a few examples, I point to [5, 1, 12], where the Thue-Morse sequence makes an appearance, and [13], where the Fibonacci infinite word is studied.

A fundamental result, essentially due to Büchi [4] and Bruyère et al. [3], tells us that the first-order theory of such sequences, with addition, is decidable, and there is a relatively simple decision procedure based on automata. This decision procedure has been implemented in free software called **Walnut**, originally created by Hamoon Mousavi [11]. Therefore, in many cases, we can prove properties of such sequences of interest to the CPM community “automatically”, merely by stating the desired property in first-order logic, and invoking **Walnut**.

Recently there has been interest in a certain measure of repetitivity, based on string attractors, originally introduced by Kempa and Prezza [6], and studied further in [9, 7, 8, 10, 2]. A *string attractor* of a finite word $w = w[0..n-1]$ is a subset $S \subseteq \{0, 1, \dots, n-1\}$ such that every nonempty factor f of w has an occurrence that touches at least one of the indices of S . For example, $\{2, 3, 4\}$ is a string attractor of minimum size for the French word **entente**.

In this talk I will introduce **Walnut**, and explain how to obtain results on string attractors using it and the theory behind it. This is joint work with Luke Schaeffer [14].



© Jeffrey Shallit;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 2; pp. 2:1–2:3

Leibniz International Proceedings in Informatics



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

2 Results

As an example of the kind of thing we can prove with `Walnut`, here is one theorem:

► **Theorem 1.** *Let a_n denote the size of the smallest string attractor for the length- n prefix of the Thue-Morse word \mathbf{t} . Then*

$$a_n = \begin{cases} 1, & \text{if } n = 1; \\ 2, & \text{if } 2 \leq n \leq 6; \\ 3, & \text{if } 7 \leq n \leq 14 \text{ or } 17 \leq n \leq 24; \\ 4, & \text{if } n = 15, 16 \text{ or } n \geq 25. \end{cases}$$

More generally, we can prove

► **Theorem 2.** *Let \mathbf{w} be a k -automatic sequence. Either*

- *every factor $\mathbf{w}[i..i + \ell - 1]$ has a string attractor of constant size, and there exists a finite automaton outputting the minimum size given i and ℓ , or*
- *for all $n \geq 1$, the minimum size string attractor for the length- n prefix $\mathbf{w}[0..n - 1]$ grows as $\Theta(\log n)$,*

and we can decide which is the case for \mathbf{w} .

For more about `Walnut` and its applications in combinatorics on words, see my forthcoming book [15].

References

- 1 A. Amir, Y. Aumann, A. Levy, and Y. Roshko. Quasi-distinct parsing and optimal compression methods. In G. Kucherov and E. Ukkonen, editors, *CPM 2009*, volume 5577 of *Lecture Notes in Computer Science*, pages 12–25. Springer-Verlag, 2009.
- 2 H. Bannai, M. Funakoshi, T. I. D. Köppl, T. Mieno, and T. Nishimoto. A separation of γ and b via Thue–Morse words. In T. Lecroq and H. Touzet, editors, *SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 168–178. Springer-Verlag, 2021.
- 3 V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p -recognizable sets of integers. *Bull. Belgian Math. Soc.*, 1:191–238, 1994. Corrigendum, *Bull. Belg. Math. Soc.* 1:577, 1994.
- 4 J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960. Reprinted in S. Mac Lane and D. Siefkes, eds., *The Collected Works of J. Richard Büchi*, Springer-Verlag, 1990, pp. 398–424.
- 5 M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nordic J. Computing*, 4:172–186, 1997.
- 6 D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *STOC'18 Proceedings*, pages 827–840. ACM Press, 2018.
- 7 T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. In Y. Kohayakawa and F. K. Miyazawa, editors, *LATIN 2020*, volume 12118 of *Lecture Notes in Computer Science*, pages 207–219. Springer-Verlag, 2020.
- 8 K. Kutsukake, T. Matsumoto, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. On repetitiveness measures of Thue-Morse words. In C. Boucher and S. V. Thankachan, editors, *SPIRE 2020*, volume 12303 of *Lecture Notes in Computer Science*, pages 213–220. Springer-Verlag, 2020.
- 9 S. Mantaci, A. Restivo, G. Romana, G. Rosone, and M. Sciortino. String attractors and combinatorics on words. In *ICTCS 2019*, volume 2504 of *CEUR Workshop Proceedings*, pages 57–71, 2019. Available at <http://ceur-ws.org/Vol-2504/paper8.pdf>.

- 10 S. Mantaci, A. Restivo, G. Romana, G. Rosone, and M. Sciortino. A combinatorial view on string attractors. *Theoret. Comput. Sci.*, 850:236–248, 2021.
- 11 H. Mousavi. Automatic theorem proving in Walnut. Arxiv preprint arXiv:1603.06017 [cs.FL], 2016. [arXiv:1603.06017](#).
- 12 J. Radoszewski and W. Rytter. On the structure of compacted subword graphs of Thue-Morse words and their applications. *J. Discrete Algorithms*, 11:15–24, 2012.
- 13 W. Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theoret. Comput. Sci.*, 363:211–223, 2006.
- 14 L. Schaeffer and J. Shallit. String attractors for automatic sequences. Arxiv preprint arXiv:2012.06840 [cs.FL], 2021. [arXiv:2012.06840](#).
- 15 J. Shallit. *The Logical Approach To Automatic Sequences: Exploring Combinatorics on Words with Walnut*. Cambridge University Press, 2022. In press.

Compact Text Indexing for Advanced Pattern Matching Problems: Parameterized, Order-Isomorphic, 2D, etc.

Sharma V. Thankachan ✉

Department of Computer Science, University of Central Florida, Orlando, FL, USA

Abstract

In the past two decades, we have witnessed the design of various compact data structures for pattern matching over an indexed text [22]. Popular indexes like the FM-index [6], compressed suffix arrays/trees [15, 26], the recent r-index [8, 23], etc., capture the key functionalities of classic suffix arrays/trees [20, 28] in compact space. Mostly, they rely on the Burrows-Wheeler Transform (BWT) and its associated operations [2]. However, compactly encoding some advanced suffix tree (ST) variants, like parameterized ST [1, 19, 21], order-isomorphic/preserving ST [4], two-dimensional ST [14, 16], etc. [24, 27]- collectively known as suffix trees with missing suffix links [3], has been challenging. The previous techniques are not easily extendable because these variants do not hold some structural properties of the standard ST that enable compression. However, some limited progress has been made in these directions recently [11, 7, 5, 25, 10, 18, 17, 12, 13, 9]. This talk will briefly survey them and highlight some interesting open problems.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Text Indexing, Suffix Trees, String Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.3

Category Invited Talk

Funding Supported in part by the US NSF grants CCF-1527435, CCF-2112643 and CCF-2137057.

Acknowledgements I want to thank my collaborators Rahul Shah and Arnab Ganguly.

References

- 1 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, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993. doi:10.1145/167088.167115.
- 2 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.
- 3 Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003. doi:10.1137/S0097539701424465.
- 4 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.
- 5 Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A compact index for order-preserving pattern matching. In *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4-7, 2017*, pages 72–81, 2017. doi:10.1109/DCC.2017.35.
- 6 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 7 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.
- 8 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. doi:10.1145/3375890.



© Sharma V. Thankachan;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 3; pp. 3:1–3:3

Leibniz International Proceedings in Informatics



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

- 9 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. A framework for designing space-efficient dictionaries for parameterized and order-preserving matching. *Theor. Comput. Sci.*, 854:52–62, 2021. doi:10.1016/j.tcs.2020.11.036.
- 10 Arnab Ganguly, Dhruvil Patel, Rahul Shah, and Sharma V. Thankachan. LF successor: Compact space indexing for order-isomorphic pattern matching. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPICs*, pages 71:1–71:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ICALP.2021.71.
- 11 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pbwt: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, January 16-19*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.
- 12 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, volume 92 of *LIPICs*, pages 35:1–35:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ISAAC.2017.35.
- 13 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Fully functional parameterized suffix trees in compact space. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- 14 Raffaele Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.
- 15 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 16 Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Constructing suffix arrays for multi-dimensional matrices. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 126–139. Springer, 1998. doi:10.1007/BFb0030786.
- 17 Sung-Hwan Kim and Hwan-Gue Cho. A compact index for cartesian tree matching. 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 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.18.
- 18 Sung-Hwan Kim and Hwan-Gue Cho. Simpler fin-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. doi:10.1016/j.ipl.2020.106026.
- 19 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995. doi:10.1109/SFCS.1995.492664.
- 20 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.
- 21 Juan Mendivelso, Sharma V. Thankachan, and Yoan J. Pinzón. A brief history of parameterized matching problems. *Discret. Appl. Math.*, 274:103–115, 2020. doi:10.1016/j.dam.2018.07.017.
- 22 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.
- 23 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPICs*, pages 101:1–101:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ICALP.2021.101.

- 24 Sung Gwan Park, Amihoud Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.16.
- 25 Dhrumil Patel and Rahul Shah. Inverse suffix array queries for 2-dimensional pattern matching in near-compact space. 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 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ISAAC.2021.60.
- 26 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.
- 27 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 393–406, 2000. doi:10.1007/3-540-44985-X_34.
- 28 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, 1973. doi:10.1109/SWAT.1973.13.

The Fine-Grained Complexity of Episode Matching

Philip Bille  



Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz  

Technical University of Denmark, Lyngby, Denmark

Shay Mozes  

The Interdisciplinary Center Herzliya, Israel

Teresa Anna Steiner  

Technical University of Denmark, Lyngby, Denmark

Oren Weimann  

University of Haifa, Israel

Abstract

Given two strings S and P , the *Episode Matching* problem is to find the shortest substring of S that contains P as a subsequence. The best known upper bound for this problem is $\tilde{O}(nm)$ by Das et al. (1997), where n, m are the lengths of S and P , respectively. Although the problem is well studied and has many applications in data mining, this bound has never been improved. In this paper we show why this is the case by proving that no $O((nm)^{1-\epsilon})$ algorithm (even for binary strings) exists, unless the *Strong Exponential Time Hypothesis (SETH)* is false.

We then consider the indexing version of the problem, where S is preprocessed into a data structure for answering episode matching queries P . We show that for any τ , there is a data structure using $O(n + (\frac{n}{\tau})^k)$ space that answers episode matching queries for any P of length k in $O(k \cdot \tau \cdot \log \log n)$ time. We complement this upper bound with an almost matching lower bound, showing that any data structure that answers episode matching queries for patterns of length k in time $O(n^\delta)$, must use $\Omega(n^{k-k\delta-o(1)})$ space, unless the *Strong k -Set Disjointness Conjecture* is false. Finally, for the special case of $k = 2$, we present a faster construction of the data structure using fast min-plus multiplication of bounded integer matrices.

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

Keywords and phrases Pattern matching, fine-grained complexity, longest common subsequence

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.4

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

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

Shay Mozes: Israel Science Foundation grant 810/21.

Oren Weimann: Israel Science Foundation grant 810/21.

1 Introduction

A string P is a *subsequence* of a string S if P can be obtained by deleting characters from S . Given two strings S and P , the *Episode Matching* problem [15] is to find the shortest substring of S that contains P as a subsequence. In its indexing version, we are given S in advance and we need to preprocess it into a data structure for answering episode matching queries P .



© Philip Bille, Inge Li Gørtz, Shay Mozes, Teresa Anna Steiner, and Oren Weimann;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 4; pp. 4:1–4:12

Leibniz International Proceedings in Informatics



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

1.1 Episode matching

The Episode Matching problem was introduced by Das et al. [15] in 1997 as a simplified version of the frequent episode discovery problem first studied by Mannila et al. [31]. Das et al. [15] gave an upper bound of $O(nm/\log n)$, where n is the length of S and m is the length of P . Even though the problem and its variations have been thoroughly studied [6, 11, 13, 14, 29, 30] and have many applications in data mining [6, 7, 22, 26, 31], the original $O(nm/\log n)$ upper bound has never been improved. In Section 3 we show why this is the case, by proving a lower bound conditioned on the *Strong Exponential Time Hypothesis (SETH)* (actually, on the Orthogonal Vectors Hypothesis (OV)). This continues a recent line of research on quadratic lower bounds for string problems conditioned on SETH [1, 3, 8, 9, 12, 16, 17, 19, 27, 33]. Our reduction is very simple and easily extends to binary alphabet and to unbalanced inputs. Our result is summarized by the following theorem.

► **Theorem 1.** *For any $\epsilon > 0$, Episode Matching on binary strings of lengths n and $m = n^\alpha$ (for any fixed $\alpha \leq 1$) cannot be solved in $O((mn)^{1-\epsilon})$ time, unless SETH is false.*

Related work. A related problem to episode matching is the *Longest Common Subsequence (LCS)* problem. In that problem, the goal is to find for two strings S and P (of lengths n and m respectively) the longest string that is a subsequence of both S and P . In 2015 Abboud et al. [1] and Bringmann and Künnemann [12] independently proved quadratic lower bounds for LCS conditioned on the SETH. That is, they showed that assuming SETH, there cannot be an $O(m^{2-\epsilon})$ algorithm. However, there exists an $\tilde{O}(n + m^2)$ algorithm for LCS [24]. This suggests that for unbalanced inputs where $m \ll n$, episode matching is harder than LCS.

Further, the episode matching problem can be seen as a special case of the *Approximate String Matching* problem. There, given strings S and P , the goal is to find the substring of S minimizing some distance measure to P . If this distance measure is the number of deleted characters from S , the problem is equivalent to episode matching. Another version of the approximate string matching problem uses *edit distance*. Backurs and Indyk [9] implicitly give a quadratic lower bound conditioned on SETH for that version as a stepping stone for achieving a lower bound for edit distance, using an alphabet of size seven. This bound does not however directly translate to our problem, and uses a more complicated construction and a larger alphabet.

To avoid misunderstandings, we note that in the paper by Mannila et al. [31], an *episode* was more generally defined as a collection of events that occur together. The string formulation by Das et al. [15] is a simplification of this original definition, thus, the terms *episode* and *episode matching* have been used to name different concepts in the literature, see e.g. [4, 21, 23, 32, 34]. We only consider the string formulation by Das et al. [15].

1.2 Episode Matching Indexing

Limited by the above lower bound for episode matching, a natural question to ask is what bounds we can get if we allow preprocessing. Alas, in terms of time complexity, preprocessing does not help. Namely, our reduction is such that each of the two sets in the Orthogonal Vectors instance (see Definition 6) is encoded *independently* into one of the two strings in the episode matching instance. This implies (see e.g. [18]) that episode matching cannot be solved in $O((mn)^{1-\epsilon})$ time even if we are allowed polynomial time to preprocess one of the two strings in advance. In other words, polynomial time preprocessing is not enough to guarantee subquadratic time queries. In fact, this holds even when one of the strings is fixed (cf. [2, Section 1.1.2]). We therefore focus on time-space tradeoffs.

Formally, given a string S and an integer k , the *episode matching indexing* problem asks to construct a compact data structure that can quickly report the episode matching of any pattern P of length k (i.e. compute the length of the shortest substring of S that contains P as a subsequence). Apostolico and Atallah [6] gave a linear (optimal) space data structure whose query time may be prohibitive (depending on the number of distinct minimal substrings containing a prefix of the pattern as a subsequence). In Section 4.1 we present a time-space tradeoff with faster query time:

► **Theorem 2.** *For any τ , there is a data structure using space $O(n + (\frac{n}{\tau})^k)$ that answers Episode Matching queries in time $O(k \cdot \tau \cdot \log \log n)$.*

In Section 4.2 we give the following almost matching lower bound, conditioned on the k -Set Disjointness Conjecture:

► **Theorem 3.** *For any $\delta \in [0, 1/2]$, a data structure that answers Episode Matching queries in time $O(n^\delta)$ must use $\Omega(n^{k-k\delta-o(1)})$ space, unless the Strong k -Set Disjointness Conjecture is false.*

Finally, in Section 4.3 we consider the decision version of the case $k = 2$. That is, we are given some threshold t , and a query (a, b) need only report whether S contains a substring of length at most t that starts with the letter a and ends with the letter b . In this setting, there is a simple $O(1)$ -query time $O(\sigma^2)$ -space data structure for episode matching (where σ is the size of the alphabet): precompute and store the $\sigma \times \sigma$ matrix D with the answers to every possible query. Naively, we can compute D in time $\min\{\tilde{O}(n\sigma + \sigma^2), O(nt + \sigma^2)\}$. We show that for various values of σ and t , we can compute the matrix D faster by using fast min-plus matrix multiplication of bounded integer matrices [37]:

► **Theorem 4.** *For a given threshold t , we can compute the matrix D in time*
 $O(\sigma^{1/2+\omega/2} (\frac{n}{t})^{1/2} \sqrt{n})$ *if $\sigma < \frac{n}{t}$, and*
 $O(\sigma^2 (\frac{n}{t})^{\omega/2-1} \sqrt{n})$ *if $\sigma \geq \frac{n}{t}$.*

In particular, using the current matrix multiplication exponent $\omega < 2.4$ [5], if $\sigma < \frac{n}{t}$ then we get $O(\sigma^{1.7} nt^{-1/2})$ (which is smaller than $\min(n\sigma, nt)$ for any $t > \sigma^{1.4}$), and if $\sigma \geq \frac{n}{t}$ we get $O(\sigma^{2.2} \sqrt{n})$ (which is smaller than $O(n\sigma)$ for any $\sigma < n^{0.416}$).

2 Preliminaries

In this section we will review some basic string notation and important hypotheses.

A string S of length n is a sequence $S[0] \cdots S[n-1]$ of n characters drawn from an alphabet Σ of size $|\Sigma| = \sigma$. A string $S[i \cdots j] = S[i] \cdots S[j]$ for $0 \leq i < j < n$ is called a *substring* of S . For two strings X and Y we denote their *concatenation* as $X \cdot Y$ or XY .

The Strong Exponential Time Hypothesis is a popular conjecture about the hardness of the k -SAT problem, postulated by Impagliazzo and Paturi [25]. The k -SAT problem is to decide whether there exists a satisfying assignment for a Boolean formula on n variables and clauses of width at most k .

► **Conjecture 5 (The Strong Exponential Time Hypothesis).** *There is no $\epsilon > 0$ for which k -SAT can be solved in time $O(2^{(1-\epsilon)n})$ for all $k \geq 3$.*

Instead of reducing directly from k -SAT, we reduce from the Orthogonal Vectors problem, and use two conjectures about the hardness of Orthogonal Vectors which are implied by SETH for $d = \omega(\log n)$.

4:4 The Fine-Grained Complexity of Episode Matching

► **Definition 6** (Orthogonal Vectors Problem (OV)). *Given two sets $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_m\}$ of d -dimensional binary vectors, decide whether there is an orthogonal pair of vectors $a_i \in A$ and $b_j \in B$.*

The two conjectures consider the cases of equal set size and unbalanced set size, respectively. They roughly state that any algorithm solving OV that has a polynomial dependency on the dimension (denoted $\text{poly}(d)$), cannot achieve a significantly better asymptotic running time than the product of the two set sizes.

► **Conjecture 7** (Orthogonal Vectors Hypothesis (OVH)). *For $|A| = |B| = n$, there is no $\epsilon > 0$ for which OV can be solved in time $O(n^{2-\epsilon} \text{poly}(d))$.*

► **Conjecture 8** (Unbalanced Orthogonal Vectors Hypothesis (UOVH)). *Let $0 < \alpha \leq 1$, $|A| = n$ and $|B| = m$. There is no $\epsilon > 0$ for which OV restricted to $m = \Theta(n^\alpha)$ and $d = n^{o(1)}$ can be solved in time $O((nm)^{1-\epsilon})$.*

It is known that SETH implies both OVH and UOVH [12, 36]. Finally, we consider the following conjecture of Goldstein et al. [20] on the k -Set Disjointness problem:

► **Definition 9** (k -Set Disjointness Problem). *Preprocess m sets S_1, S_2, \dots, S_m of total size $\sum_{i=1}^m |S_i| = N$ drawn from a universe U such that given (i_1, i_2, \dots, i_k) we can quickly decide whether $\bigcap_{j=1}^k S_{i_j} = \emptyset$.*

► **Conjecture 10** (Strong k -Set Disjointness Conjecture). *Any data structure for the k -Set Disjointness Problem that answers queries in time T must use $\tilde{\Omega}(N^k/T^k)$ space.*

3 Episode Matching

In this section we prove Theorem 1. We first prove it for an alphabet of size four, and then for a binary alphabet.

3.1 Alphabet of Size Four

We show how to reduce an instance of OV to Episode Matching with alphabet $\{0, 1, x, \$\}$. Let $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_m\}$ be two sets of vectors in $\{0, 1\}^d$. Without loss of generality, we assume $m \leq n$. We show how to construct a string P of length $2dm + 1$ and a string S of length $3d(4n + 1) + 1$ such that there is a pair of orthogonal vectors $a_i \in A$ and $b_j \in B$ if and only if there is a substring of S of length at most $3d(2m - 1) + 1$ that contains P as a subsequence.

Constructing P . We construct P from the set B in the following way. For every $b \in B$, we define $p(b)$ as the string of length $2d - 1$ obtained by inserting an x symbol between each consecutive entries of b . That is,

$$p(b) = b[0] x b[1] x \dots x b[d]$$

Then, P is a string of length $2dm + 1$ defined as the concatenation:

$$P = \$ p(b_1) \$ p(b_2) \$ \dots \$ p(b_m) \$$$

Constructing S . We construct S from the set A . For a vector a in $\{0, 1\}^d$, we define for each entry in a a string of length 2, called a *coordinate gadget*:

$$s(a[i]) = 01 \text{ if } a[i] = 0,$$

$$s(a[i]) = 00 \text{ if } a[i] = 1.$$

Then, $s(a)$ is a string of length $3d - 1$ defined as the concatenation:

$$s(a) = s(a[0]) \ x \ s(a[1]) \ x \cdots \ x \ s(a[d])$$

For example, the vector $a = 10001$ defines the string $s(a) = 00x01x01x01x00$. Let z be the d -dimensional zero vector, and so $s(z) = 01x01x \dots x01$. Then, S consists of $\$ s(z) \$$ followed by two copies of the concatenation of $s(a_i) \$ s(z) \$$ for $1 \leq i \leq n$. That is:

$$S = \$s(z)\$s(a_1)\$s(z)\$s(a_2)\$ \cdots \$s(z)\$s(a_n)\$s(z)\$s(a_1)\$s(z)\$s(a_2)\$ \cdots \$s(z)\$s(a_n)\$s(z)\$$$

We call a substring of S of the form “ $s(a) \$$ ” or “ $s(z) \$$ ” a *block*. A block has length $3d$. The string S contains $2n$ blocks for the elements of A , plus $2n + 1$ blocks for the separating $s(z)$, thus $4n + 1$ blocks in total. There is an extra $\$$ at the start of the string. Thus, $|S| = 3d(4n + 1) + 1 = \Theta(dn)$.

Correctness. For two strings Y and X , an *alignment L of Y in X* is a sequence $0 \leq j_0 < \cdots < j_{|Y|-1} \leq |X| - 1$ such that $Y[\ell] = X[j_\ell]$ for every $0 \leq \ell \leq |Y| - 1$. We say $Y[\ell]$ is *aligned* to $X[j_\ell]$ and that L *spans* the substring $X[j_0 \cdots j_{|Y|-1}]$. Clearly, Y is a subsequence of X if and only if there exists an alignment of Y in X .

The following Lemma establishes the translation of orthogonality in our reduction.

► **Lemma 11.** *For two vectors a and b of dimension d , the string $p(b)$ is a subsequence of $s(a)$ if and only if b and a are orthogonal.*

Proof. If a and b are orthogonal, then for every $b[i] = 1$ we have that $a[i] = 0$ and therefore $s(a[i]) = 01$. Thus we can align $b[i]$ to the 1 in $s(a[i])$. If $b[i] = 0$, we can align $b[i]$ to the first character in $s(a[i])$. We can therefore define an alignment where we align every $b[i]$ to one of the characters in $s(a[i])$, and align the i th x in $p(b)$ to the i th x in $s(a)$. Therefore $p(b)$ is a subsequence of $s(a)$.

On the other hand, if $p(b)$ is a subsequence of $s(a)$, then since $p(b)$ and $s(a)$ both contain exactly $d - 1$ x 's, any alignment of $p(b)$ in $s(a)$ *must* align the i th x in $p(b)$ to the i th x in $s(a)$. Thus, each $b[i]$ must be aligned to a character in $s(a[i])$. If $b[i] = 1$, we can align it with a character in $s(a[i])$ only if $a[i] = 0$. Thus, if $p(b)$ is a subsequence, then for every $b[i] = 1$, we must have $a[i] = 0$. Therefore a and b are orthogonal. ◀

Lemma 11 implies that any $p(b)$ is a subsequence of $s(z)$. This immediately yields a substring of S of length $3d(2m - 1) + 1$ which contains P as a subsequence: If we align P with m consecutive occurrences of $s(z)$, the resulting substring contains a $\$$ at the beginning, m blocks of the form “ $s(z) \$$ ” and $m - 1$ blocks of the form “ $s(a) \$$ ”. Next, we show that if there is no pair of orthogonal vectors, then there is no shorter substring of S that contains P as a subsequence.

▷ **Claim 12.** *If there exist no $a \in A$ and $b \in B$ which are orthogonal, then there exists no substring of S of length $< 3d(2m - 1) + 1$ which contains P as a subsequence.*

4:6 The Fine-Grained Complexity of Episode Matching

$$\begin{array}{cccccccccccc} \cdots & \$ & s(z) & \$ & s(a_{i-1}) & \$ & s(z) & \$ & s(a_i) & \$ & s(z) & \$ & s(a_{i+1}) & \$ & s(z) & \$ & \cdots \\ \cdots & \$ & p(b_{j-2}) & & & & \$ & p(b_{j-1}) & \$ & p(b_j) & \$ & p(b_{j+1}) & & & & \$ & p(b_{j+2}) & \$ & \cdots \end{array}$$

■ **Figure 1** The alignment of P and S as described in the proof of Claim 13.

Proof. Consider an alignment L of P in S . If for every $b \in B$ the substring $p(b)$ is aligned to a string of the form $s(z)$, then L spans a string of length at least $3d(2m - 1) + 1$. Consider now the case where there exists a $b \in B$ such that $p(b)$ is not fully aligned to a string of the form $s(z)$. By Lemma 11, since there is no $a \in A$ such that a and b are orthogonal, there is no $s(a)$ such that $p(b)$ is fully aligned within $s(a)$. Since no $s(a)$ or $s(z)$ contain a $\$$, this means that the alignment of the string $\$ p(b) \$$ spans a string containing either a substring of the form $\$ s(a) \$ s(z) \$$ or a substring of the form $\$ s(z) \$ s(a) \$$. Then the alignment L' defined by aligning $\$ p(b) \$$ to the $\$ s(z) \$$ in that substring, and everything else as in L , spans a substring in S no longer than the substring spanned by L . Repeating this for each $p(b)$ that is not aligned to $s(z)$ gives an alignment where every $p(b)$ is aligned to some $s(z)$, and which spans a substring in S no longer than the substring spanned by L . Since this substring contains at least m copies of $\$ s(z) \$$, it has length at least $3d(2m - 1) + 1$. \triangleleft

Next, we show how to align P in S to yield a shorter substring, if there does exist an orthogonal pair.

\triangleright **Claim 13.** If there exist $a \in A$ and $b \in B$ which are orthogonal, then there is a substring of S of length $< 3d(2m - 1) + 1$ that contains P as a subsequence.

Proof. Assume a_i and b_j are orthogonal and $j \leq i$. We align $p(b_j)$ to the first copy of $s(a_i)$, and align $p(b_{j+1}) \cdots p(b_m)$ to the next $m - j$ copies of $s(z)$ to the right of $s(a_i)$, and $p(b_1) \cdots p(b_{j-1})$ to the previous $j - 1$ copies of $s(z)$ to the left of $s(a_i)$. See Figure 1. This is possible since $j \leq i$. Let T denote the resulting substring of S .

- Case 1: $j = 1$ (or $j = m$). In this case, the substring of T spans $2m - 2$ blocks: It starts (ends) with “ $\$ s(a_i) \$$ ”, and then includes the $m - 1$ following (preceeding) $s(z)$ blocks. Between any two $s(z)$ blocks, there is another block corresponding to some $a_\ell \in A$. Thus in total, the length of T is $3d(2m - 2) + 1 < 3d(2m - 1) + 1$.
- Case 2: $1 < j < m$. The substring T starts and ends with $\$ s(z) \$$, and we align to $m - 1$ of the $s(z)$ blocks and to $s(a_i)$ which is somewhere inbetween these. Thus, T includes $m - 1 + m - 2 = 2m - 3$ blocks, and the length of T is $3d(2m - 3) + 1 < 3d(2m - 1) + 1$. If $j > i$, we align $p(b_j)$ to the *second* copy of $s(a_i)$, and again align $p(b_{j+1}) \cdots p(b_m)$ to the next $m - j$ copies of $s(z)$ to the right of $s(a_i)$, and $p(b_1) \cdots p(b_{j-1})$ to the preceding $j - 1$ copies of $s(z)$ to the left of $s(a_i)$. This is possible since $m - j \leq n - i$. The rest follows analogously. \triangleleft

Analysis. When $m = \Theta(n)$, we have that $|P| = \Theta(|S|) = \Theta(nd)$, which means that an $O((|S||P|)^{1-\epsilon})$ algorithm for Episode Matching implies an $O(n^{2-2\epsilon}d^{2-2\epsilon})$ algorithm for OV, contradicting OVH.

When $m = \Theta(n^\alpha)$ for some $0 < \alpha < 1$, and $d = n^{o(1)}$, the length of P is $2dm + 1 = mn^{o(1)}$, and the length of S is $3d(4n + 1) + 1 = n^{1+o(1)}$. This means that an $O((|S||P|)^{1-\epsilon})$ algorithm for Episode Matching implies an $(nm)^{1-\epsilon}n^{o(1)} = O((nm)^{1-\epsilon'})$ algorithm for OV for any $\epsilon' < \epsilon$, contradicting UOVH.

This proves Theorem 1 for alphabet size at least 4.

3.2 Binary Alphabet

We now prove Theorem 1 for a binary alphabet. We will use almost the same reduction as before, but replace x and $\$$ with binary gadgets.

Inner separator gadget. Instead of the separating symbol x , we define an *inner separator gadget* of the form 0^d . The strings $p(b)$ and $s(a)$ are defined as before, except that each x is substituted with the inner separator gadget 0^d , and we add an extra inner separator gadget at the beginning and end of the string. That is,

$$p(b) = 0^d b[1] 0^d b[2] 0^d \dots 0^d b[d] 0^d$$

and

$$s(a) = 0^d s(a[1]) 0^d s(a[2]) 0^d \dots 0^d s(a[d]) 0^d.$$

Outer separator gadget. Instead of the separating symbol $\$$, we define an *outer separator gadget* of the form 1^{d+1} . The strings P and S are defined as before, with the new versions of $p(b)$, $s(a)$ and $s(z)$, and every $\$$ substituted with the outer separator gadget 1^{d+1} .

For any $b \in B$, the length of the string $p(b)$ is $(d+2)d$, since it contains $d+1$ inner separator gadgets of length d . The string P consists of $m+1$ outer separator gadgets, each of length $d+1$, and m substrings $p(b)$ each of length $(d+2)d$. The length of P is therefore $(d+2)dm + (m+1)(d+1) = \Theta(md^2)$.

For any $a \in A$, the length of the string $s(a)$ is $(d+3)d$. Analogously to before, we define a *block* as a substring of the form $s(a) 1^{d+1}$. The length of a block is $(d+3)d + (d+1) = d^2 + 4d + 1$. The string S consists of $4n+1$ blocks (as before), each of length $d^2 + 4d + 1$ and an extra outer separator gadget at the beginning. The length of S is therefore $(d^2 + 4d + 1)(4n+1) + d + 1 = \Theta(nd^2)$.

Now, if we align every $p(b)$ to a copy of $s(z)$, as in the proof in Section 3.1, we need $2m-1$ blocks. Thus, we get a substring of length $w = (d^2 + 4d + 1)(2m-1) + d + 1$. Our reduction will again depend on the fact that if there are no orthogonal vectors, we cannot do much better than that. Namely, we next prove that if there is no pair of orthogonal vectors then there is no substring of S of length $< w - 2d$ that contains P as a subsequence (Claim 15), and that if there is a pair of orthogonal vectors then there is a substring of S of length $\leq w - (d^2 + 4d + 1)$ that contains P as a subsequence (Claim 16).

Algorithm. We run the assumed blackbox algorithm for Episode Matching on S and P . If the algorithm outputs a substring of length at least $w - 2d$, we conclude that there are no orthogonal vectors. If it outputs some string of a shorter length, then there are.

Correctness. The correctness proof follows along the same lines as in the alphabet four case. We begin by proving an equivalent version of Lemma 11:

► **Lemma 14.** *For two vectors a and b of dimension d , the string $p(b)$ is a subsequence of $s(a)$ if and only if b and a are orthogonal.*

Proof. If a and b are orthogonal, then by the same arguments as before, $p(b)$ is a subsequence of $s(a)$. Otherwise, a and b are not orthogonal, so there exists an i such that $b[i] = a[i] = 1$. Assume for the sake of contradiction that $p(b)$ is a subsequence of $s(a)$. Then we must align the 1 corresponding to $b[i]$ with some coordinate gadget $s(a[j]) = 01$ and $j \neq i$. There are two cases depending on whether $i > j$ or $j > i$.

4:8 The Fine-Grained Complexity of Episode Matching

- Case 1: $i > j$. Note that to the left of the 1 corresponding to $b[i]$, there are $(i - 1) + di$ characters in $p(b)$, and to the left of the 1 in $s(a[j]) = 01$ there are $2(j - 1) + dj + 1$ characters in $s(a)$: namely $j - 1$ coordinate gadgets, j inner separator gadgets, and the 0 in $s(a[j])$. But this means that the prefix of $p(b)$ up to $b[i]$ is longer than the prefix of $s(a)$ up to the 1 in $s(a[j])$, since

$$(i - 1) + id \geq j + (j + 1)d = j + d + jd > 2(j - 1) + jd + 1,$$

where the last inequality holds because $j < d$. Thus we cannot have aligned the prefix of $p(b)$ up to $b[i]$ to the prefix of $s(a)$ up to the 1 in $s(a[j])$, contradicting that $p(b)$ is a subsequence of $s(a)$.

- Case 2: $j > i$. Note that to the right of the 1 corresponding to $b[i]$, there are $d - i + (d - i + 1)d = (d - i)(d + 1) + d$ characters in $p(b)$, and to the right of the 1 in $s(a[j]) = 01$ there are $2(d - j) + (d - j + 1)d = (d - j)(d + 2) + d$ characters in $s(a)$. But this means that the suffix of $p(b)$ to the right of $b[i]$ is longer than the suffix of $s(a)$ to the right of $s(a[j])$, since

$$(d - i)(d + 1) + d \geq (d - j + 1)(d + 1) + d = (d - j)(d + 1) + 2d + 1 > (d - j)(d + 2) + d.$$

This means that we cannot have aligned the suffix of $p(b)$ to the right of $b[i]$ to the suffix of $s(a)$ to the right of $s(a[j])$, contradicting that $p(b)$ is a subsequence of $s(a)$. ◀

Next we prove that if there are no vectors $a \in A$ and $b \in B$ which are orthogonal, then we cannot do much better than aligning $p(b_1), \dots, p(b_m)$ to m consecutive copies of $s(z)$. Recall that we defined the length of that alignment as w .

▷ **Claim 15.** If there exist no $a \in A$ and $b \in B$ which are orthogonal, then there exists no substring of S of length $< w - 2d$ that contains P as a subsequence.

Proof. Assume an alignment such that $p(b_i)$ is not aligned to $s(z)$. Since there are no orthogonal vectors, there is no $s(a)$ such that $p(b_i)$ is aligned in $s(a)$. Further, since $p(b_i)$ starts and ends with a 0, its alignment cannot start or end within an outer separator gadget. Thus, the alignment of $p(b_i)$ spans a string either containing

1. a non-empty suffix of some $s(a_j)$, followed by 1^{d+1} , followed by a non-empty prefix of $s(z)$ or
2. a non-empty suffix of $s(z)$, followed by 1^{d+1} , followed by a non-empty prefix of some $s(a_j)$.

Consider case 1. Since $s(z)$ only contains d 1s, at least one character in the outer separator gadget following $p(b_i)$ cannot be aligned in $s(z)$. Thus, at least one 1 cannot be aligned before the 1^{d+1} following $s(z)$. If $i < m$, then the inner separator gadget at the beginning of $p(b_{i+1})$ cannot be aligned before the beginning of a new block. Thus, the alignment defined by aligning $p(b_i) 1^{d+1}$ to $s(z) 1^{d+1}$ spans a string no longer than the original alignment. If $i = m$, the same alignment spans a string at most d longer than the original alignment. Analogously, for case 2, we align $1^{d+1} p(b_i)$ to $1^{d+1} s(z)$. The alignment spans a string of the same length as the original alignment, or at most d longer if $i = 1$. Repeating this step for any $p(b_i)$ which is not aligned to some $s(z)$, we get an alignment where every $p(b_i)$ is aligned to some $s(z)$, and which spans a string at most $2d$ longer than the original alignment. This concludes the proof. ◀

▷ **Claim 16.** If there exist a and b which are orthogonal, then there is a substring of S of length $\leq w - (d^2 + 4d + 1)$ (i.e. it is shorter than w by at least a block's length) that contains P as a subsequence.

Proof. The proof is analogous to that of Claim 13. ◀

Analysis. The length of S is now $\Theta(nd^2)$, and the length of P is $\Theta(md^2)$. The contradictions to OVH and UOVH are constructed analogously to the alphabet four case. This proves Theorem 1 restricted to binary alphabet.

4 Episode Matching Indexing

In this section we consider the indexing version of episode matching: Given the string S and an integer k , construct a data structure that can quickly report the episode matching of any pattern P of length k .

4.1 Upper bound

We now prove Theorem 2. We call letters that appear more than τ times *frequent* letters. Note that there are at most $\frac{n}{\tau}$ frequent letters. For every k -tuple of frequent letters, we precompute and store the answer in a table. The size of this table is therefore $\left(\frac{n}{\tau}\right)^k$. Additionally, for each letter in the alphabet, we store a predecessor data structure containing all positions in S where the letter appears. Using a linear-space predecessor data structure such as *y-fast-trie* [35], this requires an additional $O(n)$ space (every position in S appears in exactly one predecessor structure) and answers predecessor/successor queries in $O(\log \log n)$ time.

To answer a query, given a pattern P of length k , if every letter in P is frequent, we simply return the precomputed answer from the table. Otherwise, suppose $P[j]$ is a non-frequent letter. For each position i s.t. $S[i] = P[j]$, we find the minimal substring of S that contains P and aligns $P[j]$ to $S[i]$ (eventually we return the smallest substring found). To do this, we start from location i and use successor queries to find $P[j+1], P[j+2], \dots, P[m-1]$ and predecessor queries to find $P[j-1], P[j-2], \dots, P[0]$. Since there are at most τ positions in S that contain $P[j]$, this takes overall $O(\tau \cdot k \log \log n)$ time.

4.2 Lower bound

We now prove Theorem 3. The proof is similar to the ones in [28] and [10].

Recall that in the k -Set Disjointness problem, given sets S_1, \dots, S_m of total size N over a universe U , we want a data structure that given (i_1, \dots, i_k) reports whether $\bigcap_{j=1}^k S_{i_j} = \emptyset$. As in [10], we can reduce k -Set Disjointness to $O(\log N)$ instances of k -Set Disjointness (each of size $O(N)$) with the property that every element in U appears in the same number of sets (call this number f). We next show a reduction from such an instance to episode matching indexing.

Define for each set S_i a unique letter α_i . For each distinct element $e \in U$ define a block consisting of the letters α_i corresponding to the sets S_i that contain e , sorted by i . We then append all such blocks in an arbitrary order and separate each two blocks by an extra block of the form $\f (where $\$$ is an extra letter not corresponding to any set). The resulting string S is of length $2N - f = O(N)$. We preprocess S into a data structure for episode matching. To answer a query (i_1, \dots, i_k) (we assume w.l.o.g. that $i_1 < i_2 < \dots < i_k$), we query the data structure for $P = \alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k}$. If the output is larger than f , we answer that the sets are disjoint, otherwise we answer that they are not. The correctness is simple: If there is an element e that appears in all sets S_{i_1}, \dots, S_{i_k} , then e 's block contains $\alpha_{i_1} \alpha_{i_2} \dots \alpha_{i_k}$ (and since we sorted the elements, they will appear in the right order). If on the other hand the sets are disjoint, then there is no block containing all letters, so the smallest substring containing all letters must include at least one $\f block (and is thus longer than f).

The total space for all the $O(\log N)$ instances is therefore $O(\log N \cdot s_{\text{episode}}(N))$ and the runtime is $O(\log N \cdot t_{\text{episode}}(N))$, where $s_{\text{episode}}(n)$ and $t_{\text{episode}}(n)$ are the space and query time for a data structure for episode Matching indexing. To prove Theorem 3, assume for contradiction that $t_{\text{episode}}(n) = O(n^\delta)$ and $s_{\text{episode}}(n) = O(n^{k-k\delta-\epsilon})$. Then the space for solving k -Set Disjointness is $O(N^{k-k\delta-\epsilon} \log N) = O(N^{k-k\delta-\epsilon'})$, for any $0 < \epsilon' < \epsilon$, and the time is $O(N^\delta \log N) = O(N^{\delta+\epsilon''})$, for arbitrarily small $\epsilon'' > 0$. Setting $\epsilon'' < \epsilon'/k$, we obtain a contradiction to Conjecture 10.

4.3 The special case of $k = 2$

We now prove Theorem 4. Recall that, given the string S , our goal is to compute the binary matrix D where $D[a, b] = 1$ if and only if S contains a substring of length at most t that starts with the letter a and ends with the letter b . Naively, we can compute D in time $\min\{\tilde{O}(n\sigma + \sigma^2), O(nt + \sigma^2)\}$. To get $\tilde{O}(n\sigma + \sigma^2)$, we construct a predecessor data structure for each letter, containing the occurrences of this letter in S . Then, for each position in S we find the succeeding occurrence of each letter in the alphabet, using the corresponding predecessor data structure. Whenever we find a pair of letters (a, b) at distance at most t , we set $D[a, b]$ to 1. To get $O(nt + \sigma^2)$, for each position in S we scan t entries forward and update D as we go. In the remainder of this section, we will show how to compute D faster using min-plus multiplication of bounded integer matrices:

► **Lemma 17** ([37]). *Given two $n \times n$ matrices A and B , where A has entries in $\{-M, \dots, M\} \cup \{\infty\}$ and B is arbitrary, we can compute their min-plus product $C = A \odot B$ (defined as $C[i, j] = \min_k(A[i, k] + B[k, j])$) in $\tilde{O}(\sqrt{M}n^{(3+\omega)/2})$ time.*

First, we divide the string S into blocks $B_1, B_2, \dots, B_{n/t}$, each of length t (except maybe the last). Let $\delta_{\text{first}}(a, j)$ (resp. $\delta_{\text{last}}(a, j)$) be the distance from the beginning (resp. end) of the j th block to the first (resp. last) a in that block, and ∞ if the block has no a . Note that ab is a subsequence of a length t substring of S if and only if one of the following two conditions holds:

Condition 1: There is a j such that $\delta_{\text{first}}(a, j) + \delta_{\text{last}}(b, j) \leq t$. To check this condition, we check if $(M_1 \odot M_2)[a, b] \leq t$ where M_1 is the $\sigma \times n/t$ matrix with $M_1[a, j] = \delta_{\text{first}}(a, j)$, and M_2 is the $n/t \times \sigma$ matrix with $M_2[j, b] = \delta_{\text{last}}(b, j)$.

Condition 2: There is a j such that $\delta_{\text{last}}(a, j) + \delta_{\text{first}}(b, j+1) \leq t$. To check this condition, we check if $(M_3 \odot M_4)[a, b] \leq t$ where M_3 is the $\sigma \times (n/t-1)$ matrix with $M_3[a, j] = \delta_{\text{last}}(a, j)$, and M_4 is the $(n/t-1) \times \sigma$ matrix with $M_4[j, b] = \delta_{\text{first}}(b, j+1)$.

By the above discussion, it only remains to compute the products $M_1 \odot M_2$ and $M_3 \odot M_4$ (as then each entry of D can be found in constant time). We focus on $M_1 \odot M_2$ (the other is symmetric). Observe that the entries of M_1 and M_2 are integers bounded by t . Therefore, we can hope to use Lemma 17 (with $M = t$) to multiply them. However, in Lemma 17 the matrices are square while our matrices are rectangular. To deal with this, we break the matrices into rectangular submatrices according to the value of σ .

- If $\sigma < n/t$. We split M_1 into $M_{1,1} \dots, M_{1, \frac{n}{t\sigma}}$, each consisting of σ consecutive columns. We split M_2 into $M_{2,1} \dots, M_{2, \frac{n}{t\sigma}}$, each consisting of σ consecutive rows. Note that $(M_1 \odot M_2)[a, b] = \min_{k=1, \dots, \frac{n}{t\sigma}} (M_{1,k} \odot M_{2,k})[a, b]$. We can thus compute $M_1 \odot M_2$ by computing $\frac{n}{t\sigma}$ min-plus products of $\sigma \times \sigma$ matrices, and setting each entry in $M_1 \odot M_2$ to be the minimum of the corresponding entries in the $\frac{n}{t\sigma}$ output matrices. Using Lemma 17, this takes time $O(\frac{n}{t\sigma}(\sigma^2 + \sigma^{(3+\omega)/2}\sqrt{t})) = O(\sigma^{1/2+\omega/2} (\frac{n}{t})^{1/2} \sqrt{n})$.

- If $\sigma \geq n/t$. We split M_1 into $M_{1,1} \dots, M_{1, \frac{t\sigma}{n}}$, each consisting of n/t consecutive rows. We split M_2 into $M_{2,1} \dots, M_{2, \frac{t\sigma}{n}}$, each consisting of n/t consecutive columns. We compute $M_1 \odot M_2$ by computing $\left(\frac{t\sigma}{n}\right)^2$ min-plus products of $n/t \times n/t$ matrices $M_{1,i} \odot M_{2,j}$ for every i, j . Using Lemma 17, this takes time $O\left(\left(\frac{t\sigma}{n}\right)^2 \cdot \left(\frac{n}{t}\right)^{(3+\omega)/2} \sqrt{t}\right) = O\left(\sigma^2 \left(\frac{n}{t}\right)^{\omega/2-1} \sqrt{n}\right)$. This proves Theorem 4.

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proc. 56th FOCS*, pages 59–78, 2015.
- 2 Amir Abboud and Virginia Vassilevska Williams. Fine-grained hardness for edit distance to a fixed sequence. In *Proc. 48th ICALP*, volume 198, pages 7:1–7:14, 2021.
- 3 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Proc. 41st ICALP*, pages 39–51, 2014.
- 4 Avinash Achar, A. Ibrahim, and P. S. Sastry. Pattern-growth based frequent serial episode discovery. *Data Knowl. Eng.*, 87:91–108, 2013.
- 5 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proc. 32nd SODA*, pages 522–539, 2021.
- 6 Alberto Apostolico and Mikhail J. Atallah. Compact recognizers of episode sequences. *Inf. Comput.*, 174(2):180–192, 2002.
- 7 Mikhail J. Atallah, Robert Gwadera, and Wojciech Szpankowski. Detection of significant sets of episodes in event sequences. In *Proc. 4th ICDM*, pages 3–10, 2004.
- 8 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Proc. 57th FOCS*, pages 457–466, 2016.
- 9 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.
- 10 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.
- 11 Luc Boasson, Patrick Cégielski, Irène Guessarian, and Yuri V. Matiyasevich. Window-accumulated subsequence matching problem is linear. *Ann. Pure Appl. Log.*, 113(1-3):59–80, 2001.
- 12 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proc. 56th FOCS*, pages 79–97, 2015.
- 13 Patrick Cégielski, Irène Guessarian, and Yuri V. Matiyasevich. Multiple serial episodes matching. *Inf. Process. Lett.*, 98(6):211–218, 2006.
- 14 Maxime Crochemore, Costas S. Iliopoulos, Christos Makris, Wojciech Rytter, Athanasios K. Tsakalidis, and T. Tsihlias. Approximate string matching with gaps. *Nord. J. Comput.*, 9(1):54–65, 2002.
- 15 Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitrios Gunopoulos, and Juha Kärkkäinen. Episode matching. In *Proc. 8th CPM*, pages 12–27, 1997.
- 16 Lech Duraj, Marvin Künnemann, and Adam Polak. Tight conditional lower bounds for longest common increasing subsequence. *Algorithmica*, 81(10):3968–3992, 2019.
- 17 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *Proc. 46th ICALP*, pages 55:1–55:15, 2019.
- 18 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 *Proc. 27th SOFSEM*, volume 12607, pages 608–622, 2021.
- 19 Daniel Gibney. An efficient elastic-degenerate text index? not likely. In *Proc. 27th SPIRE*, pages 76–88, 2020.
- 20 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Proc. 15th WADS*, pages 421–436, 2017.

- 21 Robert Gwadera, Mikhail J. Atallah, and Wojciech Szpankowski. Markov models for identification of significant episodes. In *Proc. 5th SDM*, pages 404–414, 2005.
- 22 Robert Gwadera, Mikhail J. Atallah, and Wojciech Szpankowski. Reliable detection of episodes in event sequences. *Knowl. Inf. Syst.*, 7(4):415–437, 2005.
- 23 Masahiro Hirao, Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. A practical algorithm to find the best episode patterns. In *Proc. 4th DS*, pages 435–440, 2001.
- 24 Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.
- 25 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.
- 26 Mika Klemettinen, Heikki Mannila, and Hannu Toivonen. Rule discovery in telecommunication alarm data. *J. Netw. Syst. Manag.*, 7(4):395–423, 1999.
- 27 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019.
- 28 Tsvi Kopelowitz and Robert Krauthgamer. Color-distance oracles and snippets. In *Proc. 27th CPM*, pages 24:1–24:10, 2016.
- 29 Josué Kuri, Gonzalo Navarro, and Ludovic Mé. Fast multipattern search algorithms for intrusion detection. *Fundam. Informaticae*, 56(1-2):23–49, 2003.
- 30 Veli Mäkinen, Gonzalo Navarro, and Esko Ukkonen. Transposition invariant string matching. *J. Algorithms*, 56(2):124–153, 2005.
- 31 Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.
- 32 Elżbieta Nowicka and Marcin Zawada. On the complexity of matching non-injective general episodes. *Computation and Logic in the Real World*, pages 288–296, 2007.
- 33 Adam Polak. Why is it hard to beat $O(n^2)$ for longest common weakly increasing subsequence? *Inf. Process. Lett.*, 132:1–5, 2018.
- 34 Shinichiro Tago, Tatsuya Asai, Takashi Katoh, Hiroaki Morikawa, and Hiroya Inakoshi. EVIS: A fast and scalable episode matching engine for massively parallel data streams. In *Proc. 17th DASFAA*, pages 213–223, 2012.
- 35 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.
- 36 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005.
- 37 Virginia Vassilevska Williams and Yinzhan Xu. Truly subcubic min-plus product for less structured matrices, with applications. In *Proc. 31st SODA*, pages 12–29, 2020.

Mechanical Proving with Walnut for Squares and Cubes in Partial Words

John Machacek ✉ 🏠

Department of Mathematics, University of Oregon, Eugene, OR, USA

Abstract

Walnut is a software that can prove theorems in combinatorics on words about automatic sequences. We are able to apply this software to both prove new results as well as reprove some old results on avoiding squares and cubes in partial words. We also define the notion of an antisquare in a partial word and begin the study of binary partial words which contain only a fixed number of distinct squares and antisquares.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Partial words, squares, antisquares, cubes, Walnut

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.5

Funding NSF DMS-2039316.

1 Introduction

Our focus is on repetitions in partial words and the use of the software called Walnut¹ [15] to give automated proofs in situations where automatic sequences can be used. Partial words are generalizations of usual words that make use of an additional “wildcard” character which matches all other characters. Walnut has been used to give alternative proofs of previously known results and has also been used to produce new theorems in combinatorics on words (see e.g., [16]). To our knowledge this work is the first use of Walnut with partial words and contains proofs of both new results as well as previously known ones. We will define the notions which are most central to our work, but familiarity with some standard terms and ideas from combinatorics on words [13] is assumed.

A *square* or *cube* is a word of the form xx or xxx , respectively, for a nonempty word x . For example, $(010)^2 = 010010$ is a square and $(110)^3 = 110110110$ is a cube. We will consider words that avoid squares as well as those which avoid cubes. This means words that do not have a factor (i.e., contiguous substring) which is a square or cube respectively. A *squarefree* word is a word which avoids squares and a *cube-free* word is a word which avoids cubes. A *morphism* is a map $\psi : \Sigma^* \rightarrow \Delta^*$ between words over two alphabets such that $\psi(xy) = \psi(x)\psi(y)$ for all $x, y \in \Sigma^*$. We will make frequent use of morphisms to find words with a desired property.

A classic problem in combinatorics on words is constructing words avoiding squares, cubes, and other types of repetitions. Thue [20, 2] was able to construct an infinite cube-free binary word and an infinite square-free ternary word each of which can be obtained as the fixed point of a morphism. We let

$$\mathbf{tm} = 01101001100101101001011001101001 \dots$$

denote the *Thue-Morse* word which is the fixed point of the morphism $0 \mapsto 01$ and $1 \mapsto 10$ which begins with 0. We also let

$$\mathbf{vtm} = 012021012102012021020121 \dots$$

¹ We have used the version of Walnut available at <https://github.com/DistortedLight/Walnut>.



5:2 Walnut for Partial Words

denote the fixed point of the morphism $0 \mapsto 012$, $1 \mapsto 02$, and $2 \mapsto 1$ which is sometimes called the *ternary Thue-Morse word* (see e.g., [19]) or a *variant of the Thue-Morse word* (see [5]). It is the case that \mathbf{tm} is cubefree and \mathbf{vtm} is squarefree. We will make use of both of these words in constructions later.

A *partial word* is a word which can use a special character \diamond which is called a *hole* or *wildcard*. For partial words a square or cube is a partial word \mathbf{w} which is *contained* in a square $\mathbf{u} = xx$ or a cube $\mathbf{u} = xxx$ respectively in the sense the $\mathbf{w}[i] = \mathbf{u}[i]$ whenever $\mathbf{w}[i] \neq \diamond$. In this case we write $\mathbf{w} \subset \mathbf{u}$. The *order* of the square or cube is the length of x , which we denote by $|x|$. For example, $01101 \diamond 011$ is a partial word which is a cube of order 3 since it is contained in $(011)^3 = 011011011$. It is clear the presence of holes makes it more difficult to avoid squares or cubes. All words are partial words with no holes. When we wish to emphasize that a (partial) word has no holes we will refer to the word as a *full word*.

2 Squares, antisquares, cubes, and first-order logic

In this section we define what squares, antisquares, and cubes are in terms of first-order logic. This allows for seamless use with Walnut and highlights some differences between full words and partial words. All variables we quantify over are taken from the nonnegative integers unless specified otherwise.

For a full word \mathbf{w} containing a *square* means

$$\exists j \exists (n > 0) \forall i, (i < n) \implies (\mathbf{w}[j+i] = \mathbf{w}[j+n+i])$$

while for a partial word it means

$$\exists j \exists (n > 0) \forall i, (i < n) \implies ((\mathbf{w}[j+i] = \mathbf{w}[j+n+i]) \vee (\mathbf{w}[j+i] = \diamond) \vee (\mathbf{w}[j+n+i] = \diamond))$$

both of which can be expressed in first-order logic. We see that the expression for partial words contains more clauses. A value of n for which the above is made true is called the *order* of the square.

A partial word \mathbf{w} contains an *antisquare* provided

$$\exists j \exists (n > 0) \forall i, (i < n) \implies ((\mathbf{w}[j+i] \neq \mathbf{w}[j+n+i]) \wedge (\mathbf{w}[j+i] \neq \diamond) \wedge (\mathbf{w}[j+n+i] \neq \diamond))$$

which is consistent with what was considered for binary words in [17] and differs from the notion of an *anti-power* studied in [10]. We believe this to be a natural definition of an antisquare for a partial word given how it comes from negating the latter half of the implication in the logical expression for a square. We see that replacing a letter with a hole can create a square, and dually it can remove the presence of an antisquare. A factor which is an antisquare cannot contain any holes, but since we will consider squares and antisquares together in partial words holes will still play a crucial role.

■ **Table 1** Logical operators and corresponding symbols in Walnut.

\forall	\exists	\wedge	\vee	\neg	\implies
A	E	&	 	~	=>

Lastly, we say a partial word \mathbf{w} contains a *cube* if

$$\begin{aligned} \exists j \exists (n > 0) \forall i, (i < n) \implies & \left(((\mathbf{w}[j+i] = \mathbf{w}[j+n+i]) \wedge (\mathbf{w}[j+n+i] = \mathbf{w}[j+2n+i])) \right. \\ & \vee ((\mathbf{w}[j+i] = \diamond) \wedge (\mathbf{w}[j+n+i] = \mathbf{w}[j+2n+i])) \\ & \vee ((\mathbf{w}[j+n+i] = \diamond) \wedge (\mathbf{w}[j+i] = \mathbf{w}[j+2n+i])) \\ & \vee ((\mathbf{w}[j+2n+i] = \diamond) \wedge (\mathbf{w}[j+i] = \mathbf{w}[j+n+i])) \\ & \vee ((\mathbf{w}[j+i] = \diamond) \wedge (\mathbf{w}[j+n+i] = \diamond)) \\ & \vee ((\mathbf{w}[j+i] = \diamond) \wedge (\mathbf{w}[j+2n+i] = \diamond)) \\ & \left. \vee ((\mathbf{w}[j+n+i] = \diamond) \wedge (\mathbf{w}[j+2n+i] = \diamond)) \right) \end{aligned}$$

where we find a more drastic difference compared to what would be the first-order logic for the case of full words. We note that in the logical expression for a cube we would only need

$$(\mathbf{w}[j+i] = \mathbf{w}[j+n+i]) \wedge (\mathbf{w}[j+n+i] = \mathbf{w}[j+2n+i])$$

in the latter half of the implication for full words. One could continue to consider higher powers, and the number of additional clauses in the partial word version will continue to grow. We will restrict our attention to squares and cubes.

The proofs of many results in this paper are given by short snippets of Walnut code. We now explain a few aspects of Walnut to make these snippets more readable to a reader that does not have prior experience with this language. A more detailed explanation can be found in [15]. One can see in Table 1 how usual logical symbols are represented in Walnut. Our alphabet will always be $\{0, 1, \dots, N\}$ for some N , and we will use $N+1$ to denote the hole \diamond . For example, the binary partial word $0\diamond 10$ in Walnut would be `0210`. The symbol `@` is used to denote a character of the alphabet as oppose of an integer which can be the index of a position in a word. So, `W[2] = @3` is used to say that the character 3 is in position 2 of the word W . Lastly, morphisms can be defined intuitively where `0->010` encodes $0 \mapsto 010$ the image of 0.

3 Results

3.1 Avoiding long squares in binary

In this subsection we look at binary partial words which only contain short squares and antisquares. It is not possible to completely avoid squares since any binary (full) word of length at least 4 will contain some square. Let the morphism $h : \{0, 1, 2\} \rightarrow \{0, 1, \diamond\}$ be defined by

$$h(0) = 1100$$

$$h(1) = 011\diamond$$

$$h(2) = 1010$$

5:4 Walnut for Partial Words

which is a partial word variant of a morphism from [11, Section 2] that itself is a variant of a morphism used in [9] to produce a binary word which has no squares of order 3 or more. In particular, if the hole in the definition of the morphism is replaced with a 1, then the image under h of any ternary square-free word will be a binary word where all squares have order less than 3. With the addition of the hole we no longer avoid squares of order 3, but the resulting partial word will avoid squares of order 4 or more.

► **Theorem 1.** *The partial word $h(\mathbf{vtm})$ is a binary partial word with infinitely many holes that avoids squares of order 4 or more.*

Proof. The word \mathbf{vtm} is in Walnut as \mathbf{VTM} . So, all we need to do is define the morphism h and apply it to \mathbf{vtm} and check for squares of order 4 or more. In the code below \mathbf{Wh} denotes the image of this morphism. Recall, Walnut only uses $0, 1, \dots$, as letters. So, the image of h is a binary partial word represented over $\{0, 1, 2\}$ where 2 plays the role of \diamond . Running the following in Walnut

```
morphism h "0->1100, 1->0112, 2->1010";
image Wh h VTM;
eval no_sq "?msd_2 ~ Ej En Ai (n>3) & ((i<n)=>((Wh[j+i]=Wh[j+n+i])
| Wh[i+j]=@2 | Wh[i+n+j]=@2))";
```

returns “TRUE” which proves the result. ◀

► **Remark 2.** To our knowledge the construction in Theorem 1 is new in the context of partial words, but the result on avoiding long squares is not optimal. In [4, Theorem 4] an infinite binary partial word with infinitely many holes is constructed so that the only squares compatible with factors of it are 0^2 , 1^2 , $(01)^2$, and $(11)^2$. This is based on a construction for full words from [18]. We have been unable to automate a version of this construction in Walnut due to the space required for the computation.

We now consider the morphism $f : \{0, 1, \dots, 7\}^* \rightarrow \{0, 1, \dots, 7\}^*$ given by

$$\begin{array}{ll} f(0) = 01 & f(1) = 23 \\ f(2) = 24 & f(3) = 51 \\ f(4) = 06 & f(5) = 01 \\ f(6) = 74 & f(7) = 24 \end{array}$$

along with the coding $g : \{0, 1, \dots, 7\}^* \rightarrow \{0, 1, \diamond\}$ by $g(m) = m \pmod{2}$ for $m \neq 6$ and $g(6) = \diamond$. Applying g to the fixed point of f will give us a word avoiding both squares and antisquares of large length. Since $f(0) = 01$ we may iterate applying f to 0 to obtain the unique fixed point of the morphism f we denote by $f^\omega(0)$. We will make use of the notation f^ω to denote fixed points of morphisms elsewhere as well.

► **Theorem 3.** *The partial word $g(f^\omega(0))$ is a binary partial word with infinitely many holes that avoids squares of order 7 or more and avoids antisquares of order 3 or more.*

Proof. We establish the theorem by running the following in Walnut

```
morphism f "0->01, 1->23, 2->24, 3->51, 4->06, 5->01, 6->74, 7->24";
morphism g "0->0, 1->1, 2->0, 3->1, 4->0, 5->1, 6->2, 7->1";
promote Wf f;
```

■ **Table 2** The length of the longest binary partial word with a single hole that contains at most a squares and at most b antisquares.

$a \backslash b$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	...
1	3	4	5	5	5	5	5	5	5	5	5	5	5	5	...
2	5	7	9	10	11	11	12	12	14	14	15	16	16	16	...
3	7	11	14	19	19	19	19	22	26	30	34	52	97		
4	9	15	22	27	30	45	54	103	397						
5	11	19	35	40	74										
6	13	23	47	50											
7	15	27	59												
8	17	31	147												
9	19	35													
10	21	39													
⋮	⋮	⋮													

```
image Wg g Wf;
eval no_sq "?msd_2 ~ Ej En Ai (n>6) & ((i<n)=>((Wg[j+i]=Wg[j+n+i])
| Wg[i+j]=@2 | Wg[i+n+j]=@2))";

eval no_anti "?msd_2 ~ Ej En Ai (n>2) & ((i<n)=>((Wg[j+i]!=Wg[j+n+i])
& Wg[i+j]!=@2 & Wg[i+n+j]!=@2))";
```

which outputs “TRUE” twice. ◀

► **Remark 4.** In was shown in [17, Theorem 9] that the full word obtained by coding the fixing point of f with $m \mapsto m \pmod{2}$ for all $m \in \{0, 1, \dots, 7\}$ avoids both squares and antisquares of order 3 or more. Since adding holes cannot make any antisquares the fact about avoiding antisquares in Theorem 3 is immediate.

In [17] for any fixed a and b , the problem of finding the longest binary full word which contains at most a distinct squares and at most b distinct antisquares was solved. One can consider versions of the same problem for partial words. For example, we can ask for the length of the longest binary partial word with a fixed number of holes that contains at most a distinct squares and at most b distinct antisquares. We will focus on the case of partial words with a single hole. Counting distinct squares has received much attention for both words and partial words. It is known that even the addition of a single hole can fundamentally change distinct squares [6, 7, 12, 14]. Unlike if we were simply avoiding squares, the length of this longest binary partial word can be shorter or longer than that of the corresponding full word. This is since replacing a letter in a binary full word can possibly create a square while it also has the potential to remove an antisquare.

Consider the following example for $a = 4$ and $b = 5$. The length 32 partial word

◊0111010000011010000110000010000

contains only the squares 0^2 , 1^2 , $(00)^2$, and $(10)^2$. It also contains only the antisquares 01, 10, 0011, 0110, and 1100. For full words the optimal length is 31 which was originally computed in [17, Figure 1]. One binary full word giving witness to this length is

0111010000011010000110000010000

which is obtained from the partial word above by removing the hole. Notice prepending 0 to this full word creates the new antisquare 001110 while prepending 1 creates 1011101000. Prepending \diamond allows us to avoid new antisquares. Moreover, we can create even longer partial words with one hole and only 4 distinct squares and 5 distinct antisquares. The optimal length is 45. The partial word

$$000010000011000010110000011 \diamond 00101100000101110$$

has length 45 and only contains the squares 0^2 , 1^2 , $(00)^2$, and $(01)^2$ as well as only the antisquares 01, 10, 0011, 0110, and 1100. In Table 2 we list the optimal lengths for many values of a and b .

► **Theorem 5.** *The values in Table 2 are correct.*

Proof. The first three rows and first two columns are each infinite sequences of finite values and must be proven. Outside of these rows and columns there are only finitely many entries which can be computed. The first three rows follow from the fact that a binary partial word with one hole that contains at most 0, 1, or 2 squares has length at most 1, 5, or 16 respectively. This can be seen by direct verification of this fact, then computing the values until we reach 1, 5 or 16. Note it is clear the entries weakly increase along row and column.

For the first column, up to complementation, we consider words of the form $0^m \diamond 0^n$ or $0^m \diamond 1^n$ which are the only binary partial words with a single hole that do not contain an antisquare. Let $\ell = m + n + 1$ be the length of such a word. The number of squares such a word contains is

$$\left\lfloor \frac{m+1}{2} \right\rfloor + \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{\ell}{2} \right\rfloor$$

and so $\ell = 2a + 1$ is the longest length containing at most a distinct squares and 0 antisquares.

Now for the second column we consider partial words with a single hole and only 1 antisquare. Let us assume $a > 1$. We will look at partial words which start with 0 and contain only the antisquare 01. So, we have $0^m 1^n \diamond 0^p 1^q$ with $m > 0$. Note if $m > 1$, then $n \leq 1$ or else we have both the antisquares 01 and 0011. Similarly if $p > 1$, then $q \leq 1$. So, let us assume our partial word is $0^m 1 \diamond 0^p 1$. This partial word has $\lfloor \frac{N}{2} \rfloor + 1$ distinct squares where $N = \max(m, p + 1)$ unless $m = p + 1$ and the whole partial word then gives an additional square. The squares contained are 1^2 and 0^{2k} for $2k < N$ along with possibly $(0^m 1)^2$. So, we may take $0^{2a-2} 1 \diamond 0^{2a-2} 1$ which has length $4a - 1$ and contains a squares and 1 antisquare. This gives us one such word realizing the maximum agrees with what is found in Table 2. We also have the partial word $0^{2a-1} 1 \diamond 0^{2a-2}$ of length $4a - 1$ which contains a squares and 1 antisquare. It turns out that all other such partial words can be obtained from the two we have given by complement and reversal. This can be checked by considering the remaining cases of the possible forms of the partial word in a similar manner. ◀

► **Remark 6.** Table 2 is incomplete, and we do not know if the missing entries are finite or infinite. From [17] it is known that for full binary words the entry corresponding to $a = 5$ and $b = 5$ is finite while the remaining missing entries are infinite.

3.2 Avoiding non-trivial squares and cube with many holes

In this subsection we demonstrate how some known constructions [8] of partial words “dense” with holes avoiding powers can be obtained and verified through Walnut. The *hole sparsity* of a partial word is smallest s such that every factor of length s contains at least one hole. A

square of the form $a\diamond$ or $\diamond a$ for some letter a is called a *trivial square*. Indeed every partial word of length at least 2 which contains at least one hole as well as at least one letter will contain a trivial square. Thus for avoidance purposes we must allow trivial squares and avoid non-trivial squares. Since the presence of holes makes squares or cubes more likely, it is an interesting problem to find partial words with small hole sparsity (and hence many holes) which avoid non-trivial squares or cubes (or more generally higher powers).

Let us consider the morphism $\rho : \{0, 1, 2, 3\}^* \rightarrow \{0, 1, 2, 3\}^*$ defined by

$$\rho(0) = 03$$

$$\rho(1) = 12$$

$$\rho(2) = 01$$

$$\rho(3) = 10$$

which appears in [1, Exercise 33(c)]. Next we consider the morphism $\sigma : \{0, 1, 2, 3\}^* \rightarrow \{0, 1, 2, 3, \diamond\}^*$ defined by

$$\sigma(0) = 320\diamond$$

$$\sigma(1) = 120\diamond$$

$$\sigma(2) = 310\diamond$$

$$\sigma(3) = 130\diamond$$

whose image is a partial word over an alphabet with size 4. We are now ready to give an automated proof of the following which is in the proof of [8, Lemma 2].

► **Theorem 7.** *The partial word $\sigma(\rho^\omega(0))$ is a partial word with hole sparsity 4 over an alphabet of size 4 which avoids non-trivial squares.*

Proof. It is easy to see that the squares \diamond^2 , 0^2 , 1^2 , 2^2 , and 3^2 do not occur in $\sigma(\rho^\omega(0))$. Recall, since we are avoiding non-trivial squares $a\diamond$ and $\diamond a$ are allowed to us present for $a \in \{0, 1, 2, 3\}$. Thus, we only need to worry about squares of order n for $n > 1$. Running the following commands in Walnut

```
morphism rho "0->03, 1->12, 2->01, 3->10";
morphism sigma "0->320\diamond, 1->120\diamond, 2->310\diamond, 3->130\diamond";
promote Wrho rho;
image W sigma Wrho;
eval no_sq "?msd_2 ~ E j En Ai (n>1) & ((i<n)=>((W[j+i]=W[j+n+i])
| W[i+j]=@4 | W[i+n+j]=@4))";
```

results in an output of “TRUE” and the theorem is proven. ◀

We let τ denote the morphism defined by $\tau(0) = 01\diamond$ and $\tau(1) = 02\diamond$. We can now give an automated proof of the following which was first proven in [8, Lemma 7].

► **Theorem 8.** *The partial word $\tau(\mathbf{tm})$ is partial word with hole sparsity 3 over an alphabet of size 3 which avoids cubes.*

Proof. The word \mathbf{tm} is contained in Walnut at T. We run the following in Walnut

```
morphism tau "0->01\diamond, 1->02\diamond";
image W tau T;
eval no_cube "?msd_2 ~ E j En Ai (n>1)&((i<n)=>("
```

```

W[j+i]=W[j+n+i] & W[j+n+i]=W[j+2*n+i])
|(W[j+i]=03 & W[j+n+i]=W[j+2*n+i])|(W[j+n+i]=03 & W[j+i]=W[j+2*n+i])
|(W[j+2*n+i]=03 & W[j+i]=W[j+n+i])|(W[j+i]=03 & W[j+n+i]=03)
|(W[j+i]=03 & W[j+2*n+i]=03) |(W[j+n+i]=03 & W[j+2*n+i]=03))";

```

and it outputs “TRUE” proving the theorem. ◀

► Remark 9. Both Theorem 7 and Theorem 8 are optimal in establishing smallest hole sparsity avoiding non-trivial squares and cubes respectively for a given alphabet size [3]. For example, there does not exist an infinite partial word with hole sparsity 3 over a 4 letter alphabet which avoids non-trivial squares.

4 Conclusion

We have initiated a study of partial words paired with the theorem prover Walnut. Additionally, we have extended the definition of antisquares to partial words. We believe both directions could be a source of new problems in combinatorics on words. Furthermore, we have given alternative proofs of some results on partial words which provide machine verification. We have discussed how the logical statements expressing a square are longer for partial words than full words. So this adds some complexity to the Walnut calculations. Additionally, in Walnut partial words work with an alphabet with an extra letter which represents the hole. Let us close with an example comparing partial words with full words in Walnut.

Let us consider the morphisms g and h defined by

$$\begin{array}{ll}
 g(0) = 1100 & h(0) = 1100 \\
 g(1) = 0111 & h(1) = 011\circ \\
 g(2) = 1010 & h(2) = 1010
 \end{array}$$

where h was the morphism used in Theorem 1. To get an idea of what happens going from full words to partial words we have the deterministic finite automata with output (DFAO) for $g(\mathbf{vtm})$ and $h(\mathbf{vtm})$ shown in Figure 1 and Figure 2 respectively. Walnut works with automatic sequences using their DFAOs. We find in this case only a modest increase in the size of the DFAO, and we were indeed able to use Walnut to automate a proof in the more complex but still tractable partial word situation.

To show there are no squares of length greater than 3 in neither $g(\mathbf{vtm})$ nor $h(\mathbf{vtm})$ we may run

```

morphism h "0->1100, 1->0112, 2->1010";
image Wh h VTM;
eval no_sq "?msd_2 ~ Ej En Ai (n>3) & ((i<n)=>((Wh[j+i]=Wh[j+n+i])
| Wh[i+j]=02 | Wh[i+n+j]=02))";

morphism g "0->1100, 1->0111, 2->1010";
image Wg g VTM;
eval no_sq_full "?msd_2 ~ Ej En Ai (n>3) & ((i<n)=>(Wg[j+i]=Wg[j+n+i]))";

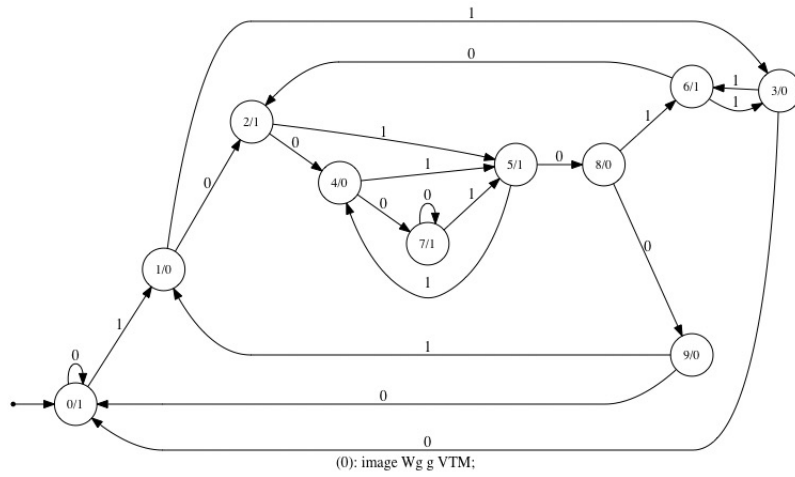
```

in Walnut. In the log files produced we will find the following two lines

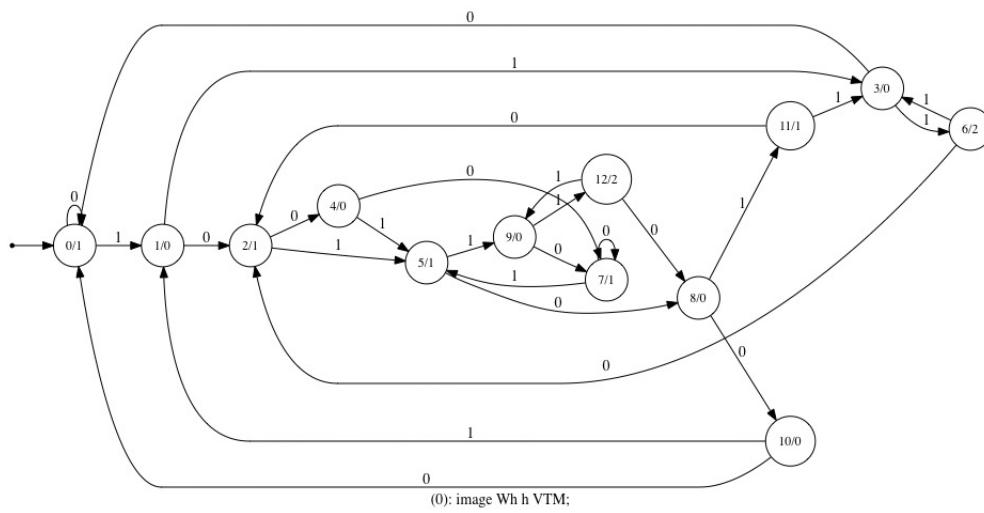
```
(i<n=>((Wh[(j+i)]=Wh[((j+n)+i)]|Wh[(i+j)]=02)|Wh[((i+n)+j)]=02)):229 states - 24ms
```

and

```
(i<n=>Wg[(j+i)]=Wg[((j+n)+i))):217 states - 10ms
```

■ **Figure 1** The DFAO for $g(\mathbf{vtm})$.



■ **Figure 2** The DFAO for $h(\mathbf{vtm})$.

showing the sizes of the automata Walnut needs to determine the exists of a square in either $g(\mathbf{vtm})$ or $h(\mathbf{vtm})$ respectively. We see the partial word version requires 229 states more states and takes to build 24ms compared 217 states and to 10ms for the full word version. The issue one encounters in Walnut computations is typically in issue of space due to building some automaton. This example, and the others we have given, suggest that partial word versions of theorems may take slightly more space in Walnut but may often be tractable with Walnut when their full word counterparts are.

References

- 1 Jean-Paul Allouche and Jeffrey Shallit. *Automatic sequences*. Cambridge University Press, Cambridge, 2003. Theory, applications, generalizations. doi:10.1017/CB09780511546563.
- 2 Jean Berstel. Axel thue’s papers on repetitions in words: a translation. Publications de Laboratoire de Combinatoire et d’Informatique Mathématique 20, Université du Québec à Montréal, 1995.
- 3 Kevin Black, Francine Blanchet-Sadri, Ian Coley, Brent Woodhouse, and Andrew Zemke. Pattern avoidance in partial words dense with holes. *J. Autom. Lang. Comb.*, 22(4):209–241, 2017. doi:10.25596/jalc-2017-209.
- 4 F. Blanchet-Sadri, Ilkyoo Choi, and Robert Mercas. Avoiding large squares in partial words. *Theoret. Comput. Sci.*, 412(29):3752–3758, 2011. doi:10.1016/j.tcs.2011.04.009.
- 5 F. Blanchet-Sadri, James D. Currie, Narad Rampersad, and Nathan Fox. Abelian complexity of fixed point of morphism $0 \mapsto 012$, $1 \mapsto 02$, $2 \mapsto 1$. *Integers*, 14:Paper No. A11, 17, 2014.
- 6 F. Blanchet-Sadri, Yang Jiao, John M. Machacek, J. D. Quigley, and Xufan Zhang. Squares in partial words. *Theoret. Comput. Sci.*, 530:42–57, 2014. doi:10.1016/j.tcs.2014.02.023.
- 7 F. Blanchet-Sadri, Robert Mercas, and Geoffrey Scott. Counting distinct squares in partial words. *Acta Cybernet.*, 19(2):465–477, 2009.
- 8 Francine Blanchet-Sadri, Kevin Black, and Andrew Zemke. Unary pattern avoidance in partial words dense with holes. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings*, volume 6638 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2011. doi:10.1007/978-3-642-21254-3_11.
- 9 R. C. Entringer, D. E. Jackson, and J. A. Schatz. On nonrepetitive sequences. *J. Combinatorial Theory Ser. A*, 16:159–164, 1974. doi:10.1016/0097-3165(74)90041-7.
- 10 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-powers in infinite words. *J. Combin. Theory Ser. A*, 157:109–119, 2018. doi:10.1016/j.jcta.2018.02.009.
- 11 Daniel Gabric and Jeffrey Shallit. The simplest binary word with only three squares. *RAIRO Theor. Inform. Appl.*, 55:Paper No. 3, 7, 2021. doi:10.1051/ita/2021001.
- 12 Vesa Halava, Tero Harju, and Tomi Kärki. On the number of squares in partial words. *RAIRO Theor. Inform. Appl.*, 44(1):125–138, 2010. doi:10.1051/ita/2010008.
- 13 M. Lothaire. *Combinatorics on words*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1997. doi:10.1017/CB09780511566097.
- 14 John Machacek. Partial words with a unique position starting a square. *Inform. Process. Lett.*, 145:44–47, 2019. doi:10.1016/j.ipl.2019.01.010.
- 15 Hamoon Mousavi. Automatic theorem proving in Walnut. arXiv:1603.06017.
- 16 Hamoon Mousavi, Luke Schaeffer, and Jeffrey Shallit. Decision algorithms for Fibonacci-automatic words, I: Basic results. *RAIRO Theor. Inform. Appl.*, 50(1):39–66, 2016. doi:10.1051/ita/2016010.
- 17 Tim Ng, Pascal Ochem, Narad Rampersad, and Jeffrey Shallit. New results on pseudosquare avoidance. In *Combinatorics on words*, volume 11682 of *Lecture Notes in Comput. Sci.*, pages 264–274. Springer, Cham, 2019. doi:10.1007/978-3-030-28796-2_21.
- 18 Narad Rampersad, Jeffrey Shallit, and Ming-wei Wang. Avoiding large squares in infinite binary words. *Theoret. Comput. Sci.*, 339(1):19–34, 2005. doi:10.1016/j.tcs.2005.01.005.

- 19 Michaël Rao, Michel Rigo, and Pavel Salimov. Avoiding 2-binomial squares and cubes. *Theoret. Comput. Sci.*, 572:83–91, 2015. doi:10.1016/j.tcs.2015.01.029.
- 20 Axel Thue. *Selected mathematical papers*. Universitetsforlaget, Oslo, 1977. With an introduction by Carl Ludwig Siegel and a biography by Viggo Brun, Edited by Trygve Nagell, Atle Selberg, Sigmund Selberg, and Knut Thalberg.

An FPT-Algorithm for Longest Common Subsequence Parameterized by the Maximum Number of Deletions

Laurent Bulteau¹  

LIGM, CNRS, Université Gustave Eiffel, F77454 Marne-la-vallée, France

Mark Jones  

Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands

Rolf Niedermeier  

Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany

Till Tantau  

Institute of Theoretical Computer Science, University of Lübeck, Germany

Abstract

In the NP-hard LONGEST COMMON SUBSEQUENCE problem (LCS), given a set of strings, the task is to find a string that can be obtained from every input string using as few deletions as possible. LCS is one of the most fundamental string problems with numerous applications in various areas, having gained a lot of attention in the algorithms and complexity research community. Significantly improving on an algorithm by Irving and Fraser [CPM'92], featured as a research challenge in a 2014 survey paper, we show that LCS is fixed-parameter tractable (FPT) when parameterized by the maximum number of deletions per input string. Given the relatively moderate running time of our algorithm (linear time when the parameter is a constant) and small parameter values to be expected in several applications, we believe that our purely theoretical analysis could finally pave the way to a new, exact and practically useful algorithm for this notoriously hard string problem.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Theory and algorithms for application domains

Keywords and phrases NP-hard string problems, multiple sequence alignment, center string, parameterized complexity, search tree algorithms, enumerative algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.6

Funding *Mark Jones*: Supported by Netherlands Organization for Scientific Research (NWO) through grants NETWORKS and OCENW.KLEIN.125.

Rolf Niedermeier: Supported by the DFG, NI 369/16, FPTinP.

Acknowledgements This work was initiated during Dagstuhl Seminar 19443, *Algorithms and Complexity in Phylogenetics*, held at Schloss Dagstuhl, Germany, in October 2019 [3].

Dedication to Rolf We dedicate this paper to our dear friend Rolf Niedermeier, who tragically passed away recently. We are deeply affected by this loss of our co-author, colleague, and advisor. Rolf has contributed tremendously to computer science and, in particular, to multivariate algorithms, and should have continued doing so for a long time. The computer science community will build on the foundations he has laid.

¹ Corresponding author



1 Introduction

With its numerous applications including bioinformatics, data compression, and computational linguistics, the NP-hard LONGEST COMMON SUBSEQUENCE (LCS) problem is among the best studied algorithmic string problems. Suiting our subsequent parameterized analysis purposes, we formally define the problem as follows.

LONGEST COMMON SUBSEQUENCE

Input: A set of k strings $\mathcal{S} = \{S_1, \dots, S_k\}$ on some alphabet Σ , each of length at most n , an integer ℓ .

Parameter: $\Delta = n - \ell$.

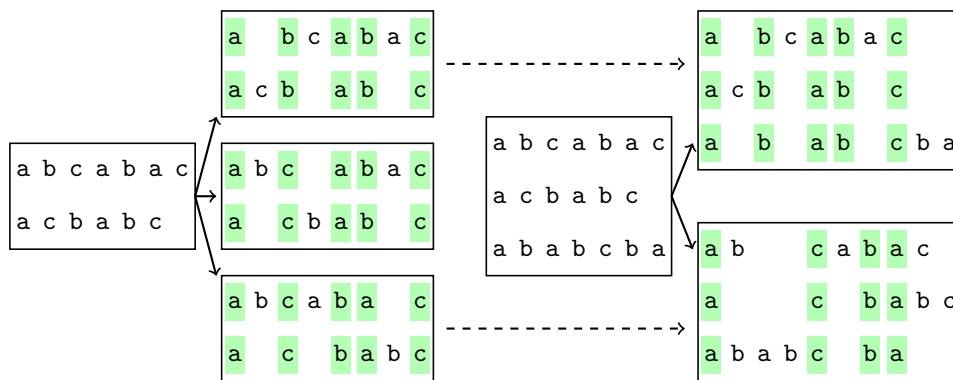
Question: Is there a string S of length at least ℓ that is a (not necessarily contiguous) subsequence of each S_i ?

For example, for $k = 3$ strings $abcabac$, $acbabc$, $ababcba$ (thus, $n = 7$), with $\ell = 5$ we have a yes-instance (with solution string $ababc$), while for $\ell = 6$ this would be a no-instance.

With straightforward dynamic programming, using the number k of strings as a parameter, LCS can be solved in $O(n^k)$ time. The problem has been shown to be W[1]-hard [16], and even W[2]-hard for parameter k , and it has no $O(n^{k-\epsilon})$ algorithm assuming the Strong Exponential Time Hypothesis (SETH) [1]. Indeed, LCS is *the* string problem having received most attention in the early years of parameterized complexity analysis [10]. Unfortunately, so far parameterized complexity analysis beyond trivial algorithmic observations mainly contributed computational hardness results. We refer to some surveys [2, 7, 8] for an overview on research results and open questions for LCS.

We remark that the special case of two input strings (that is, $k = 2$) recently attracted much attention, particularly motivated by the theoretical challenge of breaking the straightforward time bound of $O(n^2)$ [4, 6, 11]. Notably, Bringmann and Künnemann [6] (the corresponding arXiv paper has around 60 pages) also discuss the “maximum number of deletions” parameter we focus on here. We note however that in the context of $k = 2$, the parameter is used to improve over the classical $O(n^2)$ dynamic programming algorithm while maintaining a polynomial running time, while our approach requires an exponential dependency on Δ even for $k = 2$. Indirectly, this parameter already appears in the work of Irving and Fraser [13], who provided two algorithms for LCS with three or more input strings.

Irving and Fraser [13] in their 1992 paper provided an algorithm for LCS running in time $O(kn(n - \ell)^{k-1})$, implying fixed-parameter tractability with respect to the combined parameter k and $n - \ell$, where the latter coincides with our parameter Δ . We are not aware of any improvement since then and this is also reflected by a corresponding challenge featured in a 2014 survey [7, Challenge 9]. Answering positively the research challenge posed there, we improve Irving and Fraser’s result to fixed-parameter tractability with respect to only Δ . More specifically, our algorithm runs in time $O((\Delta + 1)^{\Delta+1}kn)$, which means linear time when Δ is a constant. In addition, we can *enumerate all* longest common subsequences within this time. Given that it seems natural to assume that in many applications the sought common subsequence is fairly close to every input string (which would imply small values of Δ), this promises to be of also practical relevance. The focus of our work, however, is purely theoretical. Regarding the alphabet size, we focus on the general case where the alphabet is unbounded (in particular, the alphabet size is *not* hidden in the O of the running time). The question of whether our approach can be improved for constant-size alphabets (typically $\{0, 1\}$ or $\{A, T, C, G\}$) is left open.



■ **Figure 1** Our approach towards computing the LCS of three strings `abcabac`, `acbabc`, `ababcba`. Left: compute maximal common subsequences of the first two strings (all three subsequences and their alignment with input strings are depicted). Right: compute maximal common subsequences of all three strings by comparing those obtained at the first step with the third input string (only two strings remain after filtering non-maximal common subsequences). The longest result, `ababc` is the LCS of the input strings. Filtering out strings that are shorter than a threshold prevents the number of intermediate strings from growing too fast, yielding our FPT-algorithm.

Figure 1, at a very high level, presents an example for LCS for three input strings together with the main idea behind our recursive approach towards achieving our result, the FPT-algorithm for parameter Δ . More specifically, our algorithm builds and refines an exhaustive list of maximal common subsequences after reading each input string. Maximal common subsequences have been used before in LCS-related questions, see e. g. recent advances by Sakai [17] and Conte et al. [9], but not, to the best of our knowledge, towards exactly solving LCS on multiple input strings.

2 An LCS Algorithm Using Maximal Common Subsequences

In this section, we present a linear-time algorithm for LCS when the number of deletions is a constant. Note that this does not contradict the quadratic lower bound for this problem, since this lower bound only applies to the general case where the number of deletions is unbounded. In particular, the $O(\delta n)$ algorithm by Nakatsu et al. [14] (with $\delta = \min\{|S_i|\} - \ell$) remains better than our algorithm for the two-string case. Furthermore, it is not clear if a smaller (typically constant) alphabet could be exploited in our algorithm or its analysis in order to obtain a better running time.

2.1 Definitions

Strings. The set of strings on an alphabet Σ is denoted Σ^* . The empty string is denoted ϵ , the length of a string $S \in \Sigma^*$ is denoted $|S|$. We write \cdot for the concatenation and $u \cdot T := S$ as a short-hand for “let $u \in \Sigma$ be the first character of S and T be the suffix of S starting from the second character (or $u = T = \epsilon$ if S is empty)”. We write $S[i, \dots, j]$ for the substring of S of all symbols between positions i and j inclusively.

Given two strings S_1, S_2 , we write $S_1 \preceq S_2$ (resp. $S_1 \prec S_2$) if S_1 is a (strict) subsequence of S_2 . Formally, $\epsilon \preceq S$ for any S and, if $S \preceq S'$, then for any u we have $S \preceq u \cdot S \preceq u \cdot S'$.

Longest and Maximal Common Subsequences. Given a set \mathcal{S} of strings and a nonnegative integer ℓ , let $\text{CS}_\ell(\mathcal{S})$ denote the set of all common subsequences of \mathcal{S} that have length at least ℓ . Let L be the largest integer such that $\text{CS}_L(\mathcal{S})$ is not empty, and let $\text{LCS}(\mathcal{S})$ denote an arbitrary string in $\text{CS}_L(\mathcal{S})$, i. e. a longest common subsequence of \mathcal{S} .

Let $\text{MCS}_\ell(\mathcal{S})$ denote the set of all *maximal* common subsequences of \mathcal{S} with length at least ℓ ; that is, $S \in \text{MCS}_\ell(\mathcal{S})$ iff $S \in \text{CS}_\ell(\mathcal{S})$ and there is no $S' \in \text{CS}_\ell(\mathcal{S})$ such that $S \prec S'$. Note that, if ℓ is small enough ($\ell \leq L$), then $\text{LCS}(\mathcal{S}) \in \text{MCS}_\ell(\mathcal{S})$, otherwise $\text{MCS}_\ell(\mathcal{S})$ is empty. A set of strings M is an *extended MCS* of (\mathcal{S}, ℓ) if $\text{MCS}_\ell(\mathcal{S}) \subseteq M \subseteq \text{CS}_\ell(\mathcal{S})$.

String Parameters. Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a set of strings. We write $n(\mathcal{S}) := \max_{S \in \mathcal{S}} |S|$, $m(\mathcal{S}) := \min_{S \in \mathcal{S}} |S|$. Given an integer ℓ , we write $\Delta(\mathcal{S}, \ell) = n(\mathcal{S}) - \ell$ and $\delta(\mathcal{S}, \ell) := m(\mathcal{S}) - \ell$. We omit dependencies on \mathcal{S} and ℓ when the context is clear (e. g., they are given in the lemma statement). Note that $\delta \leq \Delta$.

2.2 Main Results

► **Theorem 1.** *Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a set of strings and ℓ be an integer. Then an extended MCS of (\mathcal{S}, ℓ) with size at most $(\Delta + 1)^\delta$ can be computed in time $O(2^{\delta+\Delta}(\Delta + 1)^\delta kn)$.*

Theorem 1 directly yields an algorithm for LCS, since it suffices to test if an extended MCS of (\mathcal{S}, ℓ) is non-empty. Note that the algorithm can be adapted for the optimization formulation of LCS, i. e., when ℓ is not part of the input, with a constant factor in the time complexity (taking δ and Δ with respect to $\ell = |\text{LCS}(\mathcal{S})|$). Indeed, apply Theorem 1 for decreasing values of ℓ starting with $\ell = m$, until a non-empty set is obtained. Then, the resulting set contains the common subsequences of \mathcal{S} of size $\text{LCS}(\mathcal{S})$ (indeed, $\text{MCS}_\ell(\mathcal{S}) = \text{CS}_\ell(\mathcal{S})$ for this value of ℓ), so it contains all longest common subsequences of \mathcal{S} . The time complexity of the i -th call, $1 \leq i \leq \delta$, is upper-bounded by $O(2^{i+\Delta}(\Delta + 1)^\delta kn)$. Using $\sum_{i=1}^{\delta} 2^i = O(2^\delta)$, we get the following corollary.

► **Corollary 2.** *All longest common subsequences of \mathcal{S} (and a fortiori the value $\text{LCS}(\mathcal{S})$) can be computed in time $O(2^{\delta+\Delta}(\Delta + 1)^\delta kn)$.*

The remainder of the section is dedicated to proving Theorem 1. We first compute the number of strings and their size distribution in the MCS of two strings, then build up on this result to bound the size of the MCS of k strings.

2.3 Extended MCS for Two Strings

Algorithm 1 allows us to compute an extended MCS of two strings. Its correctness is proven using the main recursive relation for MCS given in Lemma 3, while its time complexity is analyzed in Lemmas 6 and 8.

► **Lemma 3.** *For any two non-empty strings $S, S' \in \Sigma^*$ and any ℓ , let $u \cdot T := S$ and $u' \cdot T' := S'$ for $u, u' \in \Sigma$.*

If $u = u'$, then $\text{MCS}_\ell(\{S, S'\}) \subseteq \{u \cdot X \mid X \in \text{MCS}_{\ell-1}(\{T, T'\})\}$.

If $u \neq u'$, then $\text{MCS}_\ell(\{S, S'\}) \subseteq \text{MCS}_\ell(\{S, T'\}) \cup \text{MCS}_\ell(\{T, S'\})$.

Proof. Let $R \in \text{MCS}_\ell(\{S, S'\})$, and $r \cdot X := R$.

For the first case ($u = u'$), we show that $r = u$ and X is a maximal common subsequence of $\{T, T'\}$ of length at least $\ell - 1$. Indeed, $r = u$, as otherwise the concatenation $u \cdot r \cdot X$ would also be a common subsequence of $\{S, S'\}$, with $R \prec u \cdot r \cdot X$ (contradicting the

■ **Algorithm 1** Compute a bounded-size extended MCS of two strings.

```

1  algorithm xMCS2( $\ell, S, S'$ )
2    if  $\ell > |S|$  or  $\ell > |S'|$  then return  $\emptyset$ 
3    if  $S \preceq S'$  then return  $\{S\}$ 
4    if  $S' \preceq S$  then return  $\{S'\}$ 
5     $u \cdot T := S$ 
6     $u' \cdot T' := S'$ 
7    if  $u = u'$  then
8      return  $\{u \cdot X \mid X \in \text{xMCS2}(\ell - 1, T, T')\}$ 
9    else
10   return  $\text{xMCS2}(\ell, S, T') \cup \text{xMCS2}(\ell, T, S')$ 

```

maximality of R). Note that X is a subsequence both of T and T' . Moreover X is maximal, as otherwise, if $X \prec X'$ with X' a common subsequence of T, T' , then $u \cdot X'$ would be a common subsequence of S, S' with $R \prec u \cdot X'$ (again, contradicting the maximality of R).

For the second case ($u \neq u'$), we show that R is either in $\text{MCS}_\ell(\{T, S'\})$, or in $\text{MCS}_\ell(\{S, T'\})$ (or both). Indeed, if $r \neq u$, then $R \prec S$ implies $R \prec T$, and R is a common subsequence of $\{T, S'\}$. Otherwise, $r = u \neq u'$, and R is a common subsequence of $\{S, T'\}$. In both cases, R is maximal, since for any R' , if R' is a common subsequence of (say) $\{T, S'\}$ with $R \prec R'$, then R' is also a common subsequence of $\{S, S'\}$ which contradicts the maximality of R for $\{S, S'\}$. ◀

Algorithm 1 follows the recursive relation of Lemma 3, along with trivial base cases ($\ell > \min\{|S|, |S'|\}$ or one of S or S' being a subsequence of the other). It also clearly returns only common subsequences of S and S' of length at least ℓ , so it is correct.

▶ **Corollary 4.** *Let $S, S' \in \Sigma^*$ and ℓ be an integer. Then $\text{xMCS2}(\ell, S, S')$ from Algorithm 1 returns an extended MCS of $(\{S, S'\}, \ell)$.*

▶ **Remark 5.** The first inclusion in Lemma 3 (case $u = u'$) is actually an equality, but we only need this direction for the algorithm to be correct. The second inclusion, however, may be strict: for example with $S = abcd$ and $S' = dabc$, the string $R = bc$ is a maximal common subsequence of $T = bcd$ and S' , but not of S and S' since $R \prec abc$. Such “extra” strings are actually returned by our algorithm, motivating the naming of *extended* MCS (although they could be filtered out, see Remark 7).

We now focus on the time complexity of Algorithm 1.

▶ **Lemma 6.** *Let $S, S' \in \Sigma^*$ have lengths $\ell + \delta$ and $\ell + \Delta = n$, respectively. Then $\text{xMCS2}(\ell, S, S')$ terminates in time $O(2^{\delta + \Delta n})$.*

Proof. To achieve the claimed time complexity, we first need to perform the subsequence tests in lines 3 and 4 quickly. For this, we use a precomputed subsequence table: For every pair (i, i') with $1 \in \{1, \dots, |S|\}$ and $i' \in \{1, \dots, |S'|\}$ and $|i - i'| \leq \Delta$, let $\text{sub}[i, i']$ contain **True** if $S[i, \dots, |S|]$ is a subsequence of $S'[i', \dots, |S'|]$. In other words, $\text{sub}[i, i']$ is **True** iff the i th suffix of S is a subsequence of the i' th suffix of S' . The entries of this table can be computed in time $O(n\Delta)$ by straightforward dynamic programming using the following relations:

$$\begin{aligned} \text{sub}[i, i'] &= \text{sub}[i + 1, i' + 1] && \text{if } S[i] = S'[i'], \\ \text{sub}[i, i'] &= \text{sub}[i + 1, i'] \vee \text{sub}[i, i' + 1] && \text{otherwise.} \end{aligned}$$

Note that during recursive calls, the values of Δ and δ are non-increasing, and $\Delta + \delta$ decreases by 1 in the case where two recursive calls are performed. In particular, if $\ell \leq \min\{|S|, |S'|\}$ in a recursive call, then $||S| - |S'|| \leq \Delta$, which enables us to use the precomputed table for the subsequence test. So the total number of leaves in the tree of recursive calls is at most $2^{\delta+\Delta}$, each call taking constant time, and the height of this tree is at most $\ell + \delta + \Delta \leq 2n$. Thus the algorithm takes overall time $O(2^{\delta+\Delta}n)$. ◀

► **Remark 7.** Algorithm 1 can be adapted to output only the set of maximal common subsequences, rather than an extended version of it, by simply removing non-maximal strings (which can be done in quadratic time in the size of the output set). However, this does not improve the theoretical size of the returned set since in the worst case it does not filter out any string, but adds a quadratic running time to the complexity. It should be an important step in an implementation of the algorithm, though, since an additive quadratic computation time would probably be quickly compensated by pruning a possibly exponential search tree.

The proof of Lemma 6 gives a first bound on the number of strings returned by xMCS2 (namely, at most $2^{\delta+\Delta}$). We know that all strings have lengths between ℓ and m . However, we will need an additional ingredient for a more precise analysis of our algorithm for k rather than just two strings: There cannot be many strings of length almost m . Intuitively, a long string in the returned set corresponds to a leaf in the search tree with few branching nodes among its ancestors, which actually helps reducing the size of the search tree. On the other hand, a short string in the returned set will cause less branchings in our next algorithm. Thus, the following lemma describes the repartition of the number of maximal common subsequences of two strings based on their lengths. Note that we would obtain the same bound if we used the filtering step from Remark 7 (i. e., the same formula applies to the set $\text{MCS}_\ell(\{S, S'\})$).

► **Lemma 8.** *Let ℓ , d , and d' be integers. Let $S, S' \in \Sigma^*$ be strings of lengths $\ell + d$ and $\ell + d'$, respectively, so $\{\delta, \Delta\} = \{d, d'\}$. Let N_i be the number of strings in $\text{xMCS2}(\ell, S, S')$ of length exactly $\ell + d' - i$. Then*

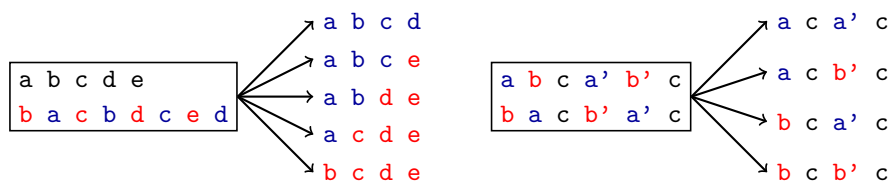
$$\sum_{i=0}^{d'} \frac{N_i}{(d+1)^i} \leq 1.$$

Proof. We prove this property by induction on $|S| + |S'|$.

If $\ell > \min\{|S|, |S'|\}$, then $\text{xMCS2}(\ell, \{S, S'\})$ is empty, and the inequality is valid. If S or S' is a subsequence of the other, then $|\text{xMCS2}(\ell, S, S')| = 1$, so we have $N_i = 1$ for some i and $N_j = 0$ for $j \neq i$. The above inequality is true in this case as well. Note that this includes the cases where S or S' are empty. In the remaining cases, S and S' are not subsequences of each other, so in particular they are not empty. Let $u \cdot T := S$ and $u' \cdot T' := S'$.

If $u = u'$, then N_i is upper-bounded by the number of strings of length $(\ell - 1) + d' - i$ in $\text{xMCS2}(\ell - 1, T, T')$, so we can directly apply the property by induction to get $\sum_{i=0}^{d'} \frac{N_i}{(d+1)^i} \leq 1$.

Otherwise ($u \neq u'$), let N_i^a (resp. N_i^b) be the number of strings of length $\ell + d' - i$ in $\text{xMCS2}(\ell, S, T')$ (resp. $\text{xMCS2}(\ell, T, S')$). We have $N_i \leq N_i^a + N_i^b \leq 2N_i$ (accounting for the fact that a string counted in N_i must be counted once in one of N_i^a, N_i^b , and at most twice in total). Note that $N_0 = 0$ (otherwise, S and S' have a common subsequence of length $\ell + d' = |S'|$, which implies that S' is a subsequence of S). Thus $N_0^a = N_0^b = 0$.



■ **Figure 2** Examples of pairs of strings $\{S, S'\}$ with large $|\text{MCS}_\ell(S, S')|$. Left: A pair with $\delta = 1$, $\Delta = 4$, and $|\text{MCS}_\ell(S, S')| = 5 = 1 + \frac{\Delta}{\delta}$, showing that a dependency on Δ is unavoidable. Right: A pair with 2^δ maximal common subsequences, with $\delta = \Delta = 2$. Proposition 9 is a generalization of both examples that yields strings with $|\text{MCS}_\ell(S, S')| = (\frac{\Delta}{\delta} + 1)^\delta$.

We apply the induction hypothesis first on the pair $\{S, T'\}$. Note that d' decreases by 1 and indices of N_i are shifted by 1, which gives $\sum_{i=0}^{d'-1} N_{i+1}^a / (d+1)^i \leq 1$, so

$$\sum_{i=0}^{d'} \frac{N_i^a}{(d+1)^i} = N_0^a + \frac{1}{d+1} \sum_{i=1}^{d'} \frac{N_i^a}{(d+1)^{i-1}} \leq \frac{1}{d+1}.$$

Then the induction hypothesis on $\{T, S'\}$ (where d decreases by 1) gives $\sum_{i=0}^{d'} N_i^b / d^i \leq 1$, so

$$\begin{aligned} \sum_{i=0}^{d'} \frac{N_i^b}{(d+1)^i} &= N_0^b + \sum_{i=1}^{d'} \frac{N_i^b}{(d+1)^i} \\ &= \frac{d}{d+1} \sum_{i=1}^{d'} \frac{d^{i-1}}{(d+1)^{i-1}} \frac{N_i^b}{d^i} \\ &\leq \frac{d}{d+1} \sum_{i=1}^{d'} \frac{N_i^b}{d^i} \leq \frac{d}{d+1}. \end{aligned}$$

Combining both inequalities yields:

$$\begin{aligned} \sum_{i=0}^{d'} \frac{N_i}{(d+1)^i} &\leq \sum_{i=0}^{d'} \frac{N_i^a}{(d+1)^i} + \sum_{i=0}^{d'} \frac{N_i^b}{(d+1)^i} \\ &\leq \frac{d}{d+1} + \frac{1}{d+1} = 1. \end{aligned} \quad \blacktriangleleft$$

Lemma 8 yields an upper bound of $(\Delta + 1)^\delta$ on the size of $\text{MCS}_\ell(S, S')$ (indeed, using $d = \Delta$ and $d' = \delta$, we have $\frac{|\text{MCS}_\ell(S, S')|}{(\Delta+1)^\delta} \leq \sum_{i=0}^{\delta} \frac{N_i}{(\Delta+1)^\delta} \leq 1$). Examples (see Figure 2 and Proposition 9) indicate that this bound is close to being tight, since there exist instances where $|\text{MCS}_\ell(S, S')| = (\frac{\Delta}{\delta} + 1)^\delta$.

► **Proposition 9.** *For any integers u and v , there exist an ℓ , an alphabet Σ , and strings $S, S' \in \Sigma^*$ of lengths $\ell + v$ and $\ell + uv$, respectively, such that $|\text{MCS}_\ell(S, S')| \geq (u + 1)^v = (\frac{\Delta}{\delta} + 1)^\delta$.*

Proof. Let $\ell = uv$, and $\Sigma = \{x_{i,j} \mid i \in \{1, \dots, v\}, j \in \{1, \dots, u+1\}\}$ be an alphabet of size $(u+1)v$. Using \prod as the concatenation operator, let $S = \prod_{i=1}^v S_i$ and $S' = \prod_{i=1}^v S'_i$ with

$$S_i = \prod_{j=1}^{u+1} x_{i,j} \text{ and}$$

$$S'_i = \prod_{j=1}^u x_{i,j+1} x_{i,j}.$$

Note that the length of S is indeed $|\Sigma| = \ell + v = uv + v = (u+1)v$ and the length of S' is $2uv = \ell + uv$. Since S_i and $S'_{i'}$ only have common characters for $i = i'$, a common subsequence T of S and S' is of the form $T = \prod_{i=1}^v T_i$, where T_i is a common subsequence of S_i and S'_i . Each T_i has length at most u (since S_i is not a subsequence of S'_i , any common subsequence has length at most $|S_i| - 1 = u$). If T has length at least $\ell = uv$, then each T_i has length exactly u . There are precisely $u+1$ such common subsequences for each i (all proper subsequences of S_i are also subsequences of S'_i). Counting all combinations of strings T_i , there are a total of $(u+1)^v$ common subsequences of S and S' of length ℓ , and they are all maximal. So $|\text{MCS}_\ell(S, S')| = (u+1)^v$. \blacktriangleleft

2.4 Extended MCS of k Strings

We now present our algorithm computing an extended MCS for any number k of strings, using xMCS2 as a subroutine, see Algorithm 2. We first give the recurrence relation on MCS on which the algorithm is based.

► Lemma 10. *Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a set of at least two strings and let ℓ be an integer. Let $M' = \text{MCS}_\ell(\{S_1, \dots, S_{k-1}\})$, then*

$$\text{MCS}_\ell(\mathcal{S}) \subseteq \bigcup_{S' \in M'} \text{MCS}_\ell(\{S', S_k\}).$$

Proof. Consider some string $S \in \text{MCS}_\ell(\mathcal{S})$. Then S is, in particular, a common subsequence of $\{S_1, \dots, S_{k-1}\}$ of length at least ℓ , and so $S \in \text{CS}_\ell(\{S_1, \dots, S_{k-1}\})$. By definition of MCS, there exists a string S' in $\text{MCS}_\ell(\{S_1, \dots, S_{k-1}\})$ such that $S \preceq S'$

Since S is a subsequence of both S' and S_k , we have that $S \in \text{CS}_\ell(\{S', S_k\})$. To see that S is also in $\text{MCS}_\ell(\{S', S_k\})$, assume that $S'' \in \text{CS}_\ell(\{S', S_k\})$ and $S \preceq S''$. Then S'' is in $\text{CS}_\ell(\mathcal{S})$ as $S' \in \text{CS}_\ell(\{S_1, \dots, S_{k-1}\})$; and since S is maximal in $\text{CS}_\ell(\mathcal{S})$, we have $S = S''$.

Thus, S is in $\text{MCS}_\ell(\{S', S_k\})$ for some $S' \in \text{MCS}_\ell(\{S_1, \dots, S_{k-1}\})$, which gives the desired inclusion. \blacktriangleleft

► Remark 11. We note that the containment in Lemma 10 may sometimes be strict, as can be seen in the following example with $\ell = 1$. Take $S_1 = abc$ and $S_2 = acb$. Then $\text{MCS}_\ell(\{S_1, S_2\}) = \{ab, ac\}$. Combining strings ab and ac with $S_3 = aab$ yields respectively $\text{MCS}_\ell(\{S_3, ab\}) = \{ab\}$ and $\text{MCS}_\ell(\{S_3, ac\}) = \{a\}$. However, only ab (and not a) is part of $\text{MCS}_\ell(\{S_1, S_2, S_3\})$. As for xMCS2 , xMCSk outputs these extra strings to avoid a costly filtering step without any gain in the worst case.

► Corollary 12. *Given \mathcal{S} and ℓ , Algorithm 2 correctly computes an extended MCS of (\mathcal{S}, ℓ) .*

We now upper-bound the number of strings at any point in the set M of the algorithm. The key point here is that this bound does not depend on k or n . This may seem counter-intuitive, compared to the following upper bound: the algorithm starts with a single string,

■ **Algorithm 2** Compute a bounded-size extended MCS of k strings.

```

1  algorithm xMCSk( $\ell, S_1, \dots, S_k$ )
2  assert  $\forall i: |S_i| \geq |S_1|$ 
3  if  $k = 1$  then
4    if  $|S_1| \geq \ell$  then return  $\{S_1\}$  else return  $\emptyset$ 
5  else
6     $M' \leftarrow \text{xMCSk}(\ell, S_1, \dots, S_{k-1})$ 
7     $M \leftarrow \emptyset$ 
8    for  $S'$  in  $M'$  do
9       $M \leftarrow M \cup \text{xMCS2}(\ell, S_k, S')$ 
10   return  $M$ 

```

and each recursive call may replace any string by up to $2^{\delta+\Delta}$ strings (cf. the complexity of xMCS2). There are k recursive calls, so this would give a bound of $2^{k(\delta+\Delta)}$ strings in total. The key argument here is that whenever a string is replaced, new strings are strictly shorter than the former. Since we only allow for at most δ deletions (starting from a minimal length input string), this gives a bound depending on δ and Δ only. Our more precise analysis in Lemma 13 allows us to shrink this quantity from $2^{O(\Delta\delta)}$ to $2^{O(\log(\Delta)\delta)}$.

► **Lemma 13.** *Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a set of strings with S_1 of minimal length (i. e. $|S_1| = m$), and ℓ be an integer. Then*

$$|\text{xMCSk}(\ell, \mathcal{S})| \leq (\Delta + 1)^\delta.$$

Proof. We prove the following claim by induction on k : *Let $d \geq \Delta$ and let N_i be the number of length- $(\ell + \delta - i)$ strings in $\text{xMCSk}(\ell, \mathcal{S})$. Then*

$$\sum_{i=0}^{\delta} \frac{N_i}{(d+1)^i} \leq 1.$$

The lemma's statement follows easily from this claim for $d = \Delta$:

$$\frac{|\text{xMCSk}(\ell, \mathcal{S})|}{(\Delta + 1)^\delta} = \sum_{i=0}^{\delta} \frac{N_i}{(\Delta + 1)^\delta} \leq \sum_{i=0}^{\delta} \frac{N_i}{(\Delta + 1)^i} \leq 1$$

For the inductive proof of the claim, we start with $k = 1$. Then we have a single string in $\text{xMCSk}(\ell, \mathcal{S})$, namely, S_1 , so $N_i = 1$ for exactly one value of i and 0 otherwise, and the formula is satisfied.

For $k \geq 2$, we have $M' = \text{xMCSk}(\ell, \{S_1, \dots, S_{k-1}\})$. Consider the for-loop in lines 8–9. We assume that when we iterate with $S' \in M'$, the string S' is immediately removed from M' . At any point of the loop, we write σ for the quantity $\sum_{i=0}^{\delta} \frac{N_i}{(d+1)^i}$, where N_i denotes the number of strings of length $\ell + \delta - i$ in $M' \cup M$. Note that by induction, before the first iteration of the loop, $\sigma \leq 1$ as $\delta(\{S_1, \dots, S_{k-1}\}, \ell) = \delta$ since $|S_1|$ is minimal, and $d \geq \Delta \geq \Delta(\{S_1, \dots, S_{k-1}\}, \ell)$.

We show that σ may only decrease after each iteration. Consider the iteration for string S' , let $d' = |S'| - \ell$ and $j = \delta - d'$ (since S' is a subsequence of S_1 , it has length at most $\ell + \delta$, so $d' \leq \delta$ and $j \geq 0$).

First, removing S' from M' makes N_j decrease by one, so σ decreases by $\frac{1}{(d+1)^j}$. Then, we add strings from $\text{xMCS2}(\{S_k, S'\})$ to M . Write D_i for the number of such strings of length $\ell + d' - i$. Note that for each pair (i, j) with $j \leq i \leq \delta$, N_i increases by D_{i-j} . By

Lemma 8, $\sum_{i=0}^{d'} \frac{D_i}{(|S_k| - \ell + 1)^i} \leq 1$. Since $d \geq \Delta \geq |S_k| - \ell$, $\sum_{i=0}^{d'} \frac{D_i}{(d+1)^i} \leq 1$. Then σ increases by $\sum_{i=0}^{d'} \frac{D_i}{(d+1)^{j+i}} = \frac{1}{(d+1)^j} \sum_{i=j}^{\delta} \frac{D_{i-j}}{(d+1)^i} \leq \frac{1}{(d+1)^j}$. Overall, σ may not increase between two steps, so at the end of the for-loop, $\sum_{i=0}^{\delta} \frac{N_i}{(d+1)^i} \leq 1$. ◀

We can now conclude with the proof of Theorem 1.

Proof of Theorem 1. Given \mathcal{S} and ℓ , Algorithm 2 computes an extended MCS of (\mathcal{S}, ℓ) (Corollary 12) of size at most $(\Delta + 1)^\delta$ (Lemma 13). Its running time is bounded by k times the complexity of the for-loop, which requires at most $(\Delta + 1)^\delta$ calls to `xMCS2`, each taking time $O(2^{\delta+\Delta}n)$ (Lemma 6). This gives the overall complexity of $O(2^{\delta+\Delta}(\Delta + 1)^\delta kn)$. ◀

3 Conclusion

Regarding LCS, we have proposed an FPT algorithm for the parameter Δ , i. e., the maximum number of deletions per input string. We leave open whether the complexity can be improved, e. g. using only parameter δ , i. e., the smallest number of deletions per input strings. In other words, the goal is to find an LCS of size ℓ in a set of strings where *one* input string has size at most $\ell + \delta$ (and other strings might be arbitrarily long). Such an algorithm may not compute and store explicitly each MCS, since the number of maximal common subsequences, even with only two input strings, can grow in $(1 + \frac{\Delta}{\delta})^\delta$. Also, it is open whether any improvement can be obtained when the alphabet size is bounded, or when each character has a bounded number of occurrences in each string.

A longest common subsequence can be interpreted as a string that can be obtained with a minimal number of edits (deletions only) from all input strings. Generalizing this notion to other edits (insertions and substitutions) yields the NP-hard `CENTER STRING` problem² [15, 12], which is highly related to the problem of `MULTIPLE SEQUENCE ALIGNMENT` in bioinformatics. In future work, one may try to extend our approach in order to design an FPT algorithm for `CENTER STRING`, parameterized by the maximum distance to the input strings. Allowing for a small number of outliers (input strings that are discarded in order to obtain a better solution [5]) would also yield a useful extension of our algorithm.

Finally, a more practical objective towards algorithm engineering would be to design an efficient data structure to store all maximal common subsequences of any number of strings, thus reducing the memory footprint of our algorithm.

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 59–78. IEEE Computer Society, 2015.
- 2 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000*, pages 39–48. IEEE Computer Society, 2000.
- 3 Magnus Bordewich, Britta Dorn, Simone Linz, and Rolf Niedermeier. Algorithms and complexity in phylogenetics (dagstuhl seminar 19443). *Dagstuhl Reports*, 9(10):134–151, 2019.
- 4 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: beyond worst case. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 1601–1620. SIAM, 2020.

² More specifically, given k strings, find a string that minimizes the maximum edit distance to the k strings.

- 5 Christina Boucher and Bin Ma. Closest string with outliers. *BMC Bioinformatics*, 12(1):1–7, 2011.
- 6 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1216–1235. SIAM, 2018.
- 7 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, 114, 2014.
- 8 Laurent Bulteau and Mathias Weller. Parameterized algorithms in bioinformatics: An overview. *Algorithms*, 12(12):256, 2019.
- 9 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Polynomial-delay enumeration of maximal common subsequences. In *26th International Symposium on String Processing and Information Retrieval, SPIRE 2019*, pages 189–202. Springer, 2019.
- 10 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1999.
- 11 MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the \sqrt{n} barrier. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1181–1200. SIAM, 2019.
- 12 Gary Hoppenworth, Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. The fine-grained complexity of median and center string problems under edit distance. In *28th Annual European Symposium on Algorithms, ESA 2020*, volume 173 of *LIPICs*, pages 61:1–61:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 13 Robert W. Irving and Campbell Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *Third Annual Symposium on Combinatorial Pattern Matching, CPM 1992*, volume 644 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 1992.
- 14 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.
- 15 François Nicolas and Eric Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *Journal of Discrete Algorithms*, 3(2):390–415, 2005.
- 16 Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003.
- 17 Yoshifumi Sakai. Maximal Common Subsequence Algorithms. In *29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

Beyond the Longest Letter-Duplicated Subsequence Problem

Wenfeng Lai ✉

College of Computer Science and Technology, Shandong University, Qingdao, China

Adiesha Liyanage ✉

Gianforte School of Computing, Montana State University, Bozeman, MT, USA

Binhai Zhu ✉ 🏠

Gianforte School of Computing, Montana State University, Bozeman, MT, USA

Peng Zou ✉

Gianforte School of Computing, Montana State University, Bozeman, MT, USA

Abstract

Motivated by computing duplication patterns in sequences, a new fundamental problem called the longest letter-duplicated subsequence (LLDS) is proposed. Given a sequence S of length n , a letter-duplicated subsequence is a subsequence of S in the form of $x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ with $x_i \in \Sigma$, $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. A linear time algorithm for computing the longest letter-duplicated subsequence (LLDS) of S can be easily obtained. In this paper, we focus on two variants of this problem. We first consider the constrained version when Σ is unbounded, each letter appears in S at least 6 times and all the letters in Σ must appear in the solution. We show that the problem is NP-hard (a further twist indicates that the problem does not admit any polynomial time approximation). The reduction is from possibly the simplest version of SAT that is NP-complete, $(\leq 2, 1, \leq 3)$ -SAT, where each variable appears at most twice positively and exact once negatively, and each clause contains at most three literals and some clauses must contain exactly two literals. (We hope that this technique will serve as a general tool to help us proving the NP-hardness for some more tricky sequence problems involving only one sequence – much harder than with at least two input sequences, which we apply successfully at the end of the paper on some extra variations of the LLDS problem.) We then show that when each letter appears in S at most 3 times, then the problem admits a factor $1.5 - O(\frac{1}{n})$ approximation. Finally, we consider the weighted version, where the weight of a block $x_i^{d_i}$ ($d_i \geq 2$) could be any positive function which might not grow with d_i . We give a non-trivial $O(n^2)$ time dynamic programming algorithm for this version, i.e., computing an LD-subsequence of S whose weight is maximized.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Segmental duplications, Tandem duplications, Longest common subsequence, NP-completeness, Dynamic programming

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.7

Related Version *Full Version*: <https://arxiv.org/abs/2112.05725>

Funding This research is partially supported by NNSF of China under project 61872427, 61732009 and 61628207.

1 Introduction

In biology, duplication is an important part of evolution. There are two kinds of duplications: arbitrary segmental duplications (i.e., select a segment and paste it somewhere else) and tandem duplications (which is in the form of $X \rightarrow XX$, where X is any segment of the input sequence). It is known that the former duplications occur frequently in cancer genomes [16, 12, 3]. On the other hand, the latter are common under different scenarios, for



© Wenfeng Lai, Adiesha Liyanage, Binhai Zhu, and Peng Zou;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 7; pp. 7:1–7:12

Leibniz International Proceedings in Informatics



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

example, it is known that the tandem duplication of 3 nucleotides CAG is closely related to the Huntington disease [11]. In addition, tandem duplications can occur at the genome level (acrossing different genes) for certain types of cancer [13]. In fact, as early as in 1980, Szostak and Wu provided evidence that gene duplication is the main driving force behind evolution, and the majority of duplications are tandem [17]. Consequently, it was not a surprise that in the first sequenced human genome around 3% of the genetic contents are in the form of tandem repeats [9].

Independently, tandem duplications were also studied in *copying systems* [5]; as well as in formal languages [1, 4, 19]. In 2004, Leupold et al. posed a fundamental question regarding tandem duplications: what is the complexity to compute the minimum tandem duplication distance between two sequences A and B (i.e., the minimum number of tandem duplications to convert A to B). In 2020, Lafond et al. answered this open question by proving that this problem is NP-hard for an unbounded alphabet [7]. In fact, Lafond et al. proved later that the problem is NP-hard even if $|\Sigma| \geq 4$ by encoding each letter in the unbounded alphabet proof with a square-free string over a new alphabet of size 4 (modified from Leech's construction [10]), which covers the case most relevant with biology, i.e., when $\Sigma = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$ [8]. Independently, Cicalese and Pilati showed that the problem is NP-hard for $|\Sigma| = 5$ using a different encoding method [2].

Motivated by the above applications (especially when some mutations occur after the duplications), some new problems related to duplications are proposed and studied in this paper. Given a sequence S of length n , a letter-duplicated subsequence (LDS) of S is a subsequence of S in the form $x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ with $x_i \in \Sigma$, where $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. (Each $x_i^{d_i}$ is called an LD-block.) Naturally, the problem of computing the longest letter-duplicated subsequence (LLDS) of S can be defined, and a simple linear time algorithm can be obtained. (We remark that recently a similar problem called *longest run subsequence* was studied [15], it differs from our problem in that each letter appears consecutively at most once in the solution and does not have to be repeated, and the goal is the same, i.e., the length of the subsequence is to be maximized.)

In this paper, we focus on some important variants around the LLDS problem, focusing on the constrained and weighted cases. The constraint is to demand that all letters in Σ appear in a resulting LDS, which simulates that in a genome with duplicated genes, how to compute the maximum duplicated pattern while including all the genes. Then we have two problems: feasibility testing (FT for short, which decides whether an LDS of S containing all letters in Σ exists) and the problem of maximizing the length of a resulting LDS where all letters in the alphabet appear, which we call LLDS+. It turns out that the status of these two problems change quite a bit when d , the maximum number a letter can appear in S , varies. We denote the corresponding problems as $FT(d)$ and $LLDS+(d)$ respectively. Let $|S| = n$, we summarize our main results in this paper as follows:

1. We show that when $d \geq 6$, both $FT(d)$ and (the decision version of) $LLDS+(d)$ are NP-complete, which implies that $LLDS+(d)$ does not have a polynomial-time approximation algorithm when $d \geq 6$.
2. We show that when $d = 3$, $FT(d)$ is decidable in $O(n^2)$ time, which implies that $LLDS+(3)$ admits a factor-1.5 approximation. With an increasing running time, we could improve the factor to $1.5 - O(\frac{1}{n})$.
3. When a weight of an LD-block is any positive function (i.e., it does not even have to grow with its length), we present a non-trivial $O(n^2)$ time dynamic programming solution for this Weighted-LDS problem.

Note that the parameter d , i.e., the maximum duplication number, is of practical interest in bioinformatics, since in many genomes duplication is a rare event and the number of duplicates is usually a small constant. For example, it is known that plants have undergone up to three rounds of whole genome duplications, resulting in a number of duplicates bounded by 8 [20].

At the end of paper, we will briefly mention two extra variations of the LLDS problem, where in the solution, i.e., a subsequence of S in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$, each x_i is either a substring or a subsequence of S . The latter is remotely related to computing the longest *square* subsequence of an input sequence S , for which Kosowski gave an $O(n^2)$ time algorithm [6]. Then, what Kosowski considered is the more restricted version of the latter, i.e., $x_1^{d_1}x_2^{d_1}$, with $x_1 = x_2$ and $d_1 = d_2 = 1$.

This paper is organized as follows. In Section 2 we give necessary definitions. In Section 3 we focus on showing that the LLDS+ and FT problems are NP-complete when $d \geq 6$ and some positive results when $d = 3$. In Section 4 we give polynomial-time algorithms for Weighted-LDS. We conclude the paper in Section 5.

2 Preliminaries

Let \mathbb{N} be the set of natural numbers. For $q \in \mathbb{N}$, we use $[q]$ to represent the set $\{1, 2, \dots, q\}$. Throughout this paper, a sequence S is over a finite alphabet Σ . We use $S[i]$ to denote the i -th letter in S and $S[i..j]$ to denote the substring of S starting and ending with indices i and j respectively. (Sometimes we also use $(S[i], S[j])$ as an interval representing the substring $S[i..j]$.) With the standard run-length representation, S can be represented as $y_1^{a_1}y_2^{a_2}\cdots y_q^{a_q}$, with $y_i \in \Sigma, y_j \neq y_{j+1}$ and $a_j \geq 1$, for $i \in [q], j \in [q-1]$. If a letter x appears multiple times in S , we could use $x^{(i)}$ to denote the i -th copy of it (reading from left to right). Finally, a *subsequence* of S is a string obtained by deleting some letters in S .

2.1 The LLDS Problem

A subsequence S' of S is a letter-duplicated subsequence (LDS) of S if it is in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$, with $x_i \in \Sigma, x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k], j \in [k-1]$. We call each $x_i^{d_i}$ in S' a *letter-duplicated block* (LD-block, for short). For instance, let $S = abcacabcb$, then $S_1 = aaabb$, $S_2 = cbbb$ and $S_3 = ccc$ are all letter-duplicated subsequences of S , where aaa and bb in S_1 , cc and bb in S_2 , and ccc in S_3 all form the corresponding LD-blocks. Certainly, we are interested in the longest ones – which gives us the longest letter-duplicated subsequence (LLDS) problem.

As a warm-up, we solve this problem by dynamic programming. We first have the following observation.

► **Observation 1.** *Suppose that there is an optimal LLDS solution for a given sequence S of length n , in the form of $x_1^{d_1}x_2^{d_2}\cdots x_k^{d_k}$. Then it is possible to decompose it into a generalized LD-subsequence $y_1^{e_1}y_2^{e_2}\cdots y_p^{e_p}$, where*

- $2 \leq e_i \leq 3$, for $i \in [p]$,
- $p \geq k$,
- y_j does not have to be different from y_{j+1} , for $j \in [p-1]$.

The proof is straightforward: For any natural number $\ell \geq 2$, we can decompose it as $\ell = \ell_1 + \ell_2 + \dots + \ell_z \geq 3$, such that $2 \leq \ell_j \leq 3$ for $1 \leq j \leq z$. Consequently, for every $d_i > 3$, we could decompose it into a sum of 2's and 3's. Then, clearly, given a generalized LD-subsequence, we could easily obtain the corresponding LD-subsequence by combining $y_i^{e_i}y_{i+1}^{e_{i+1}}$ when $y_i = y_{i+1}$.

7:4 The Longest Letter-Duplicated Subsequence Problem

We now design a dynamic programming algorithm for LLDS. Let $L(i)$ be the length of the optimal LLDS solution for $S[1..i]$. The recurrence for $L(i)$ is as follows.

$$\begin{aligned}
 L(0) &= 0, \\
 L(1) &= 0, \\
 L(i) &= \max \begin{cases} L(i-x-1) + 2 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-x) + 1 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-1) & \text{otherwise.} \end{cases}
 \end{aligned}$$

Note that the step involving $L(i-x) + 1$ is essentially a way to handle a generalized LD-subsequence of length 3 (by keeping $S[i-x]$ for the next level computation) and cannot be omitted following the above observation. For instance, if $S = dabcdd$ then without that step we would miss the optimal solution ddd .

The value of the optimal LLDS solution for S can be found in $L(n)$. For the running time, for each $S[x]$ we just need to scan S to find the closest $S[i]$ such that $S[x] = S[i]$. With this information, the table L can be filled in linear time. With a simple augmentation, the actual sequence corresponding to $L(n)$ can also be found in linear time. Hence LLDS can be solved in $O(n)$ time.

2.2 The Variants of LLDS

In this paper, we focus on the following variations of the LLDS problem.

► **Definition 2** (Constrained Longest Letter-Duplicated Subsequence (*LLDS+* for short)).

Input: A sequence S with length n over an alphabet Σ and an integer ℓ .

Question: Does S contain a letter-duplicated subsequence S' with length at least ℓ such that all letters in Σ appear in S' ?

► **Definition 3** (Feasibility Testing (*FT* for short)).

Input: A sequence S with length n over an alphabet Σ .

Question: Does S contain a letter-duplicated subsequence S'' such that all letters in Σ appear in S'' ?

For *LLDS+* we are really interested in the optimization version, i.e., to maximize ℓ . Note that, though looking similar, *FT* and the decision version of *LLDS+* are different: if there is no feasible solution for *FT*, certainly there is no solution for *LLDS+*; but even if there is a feasible solution for *FT*, computing an optimal solution for *LLDS+* could still be non-trivial.

Finally, let d be the maximum number of times a letter in Σ appears in S . Then, we can represent the corresponding versions for *LLDS+* and *FT* as *LLDS+(d)* and *FT(d)* respectively.

It turns out that (the decision version of) *LLDS+(d)* and *FT(d)* are both NP-complete when $d \geq 6$, while when $d = 3$ the status varies: *FT(3)* can be decided in $O(n^2)$ time, which immediately implies that *LLDS+(3)* has a factor-1.5 approximation. (If we are willing to increase the running time – still polynomial but higher than $O(n^2)$, with some simple twist we could improve the approximation factor for *LLDS+(3)* to $1.5 - O(\frac{1}{n})$.) We present the details in the next section. In Section 4, we will consider an extra version of LLDS, Weighted-LDS, where the weight of an LD-block is an arbitrary positive function.

3 Hardness with the full-appearance constraint

3.1 Hardness for LLDS+(d) and FT(d) when $d \geq 6$

We first try to prove the NP-completeness of the (decision version of) LLDS+(d), when $d \geq 6$. In fact, we need a very special version of SAT, which is possibly the simplest version of SAT remaining NP-complete.

Given a 3SAT formula ϕ , which is a conjunction of m disjunctive clauses (over n variable x_i 's), each clause F_j containing exactly 3 literals (i.e., in the form of x_i or \bar{x}_i), the problem is to find whether there is a satisfiable truth assignment for ϕ .

► **Definition 4.** ($\leq 2, 1, \leq 3$)-SAT: *this is a special case of SAT where each variable x_i appears at most twice and \bar{x}_i appears exactly once in ϕ ; moreover, each clause contains either two or three literals (which will be called 2-clause and 3-clause henceforth).*

► **Lemma 5.** ($\leq 2, 1, \leq 3$)-SAT is NP-complete.

Proof. We modify the proof by Tovey [18]. Given a 3SAT formula ϕ , without loss of generality, assume that each variable x_i and its complement \bar{x}_i appears in (different clauses of) ϕ . We convert ϕ to ϕ' in the form of ($\leq 2, 1 \leq 3$)-SAT as follows.

- if both x_i and \bar{x}_i appears once in ϕ , do nothing.
- if x_i appears twice and \bar{x}_i appears once in ϕ , do nothing.
- if \bar{x}_i appears twice and x_i appears once in ϕ , replace \bar{x}_i with a new variable z and replace x_i by \bar{z} .
- Otherwise, if the total number of literals of x_i (i.e., x_i and \bar{x}_i) is $k \geq 4$ then introduce k variables $y_{i,1}, y_{i,2}, \dots, y_{i,k}$ replacing the k literals of x_i respectively. Moreover, let $z_{i,j}$ be $y_{i,j}$ if the j -th literal of x_i is x_i and let $z_{i,j}$ be $\bar{y}_{i,j}$ if the j -th literal of x_i is \bar{x}_i . Finally, add k 2-clauses as $(z_{i,j} \vee \bar{z}_{i,j+1})$ for $j = 1..k - 1$ and $(z_{i,k} \vee \bar{z}_{i,1})$. (Note that it always holds that $\bar{\bar{z}} = z$.)

Following [18], when $k \geq 4$, the 2-clauses added will force all $z_{i,j}$'s to have all **True** values or all **False** values. (The only difference between our construction and Tovey's is that all literals appearing at least 4 times in the original clauses in ϕ are replaced by positive variables in the form of $y_{i,j}$'s; the negated literal $\bar{y}_{i,j}$ could only occur in the newly created 2-clauses – exactly once for each $y_{i,j}$. On the other hand, each $y_{i,j}$ occur twice – once in the original 3-clauses of ϕ and once in the newly created 2-clauses.) It is obvious to see that ϕ is satisfiable if and only if ϕ' is satisfiable. The transformation obviously takes $O(|\phi|)$ time. Hence the lemma is proven. ◀

We remark that ($\leq 2, 1 \leq 3$)-SAT, while seemingly similar to SAT3W (each clause has at most 3 literals and each clause has at most one negated variable [14]), is in fact different from it. (Following the Dichotomy Theorem for SAT by Schaefer [14], SAT3W is in P.) The difference is that in ϕ' we could even have a clause containing 3 negated variables.

Now let ϕ be an instance of ($\leq 2, 1, \leq 3$)-SAT where either both x_i and \bar{x}_i appear once in ϕ (we call such an x_i a *(1,1)-variable*), or x_i appears twice and \bar{x}_i appears once in ϕ (we call such an x_i a *(2,1)-variable*), for $i = 1..n$. (Note that the case when x_i appears once and \bar{x}_i does not appear in ϕ at all, or vice versa, can be easily handled. Hence we can assume that we do not have these kind of “single-appearance” literals in ϕ .) Without loss of generality, we assume $\phi = F_1 \wedge F_2 \wedge \dots \wedge F_m$ and there are n variables x_1, x_2, \dots, x_n ; moreover, we assume that F_j cannot contain x_i and \bar{x}_i at the same time. Given F_j we say $F_j F_j$ forms a *2-duplicated clause-string*.

7:6 The Longest Letter-Duplicated Subsequence Problem

Given a $(1,1)$ -sequence $T = ACCA$ over $\{A, C\}$, where A and C both appear twice, it is easy to see that the maximal (longest) LD-subsequences of T are AA or CC . Similarly, given a $(2,1)$ -sequence $T = ACABCB$ over $\{A, B, C\}$, where A, B and C all appear twice, it is easy to verify that the maximal LD-subsequences of T are $AABB$ or CC .

For each $(1,1)$ -variable x_i , i.e., both x_i and \bar{x}_i appear once in ϕ , say x_i in F_j and \bar{x}_i in F_k , we define L_i as a $(1,1)$ -sequence: $F_j F_k F_k F_j$. For each $(2,1)$ -variable x_i , i.e., x_i appears twice and \bar{x}_i appears once in ϕ , say x_i in F_j and F_k , and \bar{x}_i in F_ℓ , we define L_i as a $(2,1)$ -sequence: $F_j F_\ell F_j F_k F_\ell F_k$.

Now we proceed to construct the sequence S from an $(\leq 2, 1, \leq 3)$ -SAT instance ϕ .

$$S = g_1 g_1 L_1 g_2 g_2 \cdots g_i g_i L_i \cdots g_{n-1} g_{n-1} L_{n-1} g_n g_n L_n g_{n+1} g_{n+1}.$$

We claim the following: ϕ is satisfiable if and only if LLDS+ has a solution of length at least $2(n+1) + 4K_1 + 2K_2 + 2J$, where K_1, K_2 are the number of $(2,1)$ -variables in ϕ which are assigned **True** and **False** respectively and J is the number of $(1,1)$ -variables in ϕ .

Proof.

“Only-if part”. Suppose that ϕ is satisfiable. If a $(1,1)$ -variable x_i is assigned **True**, to have a solution for LLDS+, in L_i we select the 2-duplicated clause-string $F_j F_j$; likewise, if x_i is assigned **False** we select $F_k F_k$ instead. Similarly, if a $(2,1)$ -variable x_i is assigned **True**, to have a solution for LLDS+, in L_i we select two 2-duplicated clause-strings $F_j F_j F_k F_k$; likewise, if x_i is assigned **False** we select $F_\ell F_\ell$. Since $g_i g_i$ only occurs once in S and T , we must include them in the solution. Clearly we have a solution for LLDS+ with length $2(n+1) + 4K_1 + 2K_2 + 2J$.

“If part”. If LLDS+ has a solution of length at least $2(n+1) + 4K_1 + 2K_2 + 2J$, by definition, it must contain all $g_i g_i$'s. To find the truth assignment, we look at the contents between $g_i g_i$ and $g_{i+1} g_{i+1}$ in the solution as well as in S (i.e., L_i). If x_i is a $(1,1)$ -variable, $L_i = F_j F_k F_k F_j$ and in the solution $F_j F_j$ is kept then we assign $x_i \leftarrow \mathbf{True}$; otherwise, we assign $x_i \leftarrow \mathbf{False}$. If x_i is a $(2,1)$ -variable, $L_i = F_j F_\ell F_j F_k F_\ell F_k$ and in the solution either $F_j F_j F_k F_k$, $F_j F_j$ or $F_k F_k$ is kept then we assign $x_i \leftarrow \mathbf{True}$. (When $F_j F_j$ or $F_k F_k$ is kept, then the LLDS+ solution could be longer by augmenting this sub-solution to $F_j F_j F_k F_k$.) If in the solution $F_\ell F_\ell$ is kept instead then we assign $x_i \leftarrow \mathbf{False}$. Since all clauses must appear in a solution of LLDS+, clearly ϕ is satisfied. \blacktriangleleft

We comment that $2(n+1) + 4K_1 + 2K_2 + 2J = 2(n+1) + 2K_1 + 2n = 4n + 2 + 2K_1$, as $K_1 + K_2 + J = n$. (Note that K_1 only represents a part of the truth assignment for ϕ and it could be general, i.e., K_1 could be $\Omega(n)$.) But the former makes our arguments more clear. This reduction obviously takes $O(m+n)$ time. Note that each 3-clause F_j appears 6 times in S and each 2-clause F_ℓ appears 4 times in S respectively, while each $g_k, k \in [n+1]$, appears twice in S . Since we could arbitrarily add an LD-block w^j , with $u \notin \Sigma$ and $j \geq 6$, at the end of S , we have the following theorem.

► **Theorem 6.** *The decision version of LLDS+(d) is NP-complete for $d \geq 6$.*

We next present an example for this proof.

► **Example.** Let $\phi = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5) \wedge (\bar{x}_4 \vee \bar{x}_5)$. Then

$$\begin{aligned} S = & g_1 g_1 F_1 F_2 F_1 F_4 F_2 F_4 \cdot g_2 g_2 F_1 F_2 F_1 F_3 F_2 F_3 \cdot g_3 g_3 F_1 F_3 F_1 F_2 F_3 F_2 \\ & \cdot g_4 g_4 F_3 F_5 F_3 F_4 F_5 F_4 \cdot g_5 g_5 F_4 F_5 F_5 F_4 \cdot g_6 g_6, \end{aligned}$$

Corresponding to the truth assignment, $x_1, x_4 = \text{True}$ and $x_2, x_3, x_5 = \text{False}$, we have

$$S' = g_1g_1F_1F_1F_4F_4 \cdot g_2g_2F_2F_2 \cdot g_3g_3F_3F_3 \cdot g_4g_4F_3F_3F_4F_4 \cdot g_5g_5F_5F_5 \cdot g_6g_6,$$

which is of length $2(5 + 1) + 4 \times K_1 + 2 \times K_2 + 2 \times 1 = 12 + 4 \times 2 + 2 \times 2 + 2 = 26$.

The above theorem implies the following corollary.

► **Corollary 7.** *FT(d) is NP-complete for $d \geq 6$.*

Proof. The reduction remains the same. We just need to augment the proof in the reverse direction. Suppose there is a feasible solution S'' for S for the feasibility testing problem. Again, all $g_i g_i$'s must be in S'' . We now look at the contents between $g_i g_i$ and $g_{i+1} g_{i+1}$ in S (i.e., L_i) and S'' . Corresponding to L_i , if in S'' we have an empty string between $g_i g_i$ and $g_{i+1} g_{i+1}$, then we can assign x_i either as **True** or **False**. If $L_i = F_j F_k F_k F_j$, i.e., x_i is a (1,1)-variable, and $F_j F_j$ is kept in S'' then we assign $x_i \leftarrow \text{True}$; otherwise, we assign $x_i \leftarrow \text{False}$. If $L_i = F_j F_\ell F_j F_k F_\ell F_k$, i.e., x_i is a (2,1)-variable, and either $F_j F_j F_k F_k$, $F_j F_j$ or $F_k F_k$ is kept in S'' then we assign $x_i \leftarrow \text{True}$. If in the solution $F_\ell F_\ell$ is kept instead then we assign $x_i \leftarrow \text{False}$. By definition, all clauses must appear in S'' (solution of FT), clearly ϕ is satisfied. It is clear that FT belongs to NP as a solution can be easily checked in polynomial time. ◀

The above corollary essentially implies that the optimization version of $LLDS+(d)$, $d \geq 6$, does not admit any polynomial-time approximation algorithm (regardless of the approximation factor), since any such approximation would have to return a feasible solution. A natural direction to approach $LLDS+$ is to design a bicriteria approximation for $LLDS+$, where a factor- (α, β) bicriteria approximation algorithm is a polynomial-time algorithm which returns a solution of length at least OPT/α and includes at least N/β letters, where $N = |\Sigma|$ and OPT is the optimal solution value of $LLDS+$. We show that obtaining a bicriteria approximation algorithm for $LLDS+$ is no easier than approximating $LLDS+$ itself.

► **Corollary 8.** *If $LLDS+(d)$, $d \geq 6$, admitted a factor- $(\alpha, N^{1-\epsilon})$ bicriteria approximation for any $\epsilon < 1$, then $LLDS+(d)$, $d \geq 6$, would also admit a factor- α approximation, where N is the alphabet size.*

Proof. Suppose that a factor- $(\alpha, N^{1-\epsilon})$ bicriteria approximation algorithm \mathcal{A} exists. We construct an instance S^* for $LLDS+(6)$ as follows. (Recall that S is the sequence we constructed from an $(\leq 2, 1 \leq 3)$ -SAT instance ϕ in the proof of Theorem 1.) In addition to $\{F_i | i = 1..m\} \cup \{g_j | j = 1..n+1\}$ in the alphabet, we use a set of integers $\{1, 2, \dots, (m+n+1)^x - (m+n+1)\}$, where x is some integer to be determined. Hence,

$$\Sigma = \{F_i | i = 1..m\} \cup \{g_j | j = 1..n+1\} \cup \{1, 2, \dots, (m+n+1)^x - (m+n+1)\}.$$

We now construct S^* as

$$S^* = 1 \cdot 2 \cdots ((m+n+1)^x - (m+n+1)) \cdot S \cdot ((m+n+1)^x - (m+n+1)) \cdot ((m+n+1)^x - (m+n+1) - 1) \cdots 2 \cdot 1.$$

Clearly, any bicriteria approximation for S^* would return an approximate solution for S as including any number in $\{1, 2, \dots, (m+n+1)^x - (m+n+1)\}$ would result in a solution of size only 2.

7:8 The Longest Letter-Duplicated Subsequence Problem

Notice that we have $N = m + (n + 1) + (m + n + 1)^x - (m + n + 1) = (m + n + 1)^x$. In this case, the fraction of letters in Σ that is used to form such an approximate solution satisfies

$$\frac{m + (n + 1)}{(m + n + 1)^x} \leq \frac{1}{N^{1-\epsilon}},$$

which means it suffices to choose $x \geq \lceil 2 - \epsilon \rceil = 2$. ◀

3.2 Solving the Feasibility Testing Version for $d = 3$

For the Feasibility Testing Version, as mentioned earlier, Corollary 1 implies that the problem is NP-complete when $d \geq 6$. We next show that if $d = 3$, then the problem can be decided in polynomial time.

► **Lemma 9.** *Given a string S over Σ such that each letter in S appears at most 3 times, if a feasible solution for $FT(3)$ contains a 3-block then there is a feasible solution for $FT(3)$ which only uses 2-blocks.*

Proof. Suppose that $S = \dots a^{(1)} \dots a^{(2)} \dots a^{(3)} \dots$, and $a^{(1)}a^{(2)}a^{(3)}$ is a 3-block in a feasible solution for $FT(3)$. (Recall that the superscript only indicates the appearance order of letter a .) Then we could replace $a^{(1)}a^{(2)}a^{(3)}$ by either $a^{(1)}a^{(2)}$ or $a^{(2)}a^{(3)}$. The resulting solution is still a feasible solution for $FT(3)$. ◀

Lemma 2 implies that the $FT(3)$ problem can be solved using 2-SAT. For each letter a , we denote the interval $(a^{(1)}, a^{(2)})$ as a variable v_a , and we denote $(a^{(2)}, a^{(3)})$ as \bar{v}_a . (Clearly one cannot select $a^{(1)}a^{(2)}$ and $a^{(2)}a^{(3)}$ as 2-blocks at the same time.) Then, if another interval $(b^{(1)}, b^{(2)})$ overlaps the interval $(a^{(1)}, a^{(2)})$, we have a 2-SAT clause $\overline{v_a} \wedge v_b = (\bar{v}_a \vee \bar{v}_b)$. Forming a 2-SAT instance ϕ'' for all such overlapping intervals and it is clear that we can decide whether ϕ'' is satisfiable in $O(n^2)$ time (as we could have $O(n^2)$ pairs of overlapping intervals).

► **Theorem 10.** *Let S be a string of length n . Whether $FT(3)$ has a solution can be decided in $O(n^2)$ time.*

The theorem immediately implies that $LLDS+(3)$ has a factor-1.5 approximation as any feasible solution for $FT(3)$ would be a factor-1.5 approximation for $LLDS+(3)$. In the following, we extend this trivial observation to have a factor- $(1.5 - O(\frac{1}{n}))$ approximation for $LLDS+(3)$.

► **Corollary 11.** *Let S be a string of length n such that each letter appears at most 3 times in S . Then $LLDS+(3)$ admits a polynomial-time approximation algorithm with a factor of $1.5 - O(\frac{1}{n})$ if a feasible solution exists.*

Proof. First fix some constant (positive integer) D ($D < |\Sigma|$). Then for $t = 1$ to D , we enumerate all the sets which contains letters appearing exactly 3 times in S . For a fixed t , let such a set be $F_t = \{a_1, a_2, \dots, a_t\}$. We put the 3-blocks $a_i^{(1)}a_i^{(2)}a_i^{(3)}$, $i = 1..t$, in the solution. (If two such 3-blocks overlap, then we immediately stop to try a different set F'_t ; and if all valid sets of size t have been tried, we increment t to $t + 1$.) The substrings of S , between $a_i^{(1)}$ and $a_i^{(2)}$, and $a_i^{(2)}$ and $a_i^{(3)}$, will then be deleted. Finally, for the remaining letters we use 2-SAT to test whether all together, with the 3-blocks, they form a feasible solution (note that $a_i^{(1)}a_i^{(2)}a_i^{(3)}$ will serve as an obstacle and no valid interval for 2-SAT should contain it), this can be checked in $O(n^2)$ time following Theorem 2. Clearly, with this algorithm, either we compute the optimal solution with at most D 3-blocks, or we obtain an approximate solution of value $2|\Sigma| + D$. Since OPT is at most $3|\Sigma|$, the approximation factor is

$$\frac{3|\Sigma|}{2|\Sigma| + D} = 1.5 - O\left(\frac{1}{|\Sigma|}\right),$$

which is $1.5 - O(\frac{1}{n})$, because $|\Sigma|$ is at least $\lceil n/3 \rceil$. The running time of the algorithm is $O\left(\frac{|\Sigma|}{D} \cdot O(n^2)\right) = O(n^{D+2})$, which is polynomial as long as D is a constant. ◀

In the next section, we show that if the LD-blocks are arbitrarily positively weighted, then the problem can be solved in $O(n^2)$ time. Note that the $O(n)$ time algorithm in Section 2.1 assumes that the weight of any LD-block is its length, which has the property that $\ell(s) = \ell(s_1) + \ell(s_2)$, where $s = s_1 s_2$, s_1 and s_2 are LD-blocks on the same letter x , and $\ell(s)$ is the length of s (or the total number of letters of x in s_1 and s_2).

4 A Dynamic Programming Algorithm for Weighted-LDS

Given the input string $S = S[1..n]$, let $w_x(\ell)$ be the weight of LD-block x^ℓ , $x \in \Sigma$, $2 \leq \ell \leq d$, where d is the maximum number of times a letter appears in S . Here, the weight can be thought of as a positive function of x and ℓ and it does not even have to be increasing on ℓ . For example, it could be that $w(aaa) = w_a(3) = 8$, $w(aaaa) = w_a(4) = 5$. Given $w_x(\ell)$ for all $x \in \Sigma$ and ℓ , we aim to compute the maximum weight letter-duplicated string (Weighted-LDS) using dynamic programming.

Define $T(n)$ as the value of the optimal solution of $S[1..n]$ which contains the character $S[n]$. Define $w[i, j]$ as the maximum weight LD-block $S[j]^\ell$ ($\ell \geq 2$) starting at position i and ending at position j ; if such an LD-block does not exist, then $w[i, j] = 0$. Notice that $S[j]^\ell$ does not necessarily have to contain $S[i]$ but it must contain $S[j]$. We have the following recurrence relation.

$$T(0) = 0,$$

$$T(i) = \max_{S[y] \neq S[i]} \begin{cases} T(y) + w[y+1, i] & \text{if } w[y+1, i] > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The final solution value is $\max_n T(n)$. This algorithm clearly takes $O(n^2)$ time, assuming $w[i, j]$ is given. We compute the table $w[-, -]$ next.

1. For each pair of ℓ (bounded by d , the maximum number of times a letter appears in S) and letter x , compute

$$w'_x(\ell) = \max \begin{cases} w'_x(\ell - 1) \\ w_x(\ell) \end{cases},$$

with $w'_x(1) = w_x(1)$. This can be done in $O(d|\Sigma|) = O(n^2)$ time.

2. Compute the number of occurrence of $S[j]$ in the range of $[i, j]$, $N[i, j]$. Notice that $i \leq j$ and for the base case we have $S[0] = \emptyset$.

$$N(0, 0) = 0,$$

$$N(0, j) = N(0, k) + 1, \quad k = \max \begin{cases} \{y | s[y] = s[j], 1 \leq y < j\} \\ 0 \end{cases}$$

7:10 The Longest Letter-Duplicated Subsequence Problem

■ **Table 1** Input table for $w_x(\ell)$, with $S = ababbaca$ and $d = 4$.

$x \setminus \ell$	1	2	3	4
a	5	10	20	15
b	4	16	8	3
c	1	3	5	7

■ **Table 2** Table $w'_x(\ell)$, with $S = ababbaca$ and $d = 4$.

$x \setminus \ell$	1	2	3	4
a	5	10	20	20
b	4	16	16	16
c	1	3	5	7

And,

$$N(i, j) = \begin{cases} N(i-1, j), & \text{if } s[i-1] \neq s[j] \\ N(i-1, j) - 1, & \text{if } s[i-1] = s[j] \end{cases}$$

This step takes $O(n^2)$ time.

3. Finally, we compute

$$w[i, j] = \begin{cases} w'_{s[j]}(N(i, j)), & \text{if } N(i, j) \geq 2 \\ 0, & \text{else} \end{cases}$$

This step also takes $O(n^2)$ time. We thus have the following theorem.

► **Theorem 12.** *Let S be a string of length n over an alphabet Σ and d be the maximum number of times a letter appears in S . Given the weight function $w_x(\ell)$ for $x \in \Sigma$ and $\ell \leq d$, the maximum weight letter-duplicated subsequence (Weighted-LDS) of S can be computed in $O(n^2)$ time.*

We can run a simple example as follows. Let $S = ababbaca$. Suppose the table $w_x(\ell)$ is given as Table 1. At the first step, $w'_x(\ell)$ is the maximum weight of a LD-block made with x and of length at most ℓ . The corresponding table $w'_x(\ell)$ can be computed as Table 2. At the end of the second step, we have Table 3 computed. From Table 3, the table $w[-, -]$ can be easily computed and we omit the details. For instance, $w[1, -] = [0, 0, 10, 16, 20, 0, 20]$. With that, the optimal solution value can be computed as $T(8) = 36$, which corresponds to the optimal solution $aabbaa$.

■ **Table 3** Part of the table $N[i, j]$, with $S = ababbaca$ and $d = 4$.

$i \setminus j$	1	2	3	4	5	6	7	8
8	0	0	0	0	0	0	0	1
...
3	0	0	1	1	2	2	1	3
2	0	1	1	2	3	2	1	3
1	1	1	2	2	3	3	1	4

■ **Table 4** Summary of results on LLDS+ and FT, the ? indicates that the problem is still open.

d	$LLDS+(d)$	$FT(d)$	Approximability of $LLDS+(d)$
$d \geq 6$	NP-hard	NP-complete	No approximation
$d = 3$?	P	$1.5-O(\frac{1}{n})$
$d = 4, 5$?	?	?

5 Concluding Remarks

We consider the constrained longest letter-duplicated subsequence (LLDS+) and the corresponding feasibility testing (FT) problems in this paper, where all letters in the alphabet must occur in the solutions. We parameterize the problems with d , which is the maximum number of times a letter appears in the input sequence. For convenience, we summarize the results one more time in the following table. Obviously, we have many open problems.

We also consider the weighted version (without the “full-appearance” constraint), for which we give a non-trivial $O(n^2)$ time dynamic programming solution.

If we stick with the “full-appearance” constraint, one direction is to consider two additional variants of the problem where the solutions must be a subsequence of S , in the form of $x_1^{d_1} x_2^{d_2} \cdots x_k^{d_k}$ with x_i being a substring (resp. subsequence) of S with length at least 2, $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. Intuitively, for many cases these variants could better capture the duplicated patterns in S . At this point, the NP-completeness results (similar to Theorem 1 and Corollary 1) would still hold with minor modifications to the proofs. (This reduction is still from $(\leq 2, 1, \leq 3)$ -SAT and is additionally based on the following fact: given a $(2, 1)$ -sequence $T = ABCCAB$ over $\{A, B, C\}$, where A, B and C all appear twice, the corresponding maximal “substring-duplicated-subsequences” or “subsequence-duplicated-subsequences” of T are $ABAB = (AB)^2$ or CC .) But whether these extensions allow us to design good approximation algorithms needs further study. Note that, without the “full-appearance” constraint, when x_i is a subsequence of S , the problem is a generalization of Kosowski’s longest square subsequence problem [6] and can certainly be solved in polynomial time.

References

- 1 Daniel P. Bovet and Stefano Varricchio. On the regularity of languages on a binary alphabet generated by copying systems. *Information Processing Letters*, 44(3):119–123, 1992.
- 2 Ferdinando Cicalese and Nicolo Pilati. The tandem duplication distance problem is hard over bounded alphabets. In Paola Flocchini and Lucia Moura, editors, *Combinatorial Algorithms - 21st International Workshop, IWOCA 2021, Ottawa, Canada, July 5-7, 2021*, volume 12757 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2021.
- 3 Giovanni Ciriello, Martin L Miller, Bülent Arman Aksoy, Yasin Senbabaoglu, Nikolaus Schultz, and Chris Sander. Emerging landscape of oncogenic signatures across human cancers. *Nature Genetics*, 45:1127–1133, 2013.
- 4 Juegen Dassow, Victor Mitrana, and Gheorghe Paun. On the regularity of the duplication closure. *Bulletin of the EATCS*, 69:133–136, 1999.
- 5 Andrzej Ehrenfeucht and Grzegorz Rozenberg. On regularity of languages generated by copying systems. *Discrete Applied Mathematics*, 8(3):313–317, 1984.
- 6 Adrian Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In Alberto Apostolico and Massimo Melucci, editors, *String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings*, volume 3246 of *Lecture Notes in Computer Science*, pages 93–100. Springer, 2004.

7:12 The Longest Letter-Duplicated Subsequence Problem

- 7 Manuel Lafond, Binhai Zhu, and Peng Zou. The tandem duplication distance is NP-hard. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPICs*, pages 15:1–15:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 8 Manuel Lafond, Binhai Zhu, and Peng Zou. Computing the tandem duplication distance is NP-hard. *SIAM J. Discrete Mathematics*, 36(1):64–91, 2022.
- 9 E.S. Lander, et al., and International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.
- 10 John Leech. A problem on strings of beads. *The Mathematical Gazette*, 41(338):277–278, 1957.
- 11 Marcy Macdonald, et al., and Peter S. Harper. A novel gene containing a trinucleotide repeat that is expanded and unstable on huntington’s disease. *Cell*, 72(6):971–983, 1993.
- 12 The Cancer Genome Atlas Research Network. Integrated genomic analyses of ovarian carcinoma. *Nature*, 474:609–615, 2011.
- 13 Layla Oesper, Anna M. Ritz, Sarah J. Aerni, Ryan Drebin, and Benjamin J. Raphael. Reconstructing cancer genomes from paired-end sequencing data. *BMC Bioinformatics*, 13(Suppl 6):S10, 2012.
- 14 Thomas J. Schaefer. The complexity of satisfiability problems. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 216–226. ACM, 1978.
- 15 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.
- 16 Andrew J. Sharp, et al., and Even E. Eichler. Segmental duplications and copy-number variation in the human genome. *The American J. of Human Genetics*, 77(1):78–88, 2005.
- 17 Jack W. Szostak and Ray Wu. Unequal crossing over in the ribosomal dna of *saccharomyces cerevisiae*. *Nature*, 284:426–430, 1980.
- 18 Craig A. Tovey. A simplified np-complete satisfiability problem. *Discret. Appl. Math.*, 8(1):85–89, 1984.
- 19 Ming-Wei Wang. On the irregularity of the duplication closure. *Bulletin of the EATCS*, 70:162–163, 2000.
- 20 Chunfang Zheng, P Kerr Wall, James Leebens-Mack, Claude de Pamphilis, Victor A Albert, and David Sankoff. Gene loss under neighborhood selection following whole genome duplication and the reconstruction of the ancestral populus genome. *Journal of Bioinformatics and Computational Biology*, 7(03):499–520, 2009.

Reduction Ratio of the IS-Algorithm: Worst and Random Cases

Vincent Jugé ✉

LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-Vallée, France

Abstract

We study the IS-algorithm, a well-known linear-time algorithm for computing the suffix array of a word. This algorithm relies on transforming the input word w into another word, called the *reduced* word of w , that will be at least twice shorter; then, the algorithm recursively computes the suffix array of the reduced word. In this article, we study the *reduction ratio* of the IS-algorithm, i.e., the ratio between the lengths of the input word and the word obtained after reducing k times the input word. We investigate both worst cases, in which we find precise results, and random cases, where we prove some strong convergence phenomena. Finally, we prove that, if the input word is a randomly chosen word of length n , we should not expect much more than $\log(\log(n))$ recursive function calls.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Word combinatorics, Suffix array, IS algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.8

Related Version *Full Version*: <https://arxiv.org/abs/2204.04422>

1 Introduction

The suffix array of a word is the permutation of its suffixes that orders them for the lexicographic order. Suffix arrays were introduced in 1990 by Manber and Meyers [9] as a space-efficient alternative to suffix trees. Like suffix trees, they have been used since then in many applications [1, 3, 10]: data compression, pattern matching, plagiarism detection, . . .

Suffix arrays were first constructed *via* the construction of suffix trees. Then, various algorithms were proposed to construct suffix arrays directly [4, 5, 6, 7]. A more comprehensive list of approaches towards constructing suffix trees can be found in [14]. In 2010, a new algorithm, called the *IS-algorithm*, was proposed for constructing suffix arrays [12]. This algorithm, which is extremely efficient in practice, is recursive: except if the letters of its input word w are pairwise distinct, in which case the suffix array of w is easy to compute directly, the algorithm transforms w into a shorter word w' and deduces the suffix array of w from the suffix array of w' .

Thus, the question of knowing the *reduction ratio* $|w'|/|w|$ between the lengths of the words w' and w , as well as the number of recursive calls, is critical to evaluating the efficiency of the algorithm. More generally, denoting by $\text{is}^k(w)$ the word obtained after k recursive calls (with $\text{is}^0(w) = w$), we wish to evaluate the ratio $|\text{is}^k(w)|/|w|$ for all k , as well as computing the number of recursive calls that the algorithm will make, i.e., the maximal value of k .

In this article, we focus on these two questions in two different contexts. In Section 3, we consider worst cases, and prove that there exist arbitrarily long words w such that $|\text{is}^k(w)| \approx 2^{-k}|w|$ for all $k \leq \log_2(|w|) - 3$, thereby extending results from [2].

Then, in Section 4, we refine the work of [11] and consider words whose letters are generated by a Markov chain of order 1. In this context, and under mild conditions about the Markov chain, we prove, for each integer $k \geq 0$, that the ratio $|\text{is}^k(w)|/|w|$ almost surely tends to a given constant γ_k when $|w| \rightarrow +\infty$. Finally, in Section 5, we study the constant γ_1 (and,



© Vincent Jugé;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 8; pp. 8:1–8:23

Leibniz International Proceedings in Informatics



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

in some cases, γ_2) when the letters of w are identically and independently generated and, in Section 6, we propose upper bounds on the number of recursive steps on the IS-algorithm when the letters of w are given by a finite Markov chain.

2 Preliminaries

2.1 Definitions and notations

Let \mathcal{A} be a non-empty alphabet, endowed with a linear order \leq . For every integer $n \geq 0$, we denote by \mathcal{A}^n the set of words of length n over \mathcal{A} , i.e., the set of sequences of n letters in \mathcal{A} . We also denote by \mathcal{A}^* the set of all finite words over \mathcal{A} , i.e., the union $\bigcup_{n \geq 0} \mathcal{A}^n$, and by ε the empty word.

Let w be a finite word over \mathcal{A} . We denote by $|w|$ the length of w , and by $w_0, w_1, \dots, w_{|w|-1}$ the letters of w . We may abusively denote by w_{-k} the letter $w_{|w|-k}$, i.e., the k^{th} rightmost letter of w . For all integers i and j such that $0 \leq i \leq j \leq |w| - 1$, we also denote by $w_{i\dots j}$ the word $w_i w_{i+1} \dots w_j$. Every such word is called a *factor* of w . If $j = |w| - 1$, this word is a suffix of w , and we also denote it by $w_{i\dots}$. Finally, given two words u and v , we denote by $u \cdot v$ their concatenation, i.e., the word $u_0 u_1 \dots u_{|u|-1} v_0 v_1 \dots v_{|v|-1}$.

The *suffix array* [9] of a word $w \in \mathcal{A}^*$ is the unique permutation σ of $\{0, 1, \dots, |w| - 1\}$ such that $w_{\sigma(0)\dots} <_{\text{lex}} w_{\sigma(1)\dots} <_{\text{lex}} \dots <_{\text{lex}} w_{\sigma(|w|-1)\dots}$, where $<_{\text{lex}}$ denotes the lexicographic ordering. The IS-algorithm [12] aims at computing the suffix array of its input word w in time linear in $|w|$, when the alphabet \mathcal{A} is either a given finite set or a subset of $\{0, 1, \dots, |w| - 1\}$.

2.2 Unimodal factors and one-step reduction

Let w be a finite word over \mathcal{A} , and let $\$$ be a fictitious letter, called the *sentinel*, that is defined to be smaller than all letters in \mathcal{A} . Below, we simply denote by $\mathcal{A}_\$$ the set $\mathcal{A} \cup \{\$\}$.

An integer $i \leq |w| - 1$ is said to be *w-non-decreasing* if there exists an integer j such that $i + 1 \leq j \leq |w| - 1$ and $w_i = w_{i+1} = \dots = w_{j-1} < w_j$. If, in addition, $i \geq 1$ and $w_{i-1} > w_i$, we say that i is *w-locally minimal*.

Then, let $i_0 < i_1 < \dots < i_{k-1}$ be the *w-locally minimal* integers (with $k \geq 0$). We also set $i_k = |w|$, and we abusively set $w_{|w|} = \$$. This amounts to replacing w by the word $w \cdot \$$, whose suffix array is the same as the one of w , except that we appended the letter $\$$ to every suffix and that $\$$ is now the least non-empty suffix of $w \cdot \$$.

We define the *unimodal factors* of w , also called *LMS factors* [11, 12], as the k words $w_{i_0\dots i_1}, w_{i_1\dots i_2}, \dots, w_{i_{k-1}\dots i_k}$, which belong to $\mathcal{A}^+ \cdot (\varepsilon + \$)$. We call these factors *unimodal* because each sequence $w_{i_\ell}, w_{i_\ell+1}, \dots, w_{i_{\ell+1}}$ consists of a non-decreasing prefix followed by a non-increasing suffix, and we denote by $\text{eis}(w)$ – for *expanded IS-reduction* of w – the word over the infinite alphabet $\mathcal{A}^+ \cdot (\varepsilon + \$)$ whose letters are the unimodal factors of w .

For instance, if w is the word `COMBINATORIAL` over the latin alphabet \mathcal{A} , its unimodal factors are `BINA`, `ATO`, `ORIA` and `AL$`, and thus $\text{eis}(w)$ is the four-letter word `BINA·ATO·ORIA·AL$` over the alphabet $\mathcal{A}^+ \cdot (\varepsilon + \$)$.

In subsequent sections, we may extend to infinite words w (to which we append the letter $\$$ if w is left-infinite, but not if w is right-infinite) the notions of *w-locally minimal* integer, of unimodal factor, and of expanded IS-reduction.

The IS-algorithm roughly works as follows:

1. compute *w-locally minimal* integers and the associated unimodal factors, which form the letters of $\text{eis}(w)$;
2. sort these factors;

3. if w has ℓ distinct unimodal factors, identify each factor with an integer $i \in \{0, 1, \dots, \ell - 1\}$: factors f and f' such that $f <_{\text{lex}} f'$ are identified with integers i and i' such that $i < i'$;
4. identify the word $\text{eis}(w)$ with a word $\text{is}(w)$ over the alphabet $\{0, 1, \dots, \ell - 1\}$;
5. compute the suffix array of $\text{is}(w)$, either directly (if the letters of $\text{is}(w)$ are pairwise distinct) or recursively (if at least two letters of $\text{is}(w)$ coincide with each other);
6. based on that array, sort all suffixes of w .

As mentioned by its authors [12], steps 1, 3 and 4 of the algorithm can clearly be performed in time $\mathcal{O}(|w|)$. If \mathcal{A} is a given finite set, or a subset of $\{0, 1, \dots, |w| - 1\}$, bucket sorts allow sorting in linear time unimodal words whose rightmost letters are already sorted, thereby performing steps 2 and 6 in time $\mathcal{O}(|w|)$. Finally, no two consecutive integers $i \leq |w| - 1$ are w -locally minimal, and therefore $|\text{is}(w)| \leq |w|/2$, thereby proving that the IS-algorithm works in time $\mathcal{O}(|w|)$.

Thus, a natural question would be that of evaluating the constant hidden in this $\mathcal{O}(|w|)$ running time. To that end, we could focus closely on how each of the steps 1 to 4 and 6 is performed. However, several variants might be considered for performing each of these steps. Consequently, we focus on the step 5 and study the behaviour of the ratio $|\text{is}(w)|/|w|$ or, more generally, $|\text{is}^k(w)|/|w|$.

2.3 Markov chains and ergodicity

In Sections 4 to 6, we consider *random* words, whose letters result from a probabilistic process, and are random variables that form a (homogeneous) *Markov chain*. Below, we focus exclusively on such Markov chains, and thus abandon the epithet “homogeneous”.

Let \mathcal{S} be a countable set, let $\mu : \mathcal{S} \mapsto \mathbb{R}$ be a probability distribution, and let $M : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}$ be a function such that $\sum_{t \in \mathcal{S}} M(s, t) = 1$ for all $s \in \mathcal{S}$. A homogeneous Markov chain with *set of states* \mathcal{S} , *initial distribution* μ and *transition matrix* M is a sequence of random variables $(X_n)_{n \geq 0}$ with values in \mathcal{S} such that $\mathbb{P}(X_0 = x) = \mu(x)$ for all $x \in \mathcal{S}$ and such that, for every integer $n \geq 1$ and every tuple $(x_0, x_1, \dots, x_n) \in \mathcal{S}^{n+1}$, we have

$$\mathbb{P}(X_n = x_n \mid X_0 = x_0, X_1 = x_1, \dots, X_{n-1} = x_{n-1}) = M(x_{n-1}, x_n)$$

whenever $\mathbb{P}(X_0 = x_0, X_1 = x_1, \dots, X_{n-1} = x_{n-1}) > 0$. Below, we identify the Markov chain with the pair (M, μ) , or with the transition matrix M in contexts where the initial distribution is irrelevant and might need to be changed. We also abusively say that $(X_n)_{n \geq 0}$ is a *trajectory* of the Markov chain (M, μ) or, alternatively, is *generated* by (M, μ) .

The *underlying graph* of (M, μ) is the weighted graph $G = (V, E, \pi)$ with vertex set $V = \mathcal{S}$, edge set $E = \{(s, t) \in \mathcal{S} \times \mathcal{S} : M(s, t) > 0\}$, and whose weight function $\pi : E \mapsto \mathbb{R}$ is defined by $\pi(s, t) = M(s, t)$. We say that (M, μ) is *irreducible* if G is strongly connected, and *aperiodic* when the lengths of its cycles have no common divisor $d \geq 2$.

These notions are connected to the *ergodicity* of a Markov chain, which can be defined as follows. Given a probability distribution ν on \mathcal{S} , we denote by $M\nu$ the probability distribution defined by $(M\nu)(x) = \sum_{y \in \mathcal{S}} M(y, x)\nu(y)$. Then, the L^1 distance between two distributions ν and θ is defined as the real number $\|\nu - \theta\|_1 = \sum_{x \in \mathcal{S}} |\nu(x) - \theta(x)|$. The Markov chain M is said to be *ergodic* if there exists a positive probability distribution ν on \mathcal{S} (i.e., a probability distribution such that $\nu(x) > 0$ for all $x \in \mathcal{S}$) such that $\lim_{k \rightarrow +\infty} \|\nu - M^k \theta\|_1 = 0$ for all probability distributions θ on \mathcal{S} .

Such a distribution ν must be the unique *stationary distribution* of the Markov chain M , i.e., the unique probability distribution such that $\nu = M\nu$. Conversely, when M is irreducible and has a stationary distribution that is positive on \mathcal{S} , we say that M is *irreducible and positive recurrent*. This latter assumption relieves us from the need of aperiodicity, and yet retains some desirable properties of ergodic Markov chains.

8:4 Reduction Ratio of the IS-Algorithm

A typical example of an ergodic Markov chain arises if $\mathbb{P}(X_n = t | X_{n-1} = s) = \nu(t)$ for all s and t in \mathcal{S} , i.e., if $M\theta = \nu$ for all probability distributions θ on \mathcal{S} . In that case, the random variables $(X_n)_{n \geq 0}$ are said to be *independent and identically distributed*.

We refer the reader to [8, 13] for a comprehensive review about Markov chains and their properties, from which we present three crucial results below.

► **Proposition 1** (Corollary 1.18 and Theorem 21.14 of [8]). *Every ergodic Markov chain is irreducible and aperiodic. Conversely, every irreducible and aperiodic Markov chain is ergodic, provided that its state set is finite or that it has a positive stationary distribution.*

We are particularly interested in Theorem 4.16 of [8], on which we will base Section 4. However, we will not necessarily handle ergodic Markov chains, and therefore we shall relax the notion of ergodicity to a less stringent, *ad hoc* notion that we call *almost surely eventually positive recurrent and irreducible* (or EPRI) Markov chains.

A Markov chain M with underlying graph $G = (\mathcal{S}, E, \pi)$, is said to be EPRI if there exists a set $\mathcal{X} \subseteq \mathcal{S}$ of states, called the *terminal component* of M , such that (i) \mathcal{X} is a strongly connected component of G ; (ii) M has stationary distribution ν , i.e., a probability distribution ν such that $M\nu = \nu$, that is positive on \mathcal{X} and zero on $\mathcal{S} \setminus \mathcal{X}$; and (iii) for every initial distribution μ , the sequence generated by (M, μ) almost surely contains a vertex $x \in \mathcal{X}$.

Note that, since ν is positive on \mathcal{X} and zero elsewhere, no edge of G can leave \mathcal{X} , i.e., the set E contains no edge (x, y) such that $x \in \mathcal{X}$ and $y \notin \mathcal{X}$.

In this notion, we completely abandon any requirement to be acyclic, which prevents the L^1 convergence that characterises ergodicity. However, when focusing on *average, long-term* behaviours of a Markov chain, such as the frequency of occurrence of a given vertex or sequence of consecutive vertices, whether the Markov chain is cyclic or acyclic is irrelevant. Thus, we may just focus on irreducible, positive recurrent Markov chains. Moreover, in EPRI Markov chains, the path followed before entering the terminal component quickly vanishes. Consequently, the following result, which is usually stated for irreducible, positive recurrent Markov chains only, can be generalised to all EPRI Markov chains whose state space is either finite or countably infinite.

► **Theorem 2** (Theorem 4.16 of [8], Theorem 2.1.1 of [13]). *Let $(M, \mu) = (X_n)_{n \geq 0}$ be an EPRI Markov chain with set of states \mathcal{S} and stationary distribution ν . Let ℓ be a positive integer, let $f : \mathcal{S}^\ell \mapsto \mathbb{R}$ be a bounded function, and let*

$$\mathbb{E}_\nu[f] = \sum_{x_1, x_2, \dots, x_\ell \in \mathcal{S}} \nu(x_1) M(x_1, x_2) M(x_2, x_3) \cdots M(x_{\ell-1}, x_\ell) f(x_1, x_2, \dots, x_\ell).$$

We have

$$\mathbb{P} \left[\frac{1}{n} \sum_{k=0}^{n-1} f(X_k, X_{k+1}, \dots, X_{k+\ell-1}) \xrightarrow{n \rightarrow +\infty} \mathbb{E}_\nu[f] \right] = 1.$$

Proof. It is well-known [13] that Theorem 2 holds when M is irreducible and positive recurrent, i.e., when its state space \mathcal{S} coincides with its terminal component \mathcal{X} .

In the general case, trajectories of the Markov chain almost surely meet \mathcal{X} after a finite number of steps, say p , that depends of the trajectory. Once it meets \mathcal{X} , the trajectory starts behaving like an irreducible, positive recurrent Markov chain with state space \mathcal{X} . Thus,

$$\frac{1}{n-p} \sum_{k=p}^{n-1} f(X_k, X_{k+1}, \dots, X_{k+\ell-1})$$

converges almost surely (as $n \rightarrow +\infty$) to $\mathbb{E}_\nu[f]$. Theorem 2 follows. ◀

Finally, a crucial well-known property of irreducible, positive recurrent Markov chains whose initial distribution coincides with their stationary distribution is that they can be reversed.

► **Theorem 3** (Proposition 1.22 of [8]). *Let $(X_n)_{n \geq 0}$ be an irreducible, positive recurrent Markov chain with set of states \mathcal{S} , transition matrix M , and whose initial distribution coincides with the stationary distribution ν of M . For all integers $\ell \geq 0$, the sequence $(X_{\ell-n})_{0 \leq n \leq \ell}$ contains the first $\ell + 1$ elements of an irreducible, positive recurrent Markov chain, called the reverse Markov chain of (M, ν) , with initial distribution ν and whose transition matrix \hat{M} is defined by*

$$\hat{M}(x, y) = \frac{\nu(y)}{\nu(x)} M(y, x).$$

More generally, if M is EPRI, and provided that its initial distribution is ν , it already starts inside of its terminal component \mathcal{X} , which it cannot leave. Thus, up to deleting those states of M that do not belong to \mathcal{X} , the Markov chain M becomes irreducible and positive recurrent, and Theorem 3 applies, with the following caveat: the state space of its reverse Markov chain is restricted to \mathcal{X} , and needs not be extended to states outside of \mathcal{X} .

3 Deterministic worst case

By construction, no two consecutive integers $i \leq |w| - 1$ are w -locally minimal, and all w -locally minimal integers belong to the set $\{1, 2, \dots, |w| - 2\}$. Hence, at most $(|w| - 1)/2$ integers are w -locally minimal. This means that $|\text{is}^k(w)| + 1 \leq (|w| + 1)/2$ and, more generally, that $|\text{is}^k(w)| + 1 \leq 2^{-k}(|w| + 1)$ for every integer $k \geq 0$ and every word $w \in \mathcal{A}^*$ such that $\text{is}^k(w)$ exists. A genuine question is then: can we do better? The answer, which was known to be negative [2] when we allow alphabets \mathcal{A} with size $\log_2(|w|)$, remains negative for every fixed size $|\mathcal{A}| \geq 2$.

► **Theorem 4.** *Let \mathcal{A} be an alphabet of cardinality at least 4. For every integer $n \geq 3$, there exists a word $w \in \mathcal{A}^{2^n - 1}$ on which the IS-algorithm performs $n - 2$ recursive calls, and $|\text{is}^k(w)| + 1 = 2^{-k}(|w| + 1)$ for all $k \in \{0, 1, \dots, n - 2\}$.*

Proof. Without loss of generality, we assume that $\mathcal{A} = \{0, 1, 2, 4\}$. Let also $\mathcal{B} = \{0, 1, 2, 3, 4\}$. Then, let $\varphi: \mathcal{B}^* \mapsto \mathcal{B}^*$ and $\psi: \mathcal{B}^* \mapsto \mathcal{A}^*$ be morphisms of monoids, uniquely defined by their values on \mathcal{B} : $\varphi(0) = 02$, $\varphi(1) = 04$, $\varphi(2) = 12$, $\varphi(3) = 13$ and $\varphi(4) = 14$; $\psi(a) = a$ for all $a \in \mathcal{A}$, and $\psi(3) = 4$. We prove below that the word $\psi(\varphi^n(3)_{1\dots})$ satisfies the requirements of Theorem 4.

We say that a word $w = w_0 w_1 \dots w_k \in \mathcal{B}^*$ is *balanced* if (1) its length $|w| = k + 1$ is even, (2) its rightmost letter $w_k = 3$, (3) its suffix $w_{1\dots}$ contains each of the letters 0, 1, 2, 3, 4, and (4) for all $i \leq k - 1$, we have $w_i \in \{0, 1\}$ if i is even and $w_i \in \{2, 4\}$ if i is odd. The eight-letter word $\varphi^3(3) = 02140413$ is balanced, and φ maps each balanced word to a balanced word.

Provided that w is balanced, the $\varphi(w)_{1\dots}$ -minimal integers are 1, 3, 5, \dots , $2k - 1$, and the associated unimodal factors are $\varphi(w_1) \cdot \varphi(w_2)_0$, $\varphi(w_2) \cdot \varphi(w_3)_0$, \dots , $\varphi(w_{k-1}) \cdot \varphi(w_k)_0$, $\varphi(w_k) \cdot \$$. Since $\varphi(0)_1 = \varphi(1)_1 = 0$ and $\varphi(2)_1 = \varphi(3)_1 = \varphi(4)_1 = 1$, this means that the unimodal factors of $\varphi(w)$ are $\theta(w_1)$, $\theta(w_2)$, \dots , $\theta(w_k)$, where we set $\theta(0) = 021$, $\theta(1) = 041$, $\theta(2) = 120$, $\theta(3) = 13\$$ and $\theta(4) = 140$. The function θ is increasing, and thus, $\text{is}(\varphi(w)_{1\dots}) = w_{1\dots}$.

Moreover, if w is balanced, and since the rightmost letter of $\varphi(w)$ is its only occurrence of the letter 3, the words $\varphi(w)_{1\dots}$ and $\psi(\varphi(w)_{1\dots})$ have the same unimodal factors, except that their last factors are 13\$ and 14\$, respectively. Hence, $\text{is}(\psi(\varphi(w)_{1\dots})) = \text{is}(\varphi(w)_{1\dots}) = w_{1\dots}$. Thus, the map successively sends $\psi(\varphi^n(3)_{1\dots})$ to $\varphi^{n-1}(3)_{1\dots}$, $\varphi^{n-2}(3)_{1\dots}$, \dots , $\varphi^3(3)_{1\dots}$, and observing that $\text{is}(\varphi^3(3)_{1\dots}) = 201$ completes the proof. ◀

8:6 Reduction Ratio of the IS-Algorithm

Although the conclusions of Theorem 4 are not valid for alphabets of cardinality 2 or 3, it is still possible to find variants of this worst case. In these variants, the first step of the IS-algorithm is more efficient, with respective reduction ratios of 3 and 5/2, but every word considered after that first step belongs to an alphabet of cardinality 4, which explains why the reduction ratios we compute have similar orders of magnitude.

► **Corollary 5.** *Let \mathcal{A} be an alphabet of cardinality 2. For every integer $n \geq 3$, there exists a word $w \in \mathcal{A}^{3 \times 2^n - 2}$ on which the IS-algorithm performs $n - 1$ recursive calls, and $|\text{is}^k(w)| + 1 = 2^{1-k}(|w| + 2)/3$ for all $k \in \{1, 2, \dots, n - 1\}$.*

Proof. Let us assume that $\mathcal{A} = \{0, 1\}$, and let \mathcal{B} and φ be the alphabet and the morphism defined in the proof of Theorem 4. An immediate induction on ℓ shows that, for all $\ell \geq 3$, the word $\varphi^\ell(3)$ starts with the letter 0, ends with the letter 3, and contains $2^{\ell-2}$ letters 0, $2^{\ell-2}$ letters 1, $2^{\ell-2} - 1$ letters 2, one letter 3 (the rightmost one) and $2^{\ell-2}$ letters 4.

Then, we consider a new morphism $\psi_2: \mathcal{B}^* \mapsto \mathcal{A}^*$, such that $\psi_2(0) = 0001$, $\psi_2(1) = 001$, $\psi_2(2) = 01$, and $\psi_2(3) = \psi_2(4) = 011$. Like in the proof of Theorem 4, we prove that $\text{is}(1 \cdot \psi_2(w_{1\dots})) = \text{is}(\varphi(w)_{1\dots}) = w_{1\dots}$ when w is balanced, and having counted occurrences of each letter in $\varphi^n(3)$ allows us to conclude that the word $1 \cdot \psi_2(\varphi^n(3)_{1\dots})$ satisfies the requirements of Corollary 5. ◀

► **Corollary 6.** *Let \mathcal{A} be an alphabet of cardinality 3. For every integer $n \geq 3$, there exists a word $w \in \mathcal{A}^{5 \times 2^n - 3}$ on which the IS-algorithm performs n recursive calls, and $|\text{is}^k(w)| + 1 = 2^{2-k}(|w| + 3)/5$ for all $k \in \{1, 2, \dots, n\}$.*

Proof. The proof is the same as that of Corollary 5, except that we have now $\mathcal{A} = \{0, 1, 2\}$ and that, instead of the morphism ψ_2 , we use a new morphism $\psi_3: \mathcal{B}^* \mapsto \mathcal{A}^*$, such that $\psi_3(0) = 001$, $\psi_3(1) = 01$, $\psi_3(2) = 012$, $\psi_3(3) = \psi_3(4) = 02$. Indeed, we also have $\text{is}(1 \cdot \psi_3(w_{1\dots})) = \text{is}(\varphi(w)_{1\dots}) = w_{1\dots}$ when w is balanced, from which we conclude that the word $1 \cdot \psi_3(\varphi^n(3)_{1\dots})$ satisfies the requirements of Corollary 6. ◀

4 Words generated by an ergodic Markov chain

Let \mathcal{A} be a finite or countably infinite set. Below, we study the typical behaviour of the IS-algorithm on a word $w \in \mathcal{A}^n$ whose letters are the first n elements of an EPRI Markov chain (M, μ) with set of states \mathcal{A} . We prove below the following result, which is the main (and technically most demanding) result presented in this paper.

► **Theorem 7.** *Provided that w is generated by an EPRI Markov chain, and for all integers $k \geq 0$, there exist a constant γ_k and a sequence $(\varepsilon_n)_{n \geq 0}$ that tends to 0 such that*

$$\mathbb{P} \left[\left| \frac{|\text{is}^k(w)|}{|w|} - \gamma_k \right| \geq \varepsilon_{|w|} \right] \leq \varepsilon_{|w|}.$$

A particular case of interest arises when w is a word over a finite alphabet generated by an ergodic Markov chain. However, even in that restricted case, studying the words $\text{is}^k(w)$ for $k \geq 1$ will require us to consider words over infinite alphabets, which might be generated by Markov chains no longer ergodic, but only EPRI. That is why, facing the need to treat such a generalised setting, we chose to include it from the start in our study.

In addition, all finite-state Markov chains can be decomposed as a “sum” of EPRI Markov chains. Indeed, if the underlying graph of such a Markov chain (M, μ) has k terminal strongly connected components, the Markov chain will almost surely reach one of these

components. Thus, in order to study the Markov chain (M, μ) , we may consider, one by one, its k terminal components; for each such component K , compute the probability that (M, μ) eventually reaches K ; finally, simulate the behaviour of (M, μ) by first selecting at random which terminal component K it will reach, and then assuming that (M, μ) must reach that component, thereby transforming (M, μ) into an EPRI Markov chain. This allows us to obtain the following variant of Theorem 7.

► **Theorem 8.** *Let w be a word whose letters are generated by a finite-state Markov chain. There exist a constant κ and a probability law X over the set $\{1, 2, \dots, \kappa\}$ with the following property: For all integers $k \geq 0$, there exist constants $\gamma_{1,k}, \gamma_{2,k}, \dots, \gamma_{\kappa,k}$ and a sequence $(\varepsilon_n)_{n \geq 0}$ that tends to 0 such that, for all $i \leq \kappa$,*

$$\left| \mathbb{P} \left[\left| \frac{|\text{is}^k(w)|}{|w|} - \gamma_{i,k} \right| \leq \varepsilon_{|w|} \right] - \mathbb{P}[X = i] \right| \leq \varepsilon_{|w|}.$$

4.1 Generating letters from right to left

In [11], the letters of w are generated from right to left, i.e., the letter w_{n-k} is the k^{th} element of the Markov chain. Here, we mainly focus on this case too. Generating the letters of w from right to left makes things easier because, although being w -non-decreasing is *not* a local property, it enjoys the following local, recursive characterization: an integer i is w -non-decreasing if and only if $i \leq |w| - 2$ and either (a) $w_i < w_{i+1}$, or (b) $w_i = w_{i+1}$ and $i + 1$ is w -non-decreasing.

Below, we wish to study the sequence $w, \text{is}(w), \text{is}^2(w), \dots$ and in particular the lengths of these words. In fact, it will be easier to study the sequence $w, \text{eis}(w), \text{eis}^2(w), \dots$. These two sequences differ from each other because they do not use the same alphabets. Yet, for all $k \geq 0$, the words $\text{is}^k(w)$ and $\text{eis}^k(w)$ are “isomorphic” to each other: they have the same length, and there exists an increasing mapping φ from the letters of $\text{eis}^k(w)$ to those of $\text{is}^k(w)$, such that $\varphi(\text{eis}^k(w)_i) = \text{is}^k(w)_i$ for all $i < |\text{eis}^k(w)|$.

Following [11, 12], we transform the Markov chain (M, μ) into another Markov chain $(\overline{M}, \overline{\mu})$ that starts with the letter $\$$ and, in addition to telling which letter we produce, also tells whether the corresponding index is w -non-decreasing: instead of producing letters $a \in \mathcal{A}_\$,$ this new Markov chain shall produce pairs (a, \uparrow) or (a, \downarrow) , depending on whether the current position is w -non-decreasing or not: we produce a pair (a, \uparrow) if the former case, and (a, \downarrow) in the latter case. Formally, the Markov chain $(\overline{M}, \overline{\mu})$ is defined as follows. Its states form the set $\overline{\mathcal{S}} = \mathcal{A}_\$ \times \{\uparrow, \downarrow\}$. Its initial distribution is defined by $\overline{\mu}(\$, \uparrow) = 1$, and $\overline{\mu}(s) = 0$ whenever $s \neq (\$, \uparrow)$. Its transition matrix is then defined by

$$\begin{cases} \overline{M}((\$, \uparrow), (y, \downarrow)) = \mu(y) & \text{if } y \in \mathcal{A}; \\ \overline{M}((x, \downarrow), (y, \downarrow)) = M(x, y) & \text{if } (x, y) \in \mathcal{A}^2 \text{ and } x < y; \\ \overline{M}((x, \downarrow), (y, \uparrow)) = M(x, y) & \text{if } (x, y) \in \mathcal{A}^2 \text{ and } x > y; \\ \overline{M}((x, \downarrow), (y, \downarrow)) = M(x, y) & \text{if } (x, y) \in \mathcal{A}^2, x = y \text{ and } \downarrow = \downarrow; \\ \overline{M}((x, \downarrow), (y, \uparrow)) = 0 & \text{otherwise.} \end{cases}$$

► **Proposition 9.** *Let (M, μ) be an EPRI Markov chain whose terminal component has size at least two. The Markov chain $(\overline{M}, \overline{\mu})$ defined above is EPRI.*

Proof. Let $G = (\mathcal{A}, E, \pi)$ be the underlying graph of the Markov chain (M, μ) , let \mathcal{X} be its terminal component, and let ν be its stationary distribution. In addition, for all $x \in \mathcal{A}$, let $x^\uparrow = \{y \in \mathcal{X} : x < y \text{ and } (y, x) \in E\}$ and $x^\downarrow = \{y \in \mathcal{X} : x > y \text{ and } (y, x) \in E\}$.

8:8 Reduction Ratio of the IS-Algorithm

Since $M(x, x) < 1$ for all $x \in \mathcal{A}$, the distribution $\bar{\nu}$ on $\bar{\mathcal{S}}$ defined by $\bar{\nu}(\$, \updownarrow) = 0$ and by

$$\bar{\nu}(x, \updownarrow) = \frac{1}{1 - M(x, x)} \sum_{y \in x^\updownarrow} M(y, x) \nu(y)$$

for all $(x, \updownarrow) \in \mathcal{A} \times \{\up, \downarrow\}$ is a probability distribution, because

$$\bar{\nu}(x, \up) + \bar{\nu}(x, \downarrow) = \frac{1}{1 - M(x, x)} \sum_{y: x \neq y} M(y, x) \nu(y) = \frac{M\nu(x) - M(x, x)\nu(x)}{1 - M(x, x)} = \nu(x) \quad (1)$$

for all $x \in \mathcal{A}$. We further deduce from (1) that

$$\begin{aligned} \bar{M}\bar{\nu}(x, \updownarrow) - M(x, x)\bar{\nu}(x, \updownarrow) &= \sum_{y \in x^\updownarrow} M(y, x) (\bar{\nu}(y, \up) + \bar{\nu}(y, \downarrow)) = \sum_{y \in x^\updownarrow} M(y, x) \nu(y) \\ &= (1 - M(x, x))\bar{\nu}(x, \updownarrow), \end{aligned}$$

i.e., that $\bar{M}\bar{\nu}(x, \updownarrow) = \bar{\nu}(x, \updownarrow)$, for all $(x, \updownarrow) \in \mathcal{A} \times \{\up, \downarrow\}$. This means that $\bar{\nu}$ is a stationary distribution of $(\bar{M}, \bar{\mu})$.

This probability distribution is positive on the set

$$\bar{\mathcal{X}} \stackrel{\text{def}}{=} \{(x, \up): x \in \mathcal{X}, x^\up \neq \emptyset\} \cup \{(x, \downarrow): x \in \mathcal{X}, x^\down \neq \emptyset\}$$

and is zero outside of $\bar{\mathcal{X}}$. Since $\bar{\nu}$ is non-zero, it follows that $\bar{\mathcal{X}}$ is non-empty.

Then, let \bar{G} be the underlying graph of $(\bar{M}, \bar{\mu})$. We shall prove that $\bar{\mathcal{X}}$ satisfies the requirements (i) and (iii) of EPRI Markov chains. Hence, consider some state (x, \up) in $\bar{\mathcal{X}}$, and let y be a state in x^\up . For every state (z, \updownarrow) in $\bar{\mathcal{X}}$, the graph G contains a finite path from z to x whose second-to-last vertex is y , and thus \bar{G} contains a finite path from (z, \updownarrow) to (x, \up) . Similarly, every state (x, \down) in $\bar{\mathcal{X}}$ is accessible from every state (z, \updownarrow) in $\bar{\mathcal{X}}$, and thus $\bar{\mathcal{X}}$ satisfies the requirement (i).

Finally, consider some trajectory $(\bar{X}_n)_{n \geq 0}$ of $(\bar{M}, \bar{\mu})$. Deleting its first vertex and removing the second component of each vertex transforms $(\bar{X}_n)_{n \geq 0}$ into a trajectory $(X_n)_{n \geq 1}$ of the Markov chain M , which almost surely contains a vertex $x \in \mathcal{X}$ and then almost surely meets a vertex distinct from x ; let y be the first such vertex. The trajectory $(\bar{X}_n)_{n \geq 0}$ contains the vertex (y, \up) if $y < x$, or (y, \down) if $y > x$, and in both cases that vertex belongs to $\bar{\mathcal{X}}$. This shows that $\bar{\mathcal{X}}$ satisfies the requirement (iii). \blacktriangleleft

Using Theorem 2 for the function $f: \bar{\mathcal{S}} \times \bar{\mathcal{S}} \mapsto \mathbb{R}$ defined by

$$\begin{cases} f((x, \up), (y, \down)) = 1 & \text{for all } x, y \in \mathcal{A}; \\ f(u, v) = 0 & \text{in all other cases} \end{cases}$$

already allows us to prove a special case of Theorem 7 for $k = 1$, which was already proven in [11] in the case \mathcal{A} is finite and (M, μ) is ergodic.

However, if the terminal component of M contains only one state z , the Markov chain $(\bar{M}, \bar{\mu})$ is no longer EPRI, since its graph contains two self-loops around (z, \up) and (z, \down) , each one with weight 1. We overcome this difficulty by merging the two states (z, \up) and (z, \down) into one single state z , thereby recovering an EPRI Markov chain, and we modify the function f , redefining it by

$$\begin{cases} f((x, \up), (y, \down)) = 1 & \text{for all } x, y \in \mathcal{A} \setminus \{z\}; \\ f((x, \up), z) = 1 & \text{for all } x \in z^\up; \\ f(u, v) = 0 & \text{in all other cases.} \end{cases}$$

Tackling this special case allows us to derive the following result, whose validity does not depend on the size of the terminal component of M .

► **Corollary 10.** *If the letters of w are generated from right to left by an EPRI Markov chain, there exists a constant γ_1 such that $\mathbb{P}[|\text{eis}(w)|/|w| \rightarrow \gamma_1] = 1$.*

Moreover, since $|\text{is}^{k+1}(w)| \leq |\text{is}^k(w)|$ for all words w and all integers $k \geq 0$, we already know that Theorem 7 holds, with $\gamma_k = 0$, when the terminal component of M has size one. Henceforth, we assume that this terminal component has size at least two.

Under this assumption, let us show that the letters of the word $\text{eis}(w)$ are also generated by a Markov chain. In order to do so, we introduce the function $M^+ : \mathcal{A} \rightarrow \mathbb{R}$ defined by

$$M^+(x) = \sum_{y: x < y} M(x, y)$$

for every letter $x \in \mathcal{A}$, and the function $m : \mathcal{A}^+ \cdot (\varepsilon + \$) \rightarrow \mathbb{R}$ defined by

$$m(w_0 w_1 \cdots w_k) = M(w_1, w_0) M(w_2, w_1) \cdots M(w_k, w_{k-1})$$

and $m(w \cdot \$) = m(w) \mu(w_{-1})$ for every word $w = w_0 w_1 \cdots w_k$ in \mathcal{A}^+ . We also define the set

$$\mathcal{U}^\wedge \stackrel{\text{def}}{=} \{w_0 w_1 \cdots w_\ell \in \mathcal{A}^+ \cdot (\varepsilon + \$) : M^+(w_0) > 0 \text{ and} \\ \exists k \leq \ell, w_0 \leq \dots \leq w_{k-1} < w_k \geq \dots \geq w_{\ell-1} > w_\ell\}.$$

► **Lemma 11.** *The letters of the word $\text{eis}(w)$ are generated from right to left by the Markov chain $(\mathring{M}, \mathring{\mu})$ with set of states \mathcal{U}^\wedge , whose initial distribution is defined by*

$$\mathring{\mu}(w) = M^+(w_0) m(w) \mathbf{1}_{w_{-1}=\$}$$

for every word $w \in \mathcal{U}^\wedge$, and whose transition matrix is defined by

$$\mathring{M}(w, w') = \frac{M^+(w'_0)}{M^+(w_0)} \mathbf{1}_{w_0=w'_{-1}} m(w').$$

Proof. Let $u^{(1)}, u^{(2)}, \dots, u^{(k)}$ be unimodal words such that $u_{-1}^{(i)} = u_0^{(i+1)}$ for all $i \leq k-1$. These are the k rightmost letters of the word $\text{eis}(w)$ if and only if there exists a letter $x \in \mathcal{A}$ such that $x > u_0^{(1)}$ and $w \cdot \$$ ends with the suffix $x \cdot u^{(1)} \cdot u_{1\dots}^{(2)} \cdot u_{1\dots}^{(3)} \cdots u_{1\dots}^{(k)}$, which happens with probability

$$\mathbf{P}_x \stackrel{\text{def}}{=} M(u_0^{(1)}, x) m(u^{(1)}) m(u^{(2)}) \cdots m(u^{(k-1)}) m(u^{(k)}) \mathbf{1}_{u_{-1}^{(k)}=\$}.$$

Summing these probabilities \mathbf{P}_x for all $x > u_0^{(1)}$, we observe that $u^{(1)}, u^{(2)}, \dots, u^{(k)}$ are the rightmost letters of $\text{eis}(w)$ with probability

$$\mathbf{P} = M^+(u_0^{(1)}) m(u^{(1)}) m(u^{(2)}) \cdots m(u^{(k-1)}) m(u^{(k)}) \mathbf{1}_{u_{-1}^{(k)}=\$} \\ = \mathring{M}(u^{(2)}, u^{(1)}) \mathring{M}(u^{(3)}, u^{(2)}) \cdots \mathring{M}(u^{(k)}, u^{(k-1)}) \mathring{\mu}(u^{(k)}).$$

Finally, Corollary 10 proves that, if w is a left-infinite word whose letters are generated by (M, μ) from right to left, the word $\text{eis}(w)$ is almost surely infinite. It follows that $\mathring{\mu}$ is indeed a probability distribution and that \mathring{M} is indeed a transition matrix, i.e., that

$$\sum_{w' \in \mathcal{U}^\wedge} \mathring{\mu}(w') = 1 \text{ and } \sum_{w' \in \mathcal{U}^\wedge} \mathring{M}(w, w') = 1$$

for all words $w \in \mathcal{U}^\wedge$. ◀

8:10 Reduction Ratio of the IS-Algorithm

Our next move consists in proving that the Markov chain $(\overset{\circ}{M}, \overset{\circ}{\mu})$ is EPRI, by exhibiting its stationary distribution. To that end, we first require the following result, which roughly states that “almost surely, every letter of a left-infinite word w generated by (M, μ) belongs to a unimodal factor of w ”, and whose formal proof can be found in Appendix A.1.

► **Lemma 12.** *For all letters $x \in \mathcal{A}$ such that $M^+(x) \neq 0$, we have*

$$\bar{v}(x, \uparrow) = \sum_{w \in \mathcal{U}^\wedge : x=w_0} m(w) \bar{v}(w_{-1}, \uparrow).$$

With this result in hand, we can now prove Proposition 13, following the same lines of the proofs used for Proposition 9.

► **Proposition 13.** *Let (M, μ) be an EPRI Markov chain whose terminal component has size at least two. The Markov chain $(\overset{\circ}{M}, \overset{\circ}{\mu})$ is EPRI.*

Proof. First, let γ_1 be the constant of Corollary 10. Theorem 2 proves that

$$\gamma_1 = \sum_{(x, \uparrow) \in \bar{\mathcal{X}}} \left(\sum_{y \in \mathcal{X} : x < y} M(x, y) \bar{v}(x, \uparrow) \right) = \sum_{(x, \uparrow) \in \bar{\mathcal{X}}} M^+(x) \bar{v}(x, \uparrow).$$

Then, consider the distribution \hat{v} defined by

$$\hat{v}(w) = \frac{1}{\gamma_1} M^+(w_0) m(w) \bar{v}(w_{-1}, \uparrow)$$

Lemma 12 proves that

$$\sum_{w \in \mathcal{U}^\wedge} \hat{v}(w) = \frac{1}{\gamma_1} \sum_{x \in \mathcal{A}} M^+(x) \sum_{w \in \mathcal{U}^\wedge : x=w_0} m(w) \bar{v}(w_{-1}, \uparrow) = \frac{1}{\gamma_1} \sum_{x \in \mathcal{A}} \bar{v}(x, \uparrow) M^+(x) = 1,$$

i.e., that \hat{v} is a probability distribution.

Moreover, for every word $w \in \mathcal{U}^\wedge$, Lemma 12 also proves that

$$\begin{aligned} \overset{\circ}{M} \hat{v}(w) &= \frac{1}{\gamma_1} \sum_{w' \in \mathcal{U}^\wedge} \mathbf{1}_{w_{-1}=w'_0} M^+(w_0) m(w) m(w') \bar{v}(w'_{-1}, \uparrow) \\ &= \frac{1}{\gamma_1} M^+(w_0) m(w) \bar{v}(w_{-1}, \uparrow) = \hat{v}(w). \end{aligned}$$

This means that \hat{v} is a stationary probability distribution of $(\overset{\circ}{M}, \overset{\circ}{\mu})$.

This probability distribution is positive on the set $\overset{\circ}{\mathcal{X}} \stackrel{\text{def}}{=} \mathcal{U}^\wedge \cap \mathcal{X}^*$ and is zero outside of that set. Since \hat{v} is a probability distribution, it follows that $\overset{\circ}{\mathcal{X}} \neq \emptyset$.

Then, let G and $\overset{\circ}{G}$ be the respective underlying graphs of (M, μ) and $(\overset{\circ}{M}, \overset{\circ}{\mu})$. We shall prove that $\overset{\circ}{\mathcal{X}}$ satisfies the requirements (i) and (iii) of EPRI Markov chains.

Hence, consider two words w and w' in $\overset{\circ}{\mathcal{X}}$, and let us choose letters $x, y, z, t \in \mathcal{X}$ such that $x \in (w'_{-1})^\uparrow$, $w'_0 \in y^\downarrow$, $z \in w_{-1}^\uparrow$ and $w_0 \in t^\downarrow$. The graph G contains a finite path that starts with the letter x , then the letters of w' (listed from right to left) and then the letter y , and finishes with the letter z , the letters of w (listed from right to left), and then the letter t . Writing these letters from right to left, we obtain a word u whose leftmost unimodal factor is w and whose second rightmost unimodal factor is w' . This proves that $\overset{\circ}{G}$ contains a path from w' to w , i.e., that $\overset{\circ}{\mathcal{X}}$ satisfies the requirement (i).

Finally, consider some trajectory $(\overset{\circ}{X}_n)_{n \geq 0}$ of the Markov chain $(\overset{\circ}{M}, \overset{\circ}{\mu})$. Up to removing the first letter of every word (i.e., vertex) $w \in \mathcal{U}^\wedge$ encountered on this trajectory, reversing these shortened words, and then concatenating the resulting words, we obtain a trajectory

$(X_n)_{n \geq 0}$ of (M, μ) . That trajectory almost surely contains a vertex $x \in \mathcal{X}$, and will then keep visiting vertices in \mathcal{X} . Thus, our initial trajectory almost surely contains a word \dot{X}_n that is a word with a letter $x \in \mathcal{X}$, and all states \dot{X}_m such that $m \geq n + 1$ will then belong to the set $\mathcal{U} \cap \mathcal{X}^* = \dot{X}$, thereby showing that $\dot{\mathcal{X}}$ satisfies the requirement (iii). ◀

► **Proposition 14.** *The conclusion of Theorem 7 holds, provided that the letters of w are generated by an EPRI Markov chain from right to left.*

Proof. Let ℓ be the smallest integer, if any, such that the letters of the word $\text{eis}^\ell(w)$ are not generated, from right to left, by an EPRI Markov chain whose terminal component has size at least two.

If $\ell \geq k$, or if ℓ does not exist, applying Corollary 10 to the words $w, \text{eis}(w), \dots, \text{eis}^{k-1}(w)$ proves that, for all $i \leq k - 1$, there exists a positive constant θ_i such that

$$\mathbb{P}[|\text{eis}^{i+1}(w)|/|\text{eis}^i(w)| \rightarrow \theta_i] = 1$$

when $|\text{eis}^i(w)| \rightarrow +\infty$. In that case, the constant $\gamma_k = \theta_0 \theta_1 \cdots \theta_{k-1}$ satisfies the requirements of Theorem 7.

However, if $\ell \leq k - 1$, then $\text{eis}^\ell(w)$ is generated by an EPRI Markov chain whose terminal component has size one, i.e., consists in an absorbing state. In that case, Corollary 10 proves that $|\text{eis}^{\ell+1}(w)|/|\text{eis}^\ell(w)| \rightarrow 0$ almost surely, and thus the constant $\gamma_k = 0$ satisfies the requirements of Theorem 7. ◀

4.2 Generating letters from left to right

We focus now on the case where the letters of w are generated from left to right, i.e., the letter w_k is the $(k + 1)^{\text{th}}$ element of a Markov chain (M, μ) – we use a bold-face version of those notations used in Section 4.1.

The two following phenomena make generating the letters of w from left to right harder. First, whether an integer k is w -non-decreasing depends on the letters w_ℓ for $\ell \geq k$, and not on the letters w_ℓ for $\ell \leq k$. Second, we defined w as the prefix of length n of a right-infinite word \bar{w} . However, whether a given integer $k \leq n - 1$ is w -non-decreasing may depend on n since, for instance, $n - 1$ is *never* w -non-decreasing. We overcome this second issue by generalising the notion of non-decreasing integer and of expanded IS-reduction to infinite words, which allows us to use the following result.

► **Lemma 15.** *Let \bar{w} be a right-infinite word, let $n \geq 4$ be an integer, and let w be a word such that $n - 4 \leq |w| \leq n + 6$ and $w_{0\dots n-5} = \bar{w}_{0\dots n-5}$. Finally, let λ be the number of \bar{w} -locally minimal integers that are smaller than n . We have $\lambda - 4 \leq |\text{eis}(w)| \leq \lambda + 6$, and $\text{eis}(w)_{0\dots \lambda-5} = \text{eis}(\bar{w})_{0\dots \lambda-5}$ if $\lambda \geq 4$.*

Proof. Let $i_0 < i_1 < \dots < i_{\lambda-1}$ the \bar{w} -locally minimal integers smaller than n . By construction, we know that $i_j + 2 \leq i_{j+1}$ for all $j \leq \lambda - 2$. This means that $i_{\lambda-3} \leq n - 5$, and therefore an integer $j < i_{\lambda-3}$ is \bar{w} -locally minimal if and only if it is also w -locally minimal. Thus, the $\lambda - 4$ first unimodal factors of both w and \bar{w} are the words $\bar{w}_{i_j \dots i_{j+1}}$, where $0 \leq j \leq \lambda - 5$. This already proves that $|\text{eis}(w)| \geq \lambda - 4$ and that $\text{eis}(w)_{0\dots \lambda-5} = \text{eis}(\bar{w})_{0\dots \lambda-5}$.

Finally, if an integer $j \leq n - 5$ is locally w -minimal but not locally \bar{w} -minimal, we know that $\bar{w}_{j-1} = w_{j-1} > w_j = \bar{w}_j$, and therefore j is w -non-decreasing but not \bar{w} -non-decreasing. This means that $w_{j-1} > w_j = w_{j+1} = \dots = w_{n-5}$, and therefore there may be at most one such integer j . Furthermore, since no two consecutive integers may be w -minimal, the

8:12 Reduction Ratio of the IS-Algorithm

interval $\{n-4, n-3, \dots, n+5\}$ contains at most five w -locally minimal integers. Hence, there exist at most six w -locally minimal integers that do not belong to the set $\{i_0, i_1, \dots, i_{\lambda-1}\}$. This means that $|\text{eis}(w)| \leq \lambda + 6$. \blacktriangleleft

Lemma 15 allows us to approximate $\text{eis}(w)$ with a prefix of length λ of the word $\text{eis}(\bar{w})$, and proves that this approximation is of excellent quality. Indeed, if we set $\lambda_0 = n$, and inductively define λ_{i+1} as the number of $\text{eis}^i(\bar{w})$ -minimal integers smaller than λ_i , Lemma 15 ensures that $\lambda_i - 4 \leq |\text{eis}^i(w)| \leq \lambda_i + 6$. Thus, evaluating $|\text{eis}^i(w)|$ amounts to evaluating λ_i : this is the task on which we focus below, which allows us to identify w with an right-infinite word, thereby saving us from many technicalities.

The first hurdle we mentioned, which requires being able to “guess” whether a given integer will be w -non-increasing, is easy to overcome by proceeding as follows. When generating a new letter a , the corresponding position in the word has a given probability of being w -non-decreasing, which depends only on a . Thus, we can “guess” whether this position should be w -non-decreasing with the correct probability, and then stick to our guess. Hence, once again, we transform our Markov chain $(\mathbf{M}, \boldsymbol{\mu})$ into another Markov chain $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$ that will generate pairs of the form (w_i, \updownarrow_i) , where w_i is the $(i+1)$ th letter of our word w , whereas $\updownarrow_i = \uparrow$ if i is w -non-decreasing, and $\updownarrow_i = \downarrow$ otherwise. Note that, unlike its variant $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$, this Markov chain never generates pairs of the form $(\$, \updownarrow)$, which means that its state space is simply a subset of $\mathcal{A} \times \{\uparrow, \downarrow\}$.

Using this technique allows us to follow the same lines of proof as in Section 4.1. Therefore, we will just mention some milestone constructions and results towards proving Theorem 7, and omit their proofs, which can be found in Appendix A.2.

Assume here that the terminal component of the EPRI Markov chain $(\mathbf{M}, \boldsymbol{\mu})$ has size at least two. Before defining the new Markov chain $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$, we first define functions \mathbf{M}^\uparrow and \mathbf{M}^\downarrow by

$$\mathbf{M}^\uparrow(x) = \frac{1}{1 - \mathbf{M}(x, x)} \sum_{y: x < y} \mathbf{M}(x, y) \quad \text{and} \quad \mathbf{M}^\downarrow(x) = \frac{1}{1 - \mathbf{M}(x, x)} \sum_{y: x > y} \mathbf{M}(x, y)$$

for all $x \in \mathcal{A}$. Then, the Markov chain $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$ uses the set of states

$$\overline{\mathcal{S}} \stackrel{\text{def}}{=} \{(x, \updownarrow) \in \mathcal{A} \times \{\uparrow, \downarrow\} : \mathbf{M}^{\updownarrow}(x) \neq 0\},$$

the initial distribution defined by $\overline{\boldsymbol{\mu}}(x, \updownarrow) = \boldsymbol{\mu}(x) \mathbf{M}^{\updownarrow}(x)$ for all $(x, \updownarrow) \in \overline{\mathcal{S}}$, and the transition matrix defined by

$$\begin{cases} \overline{\mathbf{M}}((x, \uparrow), (y, \updownarrow)) = \frac{\mathbf{M}^{\updownarrow}(y)}{\mathbf{M}^\uparrow(x)} \mathbf{M}(x, y) & \text{if } x < y; \\ \overline{\mathbf{M}}((x, \downarrow), (y, \updownarrow)) = \frac{\mathbf{M}^{\updownarrow}(y)}{\mathbf{M}^\downarrow(x)} \mathbf{M}(x, y) & \text{if } x > y; \\ \overline{\mathbf{M}}((x, \updownarrow), (y, \updownarrow)) = \mathbf{M}(x, x) & \text{if } x = y \text{ and } \updownarrow = \updownarrow; \\ \overline{\mathbf{M}}((x, \updownarrow), (y, \updownarrow)) = 0 & \text{otherwise.} \end{cases}$$

As expected, when projecting every pair generated by $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$ onto its first coordinate, we recover a realisation of the Markov chain $(\mathbf{M}, \boldsymbol{\mu})$. Furthermore, since the word w is now assumed to be infinite, the k th pair generated by $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$ is of the form (a, \uparrow) if $k-1$ is w -non-decreasing, or (a, \downarrow) otherwise, except if the Markov chain keeps looping around a state (a, \uparrow) , which happens with probability 0 since the terminal component has size at least two. In addition, this new Markov chain is, unsurprisingly, EPRI.

If the terminal component of our Markov chain contains only one state, say z , we need to adapt our construction. For all $x \in \mathcal{A} \setminus \{z\}$, we have $\mathbf{M}(x, x) < 1$, and thus the above construction is well-defined on such states. Then, we just merge the two states (z, \uparrow) and (z, \downarrow) into a single *sink* state, say (z, \downarrow) , and we set $\overline{\mathbf{M}}((z, \downarrow), (z, \downarrow)) = 1$.

Fortunately, the following result does not depend on the size of the terminal component of the Markov chain.

► **Proposition 9b.** *Let $(\mathbf{M}, \boldsymbol{\mu})$ be an EPRI Markov chain. The Markov chain $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$ defined above is EPRI.*

Hence, let us consider the function $g: \overline{\mathcal{S}} \times \overline{\mathcal{S}} \mapsto \mathbb{R}$ defined by

$$\begin{cases} g((x, \downarrow), (y, \uparrow)) = 1 & \text{for all } x, y \in \mathcal{A}; \\ g(u, v) = 0 & \text{in all other cases.} \end{cases}$$

Given a realisation $(w_0, \uparrow_0), (w_1, \uparrow_1), \dots$ of the Markov chain $(\overline{\mathbf{M}}, \overline{\boldsymbol{\mu}})$, and denoting by $w = w_0 w_1 \dots$ the word obtained by projecting these pairs onto their first coordinate, an integer $i \geq 1$ is w -locally minimal if and only if $\uparrow_{i-1} = \downarrow$ and $\uparrow_i = \uparrow$, i.e., if $g((w_{i-1}, \uparrow_{i-1}), (w_i, \uparrow_i)) = 1$. Thus, using Theorem 2 for the function g and Lemma 15 allows us to prove a special case of Theorem 7 for $k = 1$, which consists in the following variant of Corollary 10.

► **Corollary 10b.** *If the letters of w are generated from left to right by an EPRI Markov chain, there exists a constant γ_1 such that $\mathbb{P}[\lambda_1/\lambda_0 \rightarrow \gamma_1] = 1$ when $\lambda_0 \rightarrow +\infty$.*

We focus below on the case where the Markov chain has a terminal component of size at least two. In that case, we show that the letters of $\text{eis}(w)$ are also generated from left to right by an EPRI Markov chain. Mimicking Section 4.1, we introduce the function \mathbf{m} defined by

$$\mathbf{m}(w_0 w_1 \dots w_k) = \mathbf{M}(w_0, w_1) \mathbf{M}(w_1, w_2) \dots \mathbf{M}(w_{k-1}, w_k)$$

for every word $w_0 w_1 \dots w_k$ in \mathcal{A}^* . We also define the sets

$$\begin{aligned} \mathcal{U}^\wedge &\stackrel{\text{def}}{=} \{w_0 w_1 \dots w_\ell \in \mathcal{A}^* : \mathbf{M}^\uparrow(w_\ell) > 0 \text{ and} \\ &\quad \exists k \leq \ell - 1, w_0 \leq \dots \leq w_{k-1} < w_k \geq w_{k+1} \geq \dots \geq w_{\ell-1} > w_\ell\} \\ \mathcal{V}^\wedge &\stackrel{\text{def}}{=} \{w_0 w_1 \dots w_\ell \in \mathcal{A}^* : \exists k \leq \ell - 1, w_0 \leq \dots \leq w_{k-1} \leq w_k \geq w_{k+1} \geq \dots \geq w_{\ell-1} > w_\ell\}. \end{aligned}$$

► **Lemma 11b.** *The letters of the word $\text{eis}(w)$ are generated from left to right by the Markov chain $(\mathring{\mathbf{M}}, \mathring{\boldsymbol{\mu}})$ with set of states \mathcal{U}^\wedge , whose initial distribution is defined by*

$$\mathring{\boldsymbol{\mu}}(w) = \sum_{w' \in \mathcal{V}^\wedge} \mathbf{1}_{w'_{-1}=w_0} \boldsymbol{\mu}(w'_0) \mathbf{m}(w') \mathbf{m}(w) \mathbf{M}^\uparrow(w_{-1}),$$

and whose transition matrix is defined by

$$\mathring{\mathbf{M}}(w, w') = \frac{\mathbf{M}^\uparrow(w'_{-1})}{\mathbf{M}^\uparrow(w_{-1})} \mathbf{m}(w') \mathbf{1}_{w_{-1}=w'_0}.$$

► **Proposition 13b.** *Let $(\mathbf{M}, \boldsymbol{\mu})$ be an EPRI Markov chain whose terminal component has size at least two. The Markov chain $(\mathring{\mathbf{M}}, \mathring{\boldsymbol{\mu}})$ is EPRI.*

The above properties allow us to prove the following result.

► **Proposition 14b.** *The conclusion of Theorem 7 holds, provided that the letters of w are generated by an EPRI Markov chain from left to right.*

8:14 Reduction Ratio of the IS-Algorithm

Proof. Let \bar{w} be the right-infinite word whose letters are generated, from left to right, by our Markov chain. Then, let ℓ be the smallest integer, if any, such that the letters of the word $\text{eis}^\ell(\bar{w})$ are *not* generated, from left to right, by an EPRI Markov chain whose terminal component has size at least two.

If $\ell \geq k$, or if ℓ does not exist, applying Corollary 10b to the words $\bar{w}, \text{eis}(\bar{w}), \dots, \text{eis}^{k-1}(\bar{w})$ proves that, for all $i \leq k-1$, there exists a positive constant θ_i such that $\mathbb{P}[\lambda_{i+1}/\lambda_i \rightarrow \theta_i] = 1$ when $\lambda_i \rightarrow +\infty$. In that case, the constant $\gamma_k = \theta_0\theta_1 \cdots \theta_{k-1}$ satisfies the requirements of Theorem 7.

However, if $\ell \leq k-1$, then $\text{eis}^\ell(\bar{w})$ is generated by an EPRI Markov chain whose terminal component has size one. In that case, $\lambda_{\ell+1}/\lambda_\ell \rightarrow 0$ when $\lambda_\ell \rightarrow +\infty$, and therefore the constant $\gamma_k = 0$ satisfies the requirements of Theorem 7. \blacktriangleleft

5 Words with independent and identically distributed letters

Theorem 7 roughly states that, if the letters of a word w are generated (either from left to right or from right to left) by an EPRI Markov chain (M, μ) , and provided that $|w|$ is large enough, the ratio $|\text{is}^k(w)|/|w|$ should be approximately equal to a given constant γ_k depending only on k and on the Markov chain.

If we are out of luck, the Markov chain (M, μ) might generate one unique infinite word of the form $w \cdot w \cdot w \cdots$, where w is one of the worst-case words provided in Theorem 4. Consequently, and given an integer $k \geq 0$, it is possible to choose the Markov chain (M, μ) in order to have the equality $\gamma_k = 2^{-k}$. This is indeed a worst case, given that $\gamma_{\ell+1} \leq \gamma_\ell/2$ for every Markov chain and every integer $\ell \geq 0$.

A specific context that will shield us from such bad cases, while being natural, is that of words whose letters w_0, w_1, \dots, w_{n-1} are independent and identically distributed random variables with values in the alphabet \mathcal{A} . Let X be their common probability law. We first recall a result that concerns cases where \mathcal{A} is finite and X is the uniform law over \mathcal{A} .

► **Proposition 16** (Lemma 3 of [11]). *Let w be a word over a finite alphabet \mathcal{A} , whose letters are sampled independently and uniformly over \mathcal{A} , i.e., $\mathbb{P}[w_i = a] = 1/|\mathcal{A}|$ for all integers $i \leq |w| - 1$ and all letters $a \in \mathcal{A}$. The constant γ_1 of Theorem 7 satisfies the equality*

$$\gamma_1 = \frac{1}{3} - \frac{1}{6|\mathcal{A}|}.$$

This shows that, in the most simple cases, the constant γ is bounded from above by $1/3$, although γ can be arbitrarily close to $1/3$ when the cardinality of \mathcal{A} increases. We prove below that this upper bound is *universal*.

► **Proposition 17.** *Let $n \geq 1$ be an integer, and let \mathcal{A} be a finite or countably infinite alphabet. Let X be a probability law on \mathcal{A} , let*

$$\Omega \stackrel{\text{def}}{=} \{t \in [0, 1] : \exists a \in \mathcal{A} \text{ such that } \mathbb{P}[X < a] < t < \mathbb{P}[X \leq a]\}$$

be a subset of $[0, 1]$ of Lebesgue measure 1, and let $f : \Omega \mapsto \mathcal{A}$ be the function such that $f(t)$ is the letter $a \in \mathcal{A}$ for which $\mathbb{P}[X < a] < t < \mathbb{P}[X \leq a]$. We extend f to a partial function $[0, 1]^n \mapsto \mathcal{A}^n$ by setting $f(u_0u_1 \cdots u_{n-1}) = f(u_0)f(u_1) \cdots f(u_{n-1})$ if each letter u_i belongs to Ω , and not defining f over $[0, 1]^n \setminus \Omega^n$.

For every word $u \in \Omega^n$, we have $|\text{is}(u)| \geq |\text{is}(f(u))|$. Furthermore, if the letters u_0, u_1, \dots, u_{n-1} are independent and distributed according to the uniform law \mathbb{U} over $[0, 1]$, they almost surely belong to Ω , and then the letters $f(u_0), f(u_1), \dots, f(u_{n-1})$ are also independent and distributed according to the law X .

Proof. First, Ω is a disjoint union of countably many intervals whose lengths $\mathbb{P}[X = a]$ sum up to 1, and thus it has Lebesgue measure 1. The last sentence of Proposition 17 is then immediate. Hence, we focus on proving that $|\text{is}(u)| \geq |\text{is}(f(u))|$ when $u \in \Omega^n$.

Given a word w , we say that a sequence of integers $a_1 < b_1 \leq a_2 < b_2 \leq \dots \leq a_{2k} < b_{2k}$ is w -alternating of size k if $b_{2k} < |w|$, $w_{a_i} > w_{b_i}$ for all odd indices i , and $w_{a_i} < w_{b_i}$ for all even indices i . One checks easily that $|\text{is}(w)|$ is the largest size of a w -alternating sequence. Since every $f(u)$ -alternating sequence is also u -alternating, Proposition 17 follows. ◀

Unfortunately, in general, the letters of the word $\text{is}(u)$ are not independent, and both inequalities $|\text{is}^2(u)| < |\text{is}^2(f(u))|$ and $|\text{is}^2(u)| > |\text{is}^2(f(u))|$ may hold, which prevents us from designing simple bijection-flavoured variants of Proposition 17 for investigating the length of $\text{is}^k(f(u))$. Yet, Proposition 17 still leads to the following result.

► **Theorem 18.** *For every alphabet \mathcal{A} and every probability law X on \mathcal{A} , we have $\gamma_1 \leq 1/3$.*

Proof. Let u and w be n -letter words whose letters are independent random variables following the laws \mathbb{U} and X , as described in the statement of Proposition 17. Each integer $i \in \{1, 2, \dots, n-2\}$ is u -minimal if and only if $u_i = \min\{u_{i-1}, u_i, u_{i+1}\}$, which happens with probability $1/3$, while 0 and $n-1$ cannot be u -minimal. It follows that

$$\mathbb{E}[|\text{is}(w)|] \leq \mathbb{E}[|\text{is}(u)|] = (n-2)/3 \leq n/3$$

and, thanks to Theorem 7, that $\gamma_1 \leq 1/3$. ◀

In view of Proposition 16 and Theorem 18, proving that $\gamma_1 \leq 1/3 - 1/(6|\mathcal{A}|)$ even if X is not uniform might be tempting. Unfortunately, the inequality is invalid when $|\mathcal{A}| = 3$ and $(p_1, p_2, p_3) = (3/8, 1/4, 3/8)$, because in that case $\gamma_1 = 9/32 > 5/18 = 1/3 - 1/(6|\mathcal{A}|)$.

However, the case $|\mathcal{A}| = 2$ is still promising. Indeed, in that case, $\gamma_1 = p_1(1-p_1) \leq 1/4$, and the letters of the word $\text{eis}(w)$ are independent and identically distributed, since the only constraints they are subject to is that they should begin with the letter 0 and end with the suffix 10. Thus, we can still use Theorem 18 to evaluate the ratio $|\text{is}^2(w)|/|\text{is}(w)|$, thereby deriving the following result, which suggests excellent performances of the IS-algorithm.

► **Proposition 19.** *If $|\mathcal{A}| = 2$, we have $\gamma_1 \leq 1/4$ and $\gamma_2 \leq 1/12$.*

6 Bounding the number of function calls

In this last section, we provide a short argument for proving that, if \mathcal{A} is finite and if the letters of the word w are generated, either from left to right or from right to left, by a (non necessarily EPRI) Markov chain (M, μ) , we should expect $\mathcal{O}(\log(\log(|w|)))$ recursive function calls. This is the object of the following result, whose formal proof can be found in Appendix A.3.

► **Theorem 20.** *Let $w \in \mathcal{A}^n$ be a word whose letters are generated by a Markov chain (M, μ) . For all integers $\ell \geq 0$, and provided that n is large enough, the IS-algorithm has a probability $\mathbb{P} \leq n^{-2^\ell}$ of performing more than $2 \log_2(\log_2(n)) + \ell$ recursive function calls.*

Proof idea. The probability that two independent trajectories of M (whose initial distributions may differ) coincide with each other on their k first steps decreases exponentially fast with k , unless they get trapped into a cycle from which they cannot escape. However,

every letter of the word $\text{eis}^\ell(w)$ represents at least 2^ℓ letters from w . Thus, if two such letters coincide, the word w must contain two identical subwords of length 2^ℓ , an event whose probability decreases severely once 2^ℓ exceeds $\log(|w|)$.

It remains to treat the case where w gets trapped into a cycle from which it cannot escape. Again, the probability that it would take more than k steps to reach that cycle decreases exponentially fast with k , and, when $\ell \geq \log_2(k)$, these n steps (i.e., letters) will all be subsumed in the same letter of the word $\text{eis}^\ell(w)$. However, all the other letters of $\text{eis}^\ell(w)$ will coincide with each other, and thus $\text{eis}^{\ell+1}(w)$ will contain at most one letter, thereby preventing subsequent recursive calls to the IS-algorithm. ◀

This result illustrates the fact that detecting as soon as possible special cases in which suffix arrays are easy to compute (here, observing that the letters of w are pairwise distinct) can result in dramatically decreasing the size of the recursive call stack. However, the notion of being a *large enough* integer n heavily depends on the Markov chain (M, μ) , as illustrated by the worst cases studied in Section 3, which can be arbitrarily well approximated by Markov chains.

References

- 1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.
- 2 Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. *Journal of Experimental Algorithmics (JEA)*, 21:1–27, 2016.
- 3 Maxime Crochemore, Lucian Ilie, and William F Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Data Compression Conference (DCC 2008)*, pages 482–488. IEEE, 2008.
- 4 Juha Kärkkäinen, Dominik Kempa, Simon J Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *2017 Proceedings of the Nineteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017.
- 5 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 943–955. Springer, 2003.
- 6 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 186–199. Springer, 2003.
- 7 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.
- 8 David Levin and Yuval Peres. *Markov chains and mixing times*, volume 107. American Mathematical Society, 2017.
- 9 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 10 Maxim Mozgovoy, Kimmo Fredriksson, Daniel White, Mike Joy, and Erkki Sutinen. Fast plagiarism detection system. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 267–270. Springer, 2005.
- 11 Cyril Nicaud. A probabilistic analysis of the reduction ratio in the suffix-array IS-algorithm. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 374–384. Springer, 2015.
- 12 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2010.
- 13 Ursula Porod. Dynamics of Markov chains for undergraduates, 2021. URL: <https://www.math.northwestern.edu/documents/book-markov-chains.pdf>.
- 14 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4–es, 2007.

A Appendix

A.1 Proving Lemma 12

We focus here on formally proving Lemma 12, whose intuitive meaning was already given in Section 4.1. To that end, we first introduce new variants of the set \mathcal{U}^\wedge . These are the sets

$$\begin{aligned}\mathcal{U}^\wedge &\stackrel{\text{def}}{=} \{w_0 w_1 \cdots w_\ell \in \mathcal{A}^* \cdot (\varepsilon + \$) : w_0 \geq \dots \geq w_{\ell-1} > w_\ell\} \\ \mathcal{U}' &\stackrel{\text{def}}{=} \{w_0 w_1 \cdots w_\ell \in \mathcal{A}^* : w_0 \leq \dots \leq w_{\ell-1} < w_\ell\}\end{aligned}$$

of non-increasing (respectively, non-decreasing) words in $\mathcal{A}^* \cdot (\varepsilon + \$)$ whose last two letters differ from each other. We can now prove the following auxiliary result, from which we will then deduce Lemma 12.

► **Lemma 11-1.** *For all letters $x \in \mathcal{A}$, we have*

$$\bar{v}(x, \downarrow) = \sum_{w \in \mathcal{U}^\wedge : x=w_0} m(w) \bar{v}(w_{-1}, \uparrow) \text{ and } \bar{v}(x, \uparrow) = \sum_{w \in \mathcal{U}' : x=w_0} m(w) \bar{v}(w_{-1}, \downarrow).$$

Proof. Up to reversing the order \leq on \mathcal{A}_s , both equalities are equivalent to each other. Hence, we focus on proving the left one. Let x be some element of \mathcal{X} , let \hat{M} the reverse transition matrix of \bar{M} , such as described in Theorem 3, and let $(Y_n)_{n \geq 0}$ be the Markov chain with first element $Y_0 = x$ and with transition matrix \hat{M} . Then, let \mathbf{T} be the stopping time defined as the smallest integer $n \geq 1$ such that Y_n belongs to the set $\{(y, \uparrow) : y \in \mathcal{X}\}$. Since \hat{M} is EPRI, the stopping time \mathbf{T} is almost surely finite.

For each word $w \in \mathcal{U}$ such that $x = w_0$ and $w_{-1}^\uparrow \neq \emptyset$, i.e., $\bar{v}(w_{-1}, \uparrow) \neq 0$, the Markov chain $(Y_n)_{n \geq 0}$ has a probability

$$\mathbf{P}_w \stackrel{\text{def}}{=} \hat{M}((w_0, \downarrow), (w_1, \downarrow)) \hat{M}((w_1, \downarrow), (w_2, \downarrow)) \cdots \hat{M}((w_{-2}, \downarrow), (w_{-1}, \uparrow))$$

of starting with the letters $(w_0, \downarrow), (w_1, \downarrow), \dots, (w_{-2}, \downarrow), (w_{-1}, \uparrow)$, in which case $\mathbf{T} = |w| - 1$. Using Theorem 3 and the construction of \bar{M} , we have

$$\mathbf{P}_w = \frac{\bar{v}(w_{-1}, \uparrow)}{\bar{v}(x, \downarrow)} M(w_1, w_0) M(w_2, w_1) \cdots M(w_{-1}, w_{-2}) = \frac{m(w) \bar{v}(w_{-1}, \uparrow)}{\bar{v}(x, \downarrow)}.$$

Conversely, whenever $\mathbf{T} < +\infty$, the Markov chain $(Y_n)_{n \geq 0}$ starts with such a sequence of letters. Consequently, the probabilities \mathbf{P}_w sum up to 1, which completes the proof. ◀

► **Lemma 12.** *For all letters $x \in \mathcal{A}$ such that $M^+(x) \neq 0$, we have*

$$\bar{v}(x, \uparrow) = \sum_{w \in \mathcal{U}^\wedge : x=w_0} m(w) \bar{v}(w_{-1}, \uparrow).$$

Proof. Let us associate every pair $(u, v) \in \mathcal{U}' \times \mathcal{U}^\wedge$ such that $u_{-1} = v_0$ with the word $w \stackrel{\text{def}}{=} u \cdot v_{1\dots} \in \mathcal{U}^\wedge$. Lemma 11-1 then proves that

$$\bar{v}(x, \uparrow) = \sum_{u \in \mathcal{U}' : x=u_0} \left(\sum_{v \in \mathcal{U}^\wedge : u_{-1}=v_0} m(u) m(v) \bar{v}(v_{-1}, \uparrow) \right) = \sum_{w \in \mathcal{U}^\wedge : x=w_0} m(w) \bar{v}(w_{-1}, \uparrow). \quad \blacktriangleleft$$

A.2 Proving Proposition 14b

We focus here on formally proving Proposition 14b, by providing complete proofs of the results mentioned in Section 4.2. These proofs had first been omitted because of their similarity to those of Section 4.1. Consequently, we list below results that were mentioned explicitly in Section 4.2 (sometimes adapting their wording) or were left implicit in Section 4.2 but whose variants had appeared in Section 4.1.

► **Proposition 9b.** *Let (M, μ) be an EPRI Markov chain whose terminal component has size at least two. The Markov chain $(\overline{M}, \overline{\mu})$ defined in Section 4.2 is EPRI.*

Proof. Let $G = (\mathcal{A}, E, \pi)$ be the underlying graph of the Markov chain (M, μ) , let \mathcal{X} be its terminal component, and let ν be its stationary distribution. In addition, for all $x \in \mathcal{A}$, let $x^\uparrow = \{y \in \mathcal{X} : x < y \text{ and } (x, y) \in E\}$ and $x^\downarrow = \{y \in \mathcal{X} : x > y \text{ and } (x, y) \in E\}$.

The distribution $\overline{\nu}$ on $\overline{\mathcal{S}}$ defined by $\overline{\nu}(x, \uparrow) = \nu(x)M^\dagger(x)$ is a probability distribution, because

$$\overline{\nu}(x, \uparrow) + \overline{\nu}(x, \downarrow) = \frac{1}{1 - M(x, x)} \sum_{y: x \neq y} M(x, y)\nu(y) = \nu(x) \quad (2)$$

for all $x \in \mathcal{A}$. We also deduce from (2) that

$$\begin{aligned} \overline{M}\overline{\nu}(x, \uparrow) - M(x, x)\overline{\nu}(x, \uparrow) &= \sum_{y: x < y} \frac{M^\dagger(x)}{M^\dagger(y)} M(y, x)\overline{\nu}(y, \downarrow) + \sum_{y: x > y} \frac{M^\dagger(x)}{M^\dagger(y)} M(y, x)\overline{\nu}(y, \uparrow) \\ &= M^\dagger(x) \sum_{y: x \neq y} M(y, x)\nu(y) \\ &= M^\dagger(x)(M\nu(x) - M(x, x)\nu(x)) = (1 - M(x, x))\overline{\nu}(x, \uparrow), \end{aligned}$$

i.e., that $\overline{M}\overline{\nu}(x, \uparrow) = \overline{\nu}(x, \uparrow)$, for all $(x, \uparrow) \in \mathcal{A} \times \{\uparrow, \downarrow\}$. This means that $\overline{\nu}$ is a stationary distribution of $(\overline{M}, \overline{\mu})$.

This probability distribution is positive on the set $\overline{\mathcal{X}} \stackrel{\text{def}}{=} \{(x, \uparrow) \in \overline{\mathcal{S}} : x \in \mathcal{X}\}$, and zero outside of $\overline{\mathcal{X}}$. Since $\overline{\nu}$ is non-zero, it follows that $\overline{\mathcal{X}}$ is non-empty.

Now, let \overline{G} be the underlying graph of $(\overline{M}, \overline{\mu})$. We shall prove that $\overline{\mathcal{X}}$ satisfies the requirements (i) and (iii) of EPRI Markov chains.

Consider two states (x, \uparrow) in $\overline{\mathcal{X}}$ and (z, \uparrow) in $\overline{\mathcal{S}}$. Let y and t be letters in x^\uparrow and z^\uparrow , respectively. The graph G contains a finite path from z to y whose second vertex is t and whose second last vertex is x . Therefore, \overline{G} contains a finite path from (z, \uparrow) to (x, \uparrow) , which shows that $\overline{\mathcal{X}}$ satisfies the requirement (i).

Finally, consider a trajectory $(Y_n)_{n \geq 0}$ of \overline{M} . Its projection onto the first component is a trajectory in \overline{G} , and almost surely contains a vertex $x \in \mathcal{X}$, followed by another vertex y . Thus, $(Y_n)_{n \geq 0}$ contains the vertex (x, \uparrow) if $x < y$, or (x, \downarrow) if $x > y$, and in both cases that vertex belongs to $\overline{\mathcal{X}}$. This shows that $\overline{\mathcal{X}}$ satisfies the requirement (iii). ◀

► **Lemma 11b.** *The letters of the word $\text{eis}(w)$ are generated from left to right by the Markov chain $(\mathring{M}, \mathring{\mu})$ with set of states \mathcal{U}^\wedge , whose initial distribution is defined by*

$$\mathring{\mu}(w) = \sum_{w' \in \mathcal{V}^\wedge} \mathbf{1}_{w'_{-1}=w_0} \mu(w'_0) \mathbf{m}(w'_0) \mathbf{m}(w') M^\dagger(w'_{-1}),$$

and whose transition matrix is defined by

$$\mathring{M}(w, w') = \frac{M^\dagger(w'_{-1})}{M^\dagger(w_{-1})} \mathbf{m}(w') \mathbf{1}_{w_{-1}=w'_0}.$$

Proof. Let $u^{(1)}, u^{(2)}, \dots, u^{(k)}$ be unimodal words such that $u_{-1}^{(i)} = u_0^{(i+1)}$ for all $i \leq k-1$. These are the k leftmost letters of the word $\text{eis}(w)$ if and only if there exists a word $v \in \mathcal{V}^\wedge$, two letters $x, y \in \mathcal{A}$ and an integer $\ell \geq 0$ such that $v_{-1} = u_0^{(1)}$, $u_{-1}^{(k)} = x < y$, and w begins with the prefix $v \cdot u_{1\dots}^{(1)} \cdot u_{1\dots}^{(2)} \cdots u_{1\dots}^{(k)} \cdot x^\ell \cdot y$. This happens with probability

$$\mathbf{P}_{v, x^{\ell-1}.y} \stackrel{\text{def}}{=} \boldsymbol{\mu}(v_0) \mathbf{m}(v) \mathbf{m}(u^{(1)}) \mathbf{m}(u^{(2)}) \cdots \mathbf{m}(u^{(k)}) \mathbf{M}(x, x)^\ell \mathbf{M}(x, y).$$

Summing these probabilities for all v, y and ℓ , we observe that $u^{(1)}, u^{(2)}, \dots, u^{(k)}$ are the left letters of $\text{eis}(w)$ with probability

$$\begin{aligned} \bar{\mathbf{P}} &= \sum_{v \in \mathcal{V}^\wedge} \mathbf{1}_{v_{-1}=w_0} \boldsymbol{\mu}(v_0) \mathbf{m}(v) \mathbf{m}(u^{(1)}) \mathbf{m}(u^{(2)}) \cdots \mathbf{m}(u^{(k)}) \mathbf{M}^\uparrow(u_{-1}^{(k)}) \\ &= \dot{\boldsymbol{\mu}}(u^{(1)}) \dot{\mathbf{M}}(u^{(1)}, u^{(2)}) \dot{\mathbf{M}}(u^{(2)}, u^{(3)}) \cdots \dot{\mathbf{M}}(u^{(k-1)}, u^{(k)}). \end{aligned}$$

Finally, Corollary 10b proves that, if w is a right-infinite word whose letters are generated by $(\mathbf{M}, \boldsymbol{\mu})$ from left to right, the word $\text{eis}(w)$ is almost surely infinite. It follows that $\dot{\boldsymbol{\mu}}$ is indeed a probability distribution that $\dot{\mathbf{M}}$ is indeed a transition matrix, i.e., that

$$\sum_{w' \in \mathcal{U}^\wedge} \dot{\boldsymbol{\mu}}(w') = 1 \quad \text{and} \quad \sum_{w' \in \mathcal{U}^\wedge} \dot{\mathbf{M}}(w, w') = 1$$

for all words $w \in \mathcal{U}^\wedge$. ◀

Then, we adapt Lemma 11-1, which requires introducing variants of the sets \mathcal{U}^\wedge and \mathcal{U}' of Section 4.1. These variants are the sets

$$\begin{aligned} \mathcal{U}^\wedge &\stackrel{\text{def}}{=} \{w_0 w_1 \cdots w_\ell \in \mathcal{A}^* : w_0 < w_1 \geq w_2 \geq \dots \geq w_{\ell-1}\} \\ \mathcal{U}' &\stackrel{\text{def}}{=} \{w_0 w_1 \cdots w_\ell \in \mathcal{A}^* : w_0 > w_1 \leq w_2 \leq \dots \leq w_{\ell-1}\}. \end{aligned}$$

► **Lemma 11-1b.** *For all letters $x \in \mathcal{A}$, we have*

$$\nu(x) = \sum_{w \in \mathcal{U}^\wedge : x=w_{-1}} \nu(w_0) \mathbf{m}(w) \quad \text{and} \quad \nu(x) = \sum_{w \in \mathcal{U}' : x=w_{-1}} \nu(w_0) \mathbf{m}(w).$$

Proof. Up to reversing the order \leq on \mathcal{A} , both equalities are equivalent to each other. Hence, we focus on proving the left one. Let x be some element of \mathcal{X} , let $\hat{\mathbf{M}}$ be the *reverse* transition matrix of \mathbf{M} , such as described in Theorem 3, and let $(\mathbf{Y}_n)_{n \geq 0}$ be the Markov chain with first element $\mathbf{Y}_0 = x$ and with transition matrix $\hat{\mathbf{M}}$. Finally, let \mathbf{T} be the stopping time defined as the smallest integer $n \geq 1$ such that $\mathbf{Y}_n < \mathbf{Y}_{n-1}$. Since $\hat{\mathbf{M}}$ is EPRI, \mathbf{T} is almost surely finite.

For each word $w \in \mathcal{U}^\wedge$ such that $x = w_{-1}$, the Markov chain $(\mathbf{Y}_n)_{n \geq 0}$ has a probability

$$\mathbf{P}_w \stackrel{\text{def}}{=} \hat{\mathbf{M}}(w_{-1}, w_{-2}) \cdots \hat{\mathbf{M}}(w_2, w_1) \hat{\mathbf{M}}(w_1, w_0)$$

of starting with the letters $w_{-1}, \dots, w_2, w_1, w_0$, in which case $\mathbf{T} = |w| - 1$. Theorem 3 thus proves that

$$\mathbf{P}_w = \frac{\nu(w_0)}{\nu(w_{-1})} \mathbf{M}(w_0, w_1) \mathbf{M}(w_1, w_2) \cdots \mathbf{M}(w_{-2}, w_{-1}) = \frac{\mathbf{m}(w) \nu(w_0)}{\nu(x)}.$$

Conversely, whenever $\mathbf{T} < 0$, the Markov chain $(\mathbf{Y}_n)_{n \geq 0}$ starts with such a sequence of letters. Consequently, the probabilities \mathbf{P}_w sum up to 1, which completes the proof. ◀

8:20 Reduction Ratio of the IS-Algorithm

Let us now introduce the function $\nu^+ : \mathcal{A} \rightarrow \mathbb{R}$ defined by

$$\nu^+(x) = \sum_{y: x < y} \nu(y) \mathbf{M}(y, x)$$

for every letter $x \in \mathcal{A}$.

► **Lemma 12b.** *For all letters $x \in \mathcal{A}$, we have*

$$\nu^+(x) = \sum_{w \in \mathcal{U}^\wedge : x = w_{-1}} \nu^+(w_0) \mathbf{m}(w).$$

Proof. We associate every pair $(u, v) \in \mathcal{U}' \times \mathcal{U}^\wedge$ such that $u_{-1} = v_0$ and $v_{-1} > x$ with the pair $(y, w) \stackrel{\text{def}}{=} (u_0, u_1 \dots \cdot v_1 \dots \cdot x) \in \mathcal{A} \times \mathcal{U}^\wedge$, which is such that $y > w_0$. This association is bijective, and thus Lemma 11-1b proves that

$$\begin{aligned} \nu^+(x) &= \sum_{y: x < y} \nu(y) \mathbf{M}(y, x) = \sum_{y: x < y} \left(\sum_{v \in \mathcal{U}^\wedge : v_{-1} = y} \left(\sum_{u \in \mathcal{U}' : u_{-1} = v_0} \nu(u_0) \mathbf{m}(u) \mathbf{m}(v) \right) \right) \\ &= \sum_{w \in \mathcal{U}^\wedge : x = w_{-1}} \nu^+(w_0) \mathbf{m}(w). \quad \blacktriangleleft \end{aligned}$$

► **Proposition 13b.** *Let (\mathbf{M}, μ) be an EPRI Markov chain whose terminal component has size at least two. The Markov chain $(\mathring{\mathbf{M}}, \mathring{\mu})$ is EPRI.*

Proof. First, let γ_1 be the constant of Corollary 10b. Theorem 2 proves that

$$\gamma_1 = \sum_{x \in \mathcal{A}} \nu^+(x) \mathbf{M}^\dagger(x).$$

Then, consider the distribution $\mathring{\nu}$ defined by

$$\mathring{\nu}(w) = \frac{1}{\gamma_1} \nu^+(w_0) \mathbf{m}(w) \mathbf{M}^\dagger(w_{-1}).$$

Lemma 12b proves that

$$\sum_{w \in \mathcal{U}^\wedge} \mathring{\nu}(w) = \sum_{y \in \mathcal{A}} \left(\sum_{w \in \mathcal{U}^\wedge : y = w_{-1}} \mathring{\nu}(w) \right) = \frac{1}{\gamma_1} \sum_{y \in \mathcal{A}} \nu^+(y) \mathbf{M}^\dagger(y) = 1,$$

i.e., that $\mathring{\nu}$ is a probability distribution.

Moreover, for every word $w \in \mathcal{U}^\wedge$, Lemma 12b proves that

$$\mathring{\mathbf{M}} \mathring{\nu}(w) = \frac{1}{\gamma_1} \sum_{w' \in \mathcal{U}^\wedge : w'_{-1} = w_0} \nu^+(w'_0) \mathbf{m}(w' \cdot w) \mathbf{M}^\dagger(w_{-1}) = \frac{1}{\gamma_1} \nu^+(w_0) \mathbf{m}(w) \mathbf{M}^\dagger(w_{-1}) = \mathring{\nu}(w).$$

This means that $\mathring{\nu}$ is a stationary probability distribution of $(\mathring{\mathbf{M}}, \mathring{\mu})$.

This probability distribution is positive on the set

$$\mathring{\mathcal{X}} \stackrel{\text{def}}{=} \{w \in \mathcal{U}^\wedge \cap \mathcal{X}^* : \exists x \in \mathcal{X}, x > w_0 \text{ and } \mathbf{m}(x \cdot w) \neq 0\}$$

and zero outside of that set. Since $\mathring{\nu}$ is a probability distribution, it follows that $\mathring{\mathcal{X}} \neq \emptyset$.

Then, let \mathbf{G} and $\mathring{\mathbf{G}}$ be the respective underlying graphs of (\mathbf{M}, μ) and $(\mathring{\mathbf{M}}, \mathring{\mu})$. We shall prove that $\mathring{\mathcal{X}}$ satisfies the requirements (i) and (iii) of EPRI Markov chains.

Hence, consider two words w and w' in $\dot{\mathcal{X}}$, and let us choose letters $x, y, z, t \in \mathcal{X}$ such that $x \in (w'_{-1})^\uparrow$, $w'_0 \in y^\downarrow$, $z \in w_{-1}^\uparrow$ and $w_0 \in t^\downarrow$. The graph \mathbf{G} contains a finite path that starts with the letter t , then the letters of w (listed from left to right) and then the letter z , and finishes with the letter y , the letters of w' (listed from left to right), and then the letter x . This path forms a word u whose leftmost unimodal factor is w and whose second rightmost unimodal factor is w' . This proves that $\dot{\mathbf{G}}$ contains a path from w to w' , i.e., that $\dot{\mathcal{X}}$ satisfies the requirement (i).

Finally, consider some trajectory $(\dot{\mathbf{Y}}_n)_{n \geq 0}$ of the Markov chain $(\dot{\mathbf{M}}, \dot{\boldsymbol{\mu}})$. Up to removing the first letter of every word (i.e., vertex) $w \in \mathcal{U}^\wedge$ encountered on this trajectory, and then concatenating the resulting words, we obtain a trajectory $(\mathbf{Y}_n)_{n \geq 0}$ of \mathbf{M} (for an initial distribution that may differ from $\boldsymbol{\mu}$). That trajectory almost surely contains a vertex $x \in \mathcal{X}$, and will then keep visiting vertices in \mathcal{X} . Thus, our initial trajectory almost surely contains a word $\dot{\mathbf{Y}}_n$ that is a word with a letter $x \in \mathcal{X}$, and all states $\dot{\mathbf{Y}}_m$ such that $m \geq n + 1$ will then belong to the set $\mathcal{U}^\wedge \cap \mathcal{X}^* = \dot{\mathcal{X}}$, thereby showing that $\dot{\mathcal{X}}$ satisfies the requirement (iii). ◀

A.3 Proving Theorem 20

► **Theorem 20.** *Let $w \in \mathcal{A}^n$ be a word whose letters are generated by a Markov chain (M, μ) . For all integers $\ell \geq 0$, and provided that n is large enough, the IS-algorithm has a probability $\mathbf{P} \leq n^{-2^\ell}$ of performing more than $2 \log_2(\log_2(n)) + \ell$ recursive function calls.*

Proof. Given a finite word v with v -locally minimal integers $i_0 < i_1 < \dots < i_{k-1}$, we abusively set $i_{k+1} = |v|$ and $v_{|v|} = \$$, so that $\text{eis}(v)_\ell = v_{i_\ell \dots i_{\ell+1}}$ for all $\ell \leq k-1$. Then, let the *source* of a word $v' = \text{eis}(v)_{a \dots b}$ be the word $v_{i_a \dots i_{b+1}-1}$, which we also denote by $\text{src}(v')$, and which is a factor of $v_1 \dots$. If two factors of $\text{eis}(v)$ coincide with each other, so do their sources, and if they do not overlap with each other, neither do their sources. Moreover, the word $\text{src}(v')$ is at least twice longer than v' .

More generally, the ℓ^{th} *source* of a factor v' of $\text{eis}^\ell(v)$, which we denote by $\text{src}^\ell(v')$, is just v' itself if $\ell = 0$, or the $(\ell - 1)^{\text{th}}$ source of $\text{src}(v')$ if $\ell \geq 1$. Thus, if two letters of $\text{eis}^\ell(v)$ coincide with each other, so do their ℓ^{th} sources, which are non-overlapping factors of $v_{2^{\ell-1} \dots}$ of length at least 2^ℓ . Moreover, since the last letter of $\text{eis}^\ell(v)$ is the only one that ends with the character $\$$, it cannot coincide with any other letter of $\text{eis}^\ell(v)$. Therefore, the ℓ^{th} sources of our two equal letters are in fact factors of the word $v_{2^{\ell-1} \dots |v| - 2^\ell}$.

In addition, we say that the word v is k -periodic except at borders of length b if $v_j = v_{j+k}$ whenever $b \leq j < j+k \leq |v| - b$. If the factor $v_{b \dots |v| - b}$ has exactly one letter, none of the integers $b+1, \dots, |v| - b$ is locally v -minimal, and thus $|\text{eis}(v)| \leq b$, thereby proving that the word $\text{eis}^\ell(v)$ cannot exist whenever $\ell \geq \log_2(b) + 1$. This case occurs in particular when $k = 1$.

Similarly, if $|v| \leq 2b + 3k$, the word $\text{eis}^\ell(v)$ cannot exist whenever $\ell \geq \log_2(\max\{b, k\}) + 3$.

If, on the contrary, the factor $v_{b \dots |v| - b}$ has at least two letters and is of length at least $3k$, there exists a factor \mathbf{f} of $\text{eis}(v)$ whose source is a word of the form $v_{j \dots j+k-1}$ for some j such that $b \leq j < j+k \leq |v| - b$. Let us then write v as a concatenation of the form $u \cdot \text{src}(\mathbf{f})^t \cdot u'$ where u and u' have length at most $b+k$, and t is a positive integer. We can also write $\text{eis}(v)$ as a word of the form $\mathbf{a} \cdot \mathbf{f}^t \cdot \mathbf{a}'$ such that $\text{src}(\mathbf{a})$ is a suffix of u and $\text{src}(\mathbf{b}) = u'$. By construction, we have

$$|\mathbf{a}| \leq |u|/2 \leq (b+k)/2, \quad |\mathbf{f}| \leq |\text{src}(\mathbf{f})|/2 = k/2 \quad \text{and} \quad |\mathbf{a}'| \leq |u'|/2 \leq (b+k)/2,$$

which means that $\text{eis}(v)$ is k' -periodic except at borders of length b' for some integers $k' \leq k/2$ and $b' \leq (b+k)/2 \leq \max\{b, k\}$. Thus, an immediate induction on k proves that the word $\text{eis}^\ell(v)$ cannot exist whenever $\ell \geq \log_2(\max\{b, k\}) + \log_2(k) + 3$.

8:22 Reduction Ratio of the IS-Algorithm

Now, let $G = (\mathcal{S}, E)$ be the underlying graph of the Markov chain (M, μ) , and let $s = |\mathcal{S}|$ be the number of states of the Markov chain. Let \mathcal{X} (respectively, \mathcal{Y}) be the set of states $x \in E$ that belong to a cyclic (respectively, non-cyclic) terminal connected component of G . Finally, let ε be the smallest non-zero edge weight in G , i.e., $\varepsilon = \min\{M(x, y) : M(x, y) > 0\}$, and let $\eta = -\log_2(1 - \varepsilon^s)/s > 0$.

From each state $x \in E$, there is a path starting at x and ending in $\mathcal{X} \cup \mathcal{Y}$. Furthermore, the shortest such path is of length at most s . It follows, for all $k \geq 0$, that

$$\mathbb{P}[X_{k+s} \in \mathcal{X} \cup \mathcal{Y} \mid X_k = x] \geq \varepsilon^s$$

and, more generally, that

$$\mathbb{P}[X_m \notin \mathcal{X} \cup \mathcal{Y}] \leq (1 - \varepsilon^s)^{m/s-1} = 2^{-(m-s)\eta}$$

for all $m \geq 0$.

Similarly, assume that $\mathcal{Y} \neq \emptyset$. Consider some state $x \in \mathcal{Y}$, and let $y \in \mathcal{Y}$ be a state accessible from x and with at least two outgoing edges (y, z) and (y, z') . Then, let p be a path from x to y . The shortest such path has length at most $s - 1$. Therefore, provided that $X_k = x$ for some integer $k \geq 0$, the trajectory $(X_i)_{i \geq k}$ has a probability at least ε^s of starting with the path p and then going to z , and a probability at least ε^s of starting with the path p and then going to z' . In particular, for each finite sequence \mathbf{q} consisting of $s + 1$ states in \mathcal{Y} , we have

$$\mathbb{P}[(X_i)_{k \leq i \leq k+s} = \mathbf{q} \mid X_k] \leq 1 - \varepsilon^s$$

and, more generally, if \mathbf{q} is a sequence consisting of $m + 1$ states in \mathcal{Y} , we have

$$\mathbb{P}[(X_i)_{k \leq i \leq k+m} = \mathbf{q} \mid X_k] \leq (1 - \varepsilon^s)^{m/s-1} = 2^{-(m-s)\eta}.$$

Finally, assume that w is a word of length $n \geq 2^{16s^2(s+1)+64s^2/\eta}$, and set $u = \log_2(n)/(4s)$, $t = 2\lceil \log_2(u) \rceil + \ell$ and $m = 2^t - 1$. Since $m \geq 2^{\ell-2}u^2 - 1$ and $2^\ell u \geq 1$, we have

$$2^{-(m-s)\eta} \leq 2^{-(2^{\ell-2}u^2-s-1)\eta} \leq 2^{-(2^\ell us(s+1)+2^{\ell+2}us/\eta-(s+1))\eta} \leq 2^{-2^{\ell+2}us} = n^{-2^{\ell+2}}.$$

In conclusion, let us consider several (non mutually exclusive) events:

- the event \mathcal{E}_1 , which occurs if $X_m \notin \mathcal{X} \cup \mathcal{Y}$;
- the event \mathcal{E}_2 , which occurs if $X_m \in \mathcal{X}$;
- for all integers u and v such that $m \leq u$, $u + m < v$ and $v + m < n - m$, the event $\mathcal{F}_{u,v}$, which occurs if $X_m \in \mathcal{Y}$ and $X_{u+i} = X_{v+i}$ whenever $0 \leq i \leq m$.

If \mathcal{E}_2 happens, the word w is k -periodic except at borders of length m , where $k \leq s$ is the length of the cycle of G to which X_m belongs. Thus, in that case, the IS-algorithm cannot make more than

$$\begin{aligned} \log_2(\max\{s, m\}) + \log_2(s) + 2 &= \log_2(m) + \log_2(4s) \\ &\leq 2\log_2(u) + \log_2(4s) + \ell \leq 2\log_2(\log_2(n)) + \ell \end{aligned}$$

recursive function calls.

Then, if the IS-algorithm makes more than $2\log_2(\log_2(n)) + \ell \geq t$ recursive function calls, two letters of the word $\text{eis}^t(w)$ must coincide with each other. This means that two non-overlapping length- m factors of the word $w_{m \dots |w|-m-1}$ must coincide with each other, and therefore that either $X_m \notin \mathcal{Y}$ or that one of the events $\mathcal{F}_{u,v}$ must have occurred. If $X_m \notin \mathcal{Y}$, and since \mathcal{E}_2 may not have occurred, this means that \mathcal{E}_1 occurred.

Moreover, the events \mathcal{E}_1 and $\mathcal{F}_{u,v}$ are rare: our above study proves that $\mathbb{P}[\mathcal{E}_1] \leq n^{-2^{\ell+2}}$; then, for all u and v , the sequence $(X_i)_{u \leq i \leq u+m}$ being fixed, the event $\mathcal{F}_{u,v}$ also occurs with probability $\mathbb{P}_{u,v} \leq n^{-2^{\ell+2}}$.

In conclusion, the IS-algorithm makes more than $2 \log_2(n) + \ell$ recursive function calls with a probability $\mathbf{P} \leq \mathbb{P}[\mathcal{E}_1] + \sum_{u,v} \mathbb{P}[\mathcal{F}_{u,v}] \leq n^2 \times n^{-2^{\ell+2}} \leq n^{-2^\ell}$. ◀

Arbitrary-Length Analogs to de Bruijn Sequences

Abhinav Nellore  

Oregon Health & Science University, Portland, OR, USA

Rachel Ward  

The University of Texas at Austin, Austin, TX, USA

Abstract

Let $\tilde{\alpha}$ be a length- L cyclic sequence of characters from a size- K alphabet \mathcal{A} such that for every positive integer $m \leq L$, the number of occurrences of any length- m string on \mathcal{A} as a substring of $\tilde{\alpha}$ is $\lfloor L/K^m \rfloor$ or $\lceil L/K^m \rceil$. When $L = K^N$ for any positive integer N , $\tilde{\alpha}$ is a de Bruijn sequence of order N , and when $L \neq K^N$, $\tilde{\alpha}$ shares many properties with de Bruijn sequences. We describe an algorithm that outputs some $\tilde{\alpha}$ for any combination of $K \geq 2$ and $L \geq 1$ in $O(L)$ time using $O(L \log K)$ space. This algorithm extends Lempel’s recursive construction of a binary de Bruijn sequence. An implementation written in Python is available at <https://github.com/nelloreward/pkl>.

2012 ACM Subject Classification Mathematics of computing \rightarrow Combinatorial algorithms; Mathematics of computing \rightarrow Combinatorics on words

Keywords and phrases de Bruijn sequence, de Bruijn word, Lempel’s D-morphism, Lempel’s homomorphism

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.9

Related Version *arXiv preprint*: <https://arxiv.org/abs/2108.07759>

Supplementary Material *Software (Source Code)*: <https://github.com/nelloreward/pkl>
archived at `swh:1:dir:28da4be18d03945d95e62f4acfc34f10e56b1772`

Acknowledgements We thank Arie Israel, Štěpán Holub, Joe Sawada, Jeffrey Shallit, and the anonymous reviewers for the 33rd Annual Symposium on Combinatorial Pattern Matching for their helpful feedback on various iterations of this manuscript. AN thanks the Oden Institute for Computational Engineering & Sciences at the University of Texas at Austin for hosting him as a visiting scholar as part of the J. Tinsley Oden Faculty Fellowship Research Program when the work contained here was initiated.

1 Introduction

1.1 Preliminaries

This paper is concerned with necklaces, otherwise known as circular strings or circular words. A *necklace* is a cyclic sequence of characters; each character has a direct predecessor and a direct successor, but no character begins or ends the sequence. So if 101 is said to be a necklace, 011 and 110 refer to the same necklace. In the remainder of this paper, the term *string* exclusively refers to a sequence of characters with a first character and a last character. A *substring* of a necklace is a string of contiguous characters whose length does not exceed the necklace’s length. So the set of length-2 substrings of the necklace 101 is $\{10, 01, 11\}$. A *rotation* of a necklace is a substring whose length is precisely the necklace’s length. A *prefix* of a string is any substring starting at the string’s first character. So 011 can be called a rotation of the necklace 101, and 10 is a prefix of that rotation.

A *de Bruijn sequence* of order N on a size- K alphabet \mathcal{A} is a length- K^N necklace that includes every possible length- N string on \mathcal{A} as a substring [69, 17, 19, 18]. There are $(K!)^{K^N-1}/K^N$ distinct de Bruijn sequences of order N on \mathcal{A} [19]. (See the appendix for a brief summary of the curious history of de Bruijn sequences.) An example for $\mathcal{A} = \{0, 1\}$ and $N = 4$ is the length-16 necklace

0000110101111001.



© Abhinav Nellore and Rachel Ward;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 9; pp. 9:1–9:20

Leibniz International Proceedings in Informatics



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

A de Bruijn sequence of order N on \mathcal{A} is optimally short in the sense that its length is K^N , and there are K^N possible length- N strings on \mathcal{A} . But more is true: because any length- m string on \mathcal{A} is a prefix of each of K^{N-m} strings on \mathcal{A} when $m \leq N$, the sequence has precisely K^{N-m} occurrences of that length- m string as a substring. So in the example above, there are 8 occurrences of 0, 4 occurrences of 00, 2 occurrences of 000, and 1 occurrence of 0000. Note by symmetry, K^{N-m} is also the expected number of occurrences of any length- m string on \mathcal{A} as a substring of a necklace of length K^N formed by drawing each of its characters uniformly at random from \mathcal{A} . More generally, by symmetry, L/K^m is the expected number of occurrences of any length- m string on \mathcal{A} for $m \leq L$ as a substring of a necklace of arbitrary length L formed by drawing each of its characters uniformly at random from \mathcal{A} .

1.2 $P_L^{(K)}$ -sequences

Consider a necklace defined as follows.

► **Definition 1** ($P_L^{(K)}$ -sequence). *A $P_L^{(K)}$ -sequence is a length- L necklace on a size- K alphabet \mathcal{A} such that for every positive integer $m \leq L$, the number of occurrences of any length- m string on \mathcal{A} as a substring of the necklace is $\lfloor L/K^m \rfloor$ or $\lceil L/K^m \rceil$.*

This paper proves by construction that a $P_L^{(K)}$ -sequence exists for any combination of $K \geq 2$ and $L \geq 1$, giving an algorithm for sequence generation that runs in $O(L)$ time using $O(L \log K)$ space.

When $L = K^N$ for any positive integer N , $\lfloor L/K^m \rfloor = \lceil L/K^m \rceil = K^{N-m}$ for $m \leq N$, and a $P_L^{(K)}$ -sequence collapses to a de Bruijn sequence of order N . When $L \neq K^N$, a $P_L^{(K)}$ -sequence is a natural interpolative generalization of a de Bruijn sequence: it is a necklace for which the number of occurrences of any length- m string on \mathcal{A} for $m \leq L$ as a substring differs by less than one from its expected value for a length- L necklace formed by drawing each of its characters uniformly at random from \mathcal{A} . When this expected value is an integer, $\lfloor L/K^m \rfloor = \lceil L/K^m \rceil$, and the number of occurrences of any length- m string on \mathcal{A} as a substring of a given $P_L^{(K)}$ -sequence is equal to the number of occurrences of any *other* length- m string on \mathcal{A} as a substring of that sequence. When this expected value is not an integer, a $P_L^{(K)}$ -sequence comes as close as it can to achieving the same end, as formalized in the proposition below.

► **Proposition 2.** *Consider a $P_L^{(K)}$ -sequence $\tilde{\alpha}$. Load across length- m strings on \mathcal{A} for $m \leq L$ is balanced in $\tilde{\alpha}$ as follows.*

1. *When L/K^m is an integer, each length- m string on \mathcal{A} occurs precisely L/K^m times as a substring of $\tilde{\alpha}$.*
2. *When L/K^m is not an integer, each of $L - K^m \lfloor L/K^m \rfloor$ length- m strings on \mathcal{A} occurs precisely $\lceil L/K^m \rceil$ times as a substring of $\tilde{\alpha}$, and each of $K^m \lceil L/K^m \rceil - L$ length- m strings on \mathcal{A} occurs precisely $\lfloor L/K^m \rfloor$ times as a substring of $\tilde{\alpha}$.*

Proof. Item 1 is manifestly true from $\lfloor L/K^m \rfloor = \lceil L/K^m \rceil$. To see why item 2 is true, consider the system of Diophantine equations

$$\begin{aligned} a \lfloor L/K^m \rfloor + b \lceil L/K^m \rceil &= L \\ a + b &= K^m \end{aligned} \tag{1}$$

Above, a represents the number of length- m strings on \mathcal{A} for which there are $\lfloor L/K^m \rfloor$ occurrences each as a substring of $\tilde{\alpha}$, and b represents the number of length- m strings on \mathcal{A} for which there are $\lceil L/K^m \rceil$ occurrences each as a substring of $\tilde{\alpha}$. The first equation says the

total number of occurrences of strings as substrings of $\tilde{\alpha}$ is L , and the second says there is a total of K^m length- m strings on \mathcal{A} . Note the equations hold only when L/K^m is nonintegral – that is, $\lfloor L/K^m \rfloor + 1 = \lceil L/K^m \rceil$. In this case, it is easily verified the unique solution to the system is $a = K^m \lceil L/K^m \rceil - L$ and $b = L - K^m \lfloor L/K^m \rfloor$. ◀

An example for $\mathcal{A} = \{0, 1\}$ and $L = 12$ is the sequence

$$000110111001. \tag{2}$$

To see why, note L/K^m for $L = 12$ and $K = 2$ is 6 for $m = 1$, 3 for $m = 2$, between 1 and 2 for $m = 3$, and between 0 and 1 for any $m \geq 4$. Further, the sequence (2) contains, as a substring, precisely

1. 6 occurrences of each string in the set $\{0, 1\}$;
2. 3 occurrences of each string in the set $\{00, 01, 10, 11\}$;
3. 2 occurrences of each string in the set $\{001, 011, 100, 110\}$, which is of size $L - K^m \lfloor L/K^m \rfloor = 12 - 2^3 \lfloor 12/2^3 \rfloor = 4$, and 1 occurrence of each string in the set $\{000, 010, 101, 111\}$, which is of size $K^m \lceil L/K^m \rceil - L = 2^3 \lceil 12/2^3 \rceil - 12 = 4$;
4. 1 occurrence of each string in the set

$$M := \{0001, 0011, 0110, 1101, 1011, 0111, 1110, 1100, 1001, 0010, 0100, 1000\},$$

which is of size $L - K^m \lfloor L/K^m \rfloor = 12 - 2^4 \lfloor 12/2^4 \rfloor = 12$, and 0 occurrences of each of the set of length-4 strings on \mathcal{A} not in M , which is of size $K^m \lceil L/K^m \rceil - L = 2^4 \lceil 12/2^4 \rceil - 12 = 4$; and

5. 0 or 1 occurrences of any length- m string for $4 < m \leq L$ due to item 4 above.

In distinct lines of work from the 1960s and 1970s, both Korobov [51, 52] and Stoneham [80, 77, 78, 79] explored the extent to which the repetends of base- K “decimal” forms of reduced proper fractions, when treated as necklaces, differed from expectation for digit content drawn uniformly at random from $[K]$. While the perspective differed from ours in that it did not demand a particular necklace length L from the outset, the efforts did uncover that certain fractions in base- K decimal form yielded what we call $P_L^{(K)}$ -sequences for particular combinations of K and L . Notably, in [80], Stoneham found that for $L + 1$ an odd prime and K a primitive root modulo $(L + 1)^2$, the repetend of the base- K decimal form of $1/(L + 1)$ is a $P_L^{(K)}$ -sequence.

1.3 $P_L^{(K)}$ -sequences vs. other de Bruijn-like sequences

Two other arbitrary-length generalizations of de Bruijn sequences have appeared in the literature:

1. What we call a *Lempel-Radchenko sequence* is a length- L necklace on a size- K alphabet \mathcal{A} such that every length- $\lceil \log_K L \rceil$ string on \mathcal{A} has at most one occurrence as a substring of the necklace. As recounted by Yoeli in [87], according to Radchenko and Filippov in [66], the existence of binary Lempel-Radchenko sequences of any length was first proved by Radchenko in his unpublished 1958 University of Leningrad PhD dissertation [65]. Other binary-case existence proofs were furnished by (1) Yoeli himself in [85] and [86]; (2) Bryant, Heath, and Killik in [8] based on the work [42] of Heath and Gribble; and (3) Golomb, Welch, and Goldstein in [40]. Explicit constructions of arbitrary-length binary Lempel-Radchenko sequences were given by Etzion in 1986 [25]. In brief, Etzion’s approach is to join necklaces derived from the pure cycling register, potentially overshooting the target length L , and subsequently remove substrings as necessary in the resulting sequence according to specific rules to achieve the target length. This takes $o(\log L)$ time per bit generated and uses $O(\log^2 L)$ space.

The existence of Lempel-Radchenko sequences of any length for any alphabet size was proved in 1971 by Lempel in [56]. In the special case where the alphabet size is a power of a prime number, one of two approaches for sequence construction effective at any length L may be used: either (1) pursue the algebraic construction described by Hemmati and Costello in their 1978 paper [43], or (2) cut out a length- L stretch of contiguous sequence generated by a linear feedback function, as described in Chapter 7, Section 5 of Golomb’s text [39]. In his 2000 paper [54], Landsberg built on Golomb’s technique, explaining in the appendix how to use it to construct a Lempel-Radchenko sequence on an alphabet of arbitrary size. The idea is to decompose the desired alphabet size into a product of powers of pairwise-distinct primes, construct length- L sequences on alphabets of sizes equal to factors in this product with Golomb’s technique, and finally write a particular linear superposition of the sequences. The time and space requirements of Hemmati and Costello’s construction, when optimized, have gone unstudied in the literature. In general, Golomb’s technique gives a length- L Lempel-Radchenko sequence in $O(L \log L)$ time using $O(\log L)$ space, and Landsberg’s generalization multiplies these complexities by the number of factors in the prime power decomposition of the alphabet size. Etzion suggested in his 1986 paper [25] that, using results from [24], his algorithm generating a binary Lempel-Radchenko sequence could be extended to generate a Lempel-Radchenko sequence for any alphabet size, but he did not do so explicitly. It is reported on Joe Sawada’s website [20] that in their recent unpublished manuscript [41], Gündoğan, Sawada, and Cameron extend Etzion’s construction to arbitrary alphabet sizes, streamlining it so it generates each character in $O(\log L)$ time using $O(\log L)$ space. Sawada’s website further includes an implementation in C.

2. A *generalized de Bruijn sequence* is a length- L Lempel-Radchenko sequence on a size- K alphabet \mathcal{A} such that every length- $\lceil \log_K L \rceil$ string on \mathcal{A} is a substring of the sequence. Generalized de Bruijn sequences were recently introduced by Gabric, Holub, and Shallit in [32, 37]. These papers also prove generalized de Bruijn sequences exist for any combination of $L \geq 1$ and $K \geq 2$. No work to date has given explicit constructions of arbitrary-length generalized de Bruijn sequences.

We prove the following.

► **Theorem 3.** *A $P_L^{(K)}$ -sequence is a generalized de Bruijn sequence and therefore also a Lempel-Radchenko sequence.*

Proof. Let $\tilde{\alpha}$ be a $P_L^{(K)}$ -sequence. The proposition is true if and only if

1. every length- $\lceil \log_K L \rceil$ substring of $\tilde{\alpha}$ occurs precisely once in the sequence, and
2. every length- $\lceil \log_K L \rceil$ string on \mathcal{A} is a substring of $\tilde{\alpha}$.

Item 1 is true because from Definition 1, $\tilde{\alpha}$ has

$$\lfloor L/K^{\lceil \log_K L \rceil} \rfloor = \begin{cases} 1 & \text{when } \log_K L \text{ is an integer} \\ 0 & \text{otherwise} \end{cases}$$

or $\lfloor L/K^{\lceil \log_K L \rceil} \rfloor = 1$ occurrences of any length- $\lceil \log_K L \rceil$ string on \mathcal{A} as a substring. Item 2 is true because from Definition 1, $\tilde{\alpha}$ has

$$\lceil L/K^{\lfloor \log_K L \rfloor} \rceil = \begin{cases} 1 & \text{when } \log_K L \text{ is an integer} \\ k & \text{otherwise for } k \in \{2, \dots, K\} \end{cases}$$

or $\lceil L/K^{\lfloor \log_K L \rfloor} \rceil \in \{1, \dots, K - 1\}$ occurrences of any length- $\lfloor \log_K L \rfloor$ string on \mathcal{A} as a substring. ◀

$P_L^{(K)}$ -sequences are more tightly constrained than generalized de Bruijn sequences and Lempel-Radchenko sequences. A length- L Lempel-Radchenko sequence imposes no requirements regarding presence or absence of particular strings as substrings; it simply requires that the number of distinct length- $\lceil \log_K L \rceil$ substrings is L . A length- L generalized de Bruijn sequence on \mathcal{A} goes a step further, requiring not only this distinctness, but also the presence of every string on \mathcal{A} smaller than $\lceil \log_K L \rceil$ as a substring. A $P_L^{(K)}$ -sequence goes yet another step further, requiring not only this presence, but also specific incidences of strings as substrings that, as best as they can, try not to bias the sequence toward inclusion of any one length- m string over another. This requirement makes $P_L^{(K)}$ -sequences, in general, more de Bruijn-like than Lempel-Radchenko sequences and generalized de Bruijn sequences.

An example (borrowed from [37]) of a Lempel-Radchenko sequence that is not a generalized de Bruijn sequence and therefore also not a $P_L^{(K)}$ -sequence for $\mathcal{A} = \{0, 1\}$ and $L = 11$ is

$$10011110000. \quad (3)$$

In this case, $\lceil \log_K L \rceil = \lceil \log_2 11 \rceil = 4$, and indeed, there is precisely one occurrence in (3) of every length-4 substring of (3). But $\lfloor \log_K L \rfloor = \lfloor \log_2 11 \rfloor = 3$, and in (3) just 7 of 8 length-3 strings on \mathcal{A} occur as substrings; the sequence is missing 101. An example of a generalized de Bruijn sequence that is not a $P_L^{(K)}$ -sequence for $\mathcal{A} = \{0, 1, 2\}$ and $L = 12$ is

$$000111101011. \quad (4)$$

Again, $\lceil \log_K L \rceil = \lceil \log_2 12 \rceil = 4$ and $\lfloor \log_K L \rfloor = \lfloor \log_2 12 \rfloor = 3$. Now, not only is there precisely one occurrence in (4) of every length-4 substring of (4), but also all 8 length-3 strings on \mathcal{A} occur as substrings. However, (4) should have $\lceil L/K \rceil = \lfloor L/K \rfloor = 12/2 = 6$ occurrences of each of 1 and 0 as substrings to be a $P_L^{(K)}$ -sequence, and it has 5 occurrences of 0 and 7 occurrences of 1. This imbalance of 0s and 1s leads to further violations of constraints on $P_L^{(K)}$ -sequences at other substring lengths. Another example of a generalized de Bruijn sequence that is not a $P_L^{(K)}$ -sequence, this time on the nonbinary alphabet $\mathcal{A} = \{0, 1, 2\}$ and for $L = 20$, is

$$02220010121120111002. \quad (5)$$

(This sequence was constructed by Landsberg in [54] using Golomb's technique from [39] as an example of a Lempel-Radchenko sequence.) Note $\lceil \log_K L \rceil = \lceil \log_3 20 \rceil = 3$ and $\lfloor \log_K L \rfloor = \lfloor \log_3 20 \rfloor = 2$, every length-3 substring occurs exactly once, and every length-2 string on \mathcal{A} is present as a substring. But (5) should have, for $m = 2$, precisely $\lceil L/K^m \rceil = 20/3^2 = 3$ or $\lfloor L/K^m \rfloor = 20/3^2 = 2$ occurrences of every length-2 string on \mathcal{A} as a substring, and there is only 1 occurrence of 21 as a substring of (5).

1.4 de Bruijn sequence constructions vs. de Bruijn-like sequence constructions

Unlike the current situation with de Bruijn-like sequences of arbitrary length, there is a veritable cornucopia of elegant constructions of de Bruijn sequences. Excellent summaries of many of these are provided on Sawada's website [20]. They include

1. greedy constructions. Prominent examples are the prefer-largest/prefer-smallest [58], prefer-same [23, 29, 3], and prefer-opposite [4] algorithms;
2. shift rules. A shift rule maps a length- N substring of a de Bruijn sequence of order N to the next length- N substring of the sequence. Shift rules are often simple, economical, and efficient; examples generating each character of a de Bruijn sequence in amortized constant time using $O(N)$ space are [73, 45, 26, 28] in the binary case and [74] in the K -ary case. See [5, 47, 35, 36, 84, 13, 88] for other specific rules;

3. concatenation rules. The best-known example, obtained by Fredricksen and Maiorana in 1978 [31], joins all Lyndon words on an ordered alphabet of size K whose lengths divide the desired order N in lexicographic order to form the lexicographically smallest (i.e., “granddaddy”) de Bruijn sequence of that order on that alphabet. (Also see [27] for Ford’s independent work generating this sequence.) The sequence is obtained in amortized constant time per character using $O(N)$ space with the efficient Lyndon word generation approach of Ruskey, Savage, and Wang [68], which builds on Fredricksen, Kessler, and Maiorana’s papers [31, 30]. Dragon, Hernandez, Sawada, Williams, and Wong recently discovered that joining the Lyndon words in colexicographic order instead also outputs a particular de Bruijn sequence, the “grandmama” sequence [22, 21]. A generic concatenation approach using colexicographic order is developed in [33, 34];
4. recursive constructions. Broadly, these approaches are based on transforming a de Bruijn sequence into a de Bruijn sequence of higher order, where the transformation can be implemented recursively. They fall into two principal classes:
 - a. the constructions of Mitchell, Etzion, and Paterson in [59], which interleave punctured and padded variants of a binary de Bruijn sequence of order N and modify the result slightly to obtain a binary de Bruijn sequence of order $2N$. If starting with a known binary de Bruijn sequence, this process takes amortized $O(1)$ time per output bit while using $O(1)$ additional space. The constructions are notable for being efficiently decodable – that is, the position of any given string on \mathcal{A} occurring exactly once in the sequence as a substring can be retrieved in time polynomial in N ;
 - b. constructions based on Lempel’s D-morphism (otherwise known as Lempel’s homomorphism) [55], whose inverse lifts any length- L necklace $\tilde{\beta}$ on a size- K alphabet \mathcal{A} to up to K necklaces on \mathcal{A} . When $\tilde{\beta}$ is a de Bruijn sequence of order N , the necklaces to which it is lifted may be joined to form a de Bruijn sequence of order $N + 1$. Efficient implementations constructing binary de Bruijn sequences of arbitrary order by repeated application of Lempel’s D-morphism are given by Annexstein [6] as well as Chang, Park, Kim, and Song [12]; in general, a length- L binary de Bruijn sequence is generated in $O(L)$ time using $O(L)$ space. Lempel confined attention to the binary case in [55]. An extension to alphabets of arbitrary size was first written by Ronse in [67] and also developed by Tuliani in [81]; it was further generalized by Alhakim and Akinwande in [1]. See [38, 2] for other generalizations as well as [81] for a decodable de Bruijn sequence construction exploiting both interleaving and Lempel’s D-morphism.

It is possible construction techniques for de Bruijn sequences have been more easily uncovered than for their arbitrary-length cousins as traditionally defined precisely because de Bruijn sequences are more tightly constrained. But $P_L^{(K)}$ -sequences are similarly constrained.

1.5 Our contribution

This paper defines $P_L^{(K)}$ -sequences. Further, it extends recursive de Bruijn sequence constructions based on Lempel’s D-morphism [55, 67, 81, 1], giving an algorithm that outputs a $P_L^{(K)}$ -sequence on the alphabet $\{0, \dots, K - 1\}$ for any combination of $L \geq 1$ and $K \geq 2$ in $O(L)$ time using $O(L \log K)$ space. The essence of our approach is to lengthen each of d_i longest runs of the same nonzero character by a single character at the i th step before lifting, where the $\{d_i\}$ are the digits of the desired length L of the $P_L^{(K)}$ -sequence when expressed in base K – that is, for $L = \sum_{i=0}^{\lceil \log_K L \rceil} d_i K^{\lceil \log_K L \rceil - i}$. Finally, this paper is accompanied by Python code at <https://github.com/nelloreward/pkl> implementing our algorithm.

We were motivated to study arbitrary-length generalizations of de Bruijn sequences by [62], which introduces *nength*, an analog to the Burrows-Wheeler transform [10] for offline string matching in labeled digraphs. In a step preceding the transform, a digraph with edges labeled on one alphabet is augmented with a directed cycle that (1) includes every vertex of the graph and (2) matches a de Bruijn-like sequence on a different alphabet. This vests each vertex with a unique tag along the cycle. But if the de Bruijn-like sequence is an arbitrary Lempel-Radchenko or generalized de Bruijn sequence, some vertices may be significantly more identifiable than others when locating matches to query strings in the graph using its *nength*, biasing performance. So in general, it is reasonable to arrange that the directed cycle matches a $P_L^{(K)}$ -sequence, which distributes identifiability across vertices as evenly as possible.

The remainder of this paper is organized as follows. The next section develops our algorithm for generating $P_L^{(K)}$ -sequences, proving space and performance guarantees. The third and final section lists some open questions.

2 Generating $P_L^{(K)}$ -sequences

2.1 Additional notation and conventions

In the development that follows, necklaces are represented by lowercase Greek letters adorned with tildes such as $\tilde{\beta}$ and $\tilde{\gamma}$, and strings are represented by unadorned lowercase Greek letters such as ω and ξ . A necklace or string's length or a set's size is denoted using $|\cdot|$. So $|\tilde{\beta}|$ is the length of the necklace $\tilde{\beta}$, and $|V|$ is the size of the set V . Necklaces and strings may be in indexed families, where for example in $\tilde{\beta}_i$, i specifies the family member. Further, a necklace or string may be written as a function of another necklace or string. So $\omega(\tilde{\beta})$ denotes that the string ω is a function of the necklace $\tilde{\beta}$. When any function's argument is clear from context, that argument may be dropped with prior warning. So $\omega(\tilde{\beta})$ may be written as, simply, ω .

The operation of *joining* two necklaces $\tilde{\beta}$ and $\tilde{\gamma}$ at a string ω to form a new necklace $\tilde{\lambda}$ refers to cycle joining, described in Chapter 6 of Golomb's text [39]. $\tilde{\lambda}$ is obtained by concatenating rotations of $\tilde{\beta}$ and $\tilde{\gamma}$ that share the prefix ω . So if $\tilde{\beta} = 00101101$ and $\tilde{\gamma} = 0110001$, joining $\tilde{\beta}$ and $\tilde{\gamma}$ at 110 gives $\tilde{\lambda} = 110100101100010$. There may be more than one occurrence of ω as a substring of at least one of $\tilde{\beta}$ and $\tilde{\gamma}$, so there may be more than one way to join them at ω . Any way is permitted in such a case. Note joining $\tilde{\beta}$ and $\tilde{\gamma}$ at ω preserves length- m substring occurrence frequencies for $m \leq |\omega| + 1$.

For any positive integer j ,

$$[j] := \{0, 1, \dots, j - 1\}.$$

While results are obtained for sequences on the alphabet $[K]$ here, they may be translated to any size- K alphabet \mathcal{A} by appropriate substitution of characters. When a string or necklace is initially declared to be on the alphabet $[K]$, but an expression y for one of its characters is written such that $y \notin [K]$, that character should be interpreted as $y - K \lfloor y/K \rfloor$. This is simply the remainder of floored division of y by K . Put another way, expressions for characters of strings on $[K]$ respect arithmetic modulo K . For example, if the first character of a string on the alphabet $[2] = \{0, 1\}$ is specified as an expression that equals 9, that character is 1.

Individual characters comprising strings are often expressed in terms of variables, so a necklace or string may be written as a comma-separated list of characters enclosed by parentheses, where in the necklace case, \circlearrowleft is included as a subscript. For example, for $i = 3$,

if $(i, i + 1, 0, 1)$ is said to be on the alphabet $[4]$, it is the string 3001, while if $(i, i + 1, 0, 1)_\circ$ is said to be on $[4]$, it is the necklace 3001. Bracket notation is used to refer to a specific character of a string or necklace. So $\omega[i]$ refers to the character at index i of ω . Further, characters of a string are indexed in order, so $\omega[i + 1]$ appears directly after $\omega[i]$ in ω . $\omega[0]$ and $\omega[|\omega| - 1]$ refer, respectively, to the first and last characters of the string ω . For a necklace, the choice of the character at index 0 is arbitrary, but in a parenthetical representation of that necklace, the character at index 0 always comes first. So an arbitrary length- L necklace $\tilde{\beta}$ always equals

$$\tilde{\beta} = (\tilde{\beta}[0], \tilde{\beta}[1], \dots, \tilde{\beta}[L - 1])_\circ$$

but not necessarily

$$\tilde{\beta} = (\tilde{\beta}[1], \tilde{\beta}[2], \dots, \tilde{\beta}[L - 1], \tilde{\beta}[0])_\circ.$$

A valid character index of a string ω is confined to $[|\omega|]$, but a valid character index of a length- L necklace $\tilde{\beta}$ is any integer j , with the stipulation

$$\tilde{\beta}[j] = \tilde{\beta}[j + L].$$

A string or necklace on $[K]$ can be summed with any integer by adding that integer to each of its characters modulo K . So for an integer j and a length- L necklace $\tilde{\beta}$,

$$\tilde{\beta} + j = j + (\tilde{\beta}[0], \tilde{\beta}[1], \dots, \tilde{\beta}[L - 1])_\circ = (\tilde{\beta}[0] + j, \tilde{\beta}[1] + j, \dots, \tilde{\beta}[L - 1] + j)_\circ.$$

Finally, \mathbf{i}_m is used as a shorthand for the length- m string (i, i, \dots, i) , \mathbf{i}_m^{++} is used as a shorthand for the length- m string $(i, i + 1, \dots, i + m - 1)$, and \mathbf{i}_m^{++} is used as a shorthand for the length- m necklace $(i, i + 1, \dots, i + m - 1)_\circ$. In a slight abuse of notation, a variable representing a string such as ω , \mathbf{i}_m , or \mathbf{i}_m^{++} can take the place of a character in a parenthetical representation of a string or necklace. So if $(\mathbf{2}_6^{++}, 3)$ is said to be a substring of a string on the alphabet $[4]$, that substring is 2301233.

2.2 Lempel's lift

Lempel's lift, defined below, realizes the simplest K -ary version of Lempel's D-morphism [55, 67, 81, 1] in inverse form.

► **Definition 4 (Lempel's lift).** Consider a length- L necklace $\tilde{\beta}$ on the alphabet $[K]$. Lempel's lift of $\tilde{\beta}$, denoted by $\{\tilde{\lambda}_i(\tilde{\beta})\}$, is the indexed family of necklaces on $[K]$ specified by

$$\tilde{\lambda}_i(\tilde{\beta}) = i + \left(\tilde{\beta}[0], \tilde{\beta}[0] + \tilde{\beta}[1], \dots, \sum_{j=0}^{d(\tilde{\beta}) \cdot L - 1} \tilde{\beta}[j] \right)_\circ \quad i \in [p(\tilde{\beta})]. \quad (6)$$

Above, $d(\tilde{\beta})$ is the smallest positive integer such that $(\sum_{j=0}^{L-1} \tilde{\beta}[j]) \cdot d(\tilde{\beta})$ is divisible by K , and $p(\tilde{\beta}) = K/d(\tilde{\beta})$.

The remainder of this subsection (i.e., Section 2.2) abbreviates functions of $\tilde{\beta}$ given above by dropping it as an argument. For example, p is written rather than $p(\tilde{\beta})$.

Observe that $\tilde{\lambda}_i$ is a discrete integral of $\tilde{\beta}$, with $i \in [p]$ the constant of integration. The number d specified in Definition 4 is the smallest positive integer $q \in \{1, 2, \dots, K\}$ such that integrating a cycle of $\tilde{\beta}$ a total of q times gives a cycle of $\tilde{\lambda}_i$. Conversely, $\tilde{\beta}$ is uniquely determined by a discrete derivative of $\tilde{\lambda}_i$, which eliminates the constant of integration:

$$\tilde{\beta}^d = (\lambda_i[1] - \lambda_i[0], \lambda_i[2] - \lambda_i[1], \dots, \lambda_i[d \cdot L - 1] - \lambda_i[d \cdot L - 2], \lambda_i[0] - \lambda_i[d \cdot L - 1])_\circ \quad i \in [p].$$

Above, the power d on the LHS denotes $\tilde{\beta}$ is concatenated with itself d times.

In the case $K = 2$, Lempel's lift of $\tilde{\beta}$ is composed of one necklace if $\tilde{\beta}$ has an odd number of 1s and two necklaces otherwise. For example, Lempel's lift of 01011 comprises only 0110110010, while Lempel's lift of 01010 comprises 01100 and 10011; further, the derivative of 0110110010 is 0101101011 = $(01011)^2$, while the derivative of each of 01100 and 10011 is 01011. As a nonbinary example for $K = 5$, observe that Lempel's lift of the length-8 necklace 02134012 comprises the single length-40 necklace

$$0231001330143341134211244120440224032230,$$

whose derivative is $(02134012)^5$.

Note the sum of the lengths of the necklaces comprising Lempel's lift of $\tilde{\beta}$ is $K \cdot L$. Other properties of the lift pertinent to constructing $P_L^{(K)}$ -sequences are as follows.

► **Lemma 5.** *Suppose $\tilde{\beta}$ is a length- L necklace on the alphabet $[K]$, and m is an integer satisfying $0 \leq m < L$. Suppose ω is a length- m string on $[K]$, and ξ_ℓ is the length- $(m+1)$ string on $[K]$ given by*

$$\xi_\ell = \ell + \left(0, \omega[0], \omega[0] + \omega[1], \dots, \sum_{j=0}^{m-1} \omega[j] \right) \quad \ell \in [K]. \quad (7)$$

Then ω occurs t times as a substring of $\tilde{\beta}$ if and only if ξ_ℓ occurs t times as a substring of the necklaces comprising Lempel's lift of $\tilde{\beta}$. When $m = 0$, ω is the length-0 string occurring as a substring at every character of $\tilde{\beta}$.

Proof. Start constructing a given $\tilde{\lambda}_i$ by integrating $\tilde{\beta}$ from its character index 0 up to character index $w < L$. If ω occurs as a substring of $\tilde{\beta}$ at index w , it follows from (6) and (7) that ξ_y occurs as a substring of $\tilde{\lambda}_i$ at its character index $w - 1$ for some $y \in [K]$, and vice versa. For $d > 1$, continue integrating $\tilde{\beta}$ past its character index w for another L characters to encounter ω again. This time, how p is defined in terms of the sum of $\tilde{\beta}$'s characters implies ω 's presence as a substring of $\tilde{\beta}$ at index w is a necessary and sufficient condition for ξ_{y+p} 's presence as a substring of $\tilde{\lambda}_i$ at its character index $w - 1 + L$. More generally, ω occurs as a substring of $\tilde{\beta}$ at its character index w if and only if ξ_{y+qp} occurs as a substring of $\tilde{\lambda}_i$ at its character index $w - 1 + qL$ for $q \in [d]$, and all occurrences of ξ_ℓ in Lempel's lift of $\tilde{\beta}$ for which the difference between ℓ and y is divisible by p are in $\tilde{\lambda}_i$. An occurrence of ξ_ℓ at any other value of ℓ is easily seen from (6) to be at a corresponding character index $w - 1 + qL$ of $\tilde{\lambda}_j$ for particular $j \in [p] \setminus \{i\}$ and $q \in [d]$. So there is an invertible map from the set of distinct occurrences of ω as a substring of $\tilde{\beta}$ into the set of distinct occurrences of ξ_ℓ as a substring of Lempel's lift of $\tilde{\beta}$ for $\ell \in [K]$, giving the lemma. ◀

► **Lemma 6.** *The number of occurrences of a given length- m string on the alphabet $[K]$ for $0 < m \leq L$ as a substring in the family of necklaces comprising Lempel's lift of a $P_L^{(K)}$ -sequence on $[K]$ is $\lfloor L/K^{m-1} \rfloor$ or $\lceil L/K^{m-1} \rceil$.*

Proof. From (7), choosing ξ_ℓ uniquely determines ω . So by Lemma 5, any length- m string on $[K]$ occurs $\lfloor L/K^{m-1} \rfloor$ or $\lceil L/K^{m-1} \rceil$ times as a substring in the family of necklaces comprising Lempel's lift of some length- L necklace $\tilde{\beta}$ if and only if a certain length- $(m-1)$ string on $[K]$ occurs $\lfloor L/K^{m-1} \rfloor$ or $\lceil L/K^{m-1} \rceil$ times as a substring of $\tilde{\beta}$ for $0 < m \leq L$. This holds by definition when $\tilde{\beta}$ is a $P_L^{(K)}$ sequence, for which every possible length- $(m-1)$ string on $[K]$ occurs $\lfloor L/K^{m-1} \rfloor$ or $\lceil L/K^{m-1} \rceil$ times for $0 < m \leq L$, giving the lemma. ◀

2.3 Algorithm and analysis

In this subsection (Section 2.3), $\tilde{\alpha}$ is reserved to denote a $P_L^{(K)}$ -sequence. Moreover, when a function from Definition 4 is invoked, and it has $\tilde{\alpha}$ as an argument, that function is abbreviated by dropping the $\tilde{\alpha}$. For example, p now refers to $p(\tilde{\alpha})$.

9:10 Arbitrary-Length Analogs to de Bruijn Sequences

Lemma 6 suggests a way to obtain a $P_{K \cdot L}^{(K)}$ -sequence from a $P_L^{(K)}$ -sequence $\tilde{\alpha}$: join the necklaces in Lempel's lift of $\tilde{\alpha}$ strategically to ensure the numbers of occurrences of specific strings as substrings do not violate the parameters of Definition 1. Below, the procedure LIFTANDJOIN includes an explicit prescription, and Theorem 8 proves it works. They are preceded by a requisite lemma extending the discussion of cycle joining from Section 2.1.

■ **Algorithm 1** Procedure LIFTANDJOIN referenced in the text.

```

// Returns the  $P_{K \cdot L}^{(K)}$ -sequence formed by joining the necklaces
// comprising Lempel's lift of an input  $P_L^{(K)}$ -sequence  $\tilde{\alpha}$  on the
// alphabet  $[K]$ , with  $K$  a clarifying input. Here,  $N := \lceil \log_K L \rceil$ .
1: procedure LIFTANDJOIN( $\tilde{\alpha}$ ,  $K$ )
2:   Construct Lempel's lift  $\{\tilde{\lambda}_i : i \in [p]\}$  of  $\tilde{\alpha}$ .
3:   if  $p = 1$  then // Case 1
4:     return  $\tilde{\lambda}_0$ 
5:   end if
6:   if  $1_N$  is a substring of  $\tilde{\alpha}$  then // Case 2
7:     Find  $k \in [K]$  such that  $\mathbf{k}_N^{++}$  is a substring of each of  $\tilde{\lambda}_0$  and  $\tilde{\lambda}_1$ .
8:     Initialize  $\tilde{\sigma}$  to  $\tilde{\lambda}_0$ .
9:     for  $j := 1$  to  $p - 1$  do
10:      Set  $\tilde{\sigma}$  to the result of joining  $\tilde{\sigma}$  and  $\tilde{\lambda}_j$  at  $\mathbf{s}_N^{++}$  for  $s = k + j - 1$ .
11:    end for
12:    return  $\tilde{\sigma}$ 
13:   end if
14:   Construct the join graph  $G = (V, E)$  defined in Theorem 8. // Case 3
15:   Initialize  $\tilde{\sigma}$  to the necklace represented by an arbitrary vertex  $v \in V$ .
16:   Starting at  $v$ , perform a depth-first traversal of the connected component
    $G_C = (V_C, E_C)$  of  $G$  for which  $v \in V_C$ , where at each vertex in  $V_C$ 
   reached by walking across a given edge in  $E_C$ , the necklace represented
   by that vertex is joined with  $\tilde{\sigma}$  at the string labeling that edge, and the
   result is assigned to  $\tilde{\sigma}$ .
17:   if  $G_C = G$  then // Case 3a
18:     return  $\tilde{\sigma}$ 
19:   end if
20:   Find  $k \in [K]$  such that  $\mathbf{k}_{N-1}^{++}$  is a substring of each of  $\tilde{\sigma}$  and  $\tilde{\sigma} + 1$ . // Case 3b
21:   Initialize  $\tilde{\zeta}$  to  $\tilde{\sigma}$ .
22:   for  $j := 1$  to  $p/|V_C| - 1$  do
23:     Set  $\tilde{\zeta}$  to the result of joining  $\tilde{\zeta}$  and  $\tilde{\sigma} + j$  at  $\mathbf{s}_{N-1}^{++}$  for  $s = k + j - 1$ .
24:   end for
25:   return  $\tilde{\zeta}$ 
26: end procedure

```

► **Lemma 7.** Consider two necklaces $\tilde{\beta}$ and $\tilde{\gamma}$ on the alphabet $[K]$, and suppose the length- $(m - 1)$ string ω is a substring of each of them. For every $k \in [K]$, suppose further that no length- m string (ω, k) is a substring of each of $\tilde{\beta}$ and $\tilde{\gamma}$, and no length- m string (k, ω) is a substring of each of $\tilde{\beta}$ and $\tilde{\gamma}$. Finally, suppose every length- $(m + 1)$ string on $[K]$ occurs either zero times or one time as a substring of the family $\{\tilde{\beta}, \tilde{\gamma}\}$. Then

1. every length- $(m + 1)$ string on $[K]$ occurs either zero times or one time as a substring of the necklace $\tilde{\sigma}$ formed by joining $\tilde{\beta}$ and $\tilde{\gamma}$ at ω , and
2. every length- w string for $w \leq m$ occurs the same number of times as a substring of $\{\tilde{\beta}, \tilde{\gamma}\}$ as it does as a substring of $\tilde{\sigma}$.

Proof. For $u, v, x, y \in [K]$, suppose the length- $(m - 1)$ string ω occurs (1) in $\tilde{\beta}$ as a substring of the length- $(m + 1)$ string (u, ω, v) , and (2) in $\tilde{\gamma}$ as a substring of the length- $(m + 1)$ string (x, ω, y) . Join $\tilde{\beta}$ and $\tilde{\gamma}$ at these occurrences of ω to obtain the necklace $\tilde{\sigma}$. The operation replaces (u, ω, v) and (x, ω, y) with (u, ω, y) and (x, ω, v) while affecting the occurrence frequencies of no other length- $(m + 1)$ strings as substrings and no length- w strings as substrings for $w \leq m$. But (u, ω, y) cannot occur elsewhere as a substring of $\tilde{\sigma}$ because if it does, then either (u, ω) or (ω, y) is a substring of each of $\tilde{\beta}$ and $\tilde{\gamma}$, a contradiction. By a parallel argument, (x, ω, v) cannot occur elsewhere in $\tilde{\sigma}$. The lemma follows. ◀

► **Theorem 8.** *Given a $P_L^{(K)}$ -sequence $\tilde{\alpha}$ on the alphabet $[K]$, suppose $N = \lceil \log_K L \rceil$. Consider Lempel's lift $\{\tilde{\lambda}_i : i \in [p]\}$ of $\tilde{\alpha}$, and define the join graph $G = (V, E)$ as an undirected graph with p vertices such that*

1. *the vertex $v_i \in V$ represents $\tilde{\lambda}_i$ for $i \in [p]$, and*
2. *an edge in E labeled by a length- N string of the form $(\mathbf{j}_{N-1}^{++}, k)$ or $(k, \mathbf{j}_{N-1}^{++})$ for $j, k \in [K]$ extends between vertex v_ℓ and vertex v_r if and only if that string occurs as a substring of each of $\tilde{\lambda}_\ell$ and $\tilde{\lambda}_r$ for $\ell, r \in [p]$.*

Then the length- KL necklace output by LIFTANDJOIN with $\tilde{\alpha}$ and K as inputs is a $P_{K \cdot L}^{(K)}$ -sequence.

Proof. Follow the logic of the LIFTANDJOIN pseudocode to prove it returns a $P_{K \cdot L}^{(K)}$ -sequence. To start, line 2 constructs Lempel's lift of $\tilde{\alpha}$, which is composed of p necklaces that together have precisely the same number of occurrences of any length- m string on $[K]$ as a substring that a $P_{K \cdot L}^{(K)}$ -sequence does, according to Lemma 8. To join the necklaces, various cases are handled in order of increasing difficulty:

Case 1: (Lines 3-5) This is the most straightforward case, where Lempel's lift has precisely one necklace. By Lemma 6 and by definition of a $P_L^{(K)}$ -sequence, the sole necklace is a $P_{K \cdot L}^{(K)}$ -sequence, and it is returned (Line 4).

Case 2: (Line 6-13) In this case, $p > 1$ and $\mathbf{1}_N$ is a substring of $\tilde{\alpha}$ so that by Lemma 5, \mathbf{k}_{N+1}^{++} is a substring of $\tilde{\lambda}_1$ for at least one $k \in [K]$. Consequently, \mathbf{s}_N^{++} is a substring of each $\tilde{\lambda}_j$ and $\tilde{\lambda}_{j-1}$ for $j \in [p] \setminus \{0\}$ and $s = k + j - 1$. Progressively joining a necklace under construction with the j th member $\tilde{\lambda}_j$ of Lempel's lift of $\tilde{\alpha}$ at \mathbf{s}_N^{++} for $s = k + j - 1$ (Lines 8-11) and j running from 1 to $p - 1$ preserves occurrence frequencies of all strings on $[K]$ whose lengths do not exceed $N + 1$. Since by Lemma 6 a length- $(N + 1)$ string occurs either once or never as a substring of Lempel's lift of $\tilde{\alpha}$, a string whose length exceeds $N + 1$ occurs either once or never as a substring of the joined necklace. So that joined necklace is a $P_{K \cdot L}^{(K)}$ -sequence, and it is returned (Line 12). When $\tilde{\alpha}$ is a de Bruijn sequence (i.e., for $L = K^N$), Case 2 is the K -ary extension [67, 81, 1] of the original join prescription of the paper [55] by Lempel introducing his D-morphism.

Case 3: (Lines 14-25) Because a length- N string on $[K]$ need not occur as a substring of $\tilde{\alpha}$, $\mathbf{1}_N$ may not be a substring of $\tilde{\alpha}$. This bars the availability of Lempel's join of Case 2. LIFTANDJOIN then looks for the closest alternative. By definition of a $P_L^{(K)}$ -sequence, $\mathbf{1}_{N-1}$ is necessarily a substring of $\tilde{\alpha}$, and so by Lemma 5, \mathbf{j}_{N-1}^{++} is a substring of each necklace in Lempel's lift of $\tilde{\alpha}$ for some $j \in [K]$. So Line 14 assembles the graph G encoding all possible joins at strings of the form $(\mathbf{j}_{N-1}^{++}, k)$ or $(k, \mathbf{j}_{N-1}^{++})$ for $j, k \in [K]$. Consider any connected component $G_C = (V_C, E_C)$ of G . A depth-first traversal of G_C prescribes a sequence of joins, which are performed to obtain a single necklace $\tilde{\sigma}$ (Line 16). Two cases are then considered.

Case 3a: (Lines 17-19) In this case, there is just one connected component of G . Since each join was performed at a length- N string, by an argument parallel to that of Case 2, $\tilde{\sigma}$ is a $P_{K \cdot L}^{(K)}$ -sequence, and it is returned (Line 18).

Case 3b: (Lines 20-25) If there are multiple connected components of G , by symmetry, G_C is related to any other connected component by translation modulo K . More precisely, applying $v_k \rightarrow v_{k+j}$ to each vertex $v_k \in V_C$, $e_{k\ell} \rightarrow e_{k+j, \ell+j}$ to each edge $e_{k\ell} \in E_C$ extending between $v_k \in V_C$ and $v_\ell \in V_C$, and $\epsilon_{k\ell} \rightarrow \epsilon_{k+j, \ell+j} + j$ to each edge label $\epsilon_{k\ell}$ corresponding to $e_{k\ell} \in E_C$ gives a different connected component, where $j \in [p/|V_C|]$ and addition operations are performed modulo K . It follows that for every $j \in [p/|V_C|]$, $\tilde{\sigma} + j$ gives the result of a sequence of joins prescribed by a different connected component of G . Because each join was performed at a length- N string, the necklaces $\{\tilde{\sigma} + j : j \in [p/|V_C|]\}$ together have the same occurrence frequency of any length- m string on $[K]$ as does Lempel's lift of $\tilde{\alpha}$ for $m \leq N + 1$. That occurrence frequency is 0 or 1 for $m = N + 1$, as it therefore also is for $m > N + 1$. Because possible joins at strings of the form $(\mathbf{s}_{N-1}^{++}, k)$ or $(k, \mathbf{s}_{N-1}^{++})$ for $s, k \in [K]$ were exhausted by prior joins, Lemma 7 guarantees that joins of the $\{\tilde{\sigma} + j : j \in [p/|V_C|]\}$ at strings of the form \mathbf{s}_{N-1}^{++} for $s \in [K]$ preserve the occurrence frequency of any length- m string on $[K]$ for $m \leq N$ while ensuring that when $m > N$, the occurrence frequency of a length- m

9:12 Arbitrary-Length Analogs to de Bruijn Sequences

string remains either 0 or 1. So when all necklaces in $\{\tilde{\sigma} + j : j \in [p/|V_C|]\}$ are joined as on Lines 21-24, the result is a $P_{K \cdot L}^{(K)}$ -sequence, and it is returned (Line 25). Note the joins are performed in exact analogy to those of Case 2.

The output of LIFTANDJOIN is thus a $P_{K \cdot L}^{(K)}$ -sequence. ◀

Repeated application of LIFTANDJOIN on a $P_L^{(K)}$ -sequence $\tilde{\alpha}$ outputs a $P_L^{(K)}$ -sequence whose length multiplies the length of $\tilde{\alpha}$ by a power of K . But this operation alone does not afford the expressive capacity to build up a $P_L^{(K)}$ -sequence of arbitrary length starting from an $\tilde{\alpha}$ of length less than K , in the same way that an arbitrary positive integer cannot be written as a power of K times a positive integer less than K . A mechanism for extending the length of $\tilde{\alpha}$ by up to $K - 1$ between applications of LIFTANDJOIN is required, where the length of the extension is determined by an appropriate digit from the base- K representation of L . The mechanism used in the iterative procedure GENERATEPKL below, which outputs a $P_L^{(K)}$ -sequence for any combination of $K \geq 2$ and $L \geq 1$, extends a given longest run of a nonzero character by a single character. Theorem 9 proves this approach works.

■ **Algorithm 2** Procedure GENERATEPKL referenced in the text.

```

// Returns a  $P_L^{(K)}$ -sequence on the alphabet  $[K]$  given  $K \geq 2$  and
//  $L \geq 1$  as inputs. Here,  $N := \lceil \log_K L \rceil$ .
1: procedure GENERATEPKL( $K, L$ )
2:   Compute the digits  $\{d_i\}$  of  $L$  in its base- $K$  representation as specified by
      $L = \sum_{i=0}^{N-1} d_i K^{N-i-1}$ .
3:   Initialize the necklace  $\tilde{\alpha}$  to  $\tilde{\mathbf{1}}_{d_0}^{++}$ .
4:   for  $j := 1$  to  $N - 1$  do
5:     Set  $\tilde{\alpha}$  to LIFTANDJOIN( $\tilde{\alpha}, K$ ).
6:     if  $d_j > 0$  then
7:       Set  $\tilde{\alpha}$  to the extension of  $\tilde{\alpha}$  by  $d_j$  characters obtained by replacing a substring
          $\mathbf{k}_{j-1}$  with  $\mathbf{k}_j$  for every  $k \in \{1, \dots, d_j\}$ .
8:     end if
9:   end for
10:  return  $\tilde{\alpha}$ 
11: end procedure

```

► **Theorem 9.** *GENERATEPKL(K, L) outputs a $P_L^{(K)}$ -sequence for any combination of $K \geq 2$ and $L \geq 1$.*

Proof. Use the notation $\tilde{\alpha}_0$ to denote the value of $\tilde{\alpha}$ after Line 3 of GENERATEPKL is executed and the notation $\tilde{\alpha}_j$ to denote the value of $\tilde{\alpha}$ after step j of the **for** loop of GENERATEPKL. Prove the theorem by induction, showing that if $\tilde{\alpha}_{j-1}$ is a $P_{L_{j-1}}^{(K)}$ -sequence of length L_{j-1} , and $\mathbf{0}_m$ occurs $\lfloor L_{j-1}/K^m \rfloor$ times as a substring of $\tilde{\alpha}_{j-1}$ for all $m \leq L_{j-1}$, then $\tilde{\alpha}_j$ is a $P_{L_j}^{(K)}$ -sequence of length $L_j = K \cdot L_{j-1} + d_j$, and $\mathbf{0}_n$ occurs $\lfloor L_j/K^n \rfloor$ times as a substring of $\tilde{\alpha}_j$ for all $n \leq L_j$. The base case for the induction holds: $\tilde{\alpha}_0$, as initialized on Line 3, is the $P_{L_0}^{(K)}$ -sequence $\tilde{\mathbf{1}}_{d_0}^{++}$ of length $L_0 = d_0$, in which $\mathbf{0}_m$ occurs as a substring $\lfloor d_0/K^m \rfloor = 0$ times for $1 \leq m \leq d_0$ and $\lfloor d_0/K^m \rfloor = d_0$ times for $m = 0$. Now suppose that $\tilde{\alpha}_{j-1}$ is a $P_{L_{j-1}}^{(K)}$ -sequence of length L_{j-1} , and $\mathbf{0}_m$ occurs $\lfloor L_{j-1}/K^m \rfloor$ times as a substring of $\tilde{\alpha}_{j-1}$ for all $m \leq L_{j-1}$. Then for every $k \in [K]$, \mathbf{k}_{m+1} occurs $\lfloor L_{j-1}/K^m \rfloor = \lfloor (K \cdot L_{j-1})/K^{m+1} \rfloor$ times as a substring of LIFTANDJOIN($\tilde{\alpha}_{j-1}, K$), obtained on Line 5. This follows from

1. Lemma 5, which says there are t occurrences of $\mathbf{0}_m$ as a substring of a necklace if and only if there are t occurrences of \mathbf{k}_{m+1} as a substring in Lempel's lift of that necklace, and
2. how all joins of necklaces in Lempel's lift prescribed by LIFTANDJOIN, including those permitted by Lemma 7, do not affect occurrences of substrings of the form \mathbf{k}_{m+1} .

The extension performed on Line 7 increases the number of occurrences of \mathbf{k}_m , for $k = 1, 2, \dots, d_j$, from $\lfloor (K \cdot L_{j-1})/K^{m+1} \rfloor$ to $\lceil (K \cdot L_{j-1})/K^{m+1} \rceil$ without affecting the numbers of occurrences of any other length- m strings as substrings for $m \leq j$. The longest string of 0s is never extended, and the number of occurrences of $\mathbf{0}_n$ remains $\lfloor (K \cdot L_{j-1})/K^{n+1} \rfloor$ for all $n \leq L_j$. So the resulting necklace $\tilde{\alpha}_j$ is a $P_{L_j}^{(K)}$ -sequence of length $L_j = K \cdot L_{j-1} + d_j$, and $\mathbf{0}_n$ occurs $\lfloor L_j/K^n \rfloor$ times as a substring of $\tilde{\alpha}_j$ for all $n \leq L_j$.

The **for** loop thus encodes the recursion

$$L_j = K \cdot L_{j-1} + d_j \quad j \in [N-1] \setminus \{0\} \quad (8)$$

with initial condition $L_0 = d_0$. The formula $L = L_{N-1} = \sum_{i=0}^{N-1} d_i K^{N-i-1}$ follows, concluding the proof. \blacktriangleleft

In the binary case, GENERATEPKL and the joins it requires of its subroutine LIFTANDJOIN collapse to a particularly simple algorithm, which is given in the procedure GENERATEP2L below.

■ **Algorithm 3** Procedure GENERATEP2L referenced in the text.

```

// Returns a  $P_L^{(2)}$ -sequence on the alphabet  $\{0,1\}$  given  $L \geq 1$  as
// an input. Here,  $N := \lceil \log_2 L \rceil$ .
1: procedure GENERATEP2L( $L$ )
2:   Compute the digits  $\{d_i\}$  of  $L$  in its binary representation as specified by  $L = \sum_{i=0}^{N-1} d_i 2^{N-i-1}$ .
3:   Initialize the necklace  $\tilde{\alpha}$  to the single character 1.
4:   for  $j := 1$  to  $N-1$  do
5:     Construct Lempel's lift  $\{\tilde{\lambda}_i : i \in [p]\}$  of  $\tilde{\alpha}$ .
6:     if  $p = 1$  then
7:       //  $\tilde{\alpha}$  has an odd number of 1s.
8:       Set  $\tilde{\alpha}$  to  $\tilde{\lambda}_0$ .
9:     else if  $\mathbf{1}_{j-1}$  is a substring of  $\tilde{\alpha}$  then
10:      Set  $\tilde{\alpha}$  to the result of joining  $\tilde{\lambda}_0$  and  $\tilde{\lambda}_1$  at  $\mathbf{0}_{j-1}^{++}$ .
11:     else if  $\tilde{\lambda}_0$  and  $\tilde{\lambda}_1$  can be joined at  $(\mathbf{0}_{j-2}^{++}, k)$  or  $(k, \mathbf{0}_{j-2}^{++})$  for  $k \in [2]$  then
12:      Set  $\tilde{\alpha}$  to the result of joining  $\tilde{\lambda}_0$  and  $\tilde{\lambda}_1$  at  $(\mathbf{0}_{j-2}^{++}, k)$  or  $(k, \mathbf{0}_{j-2}^{++})$  for  $k \in [2]$ .
13:     else
14:      Set  $\tilde{\alpha}$  to the result of joining  $\tilde{\lambda}_0$  and  $\tilde{\lambda}_1$  at  $\mathbf{0}_{j-2}^{++}$ .
15:     end if
16:     if  $d_j = 1$  then
17:       Set  $\tilde{\alpha}$  to the extension of  $\tilde{\alpha}$  by a single character obtained by replacing  $\mathbf{1}_{j-1}$  with  $\mathbf{1}_j$ .
18:     end if
19:   end for
20: return  $\tilde{\alpha}$ 
end procedure

```

Below is the final theorem of this paper, which proves complexity results.

► **Theorem 10.** *GENERATEPKL outputs a $P_L^{(K)}$ -sequence in $O(L)$ time using $O(L \log K)$ space.*

Proof. The space required by GENERATEPKL is dominated by storage of the final $P_L^{(K)}$ -sequence itself, which is $O(L \log K)$.

To see why the algorithm takes $O(L)$ time, consider first the case $L < K$. GENERATEPKL then initializes $\tilde{\alpha}$ to the positive integers in order up to and including L (Line 3), which scales as L . It subsequently skips the **for** loop and returns $\tilde{\alpha}$.

Now consider the opposite case $L \geq K$. Expressing L in base K (Line 2) scales as $\log_K L$, and initializing $\tilde{\alpha}$ (Line 3) scales as K . Focus on Line 5's call of LIFTANDJOIN at step j of the **for** loop, where the length- L_{j-1} necklace $\tilde{\alpha}_{j-1}$ is passed to LIFTANDJOIN in the notation of Theorem 9's proof. Constructing Lempel's lift of $\tilde{\alpha}_{j-1}$ (Line 2 of LIFTANDJOIN) scales as $K \cdot L_{j-1}$, the total length of the necklaces constructed. Addressing Case 1 (Lines 3-5) takes constant time. Addressing Case 2 (Lines 6-13) involves searching $\tilde{\alpha}_{j-1}$ for $\mathbf{1}_N$, which scales as L_{j-1} , and successively joining the necklaces comprising Lempel's lift, which scales as $K \cdot L_{j-1}$ if implemented as, e.g., a sequence of rotations and concatenations in which indexes of join substrings are tracked. Addressing Case 3 in its entirety (Lines 14-24) involves (1) constructing the join graph G , which is dominated by the $K \cdot L_{j-1}$ scaling of searching Lempel's lift for strings of the form s_{N-1}^{++} for $s \in [K]$, (2) performing a depth-first traversal of a connected component of the join graph, which takes time linear in a number of at most K vertices, and (3) joining necklaces, which also scales as $K \cdot L_{j-1}$. So LIFTANDJOIN

is dominated by a $K \cdot L_{j-1}$ scaling at step j of the **for** loop of GENERATEPKL. Refocusing on GENERATEPKL, extending LIFTANDJOIN($\tilde{\alpha}_{j-1}, K$) (Line 7) involves searching for longest runs of the same character and inserting characters as necessary, scaling as $K \cdot L_{j-1}$ if performed in one pass through the necklace. Therefore, step j of the **for** loop scales as $K \cdot L_{j-1}$, and from the recursion (8), executing all iterations of the **for** loop scales as L . The time taken by the **for** loop dominates that of Lines 2 and 3. So the overall scaling is L for the two cases $L \geq K$ and $L < K$, and GENERATEPKL takes $O(L)$ time. ◀

3 Discussion

In this paper, we have introduced $P_L^{(K)}$ -sequences as arbitrary-length analogs to de Bruijn sequences. We have shown by explicit construction that a $P_L^{(K)}$ -sequence exists for any combination of $K \geq 2$ and $L \geq 1$, giving an $O(L)$ -time, $O(L \log K)$ -space algorithm extending Lempel's recursive construction of a binary de Bruijn sequence. An implementation of the algorithm in Python is available at <https://github.com/nelloreward/pkl>.

We conclude with several open questions suggested by our work:

1. What is the number of distinct $P_L^{(K)}$ -sequences on \mathcal{A} for every possible combination of K and L ? As Gabric, Holub, and Shallit did in [32, 37] for generalized de Bruijn sequences, we have counted $P_L^{(2)}$ -sequences for L up to 32 by exhaustive search. Table 1 displays our results, which can be reproduced using code at <https://github.com/nelloreward/pkl>. Note the counts do not increase monotonically with L .
2. Can the algorithm for $P_L^{(K)}$ -sequence generation presented here or a variant be encoded in a shift rule? This would reduce the space it requires, perhaps at the expense of performance. An obstacle to deriving a shift rule from the algorithm that works for all values of L at a given alphabet size K is that it would have to account for cases like those of LIFTANDJOIN. See [61, 76, 2] for work along the lines of mathematically unrolling Lempel's recursion and generalizations.
3. Are there elegant constructions of $P_L^{(K)}$ -sequences for any possible combination of K and L that extend constructions of de Bruijn sequences besides Lempel's recursive construction? There is a considerable body of literature on constructing universal cycles. (See, e.g., [46, 48, 71, 72, 83, 36, 75]). Introduced by Chung, Diaconis, and Graham in [14], a *universal cycle* is a length- L necklace in which every string in a size- L set S of length- m strings occurs as a substring. It is possible a set S curated to ensure the universal cycle is a $P_L^{(K)}$ -sequence is compatible with existing universal cycle constructions or extensions.
4. Can an efficiently decodable $P_L^{(K)}$ -sequence be constructed for any possible combination of K and L ? Toward answering this question, it may be worth further investigating the efficient decoding of Lempel's recursive construction of a de Bruijn sequence. (See [64] for the binary case and [81] for the K -ary case.) Other efficiently decodable constructions of de Bruijn sequences are given in [50, 70].
5. What other properties that can be exhibited by a necklace are preserved under Lempel's D-morphism, and how can they be exploited to recursively construct other useful sequences? While this work was being prepared, Mitchell and Wild posted [60] to arXiv, which shows binary orientable sequences can be constructed recursively using Lempel's D-morphism. An *orientable sequence* is a necklace \tilde{v} for which each length- n substring has precisely one occurrence in precisely one of \tilde{v} and the reverse of \tilde{v} [9, 15]. It is perhaps unsurprising that Lempel's D-morphism, a kind of derivative, is so versatile and that orientability, $P_L^{(K)}$ -sequence composition, and efficient decodability can be preserved by its inverse.

■ **Table 1** Numbers of distinct $P_L^{(2)}$ sequences on \mathcal{A} for various values of L .

L	Number of distinct $P_L^{(2)}$ -sequences	L	Number of distinct $P_L^{(2)}$ -sequences
1	2	17	32
2	1	18	36
3	2	19	68
4	1	20	57
5	2	21	138
6	3	22	123
7	4	23	252
8	2	24	378
9	4	25	504
10	3	26	420
11	6	27	1296
12	9	28	1520
13	12	29	2176
14	20	30	2816
15	32	31	4096
16	16	32	2048

References

- 1 Abbas Alhakim and Mufutau Akinwande. A recursive construction of nonbinary de Bruijn sequences. *Designs, Codes and Cryptography*, 60(2):155–169, 2011.
- 2 Abbas Alhakim and Maher Nouiehed. Stretching de Bruijn sequences. *Designs, Codes and Cryptography*, 85(2):381–394, 2017.
- 3 Abbas Alhakim, Evan Sala, and Joe Sawada. Revisiting the prefer-same and prefer-opposite de Bruijn sequence constructions. *Theoretical Computer Science*, 852:73–77, 2021.
- 4 Abbas M Alhakim. A simple combinatorial algorithm for de Bruijn sequences. *The American Mathematical Monthly*, 117(8):728–732, 2010.
- 5 Gal Amram, Yair Ashlagi, Amir Rubin, Yotam Svoray, Moshe Schwartz, and Gera Weiss. An efficient shift rule for the prefer-max de Bruijn sequence. *Discrete Mathematics*, 342(1):226–232, 2019.
- 6 Fred S. Annexstein. Generating de Bruijn sequences: An efficient implementation. *IEEE Transactions on Computers*, 46(2):198–200, 1997.
- 7 Charles Philip Brown. *Sanskrit Prosody and Numerical Symbols Explained*. Trübner & Company, 1869.
- 8 PR Bryant, FG Heath, and RD Killick. Counting with feedback shift registers by means of a jump technique. *IRE Transactions on Electronic Computers*, pages 285–286, 1962.
- 9 John Burns and Chris J Mitchell. *Coding Schemes for Two-Dimensional Position Sensing*. Hewlett-Packard Laboratories, Technical Publications Department, 1992.
- 10 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.
- 11 Camille Flye Sainte-Marie. <http://henripoincare.fr/s/correspondance/item/14609>. Accessed: 2021-07-23.
- 12 Taejoo Chang, Bongjoo Park, Yun Hee Kim, and Iickho Song. An efficient implementation of the D-homomorphism for generation of de Bruijn sequences. *IEEE Transactions on Information Theory*, 45(4):1280–1283, 1999.

9:16 Arbitrary-Length Analogs to de Bruijn Sequences

- 13 Zuling Chang, Martianus Frederic Ezerman, Pinhui Ke, and Qiang Wang. General criteria for successor rules to efficiently generate binary de Bruijn sequences. *arXiv preprint*, 2019. [arXiv:1911.06670](https://arxiv.org/abs/1911.06670).
- 14 Fan Chung, Persi Diaconis, and Ron Graham. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110(1-3):43–59, 1992.
- 15 ZD Dai, KM Martin, MJB Robshaw, and PR Wild. Orientable sequences. In *Institute of Mathematics and Its Applications Conference Series*, volume 45, pages 97–97. Oxford University Press, 1993.
- 16 NG de Bruijn. In memoriam T. van Aardenne-Ehrenfest, 1905-1984. *Nieuw Archief voor Wiskunde*, 4(2):235–236, 1985.
- 17 Nicolaas Govert De Bruijn. A combinatorial problem. In *Proc. Koninklijke Nederlandse Academie van Wetenschappen*, volume 49, pages 758–764, 1946.
- 18 Nicolaas Govert de Bruijn. Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of 2^n zeros and ones that show each n -letter word exactly once. *EUT report. WSK, Dept. of Mathematics and Computing Science*, 75, 1975.
- 19 Nicolaas Govert de Bruijn and Tanja van Aardenne-Ehrenfest. Circuits and trees in oriented linear graphs. *Simon Stevin*, 28:203–217, 1951.
- 20 De Bruijn sequence and universal cycle constructions. <https://debruijnsequence.org/>. Accessed: 2021-07-24.
- 21 Patrick Baxter Dragon, Oscar I Hernandez, Joe Sawada, Aaron Williams, and Dennis Wong. Constructing de Bruijn sequences with co-lexicographic order: The k -ary grandmama sequence. *European Journal of Combinatorics*, 72:1–11, 2018.
- 22 Patrick Baxter Dragon, Oscar I Hernandez, and Aaron Williams. The grandmama de Bruijn sequence for binary strings. In *LATIN 2016: Theoretical Informatics*, pages 347–361. Springer, 2016.
- 23 Cornelius Eldert, HM Gurk, HJ Gray, and Morris Rubinoff. Shifting counters. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 77(1):70–74, 1958.
- 24 Tuvi Etzion. An algorithm for constructing m -ary de Bruijn sequences. *Journal of algorithms*, 7(3):331–340, 1986.
- 25 Tuvi Etzion. An algorithm for generating shift-register cycles. *Theoretical computer science*, 44:209–224, 1986.
- 26 Tuvi Etzion and Abraham Lempel. Algorithms for the generation of full-length shift-register sequences. *IEEE Transactions on Information Theory*, 30(3):480–484, 1984.
- 27 LR Ford Jr. A cyclic arrangement of m -tuples. *Report No. P-1071, RAND Corp*, 1957.
- 28 Harold Fredricksen. A class of nonlinear de Bruijn cycles. *Journal of Combinatorial Theory, Series A*, 19(2):192–199, 1975.
- 29 Harold Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *SIAM review*, 24(2):195–221, 1982.
- 30 Harold Fredricksen and Irving J Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete mathematics*, 61(2-3):181–188, 1986.
- 31 Harold Fredricksen and James Maiorana. Necklaces of beads in k colors and k -ary de Bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978.
- 32 Daniel Gabric, Štěpán Holub, and Jeffrey Shallit. Generalized de Bruijn words and the state complexity of conjugate sets. In *International Conference on Descriptive Complexity of Formal Systems*, pages 137–146. Springer, 2019.

- 33 Daniel Gabric and Joe Sawada. A de Bruijn sequence construction by concatenating cycles of the complemented cycling register. In *International Conference on Combinatorics on Words*, pages 49–58. Springer, 2017.
- 34 Daniel Gabric and Joe Sawada. Constructing de Bruijn sequences by concatenating smaller universal cycles. *Theoretical Computer Science*, 743:12–22, 2018.
- 35 Daniel Gabric, Joe Sawada, Aaron Williams, and Dennis Wong. A framework for constructing de Bruijn sequences via simple successor rules. *Discrete Mathematics*, 341(11):2977–2987, 2018.
- 36 Daniel Gabric, Joe Sawada, Aaron Williams, and Dennis Wong. A successor rule framework for constructing k -ary de Bruijn sequences and universal cycles. *IEEE Transactions on Information Theory*, 66(1):679–687, 2019.
- 37 Daniel Gabric, Štěpán Holub, and Jeffrey Shallit. Maximal state complexity and generalized de Bruijn words. *Information and Computation*, page 104689, 2021. doi:10.1016/j.ic.2021.104689.
- 38 R Games. A generalized recursive construction for de Bruijn sequences. *IEEE transactions on information theory*, 29(6):843–850, 1983.
- 39 Solomon W Golomb. *Shift Register Sequences: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models*. World Scientific, 2017.
- 40 Solomon W Golomb, Lloyd R Welch, and Richard M Goldstein. Cycles from nonlinear shift registers. Technical report, Jet Propulsion Lab, Pasadena, CA, 1959.
- 41 Aysu Gündoğan, Ben Cameron, and Joe Sawada. Cut-down de Bruijn sequences, 2021. Unpublished manuscript.
- 42 FG Heath and MW Gribble. Chain codes and their electronic applications. *Proceedings of the IEE-Part C: Monographs*, 108(13):50–57, 1961.
- 43 Farhad Hemmati and Daniel J Costello. An algebraic construction for q -ary shift register sequences. *IEEE Transactions on Computers*, 100(12):1192–1195, 1978.
- 44 Patricia Hersh, Thomas Lam, Pavlo Pylyavskyy, and Victor Reiner. *The Mathematical Legacy of Richard P. Stanley*, volume 100. American Mathematical Soc., 2016.
- 45 Yuejiang Huang. A new algorithm for the generation of binary de Bruijn sequences. *Journal of Algorithms*, 11(1):44–51, 1990.
- 46 Glenn Hurlbert and Garth Isaak. Equivalence class universal cycles for permutations. *Discrete Mathematics*, 149(1-3):123–129, 1996.
- 47 Cees JA Jansen, Wouter G Franx, and Dick E Boekee. An efficient algorithm for the generation of DeBruijn cycles. *IEEE Transactions on Information Theory*, 37(5):1475–1478, 1991.
- 48 J Robert Johnson. Universal cycles for permutations. *Discrete Mathematics*, 309(17):5264–5270, 2009.
- 49 Subhash Kak. Yamatarajabhanasalam: An interesting combinatoric sutra. *Indian Journal of History of Science*, 35(2):123–128, 2000.
- 50 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient ranking of Lyndon words and decoding lexicographically minimal de Bruijn sequence. *SIAM Journal on Discrete Mathematics*, 30(4):2027–2046, 2016.
- 51 Nikolai Mikhailovich Korobov. Trigonometric sums with exponential functions and the distribution of signs in repeating decimals. *Mathematical notes of the Academy of Sciences of the USSR*, 8(5):831–837, 1970.
- 52 Nikolai Mikhailovich Korobov. On the distribution of digits in periodic fractions. *Mathematics of the USSR-Sbornik*, 18(4):659, 1972.

- 53 Charles-Ange Laisant and Emile Michel Hyacinthe Lemoine. *L'Intermédiaire des Mathématiciens*, volume 1. Gauthier-Villars et Fils, 1894.
- 54 Max Landsberg. Feedback functions for generating cycles over a finite alphabet. *Discrete Mathematics*, 219(1-3):187–194, 2000.
- 55 Abraham Lempel. On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers. *IEEE Transactions on Computers*, 100(12):1204–1209, 1970.
- 56 Abraham Lempel. m -ary closed sequences. *Journal of Combinatorial Theory, Series A*, 10(3):253–258, 1971.
- 57 Willem Mantel. Resten van wederkerige reeksen. *Nieuw Archief voor Wiskunde*, 1:172–184, 1897.
- 58 Monroe H Martin. A problem in arrangements. *Bulletin of the American Mathematical Society*, 40(12):859–864, 1934.
- 59 Chris J Mitchell, Tuvi Etzion, and Kenneth G Paterson. A method for constructing decodable de Bruijn sequences. *IEEE Transactions on Information Theory*, 42(5):1472–1478, 1996.
- 60 Chris J. Mitchell and Peter B. Wild. Constructing orientable sequences. *arXiv preprint*, 2021. [arXiv:2108.03069](https://arxiv.org/abs/2108.03069).
- 61 Johannes Mykkeltveit, Man-Keung Siu, and Po Tong. On the cycle structure of some nonlinear shift register sequences. *Information and control*, 43(2):202–215, 1979.
- 62 Abhinav Nellore, Austin Nguyen, and Reid F. Thompson. An invertible transform for efficient string matching in labeled digraphs. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:14, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2021.20.
- 63 Nicolaas de Bruijn - biography. https://mathshistory.st-andrews.ac.uk/Biographies/De_Bruijn/. Accessed: 2021-07-23.
- 64 Kenneth G Paterson and Matthew JB Robshaw. Storage efficient decoding for a class of binary de Bruijn sequences. *Discrete mathematics*, 138(1-3):327–341, 1995.
- 65 AN Radchenko. *Code Rings and Their Use in Contactless Coding Devices*. PhD thesis, University of Leningrad, USSR, 1958.
- 66 AN Radchenko and VI Filippov. Shifting registers with logical feedback and their use as counting and coding devices. *Automation and Remote Control (English translation of Soviet Journal Automatika i Telemekhanika)*, 20:1467–1473, November 1959.
- 67 Christian Ronse. Feedback shift registers. *Lecture Notes in Computer Science*, 169, 1984.
- 68 Frank Ruskey, Carla Savage, and Terry Min Yih Wang. Generating necklaces. *Journal of Algorithms*, 13(3):414–430, 1992.
- 69 C Flye Saint-Marie. Solution to question nr. 48. *L'Intermédiaire des Mathématiciens*, 1894.
- 70 Joe Sawada and Aaron Williams. Practical algorithms to rank necklaces, Lyndon words, and de Bruijn sequences. *Journal of Discrete Algorithms*, 43:95–110, 2017.
- 71 Joe Sawada, Aaron Williams, and Dennis Wong. The lexicographically smallest universal cycle for binary strings with minimum specified weight. *Journal of Discrete Algorithms*, 28:31–40, 2014.
- 72 Joe Sawada, Aaron Williams, and Dennis Wong. Generalizing the classic greedy and necklace constructions of de Bruijn sequences and universal cycles. *the electronic journal of combinatorics*, pages P1–24, 2016.
- 73 Joe Sawada, Aaron Williams, and Dennis Wong. A surprisingly simple de Bruijn sequence construction. *Discrete Mathematics*, 339(1):127–131, 2016.

- 74 Joe Sawada, Aaron Williams, and Dennis Wong. A simple shift rule for k -ary de Bruijn sequences. *Discrete Mathematics*, 340(3):524–531, 2017.
- 75 Joe Sawada and Dennis Wong. Efficient universal cycle constructions for weak orders. *Discrete Mathematics*, 343(10):112022, 2020.
- 76 Man-Keung Siu and Po Tong. Generation of some de Bruijn sequences. *Discrete Mathematics*, 31(1):97–100, 1980.
- 77 R Stoneham. On (j, ε) -normality in the rational fractions. *Acta Arithmetica*, 16:221–238, 1970.
- 78 R Stoneham. On absolute (j, ε) -normality in the rational fractions with applications to normal numbers. *Acta Arithmetica*, 22:277–286, 1973.
- 79 R Stoneham. Normal recurring decimals, normal periodic systems, (j, ε) -normality, and normal numbers. *Acta Arithmetica*, 28(4):349–361, 1976.
- 80 RG Stoneham. The reciprocals of integral powers of primes and normal numbers. *Proceedings of the American Mathematical Society*, 15(2):200–208, 1964.
- 81 Jonathan Tuliani. De Bruijn sequences with efficient decoding algorithms. *Discrete Mathematics*, 226(1-3):313–336, 2001.
- 82 Srisa Chandra Vasu et al. *The Ashtadhyayi of Panini*, volume 6. Satyajnan Chatterji, 1897.
- 83 Dennis Wong. A new universal cycle for permutations. *Graphs and Combinatorics*, 33(6):1393–1399, 2017.
- 84 Jun-Hui Yang and Zong-Duo Dai. Construction of m -ary de Bruijn sequences. In *International Workshop on the Theory and Application of Cryptographic Techniques*, pages 357–363. Springer, 1992.
- 85 Michael Yoeli. *Nonlinear Feedback Shift Registers*. International Business Machines Corporation, Development Laboratories, Data Systems Division, 1961.
- 86 Michael Yoeli. Binary ring sequences. *The American Mathematical Monthly*, 69(9):852–855, 1962.
- 87 Michael Yoeli. Counting with nonlinear binary feedback shift registers. *IEEE Transactions on Electronic Computers*, pages 357–361, 1963.
- 88 Yunlong Zhu, Zuling Chang, Martianus Frederic Ezerman, and Qiang Wang. An efficiently generated family of binary de Bruijn sequences. *Discrete Mathematics*, 344(6):112368, 2021.

A Appendix: A brief history of de Bruijn sequences

The earliest known recorded de Bruijn sequence is the Sanskrit sutra *yamātārājabhānasalagām*, a mnemonic encoding all possible length-3 combinations of short and long vowels [49]. Historians have had some trouble placing when it was first conceived, but it may be over 2,500 years old [7], having appeared in work by the ancient Indian scholar Pāṇini (dates of birth and death unavailable). Little is known of Pāṇini beyond his foundational work *Aṣṭādhyāyī* codifying Sanskrit grammatical structure [82].

The question of whether and how many binary de Bruijn sequences of every order exist was first posed by A. de Rivière (dates of birth and death unavailable) as problem 48 of [53] in 1894. That same year, in response to the problem, the number of binary de Bruijn sequences of every order was counted in [69] by Camille Flye Sainte-Marie (1834–1926), a member of the Mathematical Society of France who was affiliated with the French military throughout his career [11]. His work was quickly forgotten.

Monroe Harnish Martin (1907–2007) was first to prove the existence of de Bruijn sequences of any order for any alphabet size in his 1934 paper [58] by explicit construction, shortly before arriving at the University of Maryland, where he spent the rest of his eminent career. Without knowing of Sainte-Marie’s work, Nicolaas Govert de Bruijn (1918–2012) [63] also counted the number of binary de Bruijn sequences of every order in his 1946 work [17]. Tatyana van Aardenne-Ehrenfest and de

9:20 Arbitrary-Length Analogs to de Bruijn Sequences

Bruijn were first to prove the formula for the number of de Bruijn sequences of any order for any alphabet size in their 1951 paper [19]. It is notable that after receiving her PhD from the University of Leiden in 1931, van Ardenne-Ehrenfest (1905–1984) made this and further significant contributions to the mathematics of sequences despite never holding paid employment as a mathematician and working as a homemaker [16].

Sainte-Marie's work was ultimately rediscovered by the well-known MIT combinatorialist Richard Peter Stanley (1944–) [44], who brought it to the attention of de Bruijn, and in 1975, de Bruijn issued an acknowledgement [18] of the work. In this acknowledgement, de Bruijn noted that as early as 1897, Willem Mantel (dates of birth and death unavailable) showed how to construct de Bruijn sequences of any order for any alphabet size that is prime [57], also in response to A. de Rivière's problem.

Partial Permutations Comparison, Maintenance and Applications

Avivit Levy  

Department of Software Engineering, Shenkar College, Ramat-Gan, Israel

Ely Porat 

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

B. Riva Shalom 

Department of Software Engineering, Shenkar College, Ramat-Gan, Israel

Abstract

This paper focuses on the concept of *partial permutations* and their use in algorithmic tasks. A *partial permutation* over Σ is a bijection $\pi_{par} : \Sigma_1 \mapsto \Sigma_2$ mapping a subset $\Sigma_1 \subset \Sigma$ to a subset $\Sigma_2 \subset \Sigma$, where $|\Sigma_1| = |\Sigma_2|$ ($|\Sigma|$ denotes the size of a set Σ). Intuitively, two partial permutations *agree* if their mapping pairs do not form *conflicts*. This notion, which is formally defined in this paper, enables a consistent as well as informatively rich comparison between partial permutations. We formalize the *Partial Permutations Agreement* problem (PPA), as follows. Given two sets A_1, A_2 of partial permutations over alphabet Σ , each of size n , output all pairs (π_i, π_j) , where $\pi_i \in A_1, \pi_j \in A_2$ and π_i *agrees* with π_j . The possibility of having a data structure for efficiently maintaining a dynamic set of partial permutations enabling to retrieve agreement of partial permutations is then studied, giving both negative and positive results. Applying our study enables to point out fruitful versus futile methods for efficient genes sequences comparison in database or automatic color transformation data augmentation technique for image processing through neural networks. It also shows that an efficient solution of strict Parameterized Dictionary Matching with One Gap (PDMOG) over general dictionary alphabets is not likely, unless the Strong Exponential Time Hypothesis (SETH) fails, thus negatively answering an open question posed lately.

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

Keywords and phrases Partial permutations, Partial words, Genes comparison, Color transformation, Dictionary matching with gaps, Parameterized matching, SETH hypothesis

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.10

Acknowledgements We would like to thank Amihod Amir for his valuable suggestions while reading a former version of the paper.

1 Introduction

Permutations are a classical mathematical concept widely used in computer science: playing a role in analyzing sorting algorithms [29], being a basic building block in randomization [19], and appearing in various fields, such as Computational Biology (e.g. [8, 24]), Pattern Matching (e.g. [4, 32]), Cryptography (e.g. [21]), and more. A permutation over an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$ is a bijection $\pi : \Sigma \mapsto \Sigma$ mapping every symbol $\sigma_i \in \Sigma$ to a distinct symbol $\sigma_j \in \Sigma$ (where it may be that $i = j$). In this paper, we focus on the concept of *partial permutations* and their use in algorithmic tasks.

A *partial permutation* over Σ is a bijection $\pi_{par} : \Sigma_1 \mapsto \Sigma_2$ mapping a subset $\Sigma_1 \subset \Sigma$ to a subset $\Sigma_2 \subset \Sigma$, where the sizes of the sets Σ_1, Σ_2 are equal, i.e., $|\Sigma_1| = |\Sigma_2|$.¹ Partial permutations are closely related to *partial words*, defined as follows. A *partial word* over Σ

¹ The subscript par is only used in this paragraph to distinguish a partial permutation from a permutation, however, throughout the paper we omit it for convenience and denote a partial permutation by π .



is a word (string) over the alphabet $\Sigma \cup \{\diamond\}$, where the symbol \diamond is treated as a *hole*². In the study of partial words, the holes are usually treated as gaps that may be filled by an arbitrary letter of Σ . Note that, a partial permutation is a partial word π such that each symbol of Σ appears in π exactly once, and all the remaining symbols of π are holes [17].

The study of partial words was initiated by [11, 34] for comparing genes, where alignment can be viewed as a construction of two partial words that are compatible in the sense defined in [11]. However, for the task of comparing genes sequences, partial permutations were suggested as an appropriate model due to diversity of genes and the incompleteness nature of such sequences [45]. Partial permutations play a role also in computational tasks other than computational biology. For example, it can be used for representing color transformations as a data augmentation technique in image processing through neural networks [26, 37]. In addition, in pattern matching algorithms strings may be mapped to other strings, as in the well-known parameterized matching and related problems [9, 35, 40].

Combinatorial aspects of partial words that have been studied include periods in partial words [11, 41], avoidability/unavoidability of sets of partial words [12, 13], squares in partial words [22], and overlap-freeness [23]. Combinatorial questions regarding partial permutations were also studied, e.g., pattern avoidance [17], enumeration [42, 31] or restricted forms [17, 14].

In this paper, we study algorithmic aspects of maintaining partial permutations. To this end, we next discuss the basic operation of comparing partial permutations, formally define the concept of their *agreement* and describe a condition on partial permutations representations that naturally perceive the agreement between two partial permutations.

1.1 Partial Permutations Comparison

Let R be any representation of a *permutation* π over an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$. Since R represents the bijection where the domain and codomain of π are identical, it should only specify the mapped pairs of symbols. Then, it holds that $R(\pi_1) = R(\pi_2)$ if and only if the permutations π_1 and π_2 are equal. We refer to this property as *the comparison axiom*.

Assume any representation R of a *partial permutation*. Since R represents a bijection having non-obvious domain and codomain, it should specify the domain and codomain sets of π (denoted by D_R and C_R , respectively) as well as the set of symbols pairs mapped by the bijection (denoted by M_R). A comparison of partial permutations based on such a representation R is more complicated. We may wish to know if two partial permutations are identical and enforce their representations to be equal, but then we put a rigid limitation on our notion of comparison. Considering the nature of partial permutations, we may rather prefer a way to compare the “agreement” between two given partial permutations. Formally,

► **Definition 1** (Conflict and Agreement of Partial Permutations). *Two partial permutations π_1, π_2 are conflicting (alternatively, contain a conflict) if either:*

1. *There exist $\sigma_i, \sigma_j \in \Sigma_2$, $\sigma_i \neq \sigma_j$, such that there exists $\sigma_k \in \Sigma_1$ where $\pi_1(\sigma_k) = \sigma_i$ and $\pi_2(\sigma_k) = \sigma_j$, or*
2. *There exist $\sigma_i, \sigma_j \in \Sigma_1$, $\sigma_i \neq \sigma_j$, such that there exists $\sigma_k \in \Sigma_2$ where $\pi_1(\sigma_i) = \pi_2(\sigma_j) = \sigma_k$.*

We say that π_1, π_2 agree if they do not contain any conflict.

² The hole symbol \diamond is not treated as a don't care symbol as is common in pattern matching, but rather as a don't know symbol.

Definition 1 enables to establish a comparison between two given partial permutations aimed at revealing whether they agree or not. Since the representation is the key to the comparison process, we have to take it into account. Note, however, that Definition 1 is independent of the chosen representation. We may, therefore, derive from it a universal condition on any representation enabling comparison of agreement between partial permutations.

► **Lemma 2** (A Universal Condition of Partial Permutations Agreement). *Let R be any representation of partial permutation bijections, and let π_1, π_2 be two partial permutations. Then, π_1, π_2 agree if and only if the following conditions hold on $R(\pi_1) = \langle D_R(\pi_1), C_R(\pi_1), M_R(\pi_1) \rangle, R(\pi_2) = \langle D_R(\pi_2), C_R(\pi_2), M_R(\pi_2) \rangle$:*

1. *For every $\sigma_k \in D_R(\pi_1) \cap D_R(\pi_2)$ and $\sigma_i \in C_R(\pi_1)$, if $(\sigma_k, \sigma_i) \in M_R(\pi_1)$ then $\sigma_i \in C_R(\pi_2)$ and $(\sigma_k, \sigma_i) \in M_R(\pi_2)$.*
2. *For every $\sigma_k \in C_R(\pi_1) \cap C_R(\pi_2)$ and $\sigma_i \in D_R(\pi_1)$, if $(\sigma_i, \sigma_k) \in M_R(\pi_1)$ then $\sigma_i \in D_R(\pi_2)$ and $(\sigma_i, \sigma_k) \in M_R(\pi_2)$.*

Proof. Obviously, the first condition of the lemma avoids a conflict of the first type of Definition 1, and the second condition avoids a conflict of the second type in Definition 1. ◀

► **Example.** *Let $\Sigma = \{0, 1, 2, 3\}$, $D_R(\pi_1) = \{1, 2\}$, $C_R(\pi_1) = \{0, 3\}$, $M_R(\pi_1) = \{(1, 3), (2, 0)\}$, $D_R(\pi_2) = \{0, 1\}$, $C_R(\pi_2) = \{1, 3\}$, $M_R(\pi_2) = \{(0, 1), (1, 3)\}$, then π_i agrees with π_j .*

The following classification of partial permutations representations will be useful for the discussion of algorithms complexity.

► **Definition 3** (Good Representation). *A representation R for partial permutations is called a good representation if, assuming word-RAM model, for every π_i ,*

1. *the size of $R(\pi_i)$ is $O(|\Sigma|)$, and*
2. *for every π_j , determining whether the universal agreement condition between $R(\pi_i)$ and $R(\pi_j)$ holds can be done in $O(|\Sigma|)$ time.*

In the paper, we discuss the existence of a data structure for maintaining a dynamic set of partial permutations supporting the operations of insert, delete and search agreement with the query over the set. A *static* partial permutations set situation is first investigated and then supporting a *dynamic* set is referred to. We employ a fine-grained complexity analysis, which have recently become an important tool (e.g., in [28, 20, 25]).

1.2 Fine-Grained Complexity Analysis

In traditional computer science theory, the typical problems considered “hard” are \mathcal{NP} -Hard and maybe even require exponential time to solve. Problems having polynomial time algorithms are considered “easy”. The best known algorithms for many such “easy” problems have high run-times, thus, are impractical, and their improvement has been a longstanding open problem with little to no progress. It may be that these algorithms are optimal, however, deriving unconditional lower bounds seems beyond current techniques.

A new, conditional theory of hardness has recently been developed, based around several plausible conjectures. The theory develops reductions between seemingly very different problems, showing that the reason why the known algorithms have been difficult to improve is likely the same, even though the known run-times of the problems might be very different. This direction of study has been termed “fine-grained complexity” theory (see e.g. [44]).

Much of fine-grained complexity is based on hypotheses of the time complexity of infamous problems, e.g., CNF-SAT, All-Pairs Shortest Paths (APSP) and 3-SUM. The hypotheses are about the word-RAM model with $O(\log n)$ bit words, where n is the input size. [27] introduced the Strong Exponential Time Hypothesis (SETH) to address CNF-SAT complexity.

10:4 Partial Permutations Comparison, Maintenance and Applications

The Strong Exponential Time Hypothesis (SETH) [27]. For every $\epsilon > 0$ there exists an integer $k \geq 3$ such that CNF-SAT on formulas with clause size at most k (called k -SAT) and n variables cannot be solved in $O(2^{(1-\epsilon)n})$ time even by a randomized algorithm.

Orthogonal Vectors. The Orthogonal Vectors (OV) problem is a core problem in the basis of many fine-grained hardness results for problems in \mathcal{P} . The problem is formally defined as follows.

► **Definition 4 (The OV Problem).** Let $d = \omega(\log n)$; given two sets $S_1, S_2 \in \{0, 1\}^d$ with $|S_1| = |S_2| = n$, determine whether there exist $a \in S_1, b \in S_2$ so that $a \cdot b = 0$, where $a \cdot b = \sum_{i=1}^d a[i] \cdot b[i]$.

It is not hard to solve OV in $O(n^2d)$ time by exhaustive search. The fastest known algorithms for the problem run in time $n^{2-1/\Theta(\log(d/\log n))}$ [1, 15]. It seems that $n^{2-o(1)}$ is necessary. This motivates the now widely used OV Hypothesis.

OV Hypothesis. No randomized algorithm can solve OV on instances of size n in $n^{2-\epsilon} \text{poly}(d)$ time for constant $\epsilon > 0$.

We describe the connection between OV and *Partial Permutation Agreement* (PPA) (formally defined in Definition 7, Section 3) problems. In fact, we show that they are equivalent, leading to both negative and positive results for PPA, derived also for the dynamic setting. Algorithmic applications to problems in computational biology, image processing and pattern matching are further described in Section 4.

This Paper Contributions. The main contributions of this paper are:

- Giving the first formal discussion from algorithmic point of view of efficient partial permutations maintenance enabling consistent as well as informatively rich comparison.
- Showing that a data structure for efficiently maintaining a dynamic set of partial permutations is not likely to exist, unless the SETH hypothesis fails.
- Describing positive results on maintaining a dynamic set of partial permutations: (1) an improvement in the general case derived via online matrix-factor multiplication, and (2) an efficient solution in a special case termed *almost full partial permutations*.
- Applying the study to reason about fruitful versus futile methods for efficient gene sequences comparison, enabling a formal understanding of this challenge, hinted in [45].
- Applying the study to form an *automatic* process of redundant augmented-data removal to avoid over-fitting of a neural network training set for image processing tasks and increasing its capability to generalize to unseen invariant data [43].
- Applying the study of partial permutation maintenance to answer negatively (unless the SETH hypothesis is false) for an open question regarding a solution over a general alphabet dictionary for the online strict PDMOG problem presented by [35] (see formal definition in Sect. 4.2), while supplying new tools for efficient solution in a special case termed *a k -saturated dictionary* (see formal definition in Sect. 4.2).

Paper Organization. The paper is organized as follows. Section 2 gives the needed preliminary discussion on a good representation of partial permutations. Section 3 studies the possibility of having a data structure for efficiently maintaining a dynamic set of partial permutations as follows. Subsection 3.1 describes the reduction from the OV problem. Subsection 3.2 shows the equivalence to the OV problem via a connection to the *Partial*

Match problem (PM) (formally defined there), which also enables to exploit some positive results for the PM problem to our purposes. Subsection 3.3 describes how to achieve an efficient solution in the special case of almost full partial permutations. Section 4 describes the applications to genes sequences comparison (Subsection 4.1), to color transformation data augmentation and to online strict PDMOG problems (Subsection 4.2). Section 5 concludes the paper with some open questions.

2 Preliminaries: A Good Representation of Partial Permutations

A permutation π over an alphabet $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$ can be represented as the string $s_\pi = \pi(\sigma_1)\pi(\sigma_2)\dots\pi(\sigma_{|\Sigma|})$ of length $|\Sigma|$. For example, let $\Sigma = \{a, b, c\}$ and $\pi = \{a \mapsto b, b \mapsto c, c \mapsto a\}$, then $s_\pi = bca$. It trivially holds that s_{π_1} matches s_{π_2} if and only if the permutations π_1 and π_2 are equal. A straightforward approach to achieve a good representation for partial permutations is to adjust the above by enabling the use of a don't care symbol \star whenever an alphabet symbol of Σ does not belong to the partial permutation domain. For example, let $\Sigma = \{a, b, c, d, e, f\}$ and $M_R(\pi) = \{(a, e), (b, c), (c, d)\}$, where $D_R(\pi) = \{a, b, c\}$ and $C_R(\pi) = \{c, d, e\}$, then this approach suggests using the string $s_\pi = ecd\star\star$ as a representation for π . Note, that an exact string comparison of the strings representing two given partial permutations according to this suggestion still allows two partial permutations to be exactly the same. The use of a don't care symbol in order to broaden the equality scope to partial permutations that *agree* requires using approximate string matching with don't care, where this special symbol indeed matches any symbol.

Nonetheless, even when string matching with don't care is applied, the comparison axiom does not hold for this representation. To see this, consider the partial permutation π_1 where, $M_R(\pi_1) = \{(a, e), (b, c), (e, d)\}$, $D_R(\pi_1) = \{a, b, e\}$ and $C_R(\pi_1) = \{c, d, e\}$. By the above suggestion the representation of π_1 would be $s_{\pi_1} = ec\star\star d\star$. Let π_2 be the partial permutation from the above example, thus $s_{\pi_2} = ecd\star\star$. It holds that s_{π_2} matches s_{π_1} , because the don't care symbol \star matches any symbol. However, the two partial permutations π_1 and π_2 contain a *conflict*. Note that, though the requirement of approximately matching the strings s_{π_1} and s_{π_2} exclude the possibility of a conflict of the first type in Definition 1, it does not exclude a conflict of the second type. Thus, it does not satisfy the universal condition for partial permutations agreement.

Correcting this flaw involves the use of the *inverse permutation*, defined as follows.

► **Definition 5** (Inverse of Partial Permutation). *Given a partial permutation π over Σ , mapping the subset $\Sigma_1 \subset \Sigma$ to the subset $\Sigma_2 \subset \Sigma$, where $|\Sigma_1| = |\Sigma_2|$, the inverse partial permutation π^{-1} of π is a bijection $\pi^{-1} : \Sigma_2 \mapsto \Sigma_1$ such that for every $\sigma_i \in \Sigma_2$, $\pi^{-1}(\sigma_i) = \sigma_j$ if and only if $\pi(\sigma_j) = \sigma_i$.*

For example, let $\Sigma = \{a, b, c, d, e, f\}$ and $\pi = \{a \mapsto e, b \mapsto c, c \mapsto d\}$ be a partial permutation, where $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{c, d, e\}$, then the inverse partial permutation of π is $\pi^{-1} = \{c \mapsto b, d \mapsto c, e \mapsto a\}$.

Now, a partial permutations representation enabling a distinction between two different partial permutations that agree and two partial permutation that disagree is simply the string $s_\pi \cdot s_{\pi^{-1}}$, where \cdot denotes strings concatenation. Note that the size of this representation is $\Theta(|\Sigma|)$. Lemma 6 below ensures that this representation satisfies the comparison axiom.

► **Lemma 6.** *Given two partial permutations π_1, π_2 , then $s_{\pi_1} \cdot s_{\pi_1^{-1}}$ matches $s_{\pi_2} \cdot s_{\pi_2^{-1}}$ if and only if the partial permutations π_1 and π_2 do not contain any conflict.*

The Don't Care Representation of Partial Permutations. Based on Lemma 6, given a partial permutation π we may call the string $s_\pi \cdot s_{\pi^{-1}}$ *the don't care representation of π* . We make the distinction between the don't care representation of π , which is a specific representation described in this section, and the universal condition on any representation described in Subsection 1.1, in order to enable the discussion in the next section to be independent of the representation, when necessary.

3 Maintaining a Dynamic Set of Partial Permutations

In this section we study the possibility of having a data structure for keeping a set of partial permutations enabling the operations: search, insert and delete on the set.

Following the discussion on Subsection 1.1, though there is exactly one partial permutation having the *same* representation as a given partial permutation π , there can be many partial permutations with a representation that satisfy the *universal condition for agreement* with the representation of π . All such partial permutations agree with the given partial permutation, though they are not identical, and may not agree with each other.

Therefore, we make a distinction between these two kinds of search operations: searching the same permutation or searching agreeing permutations. Specifically, given a partial permutation representation, we would like to support a query that returns all the partial permutations in the data structure that have a representation satisfying the universal condition for agreement, i.e., permutations that agree with the query permutation.

We will also refer to an offline batch version of this problem, formally defined as follows.

► **Definition 7** (The Partial Permutations Agreement Problem (PPA)).

Input: Sets A_1, A_2 of partial permutations over alphabet Σ , each of size n .

Output: All pairs (π_i, π_j) , $\pi_i \in A_1, \pi_j \in A_2$ and π_i agrees with π_j .

In the non-batch version of the problem the size of the two sets is different: A_1 has size n , where A_2 , the query, has size 1. We call this problem the *single query PPA* problem, denoted as SPPA. The following observation immediately follows.

► **Observation 8.** *If SPPA can be solved in query time $O(q)$ and $O(S)$ space for a set of n partial permutations, then PPA can be solved in $O(nq)$ time and $O(S)$ space.*

Note, that by Definition 3, for any *good representation* R , SPPA can be naively solved in $q = O(n \cdot |\Sigma|)$ time and $S = O(n \cdot |\Sigma|)$ space. Therefore, by Observation 8, PPA can be naively solved in $O(n^2 \cdot |\Sigma|)$ time and $O(S)$ space for any good representation of partial permutations.

3.1 Orthogonal Vectors and Partial Permutations Agreement

In this subsection, we show that PPA is not likely to be solved in $O(n^{2-\epsilon} \cdot |\Sigma|)$ time. We describe a reduction from the orthogonal vectors problem (OV). Theorem 10 follows. Corollary 12 then follows from Observation 8. The proofs are postponed to the full version.

The Reduction. Let S_1, S_2, n, d be an instance of the Orthogonal Vectors problem, we reduce it to an instance A_1, A_2, Σ of the Partial Permutations Agreement problem, where there are $v_i \in S_1, v_j \in S_2$ such that v_i, v_j are orthogonal if and only if there are $\pi_i \in A_1, \pi_j \in A_2$, such that π_i agrees with π_j .

We construct a permutation gadget for every binary vector v_i as follows. Let $v_i = (b_1^i, b_2^i, \dots, b_d^i)$. We define a partial permutation π_i over alphabet $\Sigma = \{\sigma_1, \dots, \sigma_{d+1}\}$ ($|\Sigma| = d + 1$), where π_i includes the mapping of $\sigma_\ell \in \Sigma$ to a symbol from Σ if and only if $b_\ell^i = 1$,

$\forall 1 \leq \ell \leq d$. However, $\forall 1 \leq \ell \leq d$, where $b_\ell^i = 0$, σ_ℓ does not participate in any pair defining the permutation gadget. Hence, for any representation R of the partial permutation π , we have that $\sigma_\ell \in D_R(\pi)$ if and only if $b_\ell = 1$.

The specific transformation to permutations is asymmetric, i.e., the transformation of S_1 vectors differs from that of S_2 vectors, as follows.

- For π_i associated with $v_i \in S_1$, a symbol that participates in the mapping pairs is mapped to itself. This means that for any representation R of the partial permutation π_i , we have that $\sigma_\ell \in D_R(\pi_i)$, $\sigma_\ell \in C_R(\pi_i)$ and $(\sigma_\ell, \sigma_\ell) \in M_R(\pi_i)$ if and only if $b_\ell = 1$. The additional symbol σ_{d+1} does not participate in any mapping pair. We regard it as if v_i has an additional bit $b_{d+1}^i = 0$. Therefore, for any representation R we get $\sigma_{d+1} \notin D_R(\pi_i)$, $\sigma_{d+1} \notin C_R(\pi_i)$.
- For π_j associated with $v_j \in S_2$, a symbol that participates in the mapping pairs is mapped to the symbol cyclicly to its right in the sorting of Σ_1 – the symbols that participate in the mapping pairs. This means that for any representation R of the partial permutation π_j , we have that $\sigma_\ell \in D_R(\pi_j)$, $\sigma_{\ell'} \in C_R(\pi_j)$ and $(\sigma_\ell, \sigma_{\ell'}) \in M_R(\pi_j)$, where $\sigma_{\ell'}$ is the symbol that is cyclicly to the right of σ_ℓ in the sorting of Σ_1 , if and only if $b_\ell = 1$. The additional symbol σ_{d+1} is included in the mapping pairs of π_j . Thus, $\sigma_{d+1} \in \Sigma_1$ and assumed to be ordered last. We regard it as if v_j has an additional bit $b_{d+1}^j = 1$. Therefore, for any representation R we get $\sigma_{d+1} \in D_R(\pi_j)$, $\sigma_{d+1} \in C_R(\pi_j)$.

See Figure 1 for example.

► **Lemma 9.** *OV is reducible to PPA in $O(n \cdot d)$ time and space.*

$$\begin{array}{c}
 \text{(a)} \\
 v_1 = (1, 0, 1, 1, 0, 0) \in S_1 \quad v_2 = (0, 1, 0, 0, 1, 0) \in S_2 \\
 \Downarrow \qquad \qquad \qquad \Downarrow \\
 \pi_{v_1} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} \\ \mathbf{a} & - & \mathbf{c} & \mathbf{d} & - & - & - \end{pmatrix} \quad \pi_{v_2} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} \\ - & \mathbf{e} & - & - & \mathbf{g} & - & \mathbf{b} \end{pmatrix} \\
 \hline
 \text{(b)} \\
 v_1 = (1, \boxed{1}, 1, 1, 0, 0) \in S_1 \quad v_2 = (0, \boxed{1}, 0, 0, 1, 0) \in S_2 \\
 \Downarrow \qquad \qquad \qquad \Downarrow \\
 \pi_{v_1} = \begin{pmatrix} \mathbf{a} & \boxed{\mathbf{b}} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} \\ \mathbf{a} & \boxed{\mathbf{b}} & \mathbf{c} & \mathbf{d} & - & - & - \end{pmatrix} \quad \pi_{v_2} = \begin{pmatrix} \mathbf{a} & \boxed{\mathbf{b}} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} \\ - & \boxed{\mathbf{e}} & - & - & \mathbf{g} & - & \mathbf{b} \end{pmatrix}
 \end{array}$$

■ **Figure 1** An example of the asymmetric transformation of vectors from S_1 and S_2 into permutations in A_1 and A_2 , respectively. (a) The transformation for a pair of orthogonal vectors gives a pair of permutations that agree. (b) The transformation for a pair of non-orthogonal vectors gives a pair of permutations that do not agree.

Theorem 10 follows.

► **Theorem 10.** *Let R be any good representation of partial permutations. If there exists $\epsilon > 0$ such that for any $c > 0$, PPA is solvable in $O(n^{2-\epsilon})$ time and R is used to represent the partial permutations in the sets A_1, A_2 , then the Strong Exponential Time Hypothesis (SETH) is false.*

Observation 11 below states a weaker version of Theorem 10, which refers explicitly also to the space complexity and includes the $|\Sigma|$ -parameter of the PPA problem. This will be useful for the applications, especially in Section 4.

► **Observation 11.** *Let R be any good representation of partial permutations. If there exists $\epsilon > 0$ such that for any $c > 0$, PPA is solvable in $O(n^{2-\epsilon} \cdot |\Sigma|)$ time and $O(n \cdot |\Sigma|)$ space, where $|\Sigma| = c \log n$ and R is used to represent the partial permutations in the sets A_1, A_2 , then the Strong Exponential Time Hypothesis (SETH) is false.*

► **Remark.** We include the dependence on $|\Sigma|$ in the time complexity in order to make explicit the role of this parameter. In the low-dimensional setting, where $|\Sigma|$ is slightly larger than logarithmic, it could be dropped (since sub-polynomial), where for moderate dimension even $O(n^{2-\epsilon} \text{poly}(\Sigma))$ algorithms can be ruled out under the OV Hypothesis.

Corollary 12 then follows from Theorem 10 and Observation 8.

► **Corollary 12.** *Let R be any good representation of partial permutations. If there exists $\epsilon > 0$ such that for any $c > 0$, SPPA query q can be answered in $O(n^{1-\epsilon})$ time and R is used to represent the partial permutations in the set A_1 and the query q , then the Strong Exponential Time Hypothesis (SETH) is false.*

3.2 The Partial Match Problem and Partial Permutations Agreement

In this subsection we discuss the connection between the Partial Permutations Agreement problem and another important problem – the Partial Match, formally defined as follows.

► **Definition 13** (The Partial Match Problem (PM)).

Preprocess: A set D of n binary vectors of dimension d .

Query: A vector q of dimension d over the set $\{0, 1, \star\}$, where \star is a “don’t care” symbol.

Output: All vectors $v \in D$, such that v matches the query vector q .

In the batch version of the Partial Match problem, denoted by BPM, we have instead of a single query vector, a set Q of n vectors over the set $\{0, 1, \star\}$, and the requested output is all pairs of vectors (v, q) , where $v \in D$ and $q \in Q$, such that v matches q .

The PM problem has been thoroughly studied for decades (e.g. Rivest’s PhD thesis [39]). However, there has been only minor algorithmic progress beyond the two obvious solutions of storing $2^{\Omega(d)}$ space for all possible queries, or taking $\Omega(n)$ time to try all points in the database. It was generally believed that PM is intractable for sufficiently large dimension d – this is one version of the “curse of dimensionality” hypothesis. The best known data structures for answering partial match queries are due to [16] for the general case, and [18] for queries with a bounded number of don’t care symbols. Finally, [1] point out some evidence that batch partial match (BPM) is not solvable in sub-quadratic time due to its equivalence to the OV problem. Consequently, it gives some evidence to the difficulty of the PM problem: it is not likely to be solved in $O(n^{1-\epsilon} \cdot d)$ time and space due to an observation similar to Observation 8.

The Two-Sided BPM and PPA Problems. Note that, in the definitions of the PM and BPM problems don’t care symbols are only allowed in the query vectors, but the database vectors are over $\{0, 1\}$. The good representation for partial permutations described in Section 2 gives a version of the PPA problem for the don’t care representation which can be viewed as a generalization of the BPM problem, where both database and query vectors are over the set $\{0, 1, \star\}$. We call this problem the *two-sided Batch Partial Match* problem (two-sided

BPM). The result presented in Theorem 10 is strong in the sense that it is *independent of the representation*, and means that the difficulty of PPA is not due to a choice of a too rich representation. Theorem 10, specifically, applies also for the don't care representation of partial permutations, for which the PPA problem becomes exactly the two-sided BPM problem. We, thus, get from Theorem 10 the following corollary.

► **Corollary 14.** *If there exists $\epsilon > 0$ such that for any $c > 0$, two-sided BPM is solvable in $O(n^{2-\epsilon})$ time, then the Strong Exponential Time Hypothesis (SETH) is false.*

Equivalence of PM, Two-Sided PM and SPPA Problems. In fact, we now show that these three problems: the partial match (PM), two-sided partial match (two-sided PM) and single-query partial permutations agreement (SPPA), are actually equivalent. As mentioned above, the good representation for partial permutations described in Section 2 gives a version of the PPA problem for the don't care representation which is exactly the two-sided BPM problem, where both database and query vectors are over the set $\{0, 1, \star\}$. Moving to the non-batch versions of these problems, we therefore get, that SPPA and two-sided PM are equivalent. It is, thus, enough to show that the PM and two-sided PM problems are equivalent. Since PM is a special case of two-sided PM, where no don't care symbol appears in the database vectors set D , and may appear only in the query vector q , we need only show how to convert an input of the two-sided PM to an input of PM. Such a transformation can be achieved by a special coding of the symbols of the two-sided PM problem input. Symbols of the dictionary D vectors are coded in the following way: "0" is coded by "01", "1" is coded by "10" and " \star " is coded by "00". Symbols of the query vector are coded in the following way: "0" is coded by "0 \star ", "1" is coded by " \star 0" and " \star " is coded by " $\star\star$ ". This is concluded in Lemma 15.

► **Lemma 15.** *There exists a linear time and space transformation from the two-sided PM problem d -dimensional input vectors $v \in D$ to PM problem $2d$ -dimensional input vectors $t_d(v) \in t_d(D)$, such that a d -dimensional input vector $v \in D$ matches a given d -dimensional query vector q of the two-sided PM problem if and only if the $2d$ -dimensional input vector $t_d(v) \in t_d(D)$ matches a given $2d$ -dimensional query vector $t_q(q)$ of the PM problem.*

We have, therefore, proven Corollary 16.

► **Corollary 16.** *Any algorithm Alg that solves PM in query time $O(q)$ and $O(\mathcal{S})$ space can be used to solve the two-sided PM and SPPA problems in $O(q)$ query time and $O(\mathcal{S})$ space.*

Remark on a Computational Difference of PM and Two-Sided PM. Note that the transformation from two-sided PM to PM has a blow-up in the number of don't care symbols, which is linear in the size of the vectors. Thus, despite Corollary 16, algorithms solving PM efficiently assuming a bounded number of don't cares (such as [18]) cannot be used to efficiently solve the two-sided PM or SPPA problems.

Corollary 16 enables to apply positive results on PM (e.g. [16], which is independent of the number of don't care symbols) on both the two-sided PM and SPPA problems. We are specifically interested in the following result of [33]³.

³ [33] refer to their result as a solution to PM, however, they actually solve two-sided PM.

10:10 Partial Permutations Comparison, Maintenance and Applications

► **Theorem 17** (Theorem 1.2 of [33]). *Let $\Sigma = \{\sigma_1, \dots, \sigma_k\}$ and let \star be an element such that $\star \notin \Sigma$. For any set of strings $x_1, \dots, x_n \in (\Sigma \cup \star)^d$ with $d \leq n$, after $\tilde{O}(nd)$ -time preprocessing, there is an $O(nd)$ space data structure such that, for every query string $q \in (\Sigma \cup \star)^d$, it is possible to answer online whether q matches x_i , for every $i = 1, \dots, n$, in $O(nd \log k / 2^{\Omega(\sqrt{\log d})})$ amortized time over $2^{\omega(\log d)}$ queries.*

Moreover, we observe that the same bounds of Theorem 17 can be achieved for a *dynamic* set of strings, supporting updates (insertion and deletions) in $\tilde{O}(d)$ time. We give a brief description of the [33] solution and explain our observation next.

A Dynamic Two-Sided PM Solution. First, for simplicity assume that $n = d$. The case $d \leq n$ is handled using n/d -splitting technique. Build an $n \times n$ matrix A over $(\Sigma \cup \{\star\})^{n \times n}$ such that $A[i, j] = x_i[j]$. The matrix A is then transformed to a boolean matrix, as follows. Let $S_1, T_1, \dots, S_k, T_k \in [2 \log k]$ be a collection of subsets such that for all i , $|S_i \cap T_i| = \phi$, yet for all $i \neq j$, $|S_i \cap T_j| \neq \phi$. Such a collection exists, by simply taking (for example) S_i to be the i th subset of $[2 \log k]$ having exactly $\log k$ elements (in some ordering on sets), and taking T_i to be the complement of S_i . Extend the matrix A to an $n \times (2n \log k)$ boolean matrix B , by replacing every occurrence of σ_i with the $(2 \log k)$ -dimensional row vector corresponding to S_i , and every occurrence of \star with the $(2 \log k)$ -dimensional row vector which is all-zeroes.

When a query vector $q \in (\Sigma \cup \{\star\})^n$ is received, convert q into a boolean (column) vector v by replacing each occurrence of σ_i with the $(2 \log k)$ -dimensional (column) vector corresponding to T_i , and every occurrence of \star by the $(2 \log k)$ -dimensional (column) vector which is all-zeroes. Compute Av using the Online Matrix Vector multiplication algorithm of [33]. For all $i = 1, \dots, n$, q matches x_i if and only if the i th row of B is *orthogonal* to v . The two vectors are orthogonal if and only if for all $j = 1, \dots, n$, either the i th row of B contains the all-zero vector in entries $(j-1)(2 \log k) + 1, \dots, j(2 \log k)$, or in those entries B contains the indicator vector for a set S_ℓ and correspondingly v contains either \star or a set $T_{\ell'}$ such that $S_\ell \cap T_{\ell'} \neq \phi$, i.e., x_i and q match in the j th symbol. That is, the two vectors are orthogonal if and only if q matches x_i . Therefore, Av reports for all $i = 1, \dots, n$ whether q matches x_i or not.

The important observation is that the transformation of each string to a matrix row is independent. Therefore, strings/vectors can be added/deleted from the set in time proportional to the transformation time, which is $\tilde{O}(d)$. The above solution can be still used for the dynamic set as long as we have enough (at least d) strings/vectors in the set. When the set of strings is less than d , we may use a naive solution instead.

This leads to Corollary 18.

► **Corollary 18** (Dynamic SPPA Online Computation). *Let A be a set of n partial permutations. After $\tilde{O}(n|\Sigma|)$ -time preprocessing, there is an $O(n|\Sigma|)$ space dynamic data structure supporting update operation (insertion or deletion to A) in $\tilde{O}(|\Sigma|)$ time, such that, for every query partial permutation π , it possible to answer online whether π agrees with $\pi_i \in A$, for every $i = 1, \dots, n$, in $O(n|\Sigma| \log |\Sigma| / 2^{\Omega(\sqrt{\log |\Sigma|})})$ amortized time over $2^{\omega(\log |\Sigma|)}$ queries.*

3.3 Almost Full Permutations

In this subsection we consider the PPA and SPPA problems in a special case, where there are only a few symbols in Σ that don't participate in the bijection pairs set. Formally,

► **Definition 19** (Almost Full Partial Permutation). *Let π be a partial permutation over Σ , i.e., a bijection mapping a subset $\Sigma_1 \subset \Sigma$ to a subset $\Sigma_2 \subset \Sigma$, where $|\Sigma_1| = |\Sigma_2|$. We call π an almost full partial permutation if $|\Sigma| - |\Sigma_1| = k$, where $k! = O(\text{poly}(|\Sigma|))$ and $\text{poly}(|\Sigma|)$ is some polynomial in the size of Σ .*

We first describe an efficient solution for the problems over permutations. Formally,

► **Definition 20** (The Equal Permutations Problem (EP)).

Input: Sets B_1, B_2 of size n , of permutations over alphabet Σ .

Output: All pairs (π_i, π_j) , where $\pi_i \in B_1, \pi_j \in B_2$ and $\pi_i = \pi_j$.

As above, in the non-batch version of the problem the size of the two sets is different: B_1 has size n , where B_2 , the query, has size 1. We refer to it as the *single query EP* problem, denoted as SEP.

Efficient Solution for the SEP and EP Problems. The SEP problem can be easily solved by using a *dimension reduction* in the representation of the permutation from $|\Sigma|$ dimensions to a single dimension, assigning each permutation a unique number in $O(|\Sigma|)$ time, assuming RAM model with $O(\log |\Sigma|)$ word size (as done in [35]). The unique numbers representing the permutations $\pi_i \in B_1$ are saved in a hash table, in which we look for the unique number assigned to the query permutation. The assignment of a unique number to a permutation, $\text{num}(\pi)$, is forming a $|\Sigma|$ -radix number representing π . It can be done in several ways, each with complexity $O(|\Sigma|)$, as follows.

1. Assuming, without loss of generality, that $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_{|\Sigma|-1}\}$, and define $|\sigma_i| = i$. Let $\pi = \sigma_{i_1} \sigma_{i_2} \dots \sigma_{i_{|\Sigma|}}$ then: $\text{num}(\pi) = |\sigma_{i_1}| \cdot |\Sigma|^{|\Sigma|-1} + |\sigma_{i_2}| \cdot |\Sigma|^{|\Sigma|-2} + \dots + |\sigma_{i_{|\Sigma|}}|$.
2. By using the technique suggested by [38], where permutations are ranked according to the indices that are swapped in the process of converting the current permutation to the identity permutation.

Consequently, the EP problem can be solved in $q = O(n \cdot |\Sigma|)$ time, and $\mathcal{S} = O(n)$ space due to an observation similar to Observation 8.

Now, an efficient solution for SPPA and PPA in the almost-full partial permutations special case can be achieved via reduction of SPPA to the SEP problem. This is done by creating for each almost full partial permutation π in A_1 , the $k!$ possible permutations derived from π by specifying all the choices to add the symbols that do not already appear in π . This is also done for the single query almost full partial permutation.

For example, let $\Sigma = \{a, b, c, d\}$, $k = 2$, $S_1 = \{\pi_1 = (a \mapsto b, b \mapsto a), \pi_2 = (a \mapsto c, c \mapsto a)\}$ and $q = (a \mapsto c, b \mapsto b)$. Denote the set of full permutations derived from a partial permutation π by $\text{full}(\pi)$. Hence, $\text{full}(\pi_1) = \{bacd, badc\}$, $\text{full}(\pi_2) = \{cbad, cdab\}$. Thus, $\text{full}(S_1) = \{bacd, badc, cbad, cdab\}$ and $\text{full}(q) = \{cbad, cbda\}$. Therefore, $\text{full}(q)$ and $\text{full}(S_1)$ have a matching pair due to $cbad$.

The SPPA is then solved using the above SEP solution with $O(k! \cdot |\Sigma| \cdot n)$ preprocessing time, $q = O(k! \cdot |\Sigma|)$ query time and $\mathcal{S} = O(k! \cdot n)$ space, where a hash table is used for the numbers of the $O(k! \cdot n)$ permutations derived from the n partial permutations of A_1 . Consequently, PPA can be solved in preprocessing $O(k! \cdot |\Sigma| \cdot n)$ time, $q = O(k! \cdot |\Sigma| \cdot n)$ query time and $\mathcal{S} = O(k! \cdot n)$ space due to an observation similar to Observation 8. Moreover, the solution described above supports maintenance of the database set A_1 dynamically, as each partial permutation can be deleted from or inserted to A_1 in $O(k! \cdot |\Sigma|)$ time. This gives Theorem 21.

► **Theorem 21.** *SPPA for almost full partial permutations can be solved in preprocessing $O(k! \cdot |\Sigma| \cdot n) = O(\text{poly}(|\Sigma|) \cdot n)$ time, insertion and deletion in $O(k! \cdot |\Sigma|) = O(\text{poly}(|\Sigma|))$ time, $q = O(k! \cdot |\Sigma|) = O(\text{poly}(|\Sigma|))$ query time and $S = O(k! \cdot n) = O(\text{poly}(|\Sigma|) \cdot n)$ space.*

4 Algorithmic Applications of Partial Permutations

In this section we describe specific computational tasks stemming from computational biology, image processing and pattern matching, for which our study of partial permutations applies. Due to space limitations, we give here only the applications to genes sequences comparison and solving string PDMOG. The description of the application to color transformation data augmentation is postponed to the full version.

4.1 Application to Genes Sequences Comparison

In this subsection, we describe the application of the results in Section 3 to genes sequences comparison.

Genes Sequences Comparison. A family of genomes is often modeled as a set of permutations on genes that are common to all organisms of the family, as in [10]. A limitation in this type of models, comes from the difficulty to identify a large number of genes that are common even to a relatively small set of organisms. This is due, in part, to incompleteness in functional annotations of genes in public websites, and also to the difficulty of determining orthology relationships among genes in different genomes, since this relationship is known to be many-to-many. On the contrary, ubiquitous genes, such as ribosomal genes are often the ones best preserved in different species both in sequence and order and thus provide little valuable information in a gene-order based analysis [45].

Therefore, classifying species based on genes order in case of missing genes and, thus, incomplete permutations, is suggested as a better approach [45]. Furthermore, [45] point out that the occurrence of incomplete permutations with missing elements renders the classification problem more computationally challenging and has received limited attention. The study of partial permutations in this paper, enables to address this computational problem formalization as well as better understand its algorithmic computational challenge.

Incompleteness is formalized as partial words and studied in comparing genes, where the “alphabet” is small and, therefore, repetitiveness is expected. The *domain* $D(w)$ of a partial word w is the set of all positions i such that $w[i]$ is defined, i.e., is not a hole. An alignment of two sequences can be viewed as a construction of two partial words that are *compatible* in the following sense [11]. Given two partial words x and y of the same length, we say that x is contained in y or that y contains x , and we write $x \subset y$, if $D(x) \subset D(y)$ and $x[k] = y[k]$ for all $k \in D(x)$. Two words x and y are *compatible* if there exists a word z that contains both x and y . In this case, the smallest word containing x and y is defined by $D(x) \cup D(y)$.

Note that, two equal length partial words x and y that are compatible must agree on the positions in $D(x) \cap D(y)$, however, there is no requirement on the positions in $D(x) \cup D(y) \setminus (D(x) \cap D(y))$. In particular, it may have repeating symbols. Thus, applying the notion of compatibility in order to compare words in the special case of partial permutations suffers from the following inconsistency: given two partial permutations x and y , we have that the smallest word that contains both x and y , $D(x) \cup D(y)$, is not necessarily also a partial permutation. It also has the undesirable side-effect of generating artificial sequences.

The definition of agreement between partial permutations gives a consistent comparison for genes sequences as well as preserves the original input sequences. Thus, our study enables to point out possibly fruitful versus futile methods for efficient genes sequences comparison.

Our Results Applied to Genes Sequences Comparison. Note, that the basic building block for classification tasks is the comparison operation between a pair of genes sequences. Given a set of n gene sequences over a set of d identified family of genes. Considering the formalization of gene sequences as partial permutations explained above, our definition of *agreement* between partial permutations not only suggests such a building block, but also enables to differentiate situations where the problem can be efficiently computed from situations it probably cannot.

Specifically, the application of the results in Section 3 to genes sequences comparison gives the following. For a family of d identified genes, and a set of n partial permutations representing genes sequences over d , we have that:

- For any $\epsilon > 0$, there exists $c > 0$ such that, if $d = c \log n$, then finding the genes sequences that agree with a query genes sequence is not likely to be answered in $O(n^{1-\epsilon} \cdot d)$ time using $O(n \cdot d)$ (linear) space (unless the Strong Exponential Time Hypothesis (SETH) is false). This follows from Corollary 12 and Observation 11 in Subsection 3.1.
- If $d = \Theta(\log n)$, after $\tilde{O}(n \cdot d)$ -time preprocessing, there is an $O(n \cdot d)$ space dynamic data structure supporting updates (insertion and deletion) in $\tilde{O}(d)$ time, such that, finding genes sequences that agree with a query gene sequence can be done in $O(n \cdot d \log d / 2^{\Omega(\sqrt{\log d})})$ amortized time over $2^{\omega(\log d)}$ queries. This follows from Corollary 18 in Subsection 3.2.
- If the gene sequences of both the database and query are almost full, then there is an $O(k! \cdot n) = O(\text{poly}(d) \cdot n)$ space dynamic data structure supporting updates (insertion and deletion) in $O(k! \cdot d) = O(\text{poly}(d))$ time, such that finding genes sequences that agree with a query genes sequence can be done in $O(k! \cdot d) = O(\text{poly}(d))$ time. This follows from Theorem 21 in Subsection 3.3.

4.2 Application to Solving Strict PDMOG

In this subsection, we describe the application of the results in Section 3 to solving the strict PDMOG problem.

Strict PDMOG. Two equal-length strings are a parameterized match, denoted by p -match, if there exists a bijection on their alphabet symbols under which one string matches the other. The PDMOG problem is motivated by the critical modern concern of cyber security. Network intrusion detection systems (NIDS) perform protocol analysis, content searching and content matching, in order to detect harmful software that may appear on several packets requiring *gapped matching* [30]. A *gapped pattern* P is one of the form $lp \{ \alpha, \beta \} rp$, where each sub-pattern lp , rp is a string over alphabet Σ , and $\{ \alpha, \beta \}$ matches any substring of length at least α and at most β . Several versions of gapped dictionary matching problems were studied recently (see [5, 6, 3, 2, 35, 7, 36]). The *Parameterized Dictionary Matching with One Gap problem* (PDMOG) is defined as follows [40]. Preprocess a dictionary D of d single-gap gapped patterns P_1, \dots, P_d over alphabet $\Sigma' \cup \Sigma$, such that $\Sigma' \cap \Sigma = \emptyset$, so that given a query text T of length n over alphabet $\Sigma' \cup \Sigma$, $\Sigma' \cap \Sigma = \emptyset$, output all locations ℓ in T , where there exist bijections $f_1, f_2 : \Sigma \rightarrow \Sigma$ and the following hold for any $P_i \in D$, and a gap length $g \in [\alpha_i, \beta_i]$, where α_i, β_i are the gap boundaries of P_i :

1. $\forall lp_i[j] \in \Sigma', lp_i[j] = T[\ell - |lp_i| - g - |rp_i| + j]$.
2. $\forall lp_i[j] \in \Sigma, f_1(lp_i[j]) = T[\ell - |lp_i| - g - |rp_i| + j]$.
3. $\forall rp_i[j] \in \Sigma', rp_i[j] = T[\ell - |rp_i| + j]$.
4. $\forall rp_i[j] \in \Sigma, f_2(rp_i[j]) = T[\ell - |rp_i| + j]$.

The *strict* PDMOG problem enforces both left and right sub-patterns to have the same parameterized matching (p-match) function, i.e., that $f_1 = f_2$, which is more reasonable if the encodings of both sub-patterns of a dictionary pattern are done simultaneously [35].

Online strict PDMOG was studied by [35] obtaining algorithms that are fast for some practical inputs called *alphabet saturated dictionary*, where dictionary sub-patterns contain the same alphabet symbols enabling to represent mapping functions of dictionary sub-patterns as permutations. While this assumption is reasonable if the alphabet size is relatively small and the dictionary sub-patterns are not very short, it is still a rigid restriction. Dealing with general alphabet dictionary requires a tool for efficient partial permutations representation and manipulation, which its existence is excluded in this paper (unless the SETH hypothesis is false), showing that an efficient solution for the strict PDMOG problem over a general dictionary alphabet is not likely. Thus, we answer negatively to an open question posed by [35].

Our Results Applied to strict PDMOG. The core issue of the strict PDMOG solution is that while scanning the text, the algorithm locates p-matches of the left sub-patterns of the dictionary D and maintains the partial permutations via which they were p-matched to the text. The algorithm also locates a set of right sub-patterns of the dictionary patterns in D which p-match the current text location. The algorithm needs to verify which of the right sub-patterns are p-matched via a partial permutation that *agrees* with any of the (dynamically changing) set of partial permutations that were used to p-match left sub-patterns that were located within an *active window* of locations determined by the relevant gap bounds of the dictionary D .

The online strict PDMOG was studied by [35] obtaining algorithms that are fast for some practical inputs called *alphabet saturated dictionary*, where dictionary sub-patterns contain the same alphabet symbols. Therefore, the algorithms of [35] represent mapping functions of dictionary sub-patterns as permutations and can efficiently maintain the dynamically changing set of permutations that were used in order to p-match the left sub-patterns of the dictionary D basically using the dimension reduction idea described in Subsection 3.3 for the representation of permutations⁴.

While this assumption is reasonable if the alphabet size is relatively small and the dictionary sub-patterns are not very short, it is still a rigid restriction. Dealing with general alphabet dictionary requires a tool for efficient maintenance of partial permutations. The discussion and results of Subsections 3.1, 3.2, 3.3 can be, therefore, applied to conclude regarding the possibility to efficiently solve the online strict PDMOG problem. In order to simplify the discussion and avoid getting into unnecessary details ([35] use various techniques and several parameters to specify complexity), we summarize the application of the discussion above using the following parameters: s_L - the size of the set S_L of p-matched left sub-patterns of dictionary gapped patterns within the current active window of the text, $|\Sigma|$ - the dictionary D and text T alphabet size. We also need the following definition.

► **Definition 22** (A k -Saturated Dictionary). *Let D be a gapped patterns dictionary over alphabet Σ . We call D a k -saturated dictionary if every sub-pattern in D is over Σ_1 , such that $\Sigma_1 \subseteq \Sigma$ and $k = |\Sigma| - |\Sigma_1|$, where $k! = O(\text{poly}(|\Sigma|))$ and $\text{poly}(|\Sigma|)$ is some polynomial in the size of Σ .*

⁴ The application of this idea in [35] is slightly more involved, since it is combined with the use of range reporting data structures and other details of their algorithms.

The applications to the solution of strict PDMOG that we have shown can, therefore, be summarized as follows:

- For any $\epsilon > 0$, there exists $c > 0$ such that, for a general alphabet Σ with size $|\Sigma| = c \log s_L$, finding partial permutations, via which the sub-patterns in S_L were p-matched to the text, that agree with a partial permutation p-matching of a currently located right sub-pattern is not likely to be answered in $O(s_L^{1-\epsilon} \cdot |\Sigma|)$ time using $O(s_L \cdot |\Sigma|)$ (linear) space (unless the Strong Exponential Time Hypothesis (SETH) is false). This follows from Corollary 12 and Observation 11 in Subsection 3.1.
- For a general alphabet Σ with size $|\Sigma| = \Theta(\log s_L)$, after $\tilde{O}(s_L \cdot |\Sigma|)$ -time preprocessing, there is an $O(s_L \cdot |\Sigma|)$ space dynamic data structure supporting updates (insertion and deletion) in $\tilde{O}(|\Sigma|)$ time, such that, finding partial permutations, via which the sub-patterns in S_L were p-matched to the text, that agree with a partial permutation p-matching of a currently located right sub-pattern can be done in $O(s_L |\Sigma| \log |\Sigma| / 2^{\Omega(\sqrt{\log |\Sigma|})})$ amortized time over $2^{\omega(\log |\Sigma|)}$ queries. This follows from Corollary 18 in Subsection 3.2.
- For a k -saturated dictionary D , there is an $O(k! \cdot s_L) = O(\text{poly}(|\Sigma|) \cdot s_L)$ space dynamic data structure supporting updates (insertion and deletion) in $O(k! \cdot |\Sigma|) = O(\text{poly}(|\Sigma|))$ time, such that finding partial permutations, via which the sub-patterns in S_L were p-matched to the text, that agree with a partial permutation p-matching of a currently located right sub-pattern can be done in $O(k! \cdot |\Sigma|) = O(\text{poly}(|\Sigma|))$ time. This follows from Theorem 21 in Subsection 3.3.

Note that, that if $|\Sigma| = \Theta(\log s_L)$ and $k! = O(|\Sigma|)$ (i.e., $\text{poly}(|\Sigma|)$ is actually linear in $|\Sigma|$), then the third result gives a linear (up to a logarithmic factor) space dynamic data structure for maintaining partial permutations with update and query time logarithmic in s_L .

5 Conclusion

This paper examined the use of *partial permutations* in algorithmic tasks. Some interesting related open questions are:

- Can an efficient solution for PPA/SPPA be achieved for other (practically interesting) special cases?
- What other applications require (possibly hidden) maintenance of partial permutations?

It is our belief that being a relatively basic mathematical concept, partial permutations play a hidden role in more applications. We, therefore, expect more research on the topic in order to explore their algorithmic use.

References

- 1 A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 218–230, 2015.
- 2 A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom. Mind the gap! online dictionary matching with one gap. *Algorithmica*, 81(6):2123–2157, 2019.
- 3 A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom. Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap. In *27th International Symposium on Algorithms and Computation, ISAAC*, pages 12:1–12:12, Sydney, Australia, December 12–14, 2016.
- 4 A. Amir and A. Levy. String rearrangement metrics: A survey. In *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2010.

- 5 A. Amir, A. Levy, E. Porat, and B. R. Shalom. Dictionary matching with one gap. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, volume 8486 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2014.
- 6 A. Amir, A. Levy, E. Porat, and B. R. Shalom. Dictionary matching with a few gaps. *Theor. Comput. Sci.*, 589:34–46, 2015.
- 7 A. Amir, A. Levy, E. Porat, and B. R. Shalom. Online recognition of dictionary with one gap. *Information and Computation*, 275:104633, 2020.
- 8 V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25:272–289, 1996.
- 9 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 71–80, San Diego, CA, USA, May 16-18, 1993.
- 10 A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. *Journal of Computational Biology*, 13(7):1340–1354, 2006.
- 11 J. Berstel and L. Boasson. Partial words and a theorem of Fine and Wilf. *Theoretical Computer Science*, 218(1):135–141, 1999.
- 12 B. Blakeley, F. Blanchet-Sadri, J. Gunter, and N. Rampersad. *Developments in Language Theory*, volume 5583 of *LNCS*, chapter On the Complexity of Deciding Avoidability of Sets of Partial Words, pages 113–124. Springer, 2009.
- 13 F. Blanchet-Sadri, N. C. Brownstein, A. Kalcic, J. Palumbo, and T. Weyand. Unavoidable sets of partial words. *Theory of Computing Systems*, 45(2):381–406, 2009.
- 14 A. Burstein and I. Lankham. Restricted patience sorting and barred pattern avoidance. *Permutation patterns*, 376:233–257, 2010.
- 15 T. M. Chan and R. Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing razborov-smolensky. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1246–1255, 2016.
- 16 M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 451–462, 2002.
- 17 A. Claesson, V. Jelínek, E. Jelínková, and S. Kitaev. Pattern avoidance in partial permutations. *Electronic Journal of Combinatorics*, 18(1), 2011.
- 18 R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of 36 Annual ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 2004.
- 19 T. H. Cormen, C. E. Leiserson, L. R. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 14.3: Interval Trees, pages 348–353. MIT Press and McGraw-Hill, 3rd edition, 2009.
- 20 M. Equi, V. Mäkinen, and A. 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 Computer Science (SOFSEM)*, volume 12607 of *LNCS*, pages 608–622, 2021.
- 21 B. Esslinger. *Cryptography and mathematics*, 2011.
- 22 V. Halava, T. Harju, and T. Kärki. Square-free partial words. *Information Processing Letters*, 108(5):290–292, 2008.
- 23 V. Halava, T. Harju, T. Kärki, and P. Séébold. Overlap-freeness in infinite partial words. *Theoretical Computer Science*, 410(8-10):943–948, 2009.
- 24 S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th annual ACM symposium on the theory of computing*, pages 178–187, 1995.
- 25 G. Hoppenworth, J. W. Bentley, D. Gibney, and S. V. Thankachan. The fine-grained complexity of median and center string problems under edit distance. In *28th Annual European Symposium on Algorithms, ESA*, volume 173 of *LIPICs*, pages 61:1–61:19, 2020.

- 26 A. G. Howard. Some improvements on deep convolutional neural network based image classification. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.
- 27 R. Impagliazzo and R. Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 28 K. Bingmann, P. Gawrychowski, S. Mozes, and O. Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). *ACM Trans. Algorithms*, 16(4):48:1–48:22, 2020.
- 29 D. E. Knuth. *The Art of Computer Programming*, volume 1-3 Boxed Set. Reading, Massachusetts: Addison-Wesley, 2nd edition, 1998.
- 30 M. Krishnamurthy, E. S. Seagren, R. Alder, A. W. Bayles, J. Burke, S. Carter, and E. Faskha. *How to cheat at securing linux*. Syngress Publishing, Inc., Elsevier, Inc., 2008.
- 31 C. Y. Ku and I. Leader. An Erdős-Ko-Rado theorem for partial permutations. *Discrete Mathematics*, 306(1):74–86, 2006.
- 32 G. M. Landau, A. Levy, and I. Newman. LCS approximation via embedding into locally non-repetitive strings. *Information and Computation*, 209(4):705–716, 2011.
- 33 K. G. Larsen and R. Williams. Faster online matrix-vector multiplication. In *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2182–2189, 2017.
- 34 P. Leupold. Partial words for DNA coding. In *10th International Meeting on DNA Computing (DNA 10)*, volume 3384 of *LNCS*, pages 224–234. Springer-Verlag, Berlin, 2005.
- 35 A. Levy and B. R. Shalom. Online parameterized dictionary matching with one gap. *Theoretical Computer Science*, 845(14):208–229, 2020.
- 36 A. Levy and B. R. Shalom. A comparative study of dictionary matching with gaps: Limitations, techniques and challenges. *Algorithmica*, 84:590–638, 2022.
- 37 Y. Li, G. Hu, Y. Wang, T. M. Hospedales, N. M. Robertson, and Y. Yang. DADA: differentiable automatic data augmentation. *CoRR*, abs/2003.03780, 2020. arXiv:2003.03780.
- 38 W. J. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Inf. Process. Lett.*, 79(6):281–284, 2001.
- 39 R. L. Rivest. *Analysis of Associative Retrieval Algorithms*. PhD thesis, Stanford University, 1974.
- 40 B. R. Shalom. Parameterized dictionary matching with one gap. *Theoretical Computer Science*, 854(1):1–16, 2021.
- 41 A. M. Shur and Y. V. Konovalova. On the periods of partial words. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 2136 of *LNCS*, pages 657–665. Springer-Verlag, 2001.
- 42 H. Straubing. A combinatorial proof of the Cayley-Hamilton theorem. *Discrete Mathematics*, 43(2-3):273–279, 1983.
- 43 L. Taylor and G. Nitschke. Improving deep learning with generic data augmentation. In *IEEE Symposium Series on Computational Intelligence, SSCI 2018, Bangalore, India, November 18-21, 2018*, pages 1542–1547. IEEE, 2018.
- 44 V. V. Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians (ICM)*, pages 3447–3487, 2019.
- 45 X. Zhou, A. Amir, C. Guerra, G. M. Landau, and J. Rossignac. EDoP distance between sets of incomplete permutations: Application to bacteria classification based on gene order. *Journal of Computational Biology*, 25(11):1193–1202, 2018. doi:10.1089/cmb.2018.0063.

Bi-Directional r -Indexes

Yuma Arakawa ✉

Department of Mathematical Informatics, The University of Tokyo, Japan

Gonzalo Navarro ✉

CeBiB and Department of Computer Science, University of Chile, Santiago, Chile

Kunihiko Sadakane ✉ 

Department of Mathematical Informatics, The University of Tokyo, Japan

Abstract

Indexing highly repetitive texts is important in fields such as bioinformatics and versioned repositories. The run-length compression of the Burrows-Wheeler transform (BWT) provides a compressed representation particularly well-suited to text indexing. The r -index is one such index. It enables fast locating of occurrences of a pattern within $O(r)$ words of space, where r is the number of equal-letter runs in the BWT. Its mechanism of locating is to maintain one suffix array sample along the backward-search of the pattern, and to compute all the pattern positions from that sample once the backward-search is complete. In this paper we develop this algorithm further, and propose a new bi-directional text index called the br-index, which supports extending the matched pattern both in forward and backward directions, and locating the occurrences of the pattern at any step of the search, within $O(r + r_R)$ words of space, where r_R is the number of equal-letter runs in the BWT of the reversed text. Our experiments show that the br-index captures the long repetitions of the text, and outperforms the existing indexes in text searching allowing some mismatches except in an internal part.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Compressed text indexes, Burrows-Wheeler Transform, highly repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.11

Supplementary Material *Software (Source Code)*: <https://github.com/U-Ar/br-index>
archived at `swh:1:dir:988f1c8381e90fa759b316908113e0c5cf92f228`

Funding *Gonzalo Navarro*: Funded in part by Basal Funds FB0001 and Fondecyt Grant 1-200038, ANID, Chile.

1 Introduction

A text index is a data structure equipped with search operations on a text string. The *suffix tree* [23], which is the compacted trie whose paths to the leaves spell out the suffixes of the text, enables various complex operations useful in bioinformatics [8]. The *suffix array* [14] is a simplified variant of the suffix tree with less space usage but also less functionality. It still supports the most basic searches, *counting* and *locating* the occurrences of a pattern in the text, among more sophisticated ones [11]. Compressed suffix arrays are suffix array representations that retain its functionality within further compressed space. One of those, the *FM-index* [3], is based on the *Burrows-Wheeler transform (BWT)* [2], which searches for the pattern by starting from its last character and extends the match leftwards. The *bi-directional BWT* [10] also supports rightward extension by constructing FM-indexes on both the text and the reversed text, thus using roughly twice the space of the FM-index. This extended functionality allows retrieving some of the lost suffix tree functionality.

Classical compressed suffix arrays are based on statistical compression. This cannot capture repetitions of long text substrings when indexing highly repetitive texts, so the index sizes grow proportionally to the input sizes. Large highly repetitive texts are arising in bioinformatic applications and versioned document and software stores. For those texts,



© Yuma Arakawa, Gonzalo Navarro, and Kunihiko Sadakane;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 11; pp. 11:1–11:14

Leibniz International Proceedings in Informatics



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

■ **Table 1** Comparison of space and time with the existing compressed bi-directional indexes. H is the length of the longest maximal repeat in the text. $right-extension(contraction)$ is symmetric to $left-extension(contraction)$. Here w is the number of bits in the computer word.

index	space	$left-extension$
bi-directional BWT [10]	$O(nH_k(T)) + o(n \log \sigma)$ bits	$O(\frac{\log \sigma}{\log \log n})$
Belazzougui and Cunial [1]	$O(r + r_R)$ words	$O(H^2 \log \log n)$
br-index (Theorem 1)	$O(r + r_R)$ words	$O(\sigma + \log \log_w(n/r))$
br-index (Theorem 2)	$O(r + r_R)$ words	$O(\frac{1}{\epsilon} \log^{2+\epsilon} r)$
index	$left-contraction$	$locate$
bi-directional BWT [10]	not supported	$O(occ \cdot \log^{1+\epsilon} n)$
Belazzougui and Cunial [1]	$O(H^2 \log \log n)$	not supported
br-index (Theorem 1)	not supported	$O(occ)$
br-index (Theorem 2)	not supported	$O(occ)$

indexes based on compression methods such as Lempel-Ziv and grammar compression have been proposed [17]. While those indexes can locate, and in some cases count, the pattern occurrences, they are not based on suffix arrays and therefore lack the potential to enable other more sophisticated suffix array functionalities. The $r-index$ [5, 6] is the first compressed suffix array suitable for highly repetitive texts. It is based on the run-length compression of the BWT and uses $O(r)$ space, where r , the number of equal-letter runs in the BWT, stays low on repetitive texts. The $r-index$ enables efficient *count* and *locate* queries within that space, but more complex operations that are supported on classical suffix arrays are yet to be studied. In particular, an index supporting bi-directional extensions based on this compression method has been proposed [1], but it does not support the key *locate* operation.

Our contribution. We introduce the *br-index*, an $r-index$ extension that supports bi-directional extensions along the pattern search process, within $O(r + r_R)$ words of space, where r_R is the number of equal-letter runs in the BWT of the reversed text. The simpler version of Theorem 1 is easily built on top of the $r-index$ of both of the text and its reverse. The refined version of Theorem 2 reduces the σ term in the computation time of *left-extension* and *right-extension* (where σ is the alphabet size), and is more advantageous when σ is large. Compared to the bi-directional BWT [10], the $br-index$ captures long repetitions in the text and thus compresses highly repetitive text collections. Compared to the index proposed by Belazzougui and Cunial [1], the $br-index$ enables *locate* in efficient time and is easier to implement, though it does not support contractions (i.e., the inverses of expansions). See Table 1 for a detailed comparison. We also implemented the version of Theorem 1 and compared its practical performance with the bi-directional BWT and the $r-index$.

This paper is organized as follows. In Section 2 we describe the needed concepts to present our results. In Section 3 we introduce the algorithmic details of the $br-index$. Section 4 shows the experimental results. We conclude in Section 5.

2 Preliminaries

2.1 Basic notions

In this paper, we call a sequence of characters $T = T[1]T[2] \dots T[n]$ a *string* of length n . Each character $T[i]$ ($i = 1, \dots, n$) is an element of an ordered *alphabet* $\Sigma = \{1, 2, \dots, \sigma\}$. Here we assume Σ is the *effective alphabet*, which means that each character in Σ appears at least once in T . For convenience, we assume $T[n] = 1$ and $T[i] \neq 1$ ($i = 1, \dots, n - 1$), that is,

the last character is a unique endmarker with the minimum lexicographic rank. In addition, we call the sequence of characters $T^R = T[n-1]T[n-2] \cdots T[1]$ the *reversed string*. In other words, we obtain T^R by reversing the meaningful content of the string and attaching the character 1 at the end.

We define two queries on T , where P is a sequence of m characters:

$\mathit{count}(P)$ returns the number of the occurrences of the pattern P in T .

$\mathit{locate}(P)$ returns the starting positions of the occurrences of the pattern P in T .

We write $[l, r]$ for the set of integers $\{l, l+1, \dots, r\}$ (\emptyset if $l > r$). This notation is used to describe substrings and subsequences as well; $T[l, r]$ is the substring $T[l]T[l+1] \cdots T[r]$, which is the empty string ε if $l > r$.

A *bitvector* B is an array whose elements are 0 or 1. We define two queries on a bitvector, $\mathit{rank}_1(B, j)$ returns the number of 1-bits in $B[1, j]$ and $\mathit{select}_1(B, i)$ returns the position of the i -th 1-bit in B .

A *predecessor data structure* on the totally ordered set S supports the query $\mathit{pred}(S, i)$, which returns the maximum element that is smaller than or equal to i , $\max\{s \in S \mid s \leq i\}$.

2.2 Suffix array, Burrows-Wheeler transform, and LCP array

The *suffix array* [14] of T is an array of integers $SA[1, n]$, where $SA[i]$ is the starting position in T of the i -th lexicographically smallest suffix of T , that is, the lexicographic rank of the suffix $T[SA[i], n]$ is i . We also denote the inverse of the suffix array by ISA , that is, $SA[ISA[i]] = i$ ($i = 1, \dots, n$).

The *Burrows-Wheeler transform* (BWT) [2] of T is a sequence $L[1, n]$ of characters that satisfies

$$L[i] = \begin{cases} T[SA[i] - 1] & (SA[i] \neq 1) \\ 1 & (SA[i] = 1) \end{cases}$$

Note that $L[i]$ is the character preceding the i -th suffix in lexicographic order. Exceptionally $L[i] = 1$ when the i -th suffix is the whole string T . We also define a function rank on L : $\mathit{rank}_c(L, i)$ is the number occurrences of the character c in $L[1, i]$. It is 0 if $i = 0$.

The *longest common prefix array* (LCP) of T is an array $LCP[1, n]$ of integers satisfying

$$LCP[i] = \begin{cases} \mathit{lcp}(T[SA[i-1], n], T[SA[i], n]) & (i \neq 1) \\ 0 & (i = 1) \end{cases}$$

where $\mathit{lcp}(P, P')$ is the length of the longest common prefix between strings P and P' .

2.3 Backward search

The suffix array SA and the BWT L are useful for computing count and locate of a pattern $P[1, m]$ [3]. Given P , there exists a unique range $[s, e]$ on SA corresponding to the occurrences of P (the range is empty when P does not occur in T). In this case, $SA[s, e]$ is the list of the starting positions of P in T . We can then represent (the occurrences of) P by the range $[s, e]$. With rank on L we can extend the current pattern leftwards. Specifically, we can compute the range $[s', e']$ corresponding to the pattern cP , from the character c and $[s, e]$ corresponding to P , with the following formula. We call this a *left-extension*.

$$\begin{cases} s' = C[c] + \text{rank}_c(L, s - 1) + 1 \\ e' = C[c] + \text{rank}_c(L, e) \end{cases}$$

Here, $C[1, \sigma]$ is the array of integers where $C[c]$ is the number of occurrences of the characters c' satisfying $c' < c$ in T . When cP does not occur in T , the formula yields $e' > s'$.

The *FM-index* [3] is a statistically compressed suffix array. When it computes $\text{count}(P)$ and $\text{locate}(P)$, it starts from the end of P and extends leftwards with the formula above. It starts with the empty string ε , whose *SA* range is $[1, n]$. Then, from the range $[s_{i+1}, e_{i+1}]$ corresponding to $P[i + 1, m]$ ($1 \leq i \leq m$), it obtains $[s_i, e_i]$ with

$$\begin{cases} s_i = C[P[i]] + \text{rank}_{P[i]}(L, s_{i+1} - 1) + 1 \\ e_i = C[P[i]] + \text{rank}_{P[i]}(L, e_{i+1}) \end{cases}$$

ending if $s_i > e_i$ or $i = 1$ holds. In the first case, $\text{count}(P)$ is zero, otherwise it is $e_1 - s_1 + 1$, and the results of $\text{locate}(P)$ are in $SA[s_1, e_1]$. This searching algorithm is called the *backward search*. We denote the time to compute *left-extension* by t_{LF} , whose name comes from *LF-mapping* $LF(i) = C[L[i]] + \text{rank}_{L[i]}(L, i)$. Similarly, the time to access an element of *SA* is denoted by t_{SA} .

With the backward search algorithm, count takes $O(m \cdot t_{LF})$ time and locate takes $O(m \cdot t_{LF} + \text{occ} \cdot t_{SA})$ time, where occ is the number of the occurrences of P in T . On an alphabet of size σ , the FM-index achieved $t_{LF} = O(\frac{\log \sigma}{\log \log n})$ and $t_{SA} = O(\log^{1+\epsilon} n)$ with $nH_k(T) + o(n \log \sigma)$ bits of space for any constant $0 < \epsilon < 1$, where $H_k(T)$ is the k -th empirical entropy of T [4].

2.4 Run-length compression of BWT and r-index

The size of the representation of L grows linearly with the input size n even if we use statistical compression as in the FM-index. To handle large repetitive text collections we need to capture the repetitions in T and compress them.

Mäkinen and Navarro [12] focused on equal-letter runs in L to capture the repetitiveness. A *run* of the BWT is a maximal substring of L whose characters are equal. Since the suffixes are ordered lexicographically, the sequence of their preceding characters, L , is expected to have long runs if T is highly repetitive. They showed that the number r of such runs is sensitive to the statistical entropy of T , $r \leq nH_k(T) + \sigma^k$ for any $k \geq 0$. In particular, $r \leq nH_k(T) + o(n)$ for any $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$. It was later realized that r was sensitive to the repetitiveness of T , and the run-length-based FM-index (RLFM-index), which compressed the BWT by run-length encoding, was designed [13]. The RLFM-index achieved $t_{LF} = O(\frac{\log \sigma}{\log \log r} + (\log \log n)^2)$ in $O(r)$ words of space by emulating access and rank on L . From this, we can compute count within $O(r)$ words with the RLFM-index, but locate is not supported in the same space. To do that, additional $O(n/s)$ words of space, where s is a sampling parameter, is required to store samples of *SA* at regularly spaced intervals. Since this method yields $t_{SA} = O(s \cdot t_{LF})$, saving spaces with larger s in turn worsens the time complexity.

The *r-index* [5, 6] made it possible to compute locate in $O(m \cdot (t_{LF} + \log \log_w(n/r)) + \text{occ} \cdot t_\phi)$ time within $O(r)$ words of space, without the *SA* samplings at regular intervals. To compute rank on L , it uses an updated version of the RLFM-index, which achieves $t_{LF} = O(\log \log_w(\sigma + n/r))$. The removal of *SA* samplings is achieved by maintaining one *SA* sample during the backward search and designing inverse functions ϕ and ϕ^{-1} , whose computation time is denoted by t_ϕ :

$$\phi(i) = \begin{cases} SA[ISA[i] - 1] & (ISA[i] \neq 1) \\ SA[n] & (ISA[i] = 1) \end{cases} \quad \phi^{-1}(i) = \begin{cases} SA[ISA[i] + 1] & (ISA[i] \neq n) \\ SA[1] & (ISA[i] = n) \end{cases}$$

These functions enable us to compute neighboring SA values from an SA sample. From a sample $SA[i]$, we obtain $SA[i - 1]$ by applying ϕ and $SA[i + 1]$ by applying ϕ^{-1} . They compute those functions in time $t_\phi = O(\log \log_w(n/r))$. To explain our results later, we describe next the algorithm to maintain an SA sample during the backward search.

We say character $T[i]$ is *sampled* if and only if $i = 1$ or $T[i]$ is the first or last character of a BWT run. The number of the sampled characters is $O(r)$. In addition to the RLFM-index, we store a predecessor data structure R_c for each c , with the BWT positions of all the sampled characters equal to c . We associate each BWT position $q \in R_c$ with the pair $\langle q, SA[q] - 1 \rangle$. During the backward search, we know an SA sample $(p, SA[p])$ in the current SA range $[s, e]$ and update it using R_c . Assume we are extending $P[i + 1, m]$ to $P[i, m]$ during the backward search. We want to compute the SA range $[s_i, e_i]$ corresponding to $P[i, m]$ and the new sample $p', SA[p']$ ($s_i \leq p' \leq e_i$), from the range $[s_{i+1}, e_{i+1}]$ corresponding to $P[i + 1, m]$ and the current sample $(p, SA[p])$ ($s_{i+1} \leq p \leq e_{i+1}$). $[s_i, e_i]$ is computed using the RLFM-index. If $L[p] = P[i]$, $LF(p) \in [s_i, e_i]$ holds, so the sample can be updated to $(p' = LF(p), SA[p'] = SA[p] - 1)$. In the other case, where $L[p] \neq P[i]$ but $P[i]$ still occurs somewhere else, we obtain a predecessor $\langle q, SA[q] - 1 \rangle$ by querying $pred(R_{P[i]}, e_{i+1})$. Since $L[q] = P[i]$ holds, the sample is updated to $(p' = LF(q), SA[p'] = SA[q] - 1)$.

Nishimoto and Tabei [19] recently managed to improve the times of the operations to $t_{LF} = O(1)$ and $t_\phi = O(1)$, still within $O(r)$ words, by avoiding predecessor queries.

3 Bi-directional r-index

With the r-index, we can compute *left-extension* and locate all the occurrences of the current pattern at any step of the extensions. However, the extension is unidirectional; *right-extension* cannot be carried out. The text index we propose, br-index, enables us to extend in both directions and compute *locate* at an arbitrary step, as shown in the following theorem.

► **Theorem 1.** *We can store $O(r) + O(r_R)$ words such that, at an arbitrary step of the search, we can execute left-extension in $O(\sigma t_{LF} + \log \log_w(n/r))$ time, right-extension in $O(\sigma t_{LF^R} + \log \log_w(n/r_R))$ time, compute count of the current pattern in $O(1)$ time, and compute locate of the current pattern in $O(occ)$ time, where occ is the number of the occurrences of the current pattern in the string, w is the number of bits in the computer word, and r_R is the number of runs in the BWT L^R of the reversed string T^R .*

► **Remark.** The best known upper bound of r_R by r is $r_R = O(r \log r \max(1, \log \frac{n}{r \log r}))$ [9]. In practice, their values are very close; see Section 4.

In Sections 3.1 and 3.2 we prove Theorem 1. In Section 3.3, we propose a variant using the wavelet tree [7], which achieves the improved time bounds of *left-extension* and *right-extension*, as seen in Theorem 2.

► **Theorem 2.** *For any $\epsilon > 0$, we can store $O(r) + O(r_R)$ words such that, at any arbitrary step of the search, we can execute left-extension in $O(\frac{1}{\epsilon} \log^{2+\epsilon} r)$ time, right-extension in $O(\frac{1}{\epsilon} \log^{2+\epsilon} r_R)$ time, compute count of the current pattern in $O(1)$ time, and compute locate of the current pattern in $O(occ)$ time, where occ is the number of the occurrences of the current pattern in the string.*

The key idea of the br-index is to compute *locate* efficiently by maintaining one SA sample and one SA^R sample at the same time. These samples are not necessarily starting or ending positions of the current pattern. Instead, we also maintain their offsets towards both ends, and the length of the current pattern.

3.1 Left-extension and right-extension

Updating the ranges on SA and SA^R

Let $[s, e]$ be the range on SA corresponding to the current pattern P . Similarly, let $[s_R, e_R]$ be the range on SA^R corresponding to P^R .

When we compute *left-extension* $P \rightarrow cP$, we update $[s, e]$ by $s \leftarrow C[c] + \text{rank}_c(L, s - 1) + 1, e \leftarrow C[c] + \text{rank}_c(L, e)$. To update $[s_R, e_R]$, we use another idea [10]. We count the total number acc of occurrences of patterns aP for all $a < c$, by applying LF iteratively for each such a . Since the size of the range of any pattern is equal on SA and SA^R , we can update $[s_R, e_R]$ by $s_R \leftarrow s_R + acc, e_R \leftarrow s_R + acc + e - s$. *right-extension* is symmetric. In this case, we apply LF^R , which is LF-mapping on the BWT of T^R , instead of LF .

The required structures to update the ranges are just the RLFM-indexes on T and T^R . The space used is $O(r + r_R)$ words, the time complexity is $O(\sigma t_{LF})$ when we extend leftward, and $O(\sigma t_{LF^R})$ when we extend rightward, where t_{LF^R} is the time to compute LF^R .

Updating the sample

In addition to the SA range $[s, e]$ and the SA^R range $[s_R, e_R]$, we maintain seven variables during the search: $p, j, d, p_R, j_R, d_R, len$. We call the tuple of these variables the *sample*: p is the position of the sample in SA , j is the value of $SA[p]$, and d is the offset of j to the starting position of the current pattern. That is, it holds $j = SA[p]$ and $T[j - d, j - d + |P| - 1] = P$. The corresponding values for the reversed direction are $j_R = SA^R[p_R]$ and $T^R[j_R - d_R, j_R - d_R + |P| - 1] = P^R$. Finally, len is the length of the pattern.

We note, however, that we will not be able to maintain p and p_R in all cases; we will manage without them. We still speak of those variables for reasoning about correctness.

Assume we are computing *left-extension* $P \rightarrow cP$. If the size of the range $[s, e]$ on SA corresponding to the pattern does not change, only the character c precedes P in T . In this case, we simply increment d and len . Otherwise, we compute the predecessor $\text{pred}(R_c, e)$, to obtain $\langle q, SA[q] - 1 \rangle$. We then update $j \leftarrow SA[q] - 1$ and $j_R \leftarrow n - j$. Also, offsets are updated to $d \leftarrow 0, d_R \leftarrow len$, and $len \leftarrow len + 1$. The case of *right-extension* is symmetric.

The details are shown in Algorithms 1 and 2. In the following lemma, we prove the invariant conditions that hold during the extensions. These conditions are important for the correctness of the *locate* algorithm presented in the next section.

► **Lemma 3.** *Assume we are computing left-extension and right-extension, and the current pattern is P . Then the following conditions are invariant, except when P is empty.*

- (1) $len = |P|$
- (2) $d + d_R + 1 = len$
- (3) Let $j = SA[p]$ and $j_R = SA^R[p_R]$, then $s \leq LF^d(p) \leq e$ and $s_R \leq (LF^R)^{d_R}(p_R) \leq e_R$

Proof. When we start with an empty pattern $P = \varepsilon$, we initialize the ranges and the sample with $s = s_R = 1, e = e_R = n, len = d = d_R = 0$. We then obtain an arbitrary predecessor $\langle q, SA[q] - 1 \rangle$ and set $j = y$ and $j_R = n - y$. We now prove that the invariants are maintained by *left-extension*; *right-extension* is symmetric.

Algorithm 1 Left-extension $P \rightarrow cP$.

Input: A character c and values corresponding to $P : [s, e], [s_R, e_R], j, d, len$
Output: Values corresponding to $cP : [s', e'], [s'_R, e'_R], j', j'_R, d', d'_R, len'$

```

1:  $s' \leftarrow C[c] + rank_c(L, s - 1) + 1$ 
2:  $e' \leftarrow C[c] + rank_c(L, e)$ 
3: if  $s' > e'$  then
4:    $cP$  does not occur.
5: else
6:    $acc \leftarrow 0$ 
7:   for  $a = 1$  to  $c - 1$  do
8:      $acc \leftarrow acc + rank_a(L, e) - rank_a(L, s - 1)$ 
9:   end for
10:   $[s'_R, e'_R] \leftarrow [s_R + acc, s_R + acc + e' - s']$ 
11:  if  $e' - s' \neq e - s$  ( $cP$  and  $c'P$  occur for some  $c' \neq c$ ) then
12:     $(q, j') \leftarrow pred(R_c, e), d' \leftarrow 0$ 
13:  else
14:     $j' \leftarrow j, d' \leftarrow d + 1$ 
15:  end if
16:   $j'_R \leftarrow n - j', d'_R \leftarrow len - d'$ 
17:   $len' \leftarrow len + 1$ 
18: end if

```

First, consider the case where $e' - s' \neq e - s$ in line 11 of Algorithm 1. (1) Since len' is incremented from len , $len' = |cP|$ holds. (2) $d' + d'_R + 1 = 0 + len + 1 = len'$ holds. (3) From the definition of R_c , $j' = SA[q] - 1$, so the new value for p is $p' = LF(q)$. Also, since $j'_R = n - j' = n - (SA[q] - 1) = SA^R[ISA^R[n - SA[q] + 1]]$, it holds that the new value for p_R is $p'_R = ISA^R[n - SA[q] + 1]$. Now, cP and $c'P$ ($c' \neq c$) occur in this case, which means an end of a BWT run of the character c exists in $[s, e]$. Thus, $s \leq q \leq e$ and $L[q] = c$ holds, which in turn implies $s' \leq LF(q) = p' \leq e'$. On the other hand, $SA^R[(LF^R)^{d'_R}(p'_R)] = SA^R[p'_R] - d'_R = j'_R - d'_R = (n - j') - d'_R = n - (j' + d'_R)$ holds. This position in T^R corresponds to the position $j' + d'_R = j' + len' - d' - 1 = SA[LF^{d'}(p')] + len' - 1$ in T . This is the ending position of the pattern cP in T , and the starting position of $P^R c$ in T^R . Therefore $s'_R \leq (LF^R)^{d'_R}(p'_R) \leq e'_R$ holds.

Second, consider the other case, where $e' - s' = e - s$ in line 13 of Algorithm 1. This case does not happen when P is empty since T contains at least two distinct characters. Thus, the inductive assumption can be used. That is, we assume that the three conditions hold before the execution of *left-extension*. (1) Same as the former case. (2) $d' + d'_R + 1 = d + 1 + d_R + 1 = len + 1 = len'$ holds from the inductive assumption. (3) Note that j and j_R do not change, so $p' = p$ and $p'_R = p_R$. In this case c precedes all the occurrences of P . Thus, $s'_R = s_R$ and $e'_R = e_R$, and since we also maintain $d'_R = d_R$, the relation $s_R = s'_R \leq (LF^R)^{d'_R}(p'_R) = (LF^R)^{d_R}(p_R) \leq e'_R = e_R$ stays true by induction. On the other hand, $s' = C[c] + rank_c(L, s - 1) + 1 = C[c] + rank_c(L, s)$, $e' = C[c] + rank_c(L, e)$, and $LF^{d'}(p') = LF(LF^d(p)) = C[c] + rank_c(L, LF^d(p))$ holds since $L[s] = L[LF^d(p)] = c$. Therefore, $s' \leq LF^{d'}(p') \leq e'$ holds from the inductive assumption. ◀

■ **Algorithm 2** Right-extension $P \rightarrow Pc$.

Input: A character c and values corresponding to $P : [s, e], [s_R, e_R], j_R, d_R, len$

Output: Values corresponding to $Pc : [s', e'], [s'_R, e'_R], j', j'_R, d', d'_R, len'$

```

1:  $s'_R \leftarrow C[c] + rank_c(L^R, s_R - 1) + 1$ 
2:  $e'_R \leftarrow C[c] + rank_c(L^R, e_R)$ 
3: if  $s'_R > e'_R$  then
4:    $Pc$  does not occur.
5: else
6:    $acc \leftarrow 0$ 
7:   for  $a = 1$  to  $c - 1$  do
8:      $acc \leftarrow acc + rank_a(L^R, e_R) - rank_a(L^R, s_R - 1)$ 
9:   end for
10:   $[s', e'] \leftarrow [s + acc, s + acc + e'_R - s'_R]$ 
11:  if  $e'_R - s'_R \neq e_R - s_R$  ( $Pc$  and  $Pc'$  occur for some  $c' \neq c$ ) then
12:     $(q_R, j'_R) \leftarrow pred(R_c^R, e_R), d'_R \leftarrow 0$ 
13:  else
14:     $j'_R \leftarrow j_R, d'_R \leftarrow d_R + 1$ 
15:  end if
16:   $j' \leftarrow n - j'_R, d' \leftarrow len - d'_R$ 
17:   $len' \leftarrow len + 1$ 
18: end if

```

3.2 Determining the end of *locate* with run-length compressed PLCP

We now present the algorithm for *locate*. We can obtain the values $SA[i - 1], SA[i + 1]$ from $SA[i]$, using just the functions ϕ and ϕ^{-1} of the r-index. Therefore, neighboring SA values are obtained sequentially from component j, d of the sample. However, because we do not know $p' = LF^d(p)$, we cannot determine how many values $i < p'$ and $i > p'$ are within the range $[s, e]$ corresponding to the current pattern P .

In order to determine the ends of the iterative computations of ϕ and ϕ^{-1} , we make use of the permuted LCP array $PLCP[1, n]$, which satisfies $PLCP[i] = LCP[ISA[i]]$ ($i = 1, \dots, n$). Let the current position in SA be $p' \in [s, e]$. When we are computing the value of $SA[p' - 1]$ from $SA[p']$, we compare $PLCP[SA[p']]$ with $|P|$. If $PLCP[SA[p']]$ is smaller than $|P|$, $SA[p' - 1]$ does not correspond to an occurrence of the whole pattern P . Thus, $p' = s$ holds in this case. Otherwise we go on and compute ϕ . Similarly, when we compute $SA[p' + 1]$ from $SA[p']$, we compare $PLCP[SA[p' + 1]]$ with $|P|$.

The details are shown in Algorithm 3. In the following lemma, we prove that Algorithm 3 runs properly if the invariant conditions hold. Combining Lemmas 3 and 4, we obtain the correctness of *locate*.

► **Lemma 4.** *Let $[s, e]$ be the range on SA that corresponds to the current pattern P . Assume the input of Algorithm 3 satisfies $j = SA[p], s \leq LF^d(p) \leq e, len = |P|$. Then Algorithm 3 correctly outputs all the positions of the occurrences of P .*

Proof. The correctness of ϕ, ϕ^{-1} is proved in [6, Lem. 3.5]. Since $j = SA[p], j' = j - d$ is equal to $SA[p']$ ($p' = LF^d(p)$). Provided $s \leq p' \leq e$, we have to prove

- $PLCP[SA[p']] \geq |P| \Rightarrow p' > s$
- $PLCP[SA[p']] < |P| \Rightarrow p' = s$

In the case where $PLCP[SA[p']] \geq |P|$, $PLCP[SA[p']] = LCP[ISA[SA[p']]] = LCP[p'] = lcp(T[SA[p'], n], T[SA[p' - 1], n]) \geq |P|$ holds. Since the first $|P|$ characters of $T[SA[p'], n]$ are identical to P from the assumption, the first $|P|$ characters of $T[SA[p' - 1], n]$ are also

■ **Algorithm 3** Locate the current pattern P .

Input: $p, j (= SA[p]), d, len (= |P|)$
Output: All the starting positions of the occurrences of P in T

- 1: $j' \leftarrow j - d (= SA[LF^d(p)])$
- 2: $pos \leftarrow j'$
- 3: **output** pos
- 4: **while** $PLCP[pos] \geq len$ **do**
- 5: $pos \leftarrow \phi(pos)$
- 6: **output** pos
- 7: **end while**
- 8: $pos \leftarrow j'$
- 9: **while true do**
- 10: **if** $pos = SA[n]$ **then return**
- 11: $pos \leftarrow \phi^{-1}(pos)$
- 12: **if** $PLCP[pos] < len$ **then return**
- 13: **output** pos
- 14: **end while**

the same as P . Thus, $p' - 1$ is also within the range $[s, e]$, which means $p' > s$. On the other hand, when $PLCP[SA[p']] < |P|$, $lcp(T[SA[p'], n], T[SA[p' - 1], n]) < |P|$ holds. In this case, at least one character among the first $|P|$ characters of $T[SA[p'], n]$ and $T[SA[p' - 1], n]$ differ. Since the first $|P|$ characters of $T[SA[p'], n]$ are identical to P , the first $|P|$ characters of $T[SA[p' - 1], n]$ are not the same as P . Thus, $p' - 1$ is out of the range $[s, e]$, which means $p' = s$. Similarly,

- $PLCP[SA[p' + 1]] \geq |P| \Rightarrow p' < e$
- $PLCP[SA[p' + 1]] < |P| \Rightarrow p' = e$

holds when $p' \leq n - 1$, so we can correctly decide whether $s \leq p' \leq e$ holds.

From the above arguments, we can locate all the occurrences of P using Algorithm 3. ◀

If we use a predecessor data structure to store $PLCP$ in $O(r)$ words of space, we can access one value of $PLCP$ in $O(\log \log_w(n/r))$ time [6, Lem. 3.8.]. As a more sophisticated solution, ϕ, ϕ^{-1} and $PLCP$ can be computed simultaneously in $O(1)$ time within $O(r)$ words of space, with a *move data structure* [19]. The algorithm to compute ϕ^{-1} is explained in [19]. ϕ is symmetric. We integrate a procedure to compute $PLCP$ into the algorithm. In addition to the values of ϕ and ϕ^{-1} stored in the structure, we store the values of $PLCP$ at the same sampled positions. We compute the predecessor by a *move* query, obtain its $PLCP$ value, and subtract the offset between the current position and the predecessor from the value. Therefore we obtain Theorem 1.

3.3 Improving the *extend* time with wavelet tree

In lines 7-9 of Algorithm 1, *rank* on L is computed for $O(\sigma)$ times in order to calculate the accumulated number of occurrences of $c'P$ ($c' < c$). These computations are costly when σ is large. We could easily compute the accumulated number in $O(\log \sigma)$ time on the wavelet tree of the BWT, since it is a range-counting problem [15]. This is not that simple, however, on the run-length BWT representation. We now show that polylogarithmic time is still possible, however.

Consider the sequence $L'[1, r]$ of the *run heads* in the BWT, that is, the first characters of the BWT runs. Regard L' as the 2-dimensional grid G of size $r \times \sigma$ which has r points, whose x -coordinates are the positions in L' and y -coordinates are the characters. That is, if $L'[i] = c$, there is a grid point at (i, c) . Give to that point a *weight*, equal to the length of the corresponding run in L . We can apply the following theorem on that grid (simplified for our purpose).

► **Theorem 5** ([16]). *Let a grid of size $r \times r$ store r points with associated non-negative integers whose values are at most n . For any $\epsilon > 0$, a structure of $O(\frac{1}{\epsilon} r \log n)$ bits can compute the sum of the integers in any rectangular range in time $O(\frac{1}{\epsilon} \log^{2+\epsilon} r)$.*

Since the shape of the grid is required to be $r \times r$ in Theorem 5, we extend the $r \times \sigma$ grid with an empty area. We also need a way to determine, given a position $L[i]$, the run it belongs to, and the start/end positions of that run in L . This is already supported by the r-index structures, in time $O(\log \log_w(n/r))$.

With these structures, we count the number of symbols $< c$ in $L[l, r]$ as follows. (1) Compute the runs x_1 and x_2 where l and r belong, respectively, the ending position l' of the x_1 -th run and the starting position r' of the x_2 -th run. (2) Compute, using Theorem 5, the sum of the weights of the points falling in $[x_1 + 1, x_2 - 1] \times [1, c - 1]$. (3) Add $l' - l + 1$ if $L[l] < c$, and $r - r' + 1$ if $L[r] < c$.

We thus construct the structure of Theorem 5 on L and on L^R . We obtain Theorem 2 by noting that all the times of the form $O(\log \log_w(n/r))$ come from predecessor queries, which can also be done in time $O(\log r)$ by resorting to binary search.

4 Experiments

4.1 Experimental setup

In order to test the practical performance of the index, we experimented on repetitive datasets taken from the Pizza&Chili Repetitive Corpus.¹ Their characteristics are shown in Table 2. We compared the br-index with the r-index and the bi-directional FM-index (2BWT) built on the same datasets. For the br-index, we implemented the differentially encoded *PLCP* with a sparse bitmap [22, 20]. For the 2BWT, we tested $s = 16, 32, 64, 128$ as the sampling parameter of *SA*. Also, as the components of the 2BWT, we used the wavelet trees implemented with RRR bitvectors [21].

We evaluated all the experiments in a machine with Intel Xeon CPU E5-2650 v2 clocked at 2.60GHz and the 128GB memory. The compiler was gcc 4.8.5 and the compiler options were `-std=c++11 -Ofast -march=native`.

In addition to comparing the spaces used by the indexes, we demonstrate the power of the extended primitives on a simplified variant of a popular bioinformatics query, the so-called *seed-and-extend* approach used in BLAST. In the query, we consider a pattern divided into three parts, $P = P_1P_2P_3$. We locate all the occurrences of P allowing up to k mismatches in P_1 and P_3 , while P_2 is matched exactly. Note that we do not locate the occurrences of P with mismatches in P_2 , even if the total number of mismatches in P is within k . On the 2BWT and the br-index, we execute the query by first searching for P_2 in exact form.

¹ <http://pizzachili.dcc.uchile.cl/repcorpus.html>

■ **Table 2** The statistics for the datasets. The lexicographically minimum character attached to the end is included.

datasets	n	σ	r	r_R	r/n
cere	461,286,644	6	11,574,641	11,575,583	0.0251
coreutils	205,281,778	237	4,684,460	4,732,795	0.0228
einstein.de	92,758,441	118	101,370	99,834	0.0011
einstein.en	467,626,544	140	290,239	286,698	0.0006
escherichia	112,689,515	16	15,044,487	15,045,278	0.1335
influenza	154,808,555	16	3,022,822	3,018,825	0.0195
kernel	258,961,616	161	2,791,368	2,780,096	0.0108
para	429,265,758	6	15,636,740	15,635,178	0.0364
world-leaders	46,968,181	90	573,487	583,397	0.0122

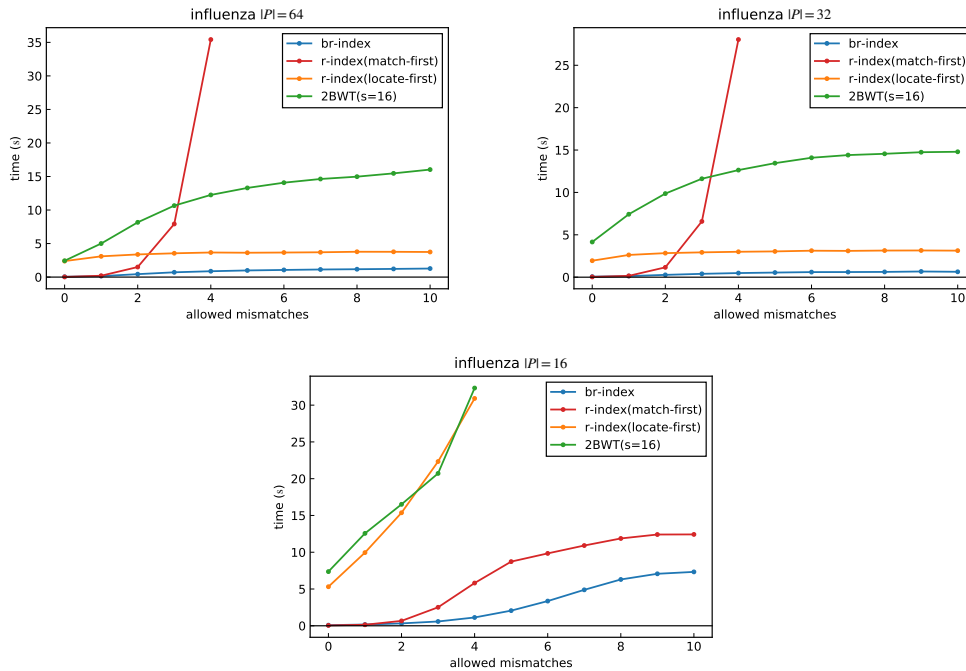
■ **Table 3** The sizes (bits/symbol) of the indexes on the repetitive datasets. s is the sampling parameter for SA .

	2BWT				r-index	br-index
	$s = 16$	$s = 32$	$s = 64$	$s = 128$		
cere	8.44	6.33	5.27	4.73	1.93	5.63
coreutils	12.80	10.68	9.61	9.07	1.87	4.92
einstein.de	11.08	8.96	7.90	7.36	0.099	0.276
einstein.en	11.97	9.86	8.79	8.24	0.057	0.162
escherichia	10.18	8.07	7.00	6.46	9.20	26.89
influenza	8.80	6.69	5.62	5.09	1.49	4.32
kernel	12.32	10.20	9.14	8.60	0.90	2.54
para	8.61	6.50	5.43	4.90	2.76	8.07
world-leaders	11.38	9.26	8.20	7.66	0.96	2.74

Then we extend the match leftwards to any P'_1P_2 , where P'_1 has $0 \leq k' \leq k$ mismatches with respect to P_1 . This is done with the usual backtracking mechanism starting from the range of P_2 , using *left-extension* on every possible symbol as long as the error threshold permits. Finally, we extend each resulting range rightwards using *right-extension*, finding P_3 with at most $k - k'$ mismatches, and report all the occurrences found.

This strategy cannot be used on the r-index, because it cannot extend rightwards. In this case, we tested two different algorithms. The first algorithm, which we call *match-first*, searches for the pattern from the end to the beginning using *left-extension*, allowing up to k mismatches when matching P_3 and P_1 . This is likely to be considerably slower because it does not restrict the matches to P_2 before starting to allow errors. The second algorithm, which we call *locate-first*, finds all the occurrences of P_2 with just the r-index, and extracts the text around each occurrence to check if the number of mismatches in P'_1 and P'_3 is within k . This algorithm is similar to the approach of BLAST, although we extract the characters around P_2 using *LF* and *FL* (the inverse function of *LF*) because we were not storing the plain text. This approach can work well if P_2 is long enough, although it scales linearly with the text size.

We extracted 100 random substrings of length 16, 32, 64 as the target patterns from *influenza*, and computed *seed-and-extend* for each pattern. P_2 is set at the middle of P , with length $\lceil |P|/3 \rceil$. The number of allowed mismatches was between 0 and 10.



■ **Figure 1** The total computation times of *seed-and-extend* query for all the target patterns on influenza with the number of allowed mismatches between 0 and 10. The 2BWT sometimes mistakenly locates positions for unknown reasons, but the number of reported patterns is very close to that of other indexes.

4.2 Experimental results

The index sizes are shown in Table 3. The br-index is smaller than the 2BWT in many cases. Exceptionally, the br-index is larger when built on *escherichia*, where r/n is relatively large. The br-index is about 3 times larger than the r-index in all cases. This is expected because we store L , L^R , $PLCP$, and the structures to compute ϕ^{-1} (in practice the r-index works with only ϕ).

Figure 1 shows the computation times of *seed-and-extend*. As it can be seen, the br-index and the 2BWT yield curves with similar shape, though the br-index is an order of magnitude faster. The match-first algorithm we use on the r-index, instead, is sharply outperformed as soon as we allow a few mismatches, as expected. When the pattern is short, the approach manages to outperform the 2BWT, but still the br-index is considerably faster. The br-index is also faster than the locate-first algorithm on the r-index in all cases, and is robust to the increase of allowed mismatches when the pattern is long. The locate-first approach, instead, worsens significantly on short patterns, because in that case P_2 has too many occurrences to verify.

5 Conclusions

We introduced the br-index, which supports the bi-directional extension of the currently searched pattern while efficiently locating all of its occurrences within $O(r + r_R)$ words, by maintaining an SA sample and its offset to the current pattern, and determining the end of the *locate* area using the run-length compressed $PLCP$. In practice, the size of the br-index

was observed to be around 3 times as large as that of the r-index [6], and comparable to that of the 2BWT [1], on repetitive datasets. Also, as an application of interleaving *left-extension* and *right-extension*, we tested the *seed-and-extend* query, which finds a pattern allowing some mismatches except in an internal part. The br-index is shown to sharply outperform the r-index on this query, and the gap is likely to grow when allowing more mismatches.

Our work can be seen as a first step towards a fully-functional compressed suffix tree whose size is as close to $O(r + r_R)$ words as possible. The br-index can serve as a component of such a suffix tree, since we can compute *child* and *weiner-link* with it: these operations correspond to *right-extension* and *left-extension*, respectively. On the other hand, *suffix-link* and *parent* are not supported because they need bi-directional pattern contraction. These operations can be carried out with the representation of the suffix tree topology or the random access to *LCP*, both of which require some queries on it. From the perspective of the computation time, the former is more promising in practice [18], while the latter is guaranteed to use $O(r \log \frac{n}{r})$ words [6]. We wonder if the functionality can be supported in $O(r + r_R)$ words, or if another reasonable repetitiveness measure can be defined within which we can represent, for example, the compressed suffix tree topology.

References

- 1 Djamel Belazzougui and Fabio Cunial. Smaller fully-functional bidirectional BWT indexes. In *International Symposium on String Processing and Information Retrieval*, pages 42–59, 2020.
- 2 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. *Digital SRC Research Report*, 1994.
- 3 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
- 4 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
- 5 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477, 2018.
- 6 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):1–54, 2020.
- 7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.
- 8 Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 9 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In *IEEE 61st Annual Symposium on Foundations of Computer Science*, pages 1002–1013, 2020.
- 10 T. W. Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and S. M. Yiu. High throughput short read alignment via bi-directional BWT. *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 31–36, 2009.
- 11 Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
- 12 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56, 2005.
- 13 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

- 14 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 15 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
- 16 Gonzalo Navarro. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science*, 772:58–72, June 2019.
- 17 Gonzalo Navarro. Indexing highly repetitive string collections, part i. *ACM Computing Surveys*, 54(2), 2021.
- 18 Gonzalo Navarro and Alberto Ordóñez. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 21(1):article 1.8, 2016.
- 19 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Leibniz International Proceedings in Informatics*, pages 101:1–101:15, 2021.
- 20 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pages 60–70, 2007.
- 21 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, 2002.
- 22 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- 23 Peter Weiner. Linear pattern matching algorithms. *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Making de Bruijn Graphs Eulerian

Giulia Bernardini ✉ 

University of Trieste, Italy
CWI, Amsterdam, The Netherlands

Huiping Chen ✉ 

King's College London, UK

Grigorios Loukides ✉ 

King's College London, UK

Solon P. Pissis ✉ 

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Leen Stougie ✉

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Michelle Sweering ✉

CWI, Amsterdam, The Netherlands

Abstract

A directed multigraph is called *Eulerian* if it has a circuit which uses each edge exactly once. Euler's theorem tells us that a weakly *connected* directed multigraph is Eulerian if and only if every node is *balanced*. Given a collection S of strings over an alphabet Σ , the *de Bruijn graph* (DBG) of order k of S is a directed multigraph $G_{S,k}(V, E)$, where V is the set of length- $(k-1)$ substrings of the strings in S , and $G_{S,k}$ contains an edge (u, v) with multiplicity $m_{u,v}$, if and only if the string $u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and this string occurs exactly $m_{u,v}$ times in total in strings in S . Let $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$ be the complete DBG of Σ^k . The Eulerian Extension (EE) problem on $G_{S,k}$ asks to extend $G_{S,k}$ with a set \mathcal{B} of nodes from $V_{\Sigma,k}$ and a smallest multiset \mathcal{A} of edges from $E_{\Sigma,k}$ to make it Eulerian. Note that extending DBGs is algorithmically much more challenging than extending general directed multigraphs because some edges in DBGs are by definition *forbidden*. Extending DBGs lies at the heart of sequence assembly [Medvedev et al., WABI 2007], one of the most important tasks in bioinformatics. The novelty of our work with respect to existing works is that we allow not only to duplicate existing edges of $G_{S,k}$ but to also add novel edges and nodes, in an effort to (i) connect multiple components and (ii) reduce the total EE cost. It is easy to show that EE on $G_{S,k}$ is NP-hard via a reduction from shortest common superstring. We further show that EE remains NP-hard, even when we are not allowed to add new nodes, via a highly non-trivial reduction from 3-SAT. We thus investigate the following two problems underlying EE in DBGs:

1. When $G_{S,k}$ is not weakly connected, we are asked to connect its $d > 1$ components using a minimum-weight spanning tree, whose edges are paths on the underlying $G_{\Sigma,k}$ and weights are the corresponding path lengths. This way of connecting guarantees that no new unbalanced node is added. We show that this problem can be solved in $\mathcal{O}(|V|k \log d + |E|)$ time, which is nearly optimal, since the size of $G_{S,k}$ is $\Theta(|V|k + |E|)$.
2. When $G_{S,k}$ is not balanced, we are asked to extend $G_{S,k}$ to $H_{S,k}(V \cup \mathcal{B}, E \cup \mathcal{A})$ such that every node of $H_{S,k}$ is balanced and the total number $|\mathcal{A}|$ of added edges is minimized. We show that this problem can be solved in the optimal $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time.

Let us stress that, although our main contributions are theoretical, the algorithms we design for the above two problems are practical. We combine the two algorithms in one method that makes any DBG Eulerian; and show experimentally that the cost of the obtained feasible solutions on real-world DBGs is substantially smaller than the corresponding cost obtained by existing greedy approaches.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching



© Giulia Bernardini, Huiping Chen, Grigorios Loukides, Solon P. Pissis, Leen Stougie, and Michelle Sweering;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 12; pp. 12:1–12:18

Leibniz International Proceedings in Informatics



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

Keywords and phrases string algorithms, graph algorithms, Eulerian graph, de Bruijn graph

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.12

Supplementary Material *Software (Source Code)*: <https://bitbucket.org/eulerian-ext/cpm2022/>
archived at `swh:1:dir:d7c2ca6a257600d6d7176b876370c456496af5a2`

Funding The work in this paper is supported in part by: the Netherlands Organisation for Scientific Research (NWO) through project OCENW.GROOT.2019.015 “Optimization for and with Machine Learning (OPTIMAL)” and Gravitation-grant NETWORKS-024.002.003; a CSC scholarship; the Leverhulme Trust RPG-2019-399 project; and 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.

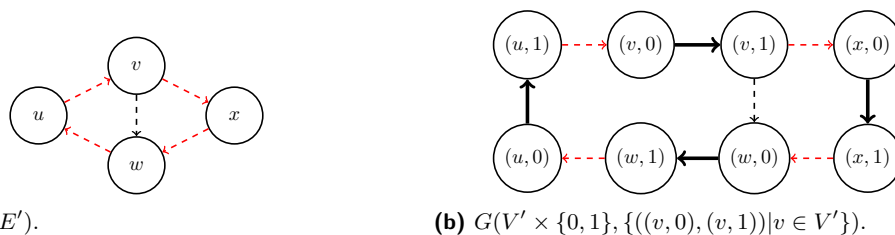
1 Introduction

We start with some basic definitions and notation on strings from [6]. Let $x = x[0] \cdots x[n-1]$ be a *string* of length $n = |x|$ over an integer alphabet $\Sigma = [0, \sigma)$ of σ *letters*. By Σ^k we denote the set of all strings of length $k > 0$. For any two positions i and $j \geq i$ of x , $x[i..j]$ is the *fragment* of x starting at position i and ending at position j ; it is represented in $\mathcal{O}(1)$ space by i and j . The fragment $x[i..j]$ is an *occurrence* of the underlying *substring* $p = x[i] \cdots x[j]$; we say that p occurs at *position* i in x . A *prefix* of x is a fragment of the form $x[0..j]$ and a *suffix* of x is a fragment of the form $x[i..n-1]$. By xy or $x \cdot y$ we denote the *concatenation* of strings x and y : $xy = x[0] \cdots x[|x|-1]y[0] \cdots y[|y|-1]$. Given strings x and y , a *suffix/prefix overlap* of x and y is a suffix of x that is a prefix of y .

The order- k de Bruijn graph (DBG) of a collection S of strings is a directed multigraph $G_{S,k}(V, E)$ such that V is the set of length- $(k-1)$ substrings of the strings in S and $G_{S,k}$ contains an edge (u, v) with multiplicity $m_{u,v}$, if and only if the string $u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and this string occurs exactly $m_{u,v}$ times in total in strings in S . When S is generated by a sequencing experiment from a genome, any Eulerian circuit of $G_{S,k}(V, E)$ corresponds to a single genome reconstruction [24, 20]. It goes without saying that genome assembly is one of the most important bioinformatics tasks [25, 29, 11, 22, 21, 26, 27, 18].

However, $G_{S,k}$ is almost surely not Eulerian in practice due to sequencing errors [21]. One could thus try to make it Eulerian by duplicating some of its *existing* edges [19]. In this case, one would naturally like to minimize the total cost of this extension. Even worse, $G_{S,k}$ would likely not be weakly connected, and thus edge duplication is not sufficient to make $G_{S,k}$ Eulerian. In this paper, we introduce the problem of making any arbitrary $G_{S,k}$ Eulerian by allowing not only to duplicate existing edges but to also add novel edges and nodes. The motivation for this is twofold. First, such a process would connect multiple components, which are often unconnected for the values of k used in practice. Second, as this is a generalization of the edge duplication problem [19], it would only reduce the total extension cost, even if the input graph is already weakly connected.

Let us now more formally lay the foundations of our work by first considering a general directed multigraph $G(V, E)$. A directed multigraph is called *Eulerian* if it has a circuit which uses each edge exactly once. Euler’s theorem tells us that a weakly *connected* directed multigraph is Eulerian if and only if every node is *balanced*: for any node $v \in V$ the in- and out-degree of v are equal. The *Eulerian Extension* (EE) problem on $G(V, E)$ asks for an Eulerian extension minimizing the total cost of the multiset \mathcal{A} of added edges according to some cost function. A smallest multiset \mathcal{A} over $V \times V$ such that $H(V, E \cup \mathcal{A})$ is Eulerian can be computed in the optimal $\mathcal{O}(|V| + |E|)$ time [5, 9]. We prove that the EE problem becomes significantly more challenging when a subset F of $V \times V$ is *forbidden* (i.e., not feasible):

(a) $\mathcal{H}(V', E')$.(b) $G(V' \times \{0, 1\}, \{(v, 0), (v, 1) \mid v \in V'\})$.

■ **Figure 1** (a) An instance of the directed Hamiltonian circuit with a solution in red and (b) the instance of Problem 1 to which it reduces. An Eulerian circuit in graph (b), with required bold edges and non-forbidden dashed edges, corresponds to finding a directed Hamiltonian circuit in graph (a). The corresponding solution is in red in graph (b).

► **Problem 1.** Given a directed multigraph $G(V, E)$ and a set $F \subset V \times V$, with $F \cap E = \emptyset$, find a multiset \mathcal{A} of edges over $(V \times V) \setminus F$ such that $H(V, E \cup \mathcal{A})$ is Eulerian and $|\mathcal{A}|$ is minimized; or report *FAIL* if not possible.

It should be clear that Problem 1 is equivalent to the EE problem when $F = \emptyset$. Note that Problem 1 is directly applicable on arbitrary dBGs, where the set F of forbidden edges is directly implied by the dBG definition: $(u, v) \in F$ if and only if $u[0] \cdot v \neq u \cdot v[k-2]$. We observe that adding nodes may shorten the length of the Eulerian circuit with respect to Problem 1. This observation leads naturally to the following generalization of Problem 1:

► **Problem 2.** Given a directed multigraph $G(V, E)$, a set $\mathcal{V} \supseteq V$, and a set $F \subset \mathcal{V} \times \mathcal{V}$, with $F \cap E = \emptyset$, find a multiset \mathcal{A} of edges over $(\mathcal{V} \times \mathcal{V}) \setminus F$ and a set of nodes $\mathcal{B} \subseteq \mathcal{V}$ such that $H(V \cup \mathcal{B}, E \cup \mathcal{A})$ is Eulerian and $|\mathcal{A}|$ is minimized; or report *FAIL* if not possible.

We will now prove that both Problems 1 and 2 are NP-hard by reducing from the directed Hamiltonian circuit [13] problem: Given a directed graph, decide whether there exists a directed circuit that visits every node of the graph exactly once.

► **Theorem 1.** Both Problems 1 and 2 are NP-hard.

Proof. We will first prove that Problem 1 is NP-hard. Consider an instance $\mathcal{H}(V', E')$ of the directed Hamiltonian circuit problem (inspect Figure 1a). We replace each node $v \in V'$ by two nodes $(v, 0), (v, 1)$ and an edge $((v, 0), (v, 1))$ with all the incoming edges incident to the tail $(v, 0)$ and all the outgoing edges incident to the head $(v, 1)$ (inspect Figure 1b).

Note that any sequence of adjacent edges on this modified graph alternates between new edges (corresponding to nodes in \mathcal{H} , the bold edges in Figure 1b) and old edges (corresponding to the edges in \mathcal{H} connecting those nodes, dashed in Figure 1b). Finding a Hamiltonian circuit in \mathcal{H} is equivalent to finding a circuit passing through all new edges in the modified graph exactly once. It follows that solving the directed Hamiltonian circuit problem on \mathcal{H} is equivalent to deciding whether the following instance of Problem 1 has a solution of size $|V'|$ (smaller solutions are not possible, larger solutions imply that \mathcal{H} is not Hamiltonian):

$$G(V, E) = G(V' \times \{0, 1\}, \{(v, 0), (v, 1) \mid v \in V'\})$$

$$F = (V \times V) \setminus (E \cup \{((u, 1), (v, 0)) \mid (u, v) \in E'\}).$$

Since the directed Hamiltonian circuit problem is NP-complete [13], it is NP-hard to decide whether a solution to Problem 1 has size at most $|V|/2$. Thus solving Problem 1 is NP-hard.

Note that Problem 2 is equivalent to the EE problem when $F = \emptyset$ and $\mathcal{V} = V$, and that Problem 2 is at least as hard as Problem 1: every instance of Problem 1 can be reduced to some instance of Problem 2 with $\mathcal{V} = V$. Therefore Problem 2 is NP-hard as well. ◀

Related Work. Both Problems 1 and 2 *on general graphs* are closely-related to the *Directed Rural Postman* problem (DRP) [23]: Given a directed (multi-)graph $\mathcal{G}(V', E')$ and a (multi-)set $\mathcal{R} \subseteq E'$ of *required* edges, we are asked to compute a minimum-cost circuit in \mathcal{G} including all edges in \mathcal{R} . It is easy to see that any instance of Problem 2 reduces to an instance of DRP with $V' = \mathcal{V}$, $E' = (\mathcal{V} \times \mathcal{V}) \setminus F$ and $\mathcal{R} = E$; a similar reduction works for Problem 1.

Problems 1 and 2 *on arbitrary dBGs* are different versions of the classic Shortest Common Superstring (SCS) problem [12]. In particular, Problem 2 is closely-related to the Multi-SCS problem [5]: Given a set S of strings and a multiplicity $f(s_i)$ of each $s_i \in S$, Multi-SCS asks for a shortest string containing at least $f(s_i)$ occurrences of each $s_i \in S$. When all strings in S are of length k , Multi-SCS is essentially Problem 2 on dBGs of order k .¹ Crochemore et al. showed that Multi-SCS can be solved in linear time when all input strings in S are of length 2. Cazaux and Rivals [4] presented a $\frac{1}{2}$ -approximation algorithm for Multi-SCS that maximizes the compression offered by the output string; and a 4-approximation algorithm for Multi-SCS that minimizes the length of the output string. In the Multi-SCCS problem [4], given a set S of strings and a multiplicity $f(s_i)$ of each $s_i \in S$, we are asked to construct a multiset C of cyclic strings of minimum total length such that every string in S occurs $f(s_i)$ times in the strings of C . Cazaux and Rivals [4] showed a linear-time implementation of a greedy algorithm that solves Multi-SCCS exactly (see also [3] for the SCCS problem).

Contributions. Let us now summarize our main contributions on arbitrary dBGs:

1. The reduction leading to Theorem 1 *does not* apply to the case in which the input to Problems 1 or 2 is an arbitrary dBG $G_{S,k}$, as not all edges implied by the reduction may be feasible in $G_{S,k}$. In Section 3 we prove that both problems are NP-hard even on arbitrary dBGs. It is easy to show that Problem 2 is NP-hard via a reduction from SCS. For Problem 1, we make a highly non-trivial reduction from 3-SAT; this is the most involved part of the paper. Since our ultimate goal is to make dBGs Eulerian, we next investigate the following two problems underlying EE in dBGs: *connect* and *balance*.
2. In Section 4, we show an *exact* greedy algorithm to make any $G_{S,k}$, consisting of $d > 1$ weakly connected components, weakly connected, by extending $G_{S,k}$ with a minimum-weight spanning tree, whose edges are paths on the underlying $G_{\Sigma,k}$ and weights are the corresponding path lengths. While there are many optimization criteria for connecting $G_{S,k}$, this way guarantees that no new unbalanced node is added. Our algorithm runs in $\mathcal{O}(|V|k \log d + |E|)$ time, which is *nearly optimal*, since the size of $G_{S,k}$ is $\Theta(|V|k + |E|)$. To achieve this time complexity, we simulate Kruskal's classic algorithm for computing minimum spanning trees [17] using an efficient method to compute shortest paths on the implicit $G_{\Sigma,k}$. This method employs an augmented and modified version of the Aho-Corasick machine [1], which we dynamically update every time we unite two components.
3. Balancing any $G_{S,k}$ with the smallest number of newly added edges can be reduced to Multi-SCCS. By employing the linear-time algorithm of Cazaux and Rivals [4] for Multi-SCCS, we obtain an $\mathcal{O}(k|E|)$ -time algorithm for balancing. In Section 5, we show an *exact* greedy algorithm for this problem that runs in the *optimal* $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time, where $|\mathcal{A}|$ is the total number of added edges. To achieve this time complexity, similar to Section 4, we simulate Cazaux and Rivals algorithm using another augmented and modified version of the Aho-Corasick machine.

¹ We say “essentially” because Multi-SCS asks for a shortest linear string, whereas Problem 2 asks for an Eulerian circuit, which on a dBG corresponds to a shortest cyclic string.

4. Although our main contributions here are theoretical, the algorithms we design are *practical*. In Section 6, we combine the algorithms of Sections 4 and 5 in one method that makes any $G_{S,k}$ Eulerian; and show experimentally that the cost of the feasible solutions obtained by this method on real-world DBGs constructed over sequencing data is substantially smaller than the cost of solutions obtained by existing string-based greedy approaches. This justifies the need for an approach specifically designed to extend DBGs.

2 Preliminaries

We fix an integer $k > 1$ and an integer alphabet Σ . Given a collection S of strings over Σ , we denote by $G_{S,k}(V, E)$ the de Bruijn graph (DBG) of order k of S (defined in Section 1). The cardinality of E (i.e., the sum of edge multiplicities) is $|E| = ||S|| - (k - 1)|S|$, where $||S||$ is the total length of the strings in S . Let $d^-(u)$ and $d^+(u)$ be, respectively, the in- and out-degree of node u of $G_{S,k}$. An undirected graph is said to be *connected* if for every pair u and v of nodes in the graph there exists a path from u to v . A directed graph is called *weakly connected* if by replacing all of its directed edges with undirected edges we obtain a connected (undirected) graph. A *spanning tree* of a weakly connected graph is a weakly connected subgraph which covers all the nodes of the graph with the minimum possible number of edges. A weakly connected graph $G_{S,k}$ is called *Eulerian* if every node u in $G_{S,k}$ is balanced, i.e., $d^+(u) = d^-(u)$. The DBG of order k of Σ^k is called the *complete de Bruijn graph* of order k over Σ ; we denote it by $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$, where $V_{\Sigma,k} = \Sigma^{k-1}$ and $E_{\Sigma,k} = \{(s[0..k-2], s[1..k-1]) \mid s \in \Sigma^k\}$.

Throughout, we assume that we are given the graph $G_{S,k}$ of an arbitrary string collection S , which we denote by $G(V, E)$.

3 Eulerian Extension of de Bruijn Graphs is NP-hard

In this section, we investigate the hardness of Problems 1 and 2 on arbitrary DBGs.

EULERIAN EXTENSION OF DE BRUIJN GRAPHS (EXTEND-DBG)

Input: A de Bruijn graph $G(V, E)$ of order k over alphabet Σ .

Output: An Eulerian graph $H(V \cup \mathcal{B}, E \cup \mathcal{A})$ with $\mathcal{B} \subseteq V_{\Sigma,k}$, \mathcal{A} over $E_{\Sigma,k}$ and minimized $|\mathcal{A}|$.

EXTEND-DBG can be solved in linear time when $k = 2$ [5]. When $k > 2$, EXTEND-DBG can be shown to be NP-hard via a simple reduction from the Length- k Shortest Common Superstring problem (k -SCS), a special case of the SCS problem in which all input strings are of length k . k -SCS is NP-hard, for any $k > 2$ [12]. Any instance of k -SCS on some alphabet Σ can be reduced to an instance of EXTEND-DBG on a DBG of order k over $\Sigma \cup \{\#\}$, with $\# \notin \Sigma$. The nodes of such DBG are the length- $(k - 1)$ prefixes and suffixes of each input string of k -SCS plus a special node $\#^{k-1}$. All the edges naturally correspond to the input strings of k -SCS, except for a special edge encoding $\#^k$. An Eulerian circuit of a minimum-size Eulerian extension of such graph then corresponds to a shortest common cyclic superstring \tilde{s} , which can be trivially transformed into a solution s to k -SCS (a shortest common linear superstring) by removing substring $\#^k$, so that the first letter of s is the first letter of \tilde{s} after the last $\#$, and the last letter of s is the last letter of \tilde{s} before the first $\#$.

Since a common superstring always exists (any concatenation of the strings is a cyclic superstring), the reduction implicitly assumes that it is always possible to connect a DBG to make it Eulerian. While this is true for EXTEND-DBG, as a path of length at most $k - 1$

exists between any two nodes if new nodes can be added to the graph, the assumption is wrong if we are *only* allowed to connect pairs of nodes of the input graph. If we tried to solve k -SCS via EXTEND-DBG with this restriction, we would only consider suffix/prefix overlaps of length $(k - 1)$ (corresponding to two consecutive edges of the dBG given by the reduction) or $(k - 2)$ (corresponding to edges added between two existing nodes when solving EXTEND-DBG), which is clearly wrong. It is therefore interesting to see if EXTEND-DBG remains NP-hard even with this restriction.

We start by formally defining this restricted version of the EE problem on dBGs.

RESTRICTED EULERIAN EXTENSION OF DE BRUIJN GRAPHS (R-EXTEND-DBG)
Input: A de Bruijn graph $G(V, E)$ of order k over alphabet Σ .
Output: An Eulerian graph $H(V, E \cup \mathcal{A})$ with \mathcal{A} over $(V \times V) \cap E_{\Sigma, k}$ and minimized $|\mathcal{A}|$; or report FAIL if not possible.

R-EXTEND-DBG can also be solved in linear time when $k = 2$ [5]. However, proving that R-EXTEND-DBG is NP-hard for $k > 2$ turns out to be significantly more challenging. We prove this via a reduction from 3-SAT, a well-known NP-hard problem [15]. Let $\{x_1, \dots, x_\ell\}$ be a set of *variables*. A *literal* is a variable x_i or a negated variable $\neg x_i$. A *clause* is a disjunction of literals. A *formula* $F = C_1 \wedge C_2 \wedge \dots \wedge C_n$ is in conjunctive normal form (CNF), if it is a conjunction of n clauses. The k -SAT problem is deciding whether a formula F in CNF form with every clause in F consisting of at most k literals is satisfiable.

► **Theorem 2.** *R-EXTEND-DBG is NP-hard if $G(V, E)$ is of order $k = 3$.*

Proof. Consider a 3-SAT instance with a set $\{x_1, \dots, x_\ell\}$ of ℓ variables and a formula $F = C_1 \wedge C_2 \wedge \dots \wedge C_n$ of n clauses, where each clause contains three literals. We construct a dBG with $k = 3$ for which solving R-EXTEND-DBG problem tells us whether F is satisfiable or not. The dBG is constructed over the alphabet:

$$\Sigma = \{x_i, \neg x_i, y_i, z_i\}_{i \in [1, \ell]} \cup \{a_j, b_j\}_{j \in [1, n]} \cup \{x_{ij}, \neg x_{ij}\}_{i \in [1, \ell]}^{j \in [1, n]} \cup \{c_1, c_2\},$$

where the letters within each set are pairwise distinct and all sets are pairwise disjoint. The dBG will consist of the union of some gadget subgraphs, as described next. For each variable x_i , we define a variable-gadget $G_v(x_i)(V_v^i, E_v^i)$, conceptually corresponding to $x_i \vee \neg x_i$, with five nodes and four edges:

$$V_v^i = \{z_i x_i, x_i y_i, y_i z_i, z_i \neg x_i, \neg x_i y_i\}, \quad E_v^i = \{x_i y_i z_i, y_i z_i x_i, \neg x_i y_i z_i, y_i z_i \neg x_i\}.$$

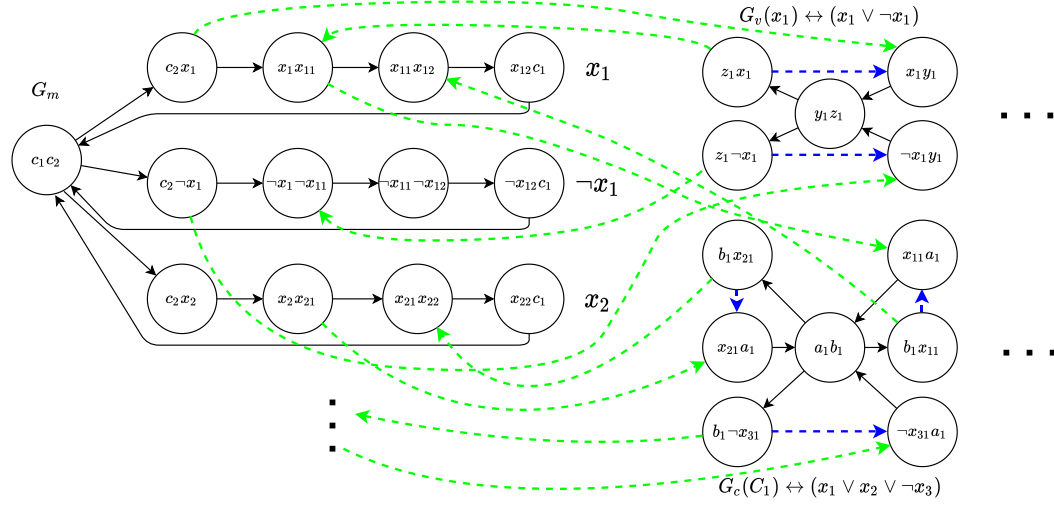
For every clause $C_j = l_1^j \vee l_2^j \vee l_3^j$, with $l_1^j, l_2^j, l_3^j \in \{x_i, \neg x_i\}_{i \in [1, \ell]}$, we define a corresponding clause-gadget $G_c(C_j)(V_c^j, E_c^j)$, with seven nodes and six edges:

$$V_c^j = \{a_j b_j, b_j l_{1j}^j, l_{1j}^j a_j, b_j l_{2j}^j, l_{2j}^j a_j, b_j l_{3j}^j, l_{3j}^j a_j\},$$

$$E_c^j = \{a_j b_j l_{1j}^j, a_j b_j l_{2j}^j, a_j b_j l_{3j}^j, l_{1j}^j a_j b_j, l_{2j}^j a_j b_j, l_{3j}^j a_j b_j\}.$$

In this definition, l_{tj}^j for each $t \in \{1, 2, 3\}$ are just placeholders, such that $l_{tj}^j = x_{ij}$ if $l_t^j = x_i$ and $l_{tj}^j = \neg x_{ij}$ if $l_t^j = \neg x_i$: for example, in Figure 2, $l_{11}^1 = x_{11}$, $l_{21}^1 = x_{21}$, and $l_{31}^1 = \neg x_{31}$ because $C_1 = (x_1 \vee x_2 \vee \neg x_3)$.

Finally, the main component-gadget $G_m(V_m, E_m)$ is daisy-shaped: it has a central node and 2ℓ *petals*, one for each variable x_i and negated variable $\neg x_i$, each consisting of a simple cycle of length $n + 3$ beginning and ending at the central node. In the following definition we use l_j for each $j \in [1, n]$ again as placeholders, to be replaced with x_{ij} in the petal of x_i , and with $\neg x_{ij}$ in the petal of $\neg x_i$, for all $i \in [1, \ell]$:



■ **Figure 2** An instance of R-EXTEND-DBG that is equivalent to the 3-SAT problem $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$. The first half of the G_m component is shown on the left. The edges of the gadgets are shown in black. Other feasible edges within components are shown in blue, while feasible edges between components are shown in green.

$$V_m = \{c_1 c_2\} \cup \{c_2 l, ll_1, l_1 l_2, \dots, l_{n-1} l_n, l_n c_1 \mid l \in \{x_i, \neg x_i\}_{i \in [1, \ell]}\}$$

$$E_m = \{c_1 c_2 l, c_2 ll_1, ll_1 l_2, l_1 l_2 l_3, \dots, l_{n-1} l_n c_1, l_n c_1 c_2 \mid l \in \{x_i, \neg x_i\}_{i \in [1, \ell]}\}.$$

Note that each of these gadgets is connected and they are all mutually disjoint. An example is shown in Figure 2. To balance the nodes, we need to add at least 2 edges for each variable-gadget and at least 3 edges for each clause-gadget. For example, adding the blue edges in Figure 2 would balance the graph. However, to additionally make the graph connected, we would need to add other additional edges.

We want to minimize the number of added edges to make the graph balanced and connected. That is equivalent to minimizing the number of nodes visited with multiplicity. We will prove that we need at least $3\ell(n+3) + 6\ell + 9n$ edges and that this is sufficient if and only if the formula F is satisfiable. Consider the cut separating all gadgets G_v from the other components. A gadget $G_v(x_i)$ can only be reached from nodes $c_2 x_i$ and $c_2 \neg x_i$ of G_m (see the green edges in Figure 2). Therefore, there must be at least ℓ additional edges leaving nodes of the form $c_2 l$ of G_m , one for each $G_v(x_i)$, with $l \in \{x_i, \neg x_i\}$. Such nodes are then visited at least 3ℓ times in total, as all the 2ℓ of them must be visited at least once by following the solid edges of G_m , and ℓ of them (one for each $G_v(x_i)$) must be visited at least once more. Since the graph must be balanced, the number of edges traversing the cut reaching the G_v gadgets equals the number of edges leaving them. Thus, the nodes of G_m of the form ll_j , which are the only ones reachable from the G_v gadgets, are also visited at least 3ℓ times. Moreover, in order for G_m to remain balanced, the nodes of the form $l_{j-1} l_j$ and $l_j l_{j+1}$ (with $l_0 = l, l_{n+1} = c_1$) on the petals on which nodes of the form $c_2 l$ or ll_j are visited twice must be visited at least twice too. It follows that there are at least $3\ell(n+3)$ visits of the nodes in G_m (each of the $n+3$ nodes on each of the 2ℓ petals are visited at least once; and the nodes of at least ℓ petals must be visited twice), while the number of visits in the G_v components is at least 6ℓ and the number of visits in the G_c components is at least $9n$ (in order for them to be balanced), yielding the desired *lower bound*.

12:8 Making de Bruijn Graphs Eulerian

For this bound to be tight, there cannot be any extra visits to nodes in G_v and G_c gadgets. Hence, for the graph to be balanced, we need the number of visits to $c_2l, ll_1, l_1l_2, \dots, l_n c_1$ to be equal for each fixed literal l . It follows that, in order to remain within $3\ell(n+3)$ visits to nodes of G_m , for exactly one of the literals x_i and $\neg x_i$ these nodes are visited twice, while for the other literal these nodes are visited only once. Note that we can only connect to $G_c(C_j)$ if for one of its literals l node $l_{j-1}l_j$ of G_m is visited twice. Therefore we can only connect the graph with the lower bound number of nodes if F is satisfiable.

We will now show that if F is satisfiable then this number is enough. We balance each G_v gadget with 2 edges and each G_c with 3 edges (blue edges in Figure 2). We also add the length- $(n+3)$ cycles in G_m corresponding to the ℓ true literals. We will show that we can connect each of the G_v and each of the G_c gadgets to G_m . That we can do so without increasing the number of edges results from the following claim.

▷ **Claim.** Let X and Y be two distinct balanced connected components with non-required (i.e., appearing with a higher multiplicity than in the original graph) edges azb and czd in X and Y , respectively. Then there exists a connected balanced graph on the nodes of X and Y with the same number of edges.

Proof. Since azb and czd are non-required, we can remove them. Note that both X and Y are still connected: since az and zb (resp. cz and zd) are the only unbalanced nodes in X (resp. Y), they must lie in the same component. Now add azd and czb . These edges are feasible because all endpoints are already present in the graph. This rebalances the graph and connects the two components. ◁

Since either x_i or $\neg x_i$ is true, one of them is the mid symbol of a non-required edge in G_m , thus we can link all G_v gadgets to G_m using the interchange of edges described in the proof of the claim (in Figure 2, we trade the blue edge in $G_v(x_i)$ and one copy of (c_2x_i, x_ix_{i1}) for the green edges, or do it for $\neg x_1$). Moreover, since we assumed that F is satisfiable, at least one literal l of each C_j is true. The edges in $G_c(C_j)$ and G_m with l in the middle are non-required, thus we can link all G_c gadgets to G_m , making the graph connected. ◀

4 Connecting de Bruijn Graphs with Paths in Near-Optimal Time

We present an exact $\mathcal{O}(|V|k \log d + |E|)$ -time algorithm for connecting any dBG $G(V, E)$ of order k by arranging its $d > 1$ weakly connected components in a tree. The tree nodes are the components themselves and the tree edges are paths of minimum total length between such components. Since our ultimate goal is to both connect and balance G , by connecting G in this way, we make sure that the new nodes we add are already *balanced*.² To formally define the connecting problem we consider, we first need the following definition.

► **Definition 3 (Condensed Graph).** *Given a dBG $G(V, E)$ of order k over an alphabet Σ with a set \mathcal{C} of weakly connected components, its condensed graph $\widehat{G}(\widehat{V}, \widehat{E})$ is a weighted directed multigraph whose nodes \widehat{V} are in a bijection with \mathcal{C} . The edges have integer weights in $[1, k-1]$: there is an edge $(i, j) \in \widehat{E}$ for each pair of nodes $u_i \in C_i, u_j \in C_j$, with $C_i, C_j \in \mathcal{C}$, and its weight is the length of a shortest path from u_i to u_j in the complete dBG $G_{\Sigma, k}$.*

We now formally define the problem we consider in this section.

² Note that the graph resulting from this algorithm would, in general, not be balanced.

■ **Algorithm 1** CONNECTING A DE BRUIJN GRAPH WITH PATHS.

-
- 1: Find the d connected components of G , construct, and preprocess the AC machine of the nodes of G
 - 2: **for** $i \in [1, d - 1]$ **do**
 - 3: Select a backward edge (s, u) encoding a longest suffix/prefix overlap
 - 4: $(s_\alpha, s_\beta) \leftarrow \text{components}(s, u)$
 - 5: Add to \mathcal{P} the path from s_α to s_β , which connects components α and β
 - 6: Update the labels of the states and the backward edges
 - 7: Prune the backward edges connecting two single-color states of the same color
-

CONNECTING DE BRUIJN GRAPHS WITH PATHS (CONNECT-DBG)

Input: A de Bruijn graph $G(V, E)$ of order k over alphabet $\Sigma = [0, \sigma], \sigma \leq (k - 1)|V|$.

Output: A minimum-weight spanning tree \mathcal{T} of the condensed graph \widehat{G} of G .

A solution \mathcal{T} to CONNECT-DBG naturally corresponds to a set \mathcal{P} of paths on $G_{\Sigma, k}$ that make G weakly connected: an edge (i, j) of \mathcal{T} corresponds to a shortest path from some node $u_i \in C_i$ to some node $u_j \in C_j$, and in turn, by the definition of DBG, such path is determined by the longest suffix/prefix overlap of u_i and u_j . Our algorithm essentially mimics the Kruskal algorithm [17] on the condensed graph \widehat{G} . However let us stress that we do not construct \widehat{G} explicitly, as it would take $\Theta(k|V|^2)$ time, and moreover using the Kruskal algorithm as-is would require $\mathcal{O}(|V|^2 \log |V|)$ time (because \widehat{G} has $\Theta(|V|^2)$ edges). We rather exploit the properties of DBGs and compute \mathcal{T} by searching for longest suffix/prefix overlaps of the nodes of G . Our algorithm greedily selects, at each iteration, a longest suffix/prefix overlap (encoding a shortest path) of any two nodes that belong to different components. To do so, we define an augmented and modified version of the Aho-Corasick (AC) machine [1] of all the nodes of G , which we dynamically update every time we unite two components. The AC machine generalizes the Knuth-Morris-Pratt [16] algorithm for a set of strings. Informally, it is a finite-state machine that resembles a trie with additional *backward edges* (also called *failure transitions*) between the various states. There is exactly one failure transition $f(u) = v$ from each state u . Suffix/prefix overlaps can then be found using the following lemma.

► **Lemma 4** (Aho-Corasick lemma [1]). *Let u and v be two strings representing two distinct states of the AC machine, and identify the states with such strings. Then, $f(u) = v$ if and only if v is the longest proper suffix of u that is also a prefix of some string in the machine.*

We first assign each connected component of G a distinct color, and modify the AC machine of the nodes of G so that we maintain the three following invariants, in any iteration i of the algorithm:

- I1. Each state has up to $d - i$ colors. Each terminal state is colored by its current connected component; each non-terminal state has the union of colors of the descending subtree.
- I2. There are no backward edges connecting two single-color states of the same color.
- I3. There are up to $k - 1$ backward edges outgoing from each terminal state s , each labeled by the color of s . There are no backward edges connecting non-terminal states.

Intuitively, we prune each backward edge connecting two single-color states colored α , because in this case all the nodes of G with the corresponding suffix/prefix overlap are in the same component α , and thus this edge cannot be used to unite unconnected components of G .

Algorithm 1 consists of four main phases: (i) preprocessing (Line 1); (ii) greedily selecting backward edges (Line 3); (iii) recoloring (Line 6); and (iv) pruning (Line 7).

(i) Preprocessing. We first identify the connected components of G , build the AC machine of its nodes and color its states according to invariant I1. We maintain the colors of a state u using a list LC_u and a dynamic hashtable HC_u . A key c of HC_u is a color of u , and its value is a pair of pointers: one to the position of c in LC_u , the second to any terminal state colored c below u . We also keep a counter $\text{colors-cnt}(u)$ of the number of distinct colors of u .

From each terminal state s , we then follow the unique path of backward edges to the root and, for each state u on this path, we add a backward edge (s, u) of the same color as s , according to invariant I3. We maintain the backward edges outgoing from s with a list LB_s of their heads and a dynamic hashtable HB_s . A key u of HB_s is a state in LB_s (the head of an outgoing backward edge); its value is a pointer to the position of u in LB_s .

We keep the backward edges incoming to u with a list LE_u of their tails, and we maintain their colors with a dynamic hashtable HE_u . A key c of HE_u is the color of one such edges; its value is the list $HE_u[c]$ of the positions in LE_u of the edges colored c . To add an incoming backward edge (s, u) of color α to u , we first append s to LE_u ; we then look up the value of α in HE_u . If we find it, we append to $HE_u[\alpha]$ the position of s in LE_u ; otherwise, we create key α and initialize $HE_u[\alpha]$ with the position of s in LE_u . Finally, we prune all the backward edges connecting two non-terminal states and, for each non-terminal state u colored c with $\text{colors-cnt}(u) = 1$, we query HE_u and prune from the machine all the backward edges (s, u) represented in the list $HE_u[c]$ (using HB_s). For each color c , we also maintain a global counter $\text{global-cnt}(c)$ of the total number of states and backward edges colored c .

(ii) Selecting backward edges. We select the backward edges in an order given by a reverse BFS, starting from the deepest states and proceeding level by level towards the root. At each visited state u of string depth (level) ℓ , we search for incoming backward edges, encoding a suffix/prefix overlap of length ℓ (Lemma 4), in the list LE_u . We select an edge of LE_u at each subsequent iteration, and only when LE_u is empty we move on to the next state. Note that the same backward edge (s, u) can be selected in multiple iterations, as it can be used to unite the component α of s with all the components coloring u , thus it will only be pruned when all such components are united with α .

To unite two components using a suffix/prefix overlap implied by a backward edge (s, u) , we select two appropriate nodes of G by $\text{components}(s, u)$, which takes as input a terminal state s of color α and a non-terminal state u , and outputs $s_\alpha = s$ and a terminal state s_β descending from u of some color $\beta \neq \alpha$; or returns FAIL if no such s_β exists (i.e., only when s and u both have the same single color α). We also add the path from s_α to s_β into \mathcal{P} .

(iii) Updating the colors. When we unite two components α and β , we change all labels α into β if $\text{global-cnt}(\alpha) \leq \text{global-cnt}(\beta)$; and change β into α otherwise. At each iteration one color is removed from the machine, and thus after iteration i there are $d - i$ distinct colors (I1). We update the colors of the states starting from the terminals and proceeding towards the root. To change color α to β in a non-terminal state u , we look up α in the hashtable HC_u , delete it from the list LC_u by following the first pointer of $HC_u[\alpha]$ and remove the entry of α from HC_u . We then look up β in HC_u : if it is not there, we insert key β in HC_u with second pointer equal to the second pointer of $HC_u[\alpha]$ and append β to LC_u . We also update the counter $\text{colors-cnt}(u)$ of the number of colors of u and the counter $\text{global-cnt}(\beta)$ of the total number of states and edges colored β accordingly: if β was already in HC_u , we decrease $\text{colors-cnt}(u)$ by one (because of deleting α) and leave $\text{global-cnt}(\beta)$ unchanged; otherwise we leave $\text{colors-cnt}(u)$ unchanged and increase $\text{global-cnt}(\beta)$ by one.

When we change the color α of a terminal state s into β , we must also change to β the color of the backward edges from s . To do so, for each edge (s, u) we look up α in HE_u . If we do not find it, then the color of all the edges with head u has already been updated. Otherwise, we access the list pointed by $\text{HE}_u[\alpha]$ (which contains s and possibly other terminal states colored α). We insert β in HE_u and copy $\text{HE}_u[\alpha]$ in $\text{HE}_u[\beta]$, if β was not already there; or we append $\text{HE}_u[\alpha]$ to the list $\text{HE}_u[\beta]$, if $\text{HE}_u[\beta]$ already existed. In both cases, we set $\text{global-cnt}(\beta)$ to $\text{global-cnt}(\beta) + \text{len}(\text{HE}_u[\alpha])$ and remove the entry of α from HE_u .

(iv) Pruning. If, after updating the colors in the machine, the only remaining color of a non-terminal state u is β (i.e., $\text{colors-cnt}(u) = 1$), we query HE_u with key β . If we find β , we prune all edges (s, u) with s in the list pointed by $\text{HE}_u[\beta]$, and also delete the entry for β from HE_u . To prune an edge (s, u) , we look up u in HB_s , delete u from LB_s following the pointer $\text{HB}_s[u]$ and finally delete the entry of u from HB_s . This ensures invariant I2, because the backward edges in $\text{HE}_u[\beta]$ are all and only those of color β with head u .

► **Theorem 5.** *CONNECT-DBG can be solved in $\mathcal{O}(|V|k \log d + |E|)$ time using $\mathcal{O}(k|V| + |E|)$ working space.*

Proof. For the correctness of Algorithm 1 we first show that, at any iteration, the backward edges in our machine represent all suffix/prefix overlaps of nodes in two currently distinct components of G . By Lemma 4, for each state u on the path of backward edges from a terminal state s to the root in the AC machine of V , the partial path ending at u encodes a suffix/prefix overlap between s and any terminal state below u ; and each possible suffix/prefix overlap between s and any other node in V corresponds to one such partial path. During preprocessing, we replace each such partial path with a single backward edge (s, u) ; and by invariants I1-I3, we only keep the backward edges (s, u) encoding an overlap between s and some node of V in a different component (some other nodes in the same component may have the same overlap, but function `components` makes the algorithm ignore them).

The correctness of Algorithm 1 then directly follows from the above and from the correctness of the Kruskal algorithm [17] for computing a minimum-weight spanning tree.

For the complexity analysis, we bound the time for each of the four main phases as follows: (i) preprocessing by $\mathcal{O}(k|V| + |E|)$; (ii) selecting backward edges by $\mathcal{O}(k|V|)$; (iii) recoloring by $\mathcal{O}(|V|k \log d)$; and (iv) pruning by $\mathcal{O}(k|V|)$. The working space is bounded by $\mathcal{O}(k|V| + |E|)$, the size of G .

(i) Preprocessing. Computing the connected components of G and giving each one a color $c \in [1, d]$ requires $\mathcal{O}(|V| + |E|)$ time, with $|E|$ the number of *distinct* edges of G [14]. Building the AC machine of V takes $\mathcal{O}(k|V|)$ time because each string is of length $k - 1$ [1, 8]. To implement HE, HC and HB we use perfect hashing, supporting insertions and deletions of key-value pairs, and to retrieve any entry with a given key. The running time per operation is $\mathcal{O}(1)$ *with high probability* [7, Theorem 1.1]. Colors are assigned to the states of the AC machine in $\mathcal{O}(k|V|)$ time, starting from the terminal states and proceeding up to the root.

For each of the $|V|$ terminal states, we follow a path of backward edges of length up to $k - 1$ (as the string depth of the machine is $k - 1$ and backward edges connect states with strictly decreasing string depth) and add up to $k - 2$ backward edges in $\mathcal{O}(1)$ time per edge (s, u) by using the hashables HE_u , HC_u and HB_s . This takes $\mathcal{O}(k|V|)$ time in total. Finally, the initial pruning of backward edges requires $\mathcal{O}(k|V|)$ total time, as we visit each non-terminal state u , look up at most one key in HE_u , and possibly delete the edges (s, u) represented by the list stored at HE_u by using the hashtable HB_s .

12:12 Making de Bruijn Graphs Eulerian

(ii) **Selecting backward edges.** Each step of the reverse BFS takes $\mathcal{O}(1)$ time, and we abort it when we have selected $d - 1$ backward edges. A state can be visited multiple times only if there are still incoming backward edges that can be selected, and in this case we select one of them at each visit. Since $d \leq |V|$, the whole visit requires $\mathcal{O}(k|V| + d) = \mathcal{O}(k|V|)$ time in total. Moreover, for each selected edge (s, u) , we compute $\mathbf{components}(s, u)$ in $\mathcal{O}(1)$ time by visiting up to two elements in the color list of u LC_u .³

(iii) **Updating the colors.** Changing color α to β in a non-terminal state u takes $\mathcal{O}(1)$ time by using HC_u . Changing from α to β the color of all the backward edges (s, u) outgoing from a terminal state s requires accessing the list at $\text{HE}_u[\alpha]$ and appending the whole list $\text{HE}_u[\alpha]$ to the (possibly empty) list $\text{HE}_u[\beta]$. This procedure amortizes to $\mathcal{O}(1)$ time for each recolored backward edge.

We next show that the algorithm does $\mathcal{O}(|V|k \log d)$ recolorings of states and edges over all iterations via an auxiliary data structure: a rooted binary tree with the d colors (components) as leaves. Each leaf c is weighted with $\mathbf{global_cnt}(c)$, which is the total number of occurrences of color c the machine. Internal nodes in the tree represent the component unions done by the algorithm, each weighted with the number of states and backward edges that are recolored in the corresponding step, i.e., the lightest weight of its two children. We remark that this tree is not part of the algorithm, but rather it is just a conceptual aid to count the number of color updates in the worst case. The total number of recolorings done by Algorithm 1 is given by the sum of all the weights on the internal nodes. Let $f(w, d)$ be the maximum such sum on a tree with d leaves with a total weight of w . We will prove the following claim by induction on d .

▷ **Claim.** $f(w, d) \leq w \log_2(d)$.

Proof.

Induction basis: If $d = 1$, then the tree consists of the root and one leaf, so $f(w, d) = 0 = w \log_2(d)$.

Induction hypothesis: For all $d' < d$, we have $f(w, d') \leq w \log_2(d')$.

Induction step: Consider a situation with d colors. The root of the tree corresponds to the final recoloring, when the last two components are merged. The two subtrees starting from its children have weight w_1 and w_2 and d_1 and d_2 leaves, respectively. Without loss of generality $w_1 \leq w_2$. We now bound $f(w, d)$:

$$\begin{aligned} f(w, d) &\leq f(w_1, d_1) + f(w_2, d_2) + \min(w_1, w_2) \leq w_1 \log_2(d_1) + w_2 \log_2(d_2) + w_1 \\ &\leq w_1 \log_2(\min(d_1, d_2)) + w_2 \log_2(\max(d_1, d_2)) + w_1 \\ &= w_1 \log_2(2 \min(d_1, d_2)) + w_2 \log_2(\max(d_1, d_2)) \\ &\leq (w_1 + w_2) \log_2(d_1 + d_2) = w \log_2(d). \end{aligned} \quad \triangleleft$$

We conclude that $f(w, d) \leq w \log_2(d)$ for all $d \in \mathbb{N}$. Observe that $w = \mathcal{O}(k|V|)$, because the color of each of the $|V|$ terminal states propagates to at most $k - 2$ non-terminal states (the depth of the machine is $k - 1$), and there are up to $k - 2$ backward edges from each terminal state; and therefore $w \log_2(d) = \mathcal{O}(|V|k \log d)$.

(iv) **Pruning.** Pruning a backward edge (s, u) requires $\mathcal{O}(1)$ time using the hash tables HE_u and HB_s . Since there are up to $k|V|$ backward edges, deletions take $\mathcal{O}(k|V|)$ time overall. ◀

³ To compute $\mathbf{components}(s, u)$ with s colored α in $\mathcal{O}(1)$ time, we maintain a pointer in the header of the color list of each state. We either select the color β of the header of LC_u or, if it is equal to α , we advance the pointer, which guarantees finding $\beta \neq \alpha$, as the lists do not contain duplicates. In both cases we follow the pointer at $\text{HC}_u[\beta]$ to find s_β in $\mathcal{O}(1)$ time.

5 Balancing de Bruijn Graphs in Optimal Time

We present an exact $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ -time algorithm for balancing any dBG $G(V, E)$ of order k so that the number $|\mathcal{A}|$ of newly added edges is minimized. As a consequence of the Euler's theorem, when the input graph G is weakly connected, our algorithm makes it Eulerian with the smallest possible cost. Let us first formally define the problem.

BALANCING DE BRUIJN GRAPHS (BALANCE-DBG)

Input: A de Bruijn graph $G(V, E)$ of order k over alphabet $\Sigma = [0, \sigma]$, $\sigma \leq (k - 1)|V|$.

Output: A balanced graph $H = (V \cup \mathcal{B}, E \cup \mathcal{A})$ with $\mathcal{B} \subseteq V_{\Sigma, k}$, \mathcal{A} over $E_{\Sigma, k}$ and minimized $|\mathcal{A}|$.

It is easy to see that BALANCE-DBG can be reduced to the Multi-SCCS problem (defined in Section 1). In particular, BALANCE-DBG reduces to an instance of Multi-SCCS with $S = E$. A greedy algorithm, which keeps merging the suffix and prefix with the longest overlap until we are left with only cyclic strings, is known to solve Multi-SCCS exactly [4]. Cazaux and Rivals showed a linear-time implementation of this algorithm [4, Theorem 10], which implies an $\mathcal{O}(k|E|)$ -time algorithm for BALANCE-DBG. In a dBG, this algorithm corresponds to finding a minimum-weight matching between the heads and the tails of the edges, where the weight is given by the length of a shortest directed path from the head to the tail. The greedy algorithm constructs a matching by repeatedly adding a feasible edge of minimum weight. Although such greedy algorithm is not exact on general weighted bipartite graphs [10], it turns out to be optimal in the *special case* of dBGs following from the optimality of the greedy algorithm for Multi-SCCS [4]. In balanced nodes, all heads and tails can be matched for a cost of zero. The greedy algorithm will match those first, so it thus suffices to only match up the excess heads and tails at unbalanced nodes. In what follows, we describe a different implementation of the greedy strategy which gives optimal time complexity for the special instances arising from BALANCE-DBG. Similar to Section 4, we employ an augmented and modified version of the AC machine.

Let $Z^+ \subset V$ be the nodes with higher out-degree d^+ than in-degree d^- , and $Z^- \subset V$ the nodes with $d^- > d^+$. We construct the AC machine of $Z^+ \cup Z^-$ and preprocess it as follows. We label by $-$ each terminal state $s \in Z^-$ and initialize a counter $m_s = d_s^- - d_s^+$; we also label by $+$ each state encoding a *prefix* of $s \in Z^+$ and initialize a counter $m_s = d_s^+ - d_s^-$ for s . In addition, for every non-terminal state u , we compute a set $D(u)$ of all its descendant terminal states $s \in Z^+$. From each terminal state s labelled $-$, we follow the unique path of backward edges to the root: for each non-terminal state u labelled $+$ on this path, we add a backward edge (s, u) . We finally prune all backward edges that do not link a $-$ state with a $+$ state: we maintain the backward edges of the machine similar to Section 4. Our algorithm first sets $\mathcal{A} = \emptyset$ and $\mathcal{B} = \emptyset$ and then iteratively adds edges to \mathcal{A} and nodes to \mathcal{B} as follows.

We traverse the machine in reverse BFS order starting at the terminal states (this traversal was proposed by Ukkonen in [28]). When we encounter the head of a backward edge at a state u , we find the terminal state $s^- \in Z^-$ at the tail of the edge and any terminal state $s^+ \in D(u) \subseteq Z^+$. Let s be the shortest string with prefix s^+ and suffix s^- (as an application of Lemma 4). We add $\min\{m_{s^+}, m_{s^-}\}$ copies of $s[i..i+k-1]$, for all $i \in [0, |s| - k]$, to \mathcal{A} ; we add $s[i..i+k-2]$ to \mathcal{B} if it is not in $V \cup \mathcal{B}$; and we decrease both m_{s^+} and m_{s^-} by $\min\{m_{s^+}, m_{s^-}\}$. When $m_{s^-} = 0$, we delete all backward edges starting from the terminal state s^- and update the edge lists of their heads accordingly. When $m_{s^+} = 0$, we delete s^+ from the D sets of its ancestors. If any $D(u)$ becomes empty, we delete all incoming edges at state u . The algorithm terminates when there are no more backward edges in the machine.

► **Theorem 6.** *BALANCE-DBG can be solved in the optimal $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$ time using $\mathcal{O}(k|V| + |E|)$ working space.*

Proof. The correctness of the algorithm follows from the observation that in order to solve BALANCE-DBG via Multi-SCCS it suffices to consider the nodes in $Z^+ \cup Z^-$, and from the fact that a greedy strategy solves Multi-SCCS exactly [4, Theorem 10] (see the discussion above). We thus conclude that the presented algorithm is correct.

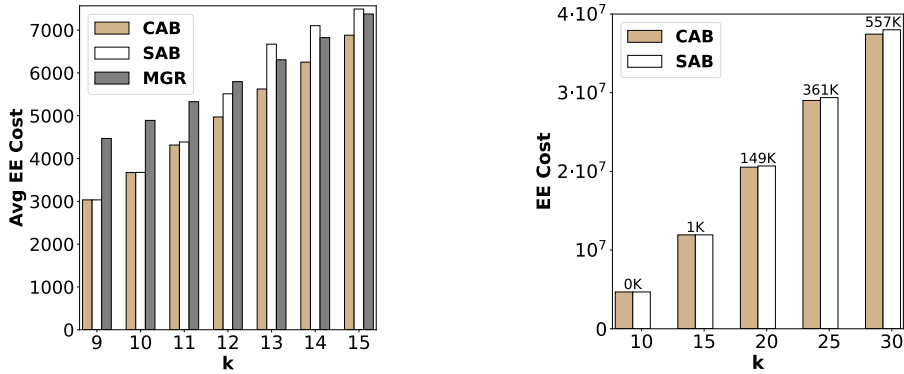
Constructing Z^+ and Z^- and computing the initial values of m counters takes $\mathcal{O}(|E|)$ time via a traversal of G . Constructing, traversing and updating the AC machine takes $\mathcal{O}(k|V|)$ time because $|Z^+ \cup Z^-| \leq |V|$ and each node in V is a string of length $k - 1$. Note that any edge added in \mathcal{A} and any node added in \mathcal{B} can be represented in $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ space using two nodes in V . Thus, the total time required to output graph $H = (V \cup \mathcal{B}, E \cup \mathcal{A})$ is $\mathcal{O}(k|V| + |E| + |\mathcal{A}|)$. The working space is bounded by $\mathcal{O}(k|V| + |E|)$, the size of G . ◀

6 Experiments

Methods and Setup. We designed a method for EXTEND-DBG based on our theoretical findings. The method first connects the input dBG based on our exact algorithm underlying Theorem 5 and then balances it by our exact algorithm underlying Theorem 6. We remark that both these algorithms are exact but their combination is generally not, which is consistent with EXTEND-DBG being NP-hard. To further help balancing, our method connects the graph using only unbalanced nodes. Our method is called CAB (for *connect* and *balance*).

We compared CAB to the $\frac{1}{2}$ -approximation algorithm for Multi-SCS that maximizes the compression offered by the output string [4]. We refer to this algorithm as MGR (for Multi-SCS *Greedy*). To specifically examine the impact of our connect framework on extension cost, we also designed a “hybrid” method, referred to as SAB (for *SCS* and *balance*). SAB first connects the graph based on the greedy algorithm [2] for SCS and then balances it by the algorithm of Theorem 6, as CAB does. The intuition is that any (shortest) common superstring s of set V corresponds to a connected extended dBG. To connect G , we consider all the potential additional edges implied by s and greedily add to G a smallest subset of them that makes G connected. The pseudocode of SAB is provided in Algorithm 2 of Appendix A.

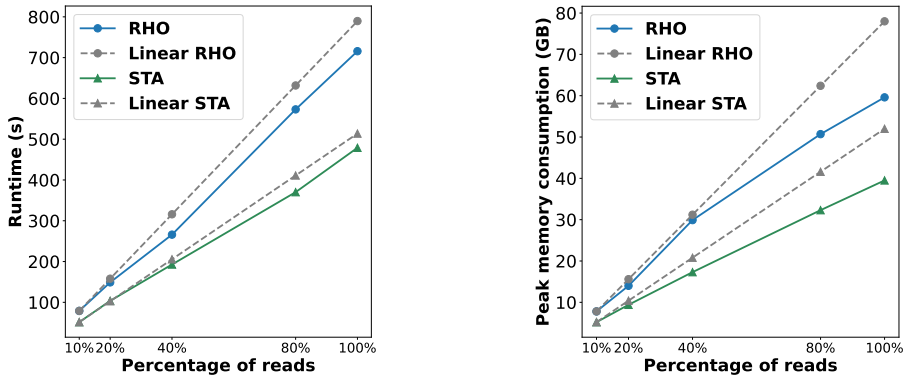
We implemented the above methods in C++ and ran them on a single core of an AMD Opteron 6386 SE 2.8GHz CPU with 252GB RAM running GNU/Linux. Our source code is available at <https://bitbucket.org/eulerian-ext/cpm2022/>. We used two whole-genome shotgun benchmark datasets that are available from <http://gage.cbc.umd.edu/data/index.html>: (i) *Rhodobacter sphaeroides* (RHO); and (ii) *Staphylococcus aureus* (STA). The number of reads in RHO and STA is 2,050,868 (Library 1) and 1,294,104 (Library 1), respectively. In both datasets, the average read length is 101bp and the insert length is 180bp. Tables 1a and 1b in Appendix A show the characteristics of the two datasets. Although MGR works in polynomial time [4], no efficient (e.g., linear-time or near-linear-time) implementation of MGR is known. This is in contrast to SAB, which uses a linear-time implementation of the greedy algorithm for SCS [2] to connect the graph. Since our implementation of MGR works in quadratic time in the input size, we used randomly selected *samples* for each dataset and every k in the comparison against MGR. The samples were constructed by selecting 650 reads from each dataset uniformly at random and had roughly 40K to 55K nodes and 60K edges.



(a) RHO samples.

(b) RHO.

■ **Figure 3** (a) Average Eulerian Extension (EE) cost vs. k over five random samples of RHO. (b) EE cost vs. k on the whole RHO dataset. The difference between the EE costs of SAB and CAB is shown on the top of each pair of bars (K stands for thousands).



(a) Runtime of CAB.

(b) Memory of CAB.

■ **Figure 4** Runtime and peak memory consumption vs. reads of CAB on RHO and STA for $k = 30$. The solid lines are the results for CAB; the dashed lines are the results that would be produced by a linear scaling of the algorithm.

Eulerian Extension (EE) Cost. Figure 3a shows the average EE cost of all methods on five random samples of the RHO dataset, for varying k . Our CAB method outperformed both MGR and SAB in all tested cases. MGR performed poorly for small k values, as the edge multiplicities are larger and the extension cost is heavily determined by balancing, whereas SAB performed poorly for larger k values, as the edge multiplicities are smaller and the EE cost is heavily determined by connecting. Our results are very promising because MGR was also orders of magnitude slower than CAB (as expected).

Figure 3b shows the EE cost on the whole RHO dataset, for varying k . We show the result only for the SAB and CAB methods, since MGR could not terminate in reasonable time. Note that there is no difference between the two methods for $k = 10$, as in this case the input graph is connected and thus both SAB and CAB balance it in the same optimal way. However, for $k > 10$, CAB outperforms SAB consistently, and the difference generally increases with k . This shows that, unlike SAB, our method is able to connect the graph with a small cost, even when the graph has a large number of components.

Analogous results to those of Figure 3 for the STA dataset are in Figure 5 of Appendix A.

Runtime and Peak Memory Consumption. Figures 4a and 4b show that the runtime and peak memory consumption of CAB scale (even better than) *linearly* with the input size, which confirms our complexity analysis (see Theorems 5 and 6). The results for SAB are omitted to avoid cluttering the figures; SAB was several times slower but consumed slightly less memory, mainly due to the space-efficient SCS algorithm [2] it employs.

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 Jarno Alanko and Tuukka Norri. Greedy shortest common superstring approximation in compact space. In *24th SPIRE*, volume 10508 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2017. doi:10.1007/978-3-319-67428-5_1.
- 3 Bastien Cazaux and Eric Rivals. A linear time algorithm for shortest cyclic cover of strings. *J. Discrete Algorithms*, 37:56–67, 2016. doi:10.1016/j.jda.2016.05.001.
- 4 Bastien Cazaux and Eric Rivals. Superstrings with multiplicities. In *29th CPM*, volume 105 of *LIPICs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CPM.2018.21.
- 5 Maxime Crochemore, Marek Cygan, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Algorithms for three versions of the shortest common superstring problem. In *21st CPM*, pages 299–309, 2010. doi:10.1007/978-3-642-13509-5_27.
- 6 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 7 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *17th ICALP*, pages 6–19, 1990. doi:10.1007/BFb0032018.
- 8 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.
- 9 Frederic Dorn, Hannes Moser, Rolf Niedermeier, and Mathias Weller. Efficient algorithms for Eulerian extension and Rural Postman. *SIAM J. Discret. Math.*, 27(1):75–94, 2013. doi:10.1137/110834810.
- 10 Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1:1–1:23, 2014. doi:10.1145/2529989.
- 11 Sara El-Metwally, Taher Hamza, Magdi Zakaria, and Mohamed Helmy. Next-generation sequence assembly: Four stages of data processing and computational challenges. *PLoS Comput. Biol.*, 9(12), 2013. doi:10.1371/journal.pcbi.1003345.
- 12 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.
- 13 Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
- 14 John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.
- 15 Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computation*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 16 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.
- 17 Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. doi:10.1090/S0002-9939-1956-0078686-7.

- 18 Paul Medvedev. Modeling biological problems in computer science: a case study in genome assembly. *Briefings Bioinform.*, 20(4):1376–1383, 2019. doi:10.1093/bib/bby003.
- 19 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *7th WABI*, volume 4645 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 2007. doi:10.1007/978-3-540-74126-8_27.
- 20 Paul Medvedev and Mihai Pop. What do Eulerian and Hamiltonian cycles have to do with genome assembly? *PLoS Computational Biology*, 17(5):1–5, May 2021. doi:10.1371/journal.pcbi.1008928.
- 21 Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010. doi:10.1016/j.ygeno.2010.03.001.
- 22 Niranjana Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14:157–167, 2013.
- 23 Clifford S. Orloff. A fundamental problem in vehicle routing. *Networks*, 4(1):35–64, 1974. doi:10.1002/net.3230040105.
- 24 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.
- 25 Michael C. Schatz, Arthur L. Delcher, and Steven L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Res.*, 20(9):1165–1173, 2010. doi:10.1101/gr.101360.109.
- 26 Jared T. Simpson and Mihai Pop. The theory and practice of genome sequence assembly. *Annu Rev Genomics Hum Genet*, 16:153–172, 2015.
- 27 Jang-il Sohn and Jin-Wu Nam. The present and future of de novo whole-genome assembly. *Briefings Bioinform.*, 19(1):23–40, 2018. doi:10.1093/bib/bbw096.
- 28 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990. doi:10.1007/BF01840391.
- 29 Bilal Wajid and Erchin Serpedin. Review of general algorithmic features for genome assemblers for next generation sequencers. *Genomics, Proteomics & Bioinformatics*, 10(2):58–73, 2012. doi:doi.org/10.1016/j.gpb.2012.05.006.

A Omitted Details from Section 6

Algorithm 2 SAB.

-
- 1: Compute the connected components of $G(V, E)$
 - 2: $s \leftarrow \text{SCS}(V)$ ▷ A (shortest) common superstring of V using the algorithm of [2]
 - 3: Let $Q_1 = u_1, \dots, u_{|V|}$ be the sequence of all nodes in V as they occur in s
 - 4: Let $Q_2 = (u_1, u_2), \dots, (u_{|V|-1}, u_{|V|})$ be the sequence of edges as they occur in Q_1
 - 5: Sort Q_2 in decreasing order w.r.t. the length of the longest suffix/prefix overlap of (u_i, u_j)
 - 6: $i \leftarrow 0$
 - 7: **while** $G'(V, E)$ is not weakly connected **do** ▷ Connects the graph
 - 8: $(u, v) \leftarrow Q_2[i]$ ▷ Gets the i th longest suffix/prefix overlap
 - 9: **if** the components where u and v lie are not currently connected **then**
 - 10: Let q be the shortest string with u as prefix and v as suffix
 - 11: Extend E with all edges $(q[p..p+k-2], q[p+1..p+k-1])$ occurring in q
 - 12: Extend V with all new nodes $q[p..p+k-1] \notin V$ occurring in q
 - 13: $i \leftarrow i + 1$
 - 14: Algorithm of Theorem 6 on graph $G'(V, E)$ to find multiset \mathcal{A} ▷ Balances the graph
-

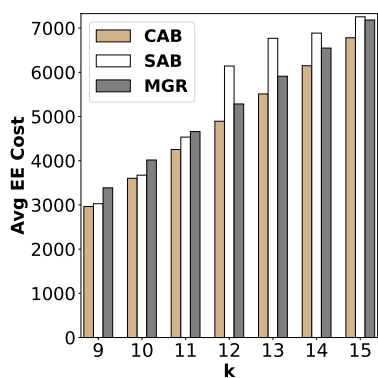
■ **Table 1** Datasets characteristics.

(a) *Rhodobacter sphaeroides* (RHO).

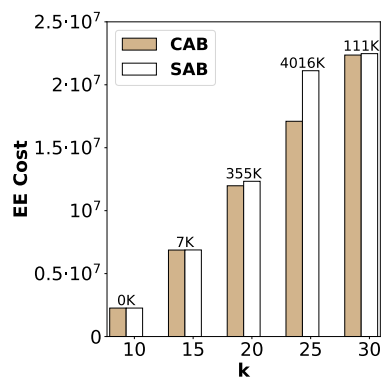
k	# nodes	# edges	# distinct edges	# components
10	1,013,904	185,506,278	3,338,995	1
15	37,858,157	175,579,617	46,337,190	528
20	61,265,275	165,433,984	62,546,892	44,386
25	64,861,977	155,232,772	65,087,335	131,266
30	65,383,451	145,014,018	65,356,249	199,627

(b) *Staphylococcus aureus* (STA).

k	# nodes	# edges	# distinct edges	# components
10	1,047,172	117,107,289	3,974,601	1
15	40,262,854	110,650,401	42,924,890	1,637
20	45,318,307	104,188,673	45,512,480	152,945
25	45,833,210	97,727,029	45,825,958	211,943
30	45,498,694	91,266,009	45,354,736	259,333



(a) STA samples.



(b) STA.

■ **Figure 5** (a) Average Eulerian Extension (EE) cost vs. k over five random samples of STA. (b) EE cost vs. k on the whole STA dataset. The difference between the EE costs of SAB and CAB is shown on the top of each pair of bars (K stands for thousands).

Back-To-Front Online Lyndon Forest Construction

Golnaz Badkobeh  

Goldsmiths University of London, UK

Maxime Crochemore  

Univ. Gustave Eiffel, Champs-sur-Marne, France

King's College London, UK

Jonas Ellert  

Department of Computer Science, Technical University of Dortmund, Germany

Cyril Nicaud  

LIGM, Univ. Gustave Eiffel, Champs-sur-Marne, France

Abstract

A Lyndon word is a word that is lexicographically smaller than all of its non-trivial rotations (e.g. **ananas** is a Lyndon word; **banana** is not a Lyndon word due to its smaller rotation **abanan**). The Lyndon forest (or equivalently Lyndon table) identifies maximal Lyndon factors of a word, and is of great combinatoric interest, e.g. when finding maximal repetitions in words. While optimal linear time algorithms for computing the Lyndon forest are known, none of them work in an online manner. We present algorithms that compute the Lyndon forest of a word in a reverse online manner, processing the input word from back to front. We assume a general ordered alphabet, i.e. the only elementary operations on symbols are comparisons of the form less-equal-greater. We start with a naive algorithm and show that, despite its quadratic worst-case behaviour, it already takes expected linear time on words drawn uniformly at random. We then introduce a much more sophisticated algorithm that takes linear time in the worst case. It borrows some ideas from the offline algorithm by Bille et al. (ICALP 2020), combined with new techniques that are necessary for the reverse online setting. While the back-to-front approach for this computation is rather natural (see Franek and Liut, PSC 2019), the steps required to achieve linear time are surprisingly intricate. We envision that our algorithm will be useful for the online computation of maximal repetitions in words.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Lyndon factorisation, Lyndon forest, Lyndon table, Lyndon array, right Lyndon tree, Cartesian tree, standard factorisation, online algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.13

Supplementary Material *Source Code*: <https://github.com/jonas-ellert/right-lyndon>
archived at `swh:1:dir:e0cf158eeec99338193603aded28b5ebc252e87b`

Funding *Jonas Ellert*: Partially supported by the French embassy in Germany (Procope mobility grant, project 0185-DEU-21-016 code 174).

1 Lyndon words

A Lyndon word is a word that is lexicographically smaller than all of its non-trivial rotations (e.g. **ananas** is a Lyndon word; **banana** is not a Lyndon word due to its smaller rotation **abanan**). The Lyndon table or equivalently the right Lyndon forest of a word (generalised from the right Lyndon tree of a Lyndon word) identifies the longest Lyndon prefix of each suffix of the word (a precise definition follows later). The article explores the complexity of algorithms for building the Lyndon table or forest of a word over a general ordered alphabet. The only elementary operations on letters of the alphabet are comparisons of the form



© Golnaz Badkobeh, Maxime Crochemore, Jonas Ellert, and Cyril Nicaud;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 13; pp. 13:1–13:23

Leibniz International Proceedings in Informatics



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

less-equal-greater. The presented algorithms process the input word $y[0..n-1]$ in a reverse online manner. When accessing position $y[i]$ for the first time, they have already computed the Lyndon table of $y[i+1..n-1]$.

Background and applications. Introduced in the field of combinatorics on words by Lyndon (see [26, 24]) and used in algebra, Lyndon words introduce structural elements in plain sequences of symbols, provided there is an ordering on the set of symbols. They have shown their usefulness for designing efficient algorithms on words. For example, they underpinned the notion of critical positions in words [24], the two-way string matching [14] and rotations of periodic words [8].

The right Lyndon tree of a Lyndon word y (based on the factorisation $y = uv$, where v is the lexicographically smallest proper suffix, or equivalently the longest proper Lyndon suffix of y) is by definition related to the sorted list of suffixes of y . Hohlweg and Reutenauer [21] showed that the right Lyndon tree is the Cartesian tree (see [30]) built from ranks of suffixes in their lexicographically sorted list (see also [15]). The list corresponds to the standard permutation of suffixes of the word and is the main component of its suffix array (see [27]), one of the major data structures for text indexing. The relation between suffix arrays and properties of Lyndon words is used by Mantaci et al. [28], by Baier [2] and by Bertram et al. [6] to compute the suffix array, as well as by Louza et al. [25] to induce the Lyndon table.

If the suffix array is given, the Lyndon table can be constructed in linear time (e.g. [19, Algorithm NSVISA]). In fact, the method is similar to the standard algorithms that build a Cartesian tree from the ranks of suffixes in their lexicographical order. However, computing the suffix array on general ordered alphabets requires $\Omega(n \lg n)$ time due to the well-known information-theoretic lower bound on comparison sorting. A linear-time algorithm that directly computes the Lyndon table (on general ordered alphabets, without requiring the suffix array) was designed by Bille et al. [7]. While it processes the word from left to right, it is not an online algorithm because it may need to look ahead arbitrarily far in the word in order to determine an entry of the Lyndon table.

The Lyndon forest is closely related to runs (maximal periodicities) in words, and is not only essential for showing theoretical properties of runs, but also for their efficient computation. Bannai et al. [3] used the Lyndon table to solve the conjecture of Kolpakov and Kucherov [22] stating that there are less than n runs in a length- n word, following a result in [12]. The key result in [3] is that every run in a word y contains as a factor a Lyndon root, according either to the alphabet ordering or its inverse, that corresponds to a node of the associated Lyndon forest. Since the Lyndon forest has a linear number of nodes according to the length of y , browsing all its nodes leads to a linear-time algorithm to report all the runs occurring in y . However, the time complexity of this technique also depends on the time required to build the forest and to extend a potential run root to an actual run. This is feasible on a linearly-sortable alphabet using an efficient longest common extension (LCE) technique (e.g. [18]). Kosolobov [23] conjectured a linear-time algorithm computing all runs for a word over a general ordered alphabet. An almost linear-time solution was given in [11], but the final positive answer is by Ellert and Fischer [17] who combined the Lyndon table algorithm by Bille et al. [7] with a new linear-time computation technique for the LCEs.

Our contributions. We present algorithms that compute the Lyndon table or Lyndon forest of a word over a general ordered alphabet. They scan the input word $y[0..n-1]$ in a reverse online manner, i.e. from the end to the beginning. When accessing $y[i]$ for the first time, they

have already computed the Lyndon forest or table of the word $y[i + 1..n - 1]$. Processing the word in a reverse manner is rather natural (see Franek and Liut [20]), but it requires intricate algorithmic techniques to obtain an efficient algorithm.

In Section 2, we present a naive algorithm for computing the Lyndon table that takes quadratic time in the worst case. We also provide a simple linear time algorithm that computes the Lyndon forest from the Lyndon table. As shown in Section 3, the naive algorithm runs in expected linear time if we fix an alphabet and a length, and then choose a random word of the chosen length over the chosen alphabet. Finally, in Section 4, we introduce a more sophisticated algorithm for constructing the Lyndon table. It takes optimal linear time in the worst case, and uses the following techniques. First, the use of both next and previous smaller suffix tables, and skipping symbol comparisons when computing longest common extensions (LCEs) associated with these tables (similarly to what has been done in [7]). Second, the additional acceleration of the LCE computation by exploiting and reusing previously computed values. Third, additional steps to ensure that we reuse each computed value at most once, which ultimately results in the linear running time.

We envision that our algorithm will be useful as a tool for the online computation of runs. For example, it could lead to an online version of the runs algorithm presented in [17].

► **Remark.** The design of the right Lyndon tree construction contrasts with the dual question of left Lyndon tree construction (see [1] and references therein). The latter is done by a far simpler algorithm than the algorithm in Section 4. But its use to build a right Lyndon tree, as done in the Lyndon bracketing by Sawada and Ryskey [29] and in [19] by Franek et al., does not seem to lead to a linear right Lyndon tree construction.

Basic definitions

Let A be an alphabet with an ordering $<$ and A^+ be the set of non-empty words with the lexicographical ordering induced by $<$. The length of a word y is denoted by $|y|$. The empty word of length 0 is denoted by ϵ . The concatenation of two words u and v is denoted by uv . The e times concatenation of a word u is written as u^e (e.g. $u^3 = uuu$). If for non-empty words u, v, y it holds $y = uv$, then we say that uv (formally (u, v)) is a non-trivial factorisation of y ; the word vu is a non-trivial rotation (or conjugate) of y ; the word u is a proper non-empty prefix of y ; and word v is a proper non-empty suffix of y . A word y is primitive if it has $|y| - 1$ distinct non-trivial rotations, or equivalently if it cannot be written as $y = u^e$ for some word u and integer $e \geq 2$. If there are non-empty u and v such that $y = uv = vu$, then y is non-primitive. A word u is strongly less than a word v , denoted by $u \ll v$, if there are words r, s and t , and letters a and b satisfying $u = ras$, $v = rbt$ and $a < b$. The word u is smaller than a word v , denoted by $u < v$, if either $u \ll v$ or u is a proper prefix of v , i.e. $v = ur$ for some non-empty word r . A Lyndon word is a non-empty word defined as follows:

► **Proposition 1** ([16, Proposition 1.2]). *Both of the following equivalent conditions define a Lyndon word y : (i) $y < vu$, for every non-trivial factorisation uv of y , (ii) $y < v$, for every proper non-empty suffix v of y .*

Note that the conditions in the definition trivially hold if $|y| = 1$, i.e. a single symbol is always a Lyndon word. The main feature of Lyndon words stands in the theorem by Chen, Fox and Lyndon (see [24]), which states that every word in A^+ can be uniquely factorised into a sequence of lexicographically non-increasing Lyndon words.

13:4 Back-To-Front Online Lyndon Forest Construction

► **Theorem 2** (Lyndon factorisation). *Any non-empty word y may be written uniquely as a lexicographically weakly decreasing product of Lyndon words, i.e. $y = x_1x_2 \cdots x_m$, where each x_k is a Lyndon word and $x_1 \geq x_2 \geq \cdots \geq x_m$.*

A fundamental property of the Lyndon factorisation is the fact that the suffix x_m is the lexicographically smallest suffix, or equivalently the longest proper Lyndon suffix, of y .

2 Lyndon tree and Lyndon table construction

The structure of the Lyndon tree of a Lyndon word derives from the next property (see [24]).

► **Property 3.** *Let y be a Lyndon word and $y = uv$, where v is the smallest or equivalently the longest proper Lyndon suffix of y . Then u is a Lyndon word.*

The (right) standard factorisation of a Lyndon word y of length $n > 1$ is the pair (u, v) of Lyndon words, simply denoted by $u \cdot v$, where u and v are as in Property 3.

The (right) Lyndon tree $\mathcal{R}(y)$ of a Lyndon word y represents recursively its complete (right) standard factorisation. It is a binary tree with $2|y| - 1$ nodes: its leaves are positions on the word and internal nodes correspond to concatenations of two consecutive Lyndon factors of the word, which as such can be viewed as inter-positions. More precisely, $\mathcal{R}(y) = \langle 0 \rangle$ if $|y| = 1$, and otherwise $\mathcal{R}(y) = \langle \mathcal{R}(u), \mathcal{R}(v) \rangle$, where $u \cdot v$ is the standard factorisation of y . The next algorithm gives a straightforward construction of the right Lyndon tree.

```
LYNDONTREE( $y$  Lyndon word of length  $n$ )
1   $\mathcal{F} \leftarrow$  stack containing only the empty word  $\epsilon$ 
2  for  $i \leftarrow n - 1$  downto 0 do
3       $(u, \mathcal{R}(u), v) \leftarrow (y[i], \langle i \rangle, \mathcal{F}.peek())$ 
4      while  $u < v$  do
5           $\mathcal{R}(uv) \leftarrow \langle \mathcal{R}(u), \mathcal{R}(v) \rangle$ 
6           $(u, v) \leftarrow (uv, \mathcal{F}.pop\text{-and-then-peek}())$ 
7       $\mathcal{F}.push(u)$ 
8  return  $\mathcal{R}(y)$ 
```

Algorithm LYNDONTREE scans the word from right to left. Just after executing an iteration of the for loop, the suffix $y[i..n-1]$ of y is decomposed into its Lyndon factorisation $z_1 \cdot z_2 \cdots z_m$. In this moment, the stack contains exactly the elements z_1, z_2, \dots, z_m (z_1 being the topmost element), and variable u stands for the first factor z_1 of the decomposition.

With an appropriate implementation of the tree, the algorithm runs in linear time if the test $u < v$ at line 4 is done in constant time. However, in the worst case, the algorithm runs in quadratic time; this is when the test is performed by mere letter comparisons. For example, if $y = \mathbf{a}^k \mathbf{c} \mathbf{a}^{k+1} \mathbf{b}$ then each factor $\mathbf{a}^\ell \mathbf{c}$ is compared with the prefix $\mathbf{a}^{\ell+1}$ of $\mathbf{a}^{k+1} \mathbf{b}$.

Lyndon forest and Lyndon table

If $x_1x_2 \cdots x_m$ is the Lyndon factorisation of the non-empty word y , then its (right) Lyndon forest is defined by the list $\mathcal{R}(x_1), \mathcal{R}(x_2), \dots, \mathcal{R}(x_m)$, i.e. the list of Lyndon trees of the Lyndon factors (the Lyndon forest is a single tree if and only if y is a Lyndon word). One could compute the Lyndon forest by simply first computing the Lyndon factorisation, and

then building the tree for each factor. A more elegant method computes the forest from the Lyndon table (sometimes called Lyndon array) of the word. It is denoted by Lyn (l in [3], \mathcal{L} in [20] and λ in [19, 7]) and is defined, for each position i on y , by

$$Lyn[i] = \max\{|w| \mid w \text{ is a Lyndon prefix of } y[i..n-1]\}.$$

An example is provided in Figure 1 (we will explain the labelling of forest nodes and the table *root* in a moment; for now, we focus on Lyn). The Lyndon factorisation of y deduces easily from Lyn . Indeed, if i is the starting position of a factor of the decomposition, the next factor starts at position $i + Lyn[i]$, which is the first position of the next smaller suffix of $y[i..n-1]$ (see Lemma 8). For the example above, we get positions 0, ($1 = 0 + Lyn[0]$), ($4 = 1 + Lyn[1]$) and ($9 = 4 + Lyn[4]$), corresponding to the Lyndon factorisation $b \cdot abb \cdot ababb \cdot aabb$ of y .

Algorithm LYNDONTABLE shown below computes Lyn using the same scheme as Algorithm LYNDONTREE. It scans the input word from right to left and implicitly concatenates two adjacent Lyndon factors u and v to form a Lyndon factor uv when $u < v$.

```

LYNDONTABLE( $y$  non-empty word of length  $n$ )
1  for  $i \leftarrow n - 1$  downto 0 do
2      ( $Lyn[i], j$ )  $\leftarrow$  ( $1, i + 1$ )
3      while  $j < n$  and  $y[i..j-1] < y[j..j + Lyn[j] - 1]$  do
4          ( $Lyn[i], j$ )  $\leftarrow$  ( $Lyn[i] + Lyn[j], j + Lyn[j]$ )
5  return  $Lyn$ 

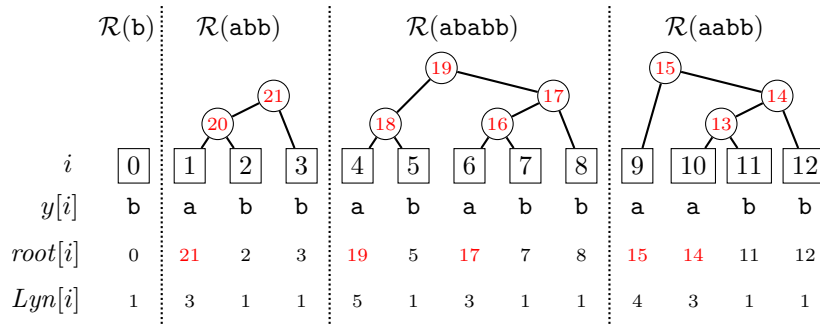
```

More details (including a proof of correctness) regarding this algorithm can be found in [13, Problem 87]). The worst-case running time is $O(|y|^2)$ in the letter-comparison model. Note, however, that the comparison of Lyndon words $y[i..j-1] < y[j..j + Lyn[j] - 1]$ in line 3 can be replaced by a suffix comparison $y[i..n-1] < y[j..n-1]$ (see [3, 15] and also [13, Problem 87]). Then, if the suffixes of y are previously sorted and their ranks are stored in a table, Algorithm LYNDONTABLE runs in linear time. The precomputation takes linear time if y is drawn from a linearly-sortable alphabet (see suffix arrays in [10, Chapter 4]).

Obtaining the Lyndon forest from the table. A possible data structure that implements the Lyndon forest uses tables *root*, *left* and *right*. Implicitly, the leaves are positions $0, \dots, n-1$ on y . For each position i , $root[i]$ is the root of the largest subtree whose leftmost leaf is i . Thus, if x_k is a factor of the Lyndon factorisation of y that starts at position i , then $root[i]$ is the root of the Lyndon tree of x_k . Internal nodes are integers larger than $n-1$. The left and right child of an internal node m are respectively $left[m]$ and $right[m]$. An example of this representation is provided in Figure 1.

As demonstrated by the example, $Lyn[i]$ is exactly the size of the subtree that is rooted in $root[i]$. This makes it easy to translate the Lyndon forest to the Lyndon table and vice versa. Algorithm LYNDONFOREST below shows the conversion from Lyndon table to forest. Note that the time required by the algorithm, apart from the time needed to compute Lyn , is linear in n . This is due to the fact that each iteration of the inner loop creates a new internal node, and there are at most $n-1$ such nodes.

13:6 Back-To-Front Online Lyndon Forest Construction



■ **Figure 1** Lyndon forest and Lyndon table of the word $y = \text{babbababbaabb}$. It holds $Lyn[4] = 5$ because ababb is the longest Lyndon factor starting at position 4.

LYNDONFOREST(y non-empty word of length n)

```

1   $Lyn \leftarrow \text{LYNDONTABLE}(y)$ 
2   $m \leftarrow n$  ▷ next available integer for a new internal node
3  for  $i \leftarrow n - 1$  downto 0 do
4       $(root[i], j) \leftarrow (i, i + 1)$ 
5      while  $j < i + Lyn[i]$  do
6           $(left[m], right[m]) \leftarrow (root[i], root[j])$ 
7           $(root[i], m) \leftarrow (m, m + 1)$ 
8           $j \leftarrow j + Lyn[j]$ 
9  return  $(root, left, right)$ 

```

3 Average running time for building a Lyndon table

In this section, we prove that the average running time of Algorithm LYNDONTABLE is linear, for the uniform distribution on size- n words, on any alphabet. The result also applies to Algorithm NAIVELYND of the next section and implies that the construction of a Lyndon forest can also be done in average linear time. We assume the alphabet $A = \{a_0, a_1, \dots, a_{\sigma-1}\}$ of size $\sigma \geq 2$, equipped with the order $a_0 < a_1 < \dots < a_{\sigma-1}$. (Note that LYNDONTABLE takes worst-case linear time for $\sigma = 1$, because then it holds $Lyn[i] = 1$ for all i , which means that the comparison in line 3 takes constant time).

For any positive n , let \mathbb{P}_n be the uniform probability on A^n , where every word u has probability $\mathbb{P}_n(u) = \sigma^{-n}$. Formally, we are considering a sequence of uniform distributions, one for each size n ; as we seek to obtain a result for every σ , we therefore have to consider that $\sigma = \sigma_n$ also depends on n (even if it can be the constant sequence and want a result for a fixed alphabet). In the following, we just require that $\sigma_n \geq 2$ for every n and write σ instead of σ_n for readability. Observe that allowing an alphabet of unbounded size is a completely different framework than in the series of articles [5, 9] dealing with the expected properties of uniform random Lyndon words, where the alphabet has a fixed size. In particular, for large σ_n , a uniform random word resembles a uniform random permutation, whose typical statistics are greatly different from the ones of a uniform random word on, say, four letters.

Let i, j be two integers with $0 \leq i < j < n$. The random variable C_{ij} is defined as the number of comparisons performed between $y[i..j-1]$ and $y[j..j+Lyn[j]-1]$ at line 3 of Algorithm LYNDONTABLE, for a random word y : if the algorithm does not compare these

two factors, then $C_{ij} = 0$, otherwise it is the number of letter comparisons performed by the naive algorithm that scans both words from left to right until it can decide. Since this is the only step of Algorithm LYNDONTABLE where letter comparisons are performed, the total number of such comparisons performed by the algorithm is the random variable $C = \sum_j C_j$, where $C_j = \sum_{i < j} C_{ij}$. In this section, we are interested in estimating the expectation $\mathbb{E}_n[C]$ of C for uniform random words of length n .

By linearity of the expectation, we have $\mathbb{E}_n[C] = \sum_j \mathbb{E}_n[C_j]$. Our proof consists in bounding from above the contribution of each $\mathbb{E}_n[C_j]$, using $\mathbb{E}_n[C_j] = \sum_{i < j} \mathbb{E}_n[C_{ij}]$. Let us first consider the cases where the position j is near the end of the word, as it has to be considered separately in our analysis and since it illustrates well the kind of techniques used for the main part of the proof.

► **Lemma 4.** *If $j \geq n - 3 \log_2 n$ then $\mathbb{E}_n[C_j] = \mathcal{O}(\log^2 n)$.*

Proof. We make the following observations: (i) the factors starting at indices i and j can only be compared once by the algorithm, (ii) the factors compared at line 3 are always Lyndon words, and (iii) the number of letter comparisons performed for given i and j , if any, is at most $3 \log_2 n$ since $j \geq n - 3 \log_2 n$. So for every word y , the number of comparisons $C_{ij}(y)$ is bounded from above by $D_{ij}(y)$, where $D_{ij}(y)$ is 0 if $y[i..j-1]$ is not Lyndon and $3 \log_2 n$ otherwise. Therefore $\mathbb{E}_n[C_{ij}] \leq \mathbb{E}_n[D_{ij}]$.

Let $\ell = j - i$ be the length of the factor $y[i..j-1]$. Since a non-primitive word cannot be a Lyndon word, if \mathcal{L} and \mathcal{P} respectively denote the set of Lyndon words and of primitive words, we have

$$\mathbb{P}_n(y[i..j-1] \in \mathcal{L}) = \mathbb{P}_\ell(\mathcal{L}) = \mathbb{P}_\ell(\mathcal{L} \cap \mathcal{P}) = \mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P}) \mathbb{P}_\ell(\mathcal{P}) \leq \mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P}).$$

But $\mathbb{P}_\ell(\mathcal{L} \mid \mathcal{P}) = \frac{1}{\ell}$ as all rotations of a primitive word are equally likely, and only one of them is a Lyndon word. Hence $y[i..j-1] \in \mathcal{L}$ with probability at most $\frac{1}{\ell}$. This yields that $\mathbb{E}_n[C_{ij}] \leq \mathbb{E}_n[D_{ij}] \leq \frac{3 \log_2 n}{\ell}$, and if we sum the contributions of all possible i , we have the announced result as $\mathbb{E}_n[C_j] \leq \sum_{i=0}^{j-1} \frac{3 \log_2 n}{j-i} \leq 3 \log_2 n \cdot (\log j + 1) = \mathcal{O}(\log^2 n)$. ◀

So when we sum the contributions of the $\mathbb{E}_n[C_j]$ for $j \geq n - 3 \log_2 n$, the expected total number of comparisons is $\mathcal{O}(\log^3 n)$, hence sublinear. Thus we can focus on estimating $\mathbb{E}_n[C_j]$ for j sufficiently far away from the end of the word. We need the following lemma. The proof is given in Appendix A.1.

► **Lemma 5.** *Let Λ be an ordered alphabet with $|\Lambda| \geq 2$ letters, $\# \notin \Lambda$ be a new letter that is greater than every letter of Λ , and y be a word selected from Λ^t uniformly at random. Then there exists a constant $c \geq 1$ such that the probability that $y\#$ is a Lyndon word is at most $\frac{c}{t}$.*

Our main technical result is stated in the following proposition.

► **Proposition 6.** *There exists a constant γ such that $\mathbb{E}_n[C_j] \leq \gamma$, for all $j < n - 3 \log_2 n$.*

Proof. Recall that if $E \subseteq A^n$, the indicator function of E is the random variable $\mathbb{1}_E$ that values 1 for elements of E and 0 otherwise. The first step of the proof is to look at the factor of a random word whose positions range from j to $j + \lceil 3 \log_2 n \rceil - 1$. Let $\mathcal{A} \subseteq A^n$ denote the set of words y such that $y[j..j + \lceil 3 \log_2 n \rceil - 1] = a_0^{\lceil 3 \log_2 n \rceil}$, i.e. it consists of the repetition of the smallest letter. Let $\overline{\mathcal{A}}$ denote the complement of \mathcal{A} in A^n . We have $C_j = \mathbb{1}_{\mathcal{A}} \cdot C_j + \mathbb{1}_{\overline{\mathcal{A}}} \cdot C_j$, hence $\mathbb{E}_n[C_j] = \mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] + \mathbb{E}_n[\mathbb{1}_{\overline{\mathcal{A}}} \cdot C_j]$.

Since for all word $y \in A^n$ we have $C_j(y) \leq C(y) \leq n^2$, it holds that $\mathbb{1}_{\mathcal{A}}(y) \cdot C_j(y) \leq \mathbb{1}_{\mathcal{A}}(y) n^2$ and therefore $\mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] \leq n^2 \mathbb{E}_n[\mathbb{1}_{\mathcal{A}}] = n^2 \mathbb{P}_n(\mathcal{A}) = \frac{n^2}{\sigma^{\lceil 3 \log_2 n \rceil}}$. As $\sigma \geq 2$, this yields that $\mathbb{E}_n[\mathbb{1}_{\mathcal{A}} \cdot C_j] \leq \frac{2}{n}$.

13:8 Back-To-Front Online Lyndon Forest Construction

We now focus on $\overline{\mathcal{A}}$. For any positive integer $k \leq \lfloor 3 \log_2 n \rfloor$ and any letter $a_s \neq a_0$, let $\mathcal{B}_{k,s}^{(n)}$ denote the set of words y in A^n for which $a_0^{k-1}a_s$ is a prefix of $y[j..n-1]$. Any word of $\overline{\mathcal{A}}$ has a factor of the form $a_0^{k-1}a_s$ starting at position j , so we have the following partition:

$$\overline{\mathcal{A}} = \bigsqcup_{k=1}^{\lfloor 3 \log_2 n \rfloor} \bigsqcup_{s=1}^{\sigma-1} \mathcal{B}_{k,s}^{(n)},$$

and therefore $\mathbb{E}_n[\mathbb{1}_{\overline{\mathcal{A}}}C_j] = \sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}}C_j]$.

Fix k and s . We want to bound from above the contribution of $\mathcal{B}_{k,s}^{(n)}$ to $\mathbb{E}_n[C_j]$ by summing the $\mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}}C_{ij}]$ for $i < j$.

For a given index $i < j - k$ (the cases $j - k \leq i < j$ will be studied separately), when the algorithm works on an input $y \in \mathcal{B}_{k,s}^{(n)}$, if it compares $y[i..j-1]$ and $y[j..j+Lyn[j]-1]$ then necessarily:

- (i) The factor $y[i..j-1]$ is a Lyndon word.
- (ii) There is no occurrence of $a_0^{k-1}a_t$ with $t < s$ in $y[i+1..j-1]$, otherwise the Lyndon factor starting at position j would have already merged, before reaching index i .

In our proof, we ask for weaker conditions, which is not an issue as we are looking for an upper bound for $\mathbb{E}_n[C_j]$. For this, we split the factor $y[i..j-1]$ of length $\ell = j - i$ into $\lambda = \lfloor \ell/k \rfloor$ blocks $y_0, \dots, y_{\lambda-1}$ of k letters, and one remaining block z of length $\ell \bmod k$ if ℓ is not a multiple of k :

$$\forall m \in \{0, \dots, \lambda-1\} : y_m = y[i+mk..i+m(k+1)-1], \text{ so that } y[i..j-1] = y_0 \cdots y_{\lambda-1} \cdot z.$$

The number λ of blocks is at least 1, as we only consider the indices i smaller than $j - k$ for now. Observe that, as y is a uniform random word, the y_m 's are uniform and independent random words of length k . Condition (ii) implies that none of the y_m is smaller than $a_0^{k-1}a_s$ for $m \geq 1$. We distinguish two cases, depending on whether y_0 is smaller than $a_0^{k-1}a_s$ or not, and define, for $i < j - k$, the following sets

$$\begin{aligned} \mathcal{B}_{k,s,i}^{<} &= \{y \in \mathcal{B}_{k,s}^{(n)} \mid y_0 < a_0^{k-1}a_s \text{ and } \forall m \in \{1, \dots, \lambda-1\}, y_m \geq a_0^{k-1}a_s\}, \\ \mathcal{B}_{k,s,i}^{\geq} &= \{y \in \mathcal{B}_{k,s}^{(n)} \mid \forall m \in \{0, \dots, \lambda-1\}, y_m \geq a_0^{k-1}a_s \text{ and } y[i..j-1] \in \mathcal{L}\}. \end{aligned}$$

As a consequence of Condition (i) and Condition (ii), if $y \in \mathcal{B}_{k,s}^{(n)}$ and $C_{ij}(y) \neq 0$ (the algorithm compares the factors at positions i and j), then necessarily $y \in \mathcal{B}_{k,s,i}^{<}$ or $y \in \mathcal{B}_{k,s,i}^{\geq}$. Therefore $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}_{k,s}^{(n)}) \leq \mathbb{P}_n(\mathcal{B}_{k,s,i}^{<}) + \mathbb{P}_n(\mathcal{B}_{k,s,i}^{\geq})$.

- The probability of $\mathcal{B}_{k,s,i}^{<}$ is $\mathbb{P}_n(\mathcal{B}_{k,s,i}^{<}) = \frac{1}{\sigma^k} \frac{s}{\sigma^k} \left(\frac{\sigma^k - s}{\sigma^k} \right)^{\lambda-1}$, as the probability to be in $\mathcal{B}_{k,s}^{(n)}$ is σ^{-k} and as there are s words of length k smaller than $a_0^{k-1}a_s$.
- To compute the probability of $\mathcal{B}_{k,s,i}^{\geq}$ observe the y_m 's are uniform independent elements of $\Lambda_{k,s} = \{w \in A^k : w \geq a_0^{k-1}a_s\}$ and z is an independent and uniform word of length $\ell \bmod k$ on A . Moreover, if $y[i..i+j-1]$ is a Lyndon word, then $y_m y_{m+1} \cdots y_{\lambda-1} z$ is greater than $y[i..i+j-1]$ for every $1 \leq m < \lambda$. Now consider $y_0 \cdots y_{\lambda-1}$ as a size- λ word on the alphabet $\Lambda_{k,s}$; the latter property implies that $y_0 \cdots y_{\lambda-1} \#$ is smaller than $y_m y_{m+1} \cdots y_{\lambda-1} \#$, where $\#$ is a new letter greater than all the letters of $\Lambda_{k,s}$. This weakens the condition that $y[i..j-1]$ is a Lyndon word, but still provides a useful upper bound: by Lemma 5 a proportion of at most $\frac{\epsilon}{\lambda}$ of the possibilities satisfy the property

(in fact we cannot apply Lemma 5 if $k = 1$ and $s = \sigma - 1$, but the inequality trivially holds as Condition (i) forces $i = j - 1$ and $c \geq 1$). Putting all together, the probability of

$$\mathcal{B}_{k,s,i}^{\geq} \text{ is bounded from above by } \frac{1}{\sigma^k} \left(\frac{\sigma^k - s}{\sigma^k} \right)^\lambda \frac{c}{\lambda}.$$

So we established that, for $i < j - k$, $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}_{k,s}^{(n)}) \leq q(k, s, \lambda)$ where

$$q(k, s, \lambda) = \frac{s}{\sigma^{2k}} \left(\frac{\sigma^k - s}{\sigma^k} \right)^{\lambda-1} + \frac{c}{\lambda \sigma^k} \left(\frac{\sigma^k - s}{\sigma^k} \right)^\lambda. \quad (1)$$

Observe that we only used conditions on the letters of indices smaller than $j + k$ to estimate the probability $\mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}_{k,s}^{(n)})$. Hence, conditioned by $C_{ij} \neq 0$ and $\mathcal{B}_{k,s}^{(n)}$, the suffix $y[j + k .. n - 1]$ of y is a uniform random word of A^{n-j-k} . For any $z \in A^{j+k}$, consider a random word $y \in A^n$ that admits z as a prefix, which is the same as saying that $y = zz'$ where z' is a uniform random word of length $n - j - k$. The number of comparisons C_{ij} performed by the algorithm between $y[i .. j - 1]$ and $y[j .. j + L_{yn}[j] - 1]$ can be bounded from above by $k + 1 + D_{i+k, j+k}(y)$, where $D_{i+k, j+k}$ is the length of the longest common prefix between $y[i + k .. n - 1]$ and $y[j + k .. n - 1]$: we get the upper bound by considering that the k first comparisons are successful and by discarding the conditions on the lengths of the factors. If we fix z , the law of $1 + D_{i+k, j+k}$ is a truncated geometric law of parameter $1 - \frac{1}{\sigma}$ (we count the number of Bernoulli trials of parameter $1 - \frac{1}{\sigma}$ until we get a success, i.e. a mismatch, or reach the end of the word). This can be in turn bounded from above by a geometric law of parameter $1 - \frac{1}{\sigma}$. Hence $1 + D_{i+k, j+k} \leq \text{Geom}(1 - \frac{1}{\sigma})$, so $\mathbb{E}[1 + D_{i+k, j+k}] \leq \frac{\sigma}{\sigma-1} \leq 2$. Let pref_m be the random variable that associates to a word its prefix of length m . For $i < j - k$, this yields

$$\begin{aligned} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} \cdot C_{ij}] &\leq \sum_{z \in \mathcal{B}_{k,s}^{(j+k)}} \mathbb{E}_n[\mathbb{1}_{\text{pref}_{j+k}=z} \cdot C_{ij}] \leq \sum_{\substack{z \in \mathcal{B}_{k,s}^{(j+k)} \\ C_{ij}(z) \neq 0}} \frac{k+2}{\sigma^{j+k}} \\ &= (k+2) \cdot \mathbb{P}_n(C_{ij} \neq 0 \text{ and } \mathcal{B}_{k,s}^{(n)}) \leq (k+2) \cdot q(k, s, \lambda). \end{aligned}$$

Now we have to sum the contributions for all $i < j - k$, all $1 \leq s < \sigma - 1$ and all $k \leq \lfloor 3 \log_2 n \rfloor$. After tedious but elementary computations of sums (the details are given in Appendix A.2), we obtain that there exists some positive constant γ_1 such that

$$\sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} \sum_{i=0}^{j-k-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} \cdot C_{ij}] \leq \gamma_1. \quad (2)$$

We still have to estimate the contribution of the indices i such that $j - k \leq i < j$. For this, we just use Condition (i): $y[i .. j - 1]$ must be a Lyndon word for C_{ij} to be positive, which happens with probability at most $\frac{1}{\ell}$. The comparisons performed by the algorithm are evaluated as previously, by bounding them by an independent geometric law of parameter $1 - \frac{1}{\sigma}$ plus k , yielding, as the probability that $y[j .. j + k - 1] = a_0^{k-1} a_s$ is σ^{-k} :

$$\mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} \cdot C_{ij}] \leq \frac{k+2}{\ell \sigma^k}.$$

Using the same kind of elementary techniques as for Eq. (2), we get that there exists a constant γ_2 such that

$$\sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} \sum_{i=j-k}^{j-1} \mathbb{E}_n[\mathbb{1}_{\mathcal{B}_{k,s}^{(n)}} \cdot C_{ij}] \leq \gamma_2. \quad (3)$$

The details are given in Appendix A.3. As a consequence, for all $j \leq n - 3 \log_2 n$ we have $\mathbb{E}_n[C_j] \leq \gamma_1 + \gamma_2$, concluding the proof. \blacktriangleleft

Eventually we can state the main result of this section whose proof directly follows from Proposition 6 and Lemma 4, by summing for all j .

► **Theorem 7.** *For any $n \geq 1$ let A_n be an alphabet having $\sigma_n \geq 2$ letters. For the uniform distribution on words of length n over A_n , the expected number of comparisons performed by Algorithm LYNDONTABLE (and by its variant NAIVELYN below) is linear.*

4 Linear computation of the Lyndon table

To describe the algorithm that computes the Lyndon table Lyn in linear time, we proceed in four steps. First, we consider the next smaller suffix table, which contains the same information as the table Lyn , and its dual version, the previous smaller suffix table. They form an important element in the left-to-right solution introduced by Bille et al. [7]. Second, we adapt another component of the left-to-right solution, namely skipping some letter comparisons when lexicographically comparing suffixes. We achieve this by efficiently computing the longest common extension (LCE) of the relevant suffixes, which is the length of their longest common prefix (see e.g. [10, Chapter 4]). In the third step, we show how to compute the LCEs even faster by reusing previously computed values. The fourth step completes the algorithm with small adjustments that lead to an overall linear running time.

4.1 Next and previous smaller suffix tables

From now on, we prepend and append an infinitely small sentinel symbol $y[-1] = y[n] = \$$ to the input word $y[0..n-1]$. This simplifies the description of algorithms, e.g. by ensuring that for any two positions it holds $y[i..n] < y[j..n] \iff y[i..n] \ll y[j..n]$. Note that this does not affect the lexicographical order of suffixes (i.e. $y[i..n-1] < y[j..n-1] \iff y[i..n] < y[j..n]$). As mentioned earlier, our algorithm uses the previous and next smaller suffix tables pss and nss , which are closely related to the Lyndon table. For each position i , where $0 \leq i < n$, these tables store the (starting) positions of the closest lexicographically smaller suffixes, formally defined by

$$\begin{aligned} pss[i] &= \max\{j \mid j < i \text{ and } y[j..n] < y[i..n]\}, \text{ and} \\ nss[i] &= \min\{j \mid j > i \text{ and } y[j..n] < y[i..n]\}. \end{aligned}$$

The tables Lyn and nss are equivalent; indeed, the longest Lyndon prefix of $y[i..n-1]$ is exactly $y[i..nss[i]-1]$. Additionally, $y[pss[i]..i-1]$ is a Lyndon word.

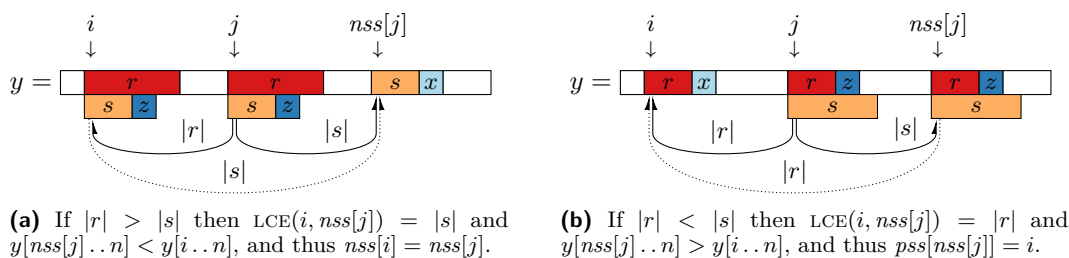
► **Lemma 8** ([19, Lemma 15]). *For any position i of a word y , it holds $Lyn[i] = nss[i] - i$.*

► **Corollary 9** ([7, Lemma 4] and Lemma 8 above). *For any position i of a word y , both $y[pss[i]..i-1]$ and $y[i..nss[i]-1]$ are Lyndon words.*

An important property of nearest smaller suffixes is that they do not *intersect*. If we draw directed edges underneath the word such that for each position i there are two outgoing edges, one to position $pss[i]$ and one to position $nss[i]$, then the resulting drawing is a planar embedding. Formally, this is expressed by the following lemma.

► **Lemma 10.** *If $i < j < nss[i]$, then $pss[j] \geq i$ and $nss[j] \leq nss[i]$.*

Proof. Because of $i < j < nss[i]$, and by the definition of nss , it holds $y[nss[i]..n] < y[i..n] < y[j..n]$. This implies $pss[j] \geq i$ and $nss[j] \leq nss[i]$. ◀



■ **Figure 2** Skipping symbols comparisons when $y[i..n] < y[j..n]$. (Best viewed in colour.)

In the remainder of the section, we show how to simultaneously compute the tables pss and nss in right-to-left order. The core of our solution is a simple folklore algorithm for the linear time computation of next and previous smaller *values*, where we substitute value comparisons for lexicographical suffix comparisons. The process is similar to the linear-time construction of the Cartesian tree [30] or the LRM-tree [4].

NAIVELYLN(y non-empty word $y[0..n-1]$ with sentinels $y[-1] = y[n] = \$$)

```

1  for  $i \leftarrow n-1$  downto 0 do
2       $j \leftarrow i+1$ 
3      while  $y[i..n] < y[j..n]$  do
4           $(\text{pss}[j], j) \leftarrow (i, \text{nss}[j])$             $\triangleright j \leftarrow j + \text{Lyn}[j]$ 
5           $(\text{nss}[i], \text{pss}[i]) \leftarrow (j, -1)$         $\triangleright \text{Lyn}[i] \leftarrow j - i$ 
6  return  $(\text{pss}, \text{nss})$ 

```

The algorithm merely adapts Algorithm LYNDONTABLE to the computation of nss , up to the use of sentinels, and adds the computation of pss . In fact, if we omit the assignment of pss entries and apply Lemma 8 (i.e. if we replace lines 4–5 with their comments), then we essentially obtain Algorithm LONGESTLYNDON described in [15, Algorithm 5] and in [13, Problem 87]. As shown in [15], the algorithm correctly computes the table Lyn (or respectively nss), and performs no more than $2n - 2$ lexicographical suffix comparisons in line 3. Note that the loop in line 3 maintains the invariant that all suffixes $y[k..n]$ with $i < k < j$ are lexicographically larger than both $y[i..n]$ and $y[j..n]$. This means that the computation of pss is correct.

If we could lexicographically compare suffixes in constant time, then NAIVELYLN would take $\mathcal{O}(n)$ time (as already mentioned in Section 2 for Algorithm LYNDONTABLE). However, for each suffix comparison, we first have to determine the length $\text{LCE}(i, j) = \min\{\ell \mid \ell \geq 0 \text{ and } y[i+\ell] \neq y[j+\ell]\}$ of the longest common prefix of two suffixes. By the definition of the lexicographical order, $y[i..n] < y[j..n]$ is equivalent to $y[i+\text{LCE}(i, j)] < y[j+\text{LCE}(i, j)]$. If we compute $\text{LCE}(i, j)$ by naive scanning, then a single suffix comparison might require $\Omega(n)$ individual symbol comparisons, and for some inputs the algorithm will take $\Omega(n^2)$ time (e.g. for the pathological word $y = a^n$).

4.2 Skipping symbol comparisons

Now we will accelerate the algorithm by exploiting previously computed LCEs. First, we show how to save comparisons for a single fixed value of i , i.e. for a single iteration of the outer loop of NAIVELYLN. During that iteration, we may have to compute the LCE between i and multiple different values of j . Assume that we have just computed $\text{LCE}(i, j) = |r|$ (with

13:12 Back-To-Front Online Lyndon Forest Construction

$y[j..n] = ru$ for some $u \in A^*$), and we discovered that $y[i..n] < y[j..n]$. Then during the next iteration of the inner loop we have to evaluate whether $y[i..n] < y[nss[j]..n]$, thus we have to compute $\text{LCE}(i, nss[j])$. Since we previously computed $nss[j]$, we must have also computed the value $\text{LCE}(j, nss[j]) = |s|$ (with $y[j..n] = sv$ for some $v \in A^*$). We observe that exactly one of the following cases holds.

- It holds $|r| > |s|$ (as depicted in Figure 2a), such that $r = szw$ for some $z \in A$ and $w \in A^*$. Let $x = y[nss[j] + |s|]$, then from $y[j..n] > y[nss[j]..n]$ follows $x < z$ and thus $sx \ll sz$. Since $y[i..n]$ has prefix $r = szw$, we have $\text{LCE}(i, nss[j]) = |s|$ and $y[nss[j]..n] < y[i..n]$. Note that this implies $nss[i] = nss[j]$, such that this is the last iteration of the inner loop.
- It holds $|r| < |s|$ (as depicted in Figure 2b), such that $s = rzw$ for some $z \in A$ and $w \in A^*$. Let $x = y[i + |r|]$, then from $y[i..n] < y[j..n]$ follows $x < z$ and thus $rx \ll rz$. Since $y[nss[j]..n]$ has prefix $s = rzw$, we have $\text{LCE}(i, nss[j]) = |r|$ and $y[nss[j]..n] > y[i..n]$. Note that this implies $pss[nss[j]] = i$.
- It holds $r = s$ and thus $\text{LCE}(i, nss[j]) \geq |s|$.

```

LCELYN( $y$  non-empty word  $y[0..n-1]$  with sentinels  $y[-1] = y[n] = \$$ )
1  for  $i \leftarrow n-1$  downto 0 do
2       $j \leftarrow i+1$ 
3      if  $y[i] \neq y[j]$  then  $\ell \leftarrow 0$ 
4      elseif  $nss[j] = j+1$  then  $\ell \leftarrow 1 + nlce[j]$ 
5      elseif  $pss[j+1] = j$  then  $\ell \leftarrow 1 + plce[j+1]$ 
6      while  $y[i+\ell] < y[j+\ell]$  do
7           $(pss[j], plce[j]) \leftarrow (i, \ell)$ 
8          if  $\ell > nlce[j]$  then
9               $\ell \leftarrow nlce[j]$ 
10              $j \leftarrow nss[j]$ 
11             elseif  $\ell < nlce[j]$  then
12                  $j \leftarrow nss[j]$ 
13             elseif  $\ell = nlce[j]$  then
14                  $j \leftarrow nss[j]$ 
15                  $\ell \leftarrow \ell + \text{SCAN-LCE}(i+\ell, j+\ell)$   $\triangleright \ell \leftarrow \text{EXTEND}(i, j, \ell)$ 
16              $(nss[i], nlce[i], pss[i]) \leftarrow (j, \ell, -1)$ 
17  return  $(pss, nss)$ 

```

The algorithm LCELYN shown above is a modification of NAIVELYLYN and exploits the new insights. It uses two auxiliary arrays $plce$ and $nlce$, in which we store $plce[i] = \text{LCE}(pss[i], i)$ and $nlce[i] = \text{LCE}(i, nss[i])$ (we update the values whenever we assign $nss[i]$ and $pss[i]$ respectively). In iteration i of the outer loop, we compute the first LCE value $\ell = \text{LCE}(i, j)$ with $j = i + 1$ in constant time by exploiting the fact that for any index j it holds either $nss[j] = j + 1$ or $pss[j + 1] = j$. Thus, if $y[i..n]$ starts with a run of a single symbol, then we have previously computed the length of this run, and we can simply assign $\text{LCE}(i, j) \leftarrow 1 + \text{LCE}(j, j + 1)$ (lines 3–5). Whenever we reach the head of the inner loop, we have already computed the value $\ell = \text{LCE}(i, j)$. Thus, we can determine the lexicographical order of $y[i..n]$ and $y[j..n]$ by comparing their first mismatching symbol (line 6). If $y[i..n] < y[j..n]$, then we enter the inner loop and assign $pss[j] \leftarrow i$ and $plce[j] \leftarrow \ell$. For the next iteration of the inner loop, we have to compute $\text{LCE}(i, nss[j])$. Here we exploit

our previous observations. If $\ell > nlce[j]$ (i.e. $|r| > |s|$ in terms of Figure 2a), then it holds $LCE(i, nss[j]) = nlce[j]$ and we continue with the next (and final) iteration of the inner loop with $\ell \leftarrow nlce[j]$ and $j \leftarrow nss[j]$ (lines 8–10). If $\ell < nlce[j]$ (i.e. $|r| < |s|$ in terms of Figure 2b), then it already holds $\ell = LCE(i, nss[j])$ and we continue with the next iteration of the inner loop with $j \leftarrow nss[j]$ (lines 11–12). Only if $\ell = nlce[j]$ we may need additional steps to compute $LCE(i, nss[j])$. However, in this case $LCE(i, nss[j]) \geq \ell$, and we can skip the first ℓ symbol comparisons when computing $LCE(i, nss[j])$. We compute the remaining part of the LCE by naive scanning, thus taking additional $LCE(i, nss[j]) - \ell + 1$ symbol comparisons (lines 13–15).

Apart from the time needed for executing line 15, algorithm LCELYN takes $\mathcal{O}(n)$ time.

4.3 Extending common prefixes with already computed LCEs

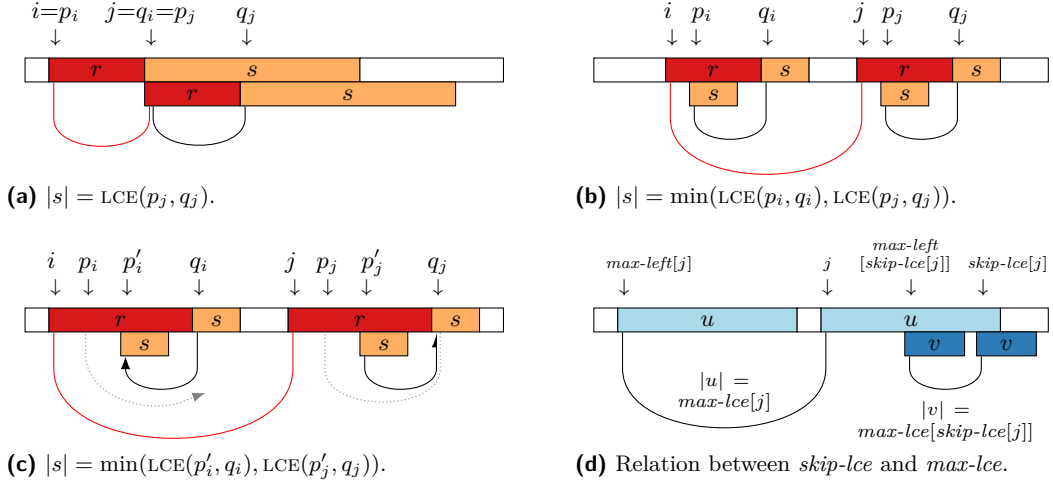
In this section, we accelerate the instruction at line 15 by replacing the naive computation of $\ell + \text{SCAN-LCE}(i + \ell, j + \ell)$ with a more sophisticated extension technique $\text{EXTEND}(i, j, \ell)$, for which we once more exploit previously computed LCEs. The technique requires $\ell > 0$. If $\ell = 0$, then we simply perform one additional symbol comparison to test $y[i] = y[j]$, which either establishes $\ell = 1$, or terminates the LCE computation in constant time.

Assume that we just reached line 15, i.e. we have to compute $LCE(i, j)$, and we have already established $LCE(i, j) \geq \ell$. If $y[q_i] \neq y[q_j]$ with $q_i = i + \ell$ and $q_j = j + \ell$, then $LCE(i, j) = \ell$, i.e. no further computation is necessary. Otherwise, let $p_j \in \{j, \dots, q_j - 1\}$ be some index with either $nss[p_j] = q_j$ or $pss[q_j] = p_j$. Such index always exists because it trivially holds either $nss[q_j - 1] = q_j$ or $pss[q_j] = q_j - 1$; we will explain how to choose p_j later. Let $p_i = p_j - (j - i)$. We compute $LCE(i, j)$ with exactly one of three methods.

1. If $p_i = i$ and $q_i = j$, then $p_i = p_j - (j - i)$ implies $p_j = p_i + j - i = j = q_i$ (as depicted in Figure 3a). From $LCE(i, j) \geq \ell$ follows $LCE(i, j) = \ell + LCE(i + \ell, j + \ell)$. Note that $LCE(i + \ell, j + \ell) = LCE(q_i, q_j) = LCE(p_j, q_j)$. Since we chose p_j such that either $nss[p_j] = q_j$ or $pss[q_j] = p_j$, we have already computed $LCE(p_j, q_j)$ and stored it either in $nlce[p_j]$ or in $plce[q_j]$. Thus we can compute $LCE(i, j) = \ell + LCE(p_j, q_j)$ in constant time.
2. If the first case does not apply, then we check whether either $nss[p_i] = q_i$ or $pss[q_i] = p_i$. If one of the two holds (as depicted in Figure 3b), then we have already computed $\ell_i = LCE(p_i, q_i)$ and stored it in $nlce[p_i]$ or in $plce[q_i]$. Analogously, we have already computed $\ell_j = LCE(p_j, q_j)$ and stored it in $nlce[p_j]$ or in $plce[q_j]$.
 - a. If $\ell_i = \ell_j$, we have established that $LCE(i, j) \geq \ell + \max(1, \ell_j)$. In this case, we say that we use $LCE(p_j, q_j)$ to extend $LCE(i, j)$. If ℓ_j is large, then we saved many symbol comparisons. We continue by recursively repeating the extension technique with $\ell \leftarrow \ell + \max(1, \ell_j)$. (We can always increase ℓ by at least 1 because we only reach this point if we initially ensured $y[q_i] = y[q_j]$.)
 - b. If $\ell_i \neq \ell_j$, then $LCE(i, j) = \ell + \min(\ell_i, \ell_j)$, and no further computation is necessary.
3. If none of the other cases apply, let $p'_i = pss[q_i]$ and $p'_j = p'_i + (j - i)$. We proceed similarly to case 2, but this time we use $\ell_i = LCE(p'_i, q_i)$ and $\ell_j = LCE(p'_j, q_j)$ (see Figure 3c). As we will show next, we have already computed ℓ_i and ℓ_j . Thus they can be used for the extension, which means that the list of cases is indeed exhaustive.

We begin the proof by showing that $p'_i > p_i$. Note that $pss[q_j] = p_j$ or $nss[p_j] = q_j$ and Corollary 9 imply that $y[p_j \dots q_j - 1] = y[p_i \dots q_i - 1]$ is a Lyndon word. Therefore, Lemma 8 implies $nss[p_i] \geq q_i$. We cannot have $nss[p_i] = q_i$ because then we would have used case 2 instead of case 3. Thus $nss[p_i] > q_i$, and due to Lemma 10 it holds $p'_i = pss[q_i] \geq p_i$. Again, we cannot have $p'_i = p_i$ because then we would have used

13:14 Back-To-Front Online Lyndon Forest Construction



■ **Figure 3** Extending common prefixes between $y[i..n]$ and $y[j..n]$ using a known lower bound $\ell = |r| \leq \text{LCE}(i, j)$ (a-c), and visualisation of skip-lce (d). (Best viewed in colour.)

case 2 instead of case 3, i.e. it holds $p'_i > p_i$ and consequently $p'_j > p_j$. Finally, from $p'_i = \text{pss}[q_i]$ and Corollary 9 follows that $y[p'_i..q_i - 1] = y[p'_j..q_j - 1]$ is a Lyndon word, such that Lemma 8 implies $\text{nss}[p'_j] \geq q_j$. Since $p_j < p'_j < q_j$ and Lemma 10 imply $\text{nss}[p'_j] \leq q_j$, we must have $\text{nss}[p'_j] = q_j$. Therefore, we have already computed $\ell_i = \text{LCE}(p'_i, q_i) = \text{plce}[q_i]$ and $\ell_j = \text{LCE}(p'_j, q_j) = \text{nlce}[p'_j]$, which means that we can compute $\text{LCE}(i, j) = \ell + \min(\ell_i, \ell_j)$ in constant time.

Note that even if $\ell_i = \ell_j$, it cannot be that $\text{LCE}(i, j) > \ell + \ell_j$. Since ℓ_i is associated with a *previous* smaller value and ℓ_j is associated with a *next* smaller value, it holds $y[q_i + \ell_j] > y[p'_i + \ell_j] = y[p'_j + \ell_j] > y[q_j + \ell_j]$.

In cases 1, 2b and 3, we take constant time to finish the computation of $\text{LCE}(i, j)$. Since we compute less than $2n$ LCEs, the total time spent for these cases is $\mathcal{O}(n)$. In case 2a, we take constant time to increase the known lower bound of $\text{LCE}(i, j)$ by $\text{LCE}(p_j, q_j)$ (however, when computing $\text{LCE}(i, j)$ we may run into this case repeatedly). Thus we should choose p_j such that we maximise $\text{LCE}(p_j, q_j)$. For this purpose, we maintain two auxiliary arrays max-left and max-lce (initialised with -1). Whenever we compute an LCE value $\text{LCE}(i, j)$ (where by design of the algorithm it always holds $i < j$), we check whether $\text{LCE}(i, j) > \text{max-lce}[j]$. If this condition holds, then we assign $\text{max-left}[j] \leftarrow i$ and $\text{max-lce}[j] \leftarrow \text{LCE}(i, j)$. For the extension technique, we always choose $p_j = \text{max-left}[q_j]$.

(Note that in general, $j \leq \text{max-left}[q_j]$. Due to the way in which we assign $\text{max-left}[q_j]$, it always holds either $\text{max-left}[q_j] = \text{pss}[q_j]$ or $\text{nss}[\text{max-left}[q_j]] = q_j$. Since also either $\text{nss}[i] = j$ or $i = \text{pss}[j]$, Lemma 10 implies $\text{max-left}[q_j] \notin \{i + 1, \dots, j - 1\}$. The iteration order of the algorithm implies $i < \text{max-left}[q_j]$.)

Apart from the invocations of case 2a, the algorithm already achieves linear time. There are $\mathcal{O}(n)$ different pairs p_j, q_j that might be used in case 2a (because for each pair it holds either $\text{nss}[p_j] = q_j$ or $p_j = \text{pss}[q_j]$). However, we may use some pairs more than once, resulting in a super-linear computation time. In the next section, we make a small change to how we update ℓ when recursing in case 2a. Perhaps counter-intuitively, we will recurse using (possibly) *smaller* values of ℓ . This allows us to use additional properties of next and previous smaller suffixes, such that each distinct pair p_j, q_j gets used in case 2a at most once.

4.4 Linear time extension of common prefixes

In this section, we ensure that we use each $\text{LCE}(p_j, q_j)$ to extend at most one LCE. This imposes a linear upper bound on the number of recursive calls to `EXTEND`, because each recursive call is preceded by an extension. For this, we need the following dynamic array.

► **Definition 11.** $\text{skip-lce}[j] = \min\{k \mid k > j \text{ and } (k + \text{max-lce}[k]) > (j + \text{max-lce}[j])\}$.

A schematic drawing of *skip-lce* is provided in Figure 3d. We maintain these values in linear time using relatively simple techniques. Whenever we have to update some value $\text{max-lce}[j]$, due to $\text{LCE}(i, j)$ for some i , we inherently discover the new value of $\text{skip-lce}[j]$. We may have to additionally assign $\text{skip-lce}[x] \leftarrow j$ for some values $x \in \{i+1, \dots, j-1\}$, but the total number of such updates is linear. The technical details can be found in Appendix B and in our well-documented implementation^{1,2} Using *skip-lce*, the only two changes needed to achieve linear time are:

- In line 15 of `LCELYN`, we call `EXTEND(i, j, 0)` if and only if $\ell = 0$, and otherwise we call `EXTEND(i, j, skip-lce[j] - j)`. We can replace ℓ with $\text{skip-lce}[j] - j$ because in this moment it holds $\text{skip-lce}[j] - j \leq \text{max-lce}[j] = \ell$. The special handling of $\ell = 0$ ensures that we compare the symbols $y[i]$ and $y[j]$.
- In case 2a of the extension technique, we replace $\ell \leftarrow \ell + \max(1, \ell_j)$ with $\ell \leftarrow \ell + \text{skip-lce}[q_j] - q_j$. This is possible due to $\text{skip-lce}[q_j] - q_j \leq \max(1, \text{max-lce}[q_j]) = \max(1, \ell_j)$.

It may seem counter-intuitive that this improves the execution time, since we no longer extend by $\ell_j = \text{max-lce}[q_j]$ but by the possibly shorter length $\text{skip-lce}[q_j] - q_j$. However, this guarantees that we use each LCE in at most one extension step. A crucial observation for showing this is that q_j only assumes values that can be obtained by repeatedly applying *skip-lce* to j . For any j , we define the repeated application of the skip function as $\text{skip-lce}^1[j] = \text{skip-lce}[j]$ and for integer $e > 1$ as $\text{skip-lce}^e[j] = \text{skip-lce}^{e-1}[\text{skip-lce}[j]]$. We then write $q_j = \text{skip-lce}^*[j]$ if and only if $\exists e : q_j = \text{skip-lce}^e[j]$. The following observation is a direct consequence of Definition 11 and readily extends to the helpful intermediate Lemma 13.

► **Observation 12.** *If $q_j = \text{skip-lce}[j]$, then for every k with $j \leq k < q_j$ it holds $k + \text{max-lce}[k] < q_j + \text{max-lce}[q_j]$.*

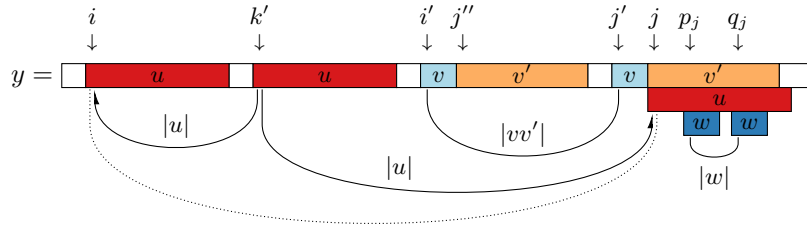
► **Lemma 13.** *If $q_j = \text{skip-lce}^*[j]$, then for every k with $j \leq k < q_j$ it holds $k + \text{max-lce}[k] < q_j + \text{max-lce}[q_j]$.*

► **Lemma 14.** *When running `LCELYN` with the modified extension technique using *skip-lce*, if we use $\text{LCE}(p_j, q_j)$ for an extension, then we will never use it for an extension again.*

Proof. For the sake of contradiction, assume that we use $\text{LCE}(p_j, q_j)$ more than once: first to compute $\text{LCE}(i', j')$, and then again to compute $\text{LCE}(i, j)$ for some i, j with $i \neq i'$ or $j \neq j'$. We have two cases to consider according to the position of j with respect to j' :

¹ see <https://archive.softwareheritage.org/swh:1:cnt:19a5c3e295db7241d03c1aafa396ba31057bbe60;origin=https://github.com/jonas-ellert/right-lyndon;visit=swh:1:snp:551e86cb1c7ff9452669f838323fad1aea23a6f2;anchor=swh:1:rev:f292fd218d0edb546530aaf517923eccb79b2736;path=/right-lyndon-extension-linear.hpp;lines=88-135>

² see same link as given in previous footnote, lines 52–71



■ **Figure 4** Supplementary drawing for the proof of Lemma 14. (Best viewed in colour.)

Case $j < j'$: Since we use $\text{LCE}(p_j, q_j)$ to extend $\text{LCE}(i, j)$, it holds $q_j = \text{skip-lce}^*[j]$. However, Lemma 13 implies that $j' + \text{max-lce}[j'] < q_j + \text{max-lce}[q_j]$. This contradicts the assumption that we used $\text{LCE}(p_j, q_j)$ to extend $\text{LCE}(i', j')$. (Note that $\text{max-lce}[q_j]$ has not changed. If it had changed, then we would have also updated $\text{max-left}[q_j]$. Thus a different p_j would have been used for the extension of $\text{LCE}(i, j)$ and $\text{LCE}(i', j')$.)

Case $j \geq j'$: This case is accompanied by a schematic drawing in Figure 4. If we are comparing i and j then either $\text{nss}[i] = j$ or $\text{pss}[j] = i$, which implies $y[k..n] > y[j..n]$ for all k with $i < k < j$. Let $y[k'..n]$ be the lexicographically smallest suffix amongst the suffixes starting between positions i and j , then it holds $\text{nss}[k'] = j$. Thus, at the time we compute $\text{LCE}(i, j)$, we have computed $\text{LCE}(k', j)$ already, and it holds $\text{max-lce}[j] \geq \text{LCE}(k', j)$. Let $j'' = i' + j - j'$, then due to our choice of k' it holds $y[j''..n] \geq y[k'..n] > y[j..n]$ and thus $\text{LCE}(j'', j) \leq \text{LCE}(k', j)$. Therefore, we have $j + \text{max-lce}[j] \geq j + \text{LCE}(k', j) \geq j + \text{LCE}(j'', j) = j' + \text{LCE}(i', j') \geq q_j + \text{LCE}(p_j, q_j)$. However, according to Lemma 13, $j + \text{max-lce}[j] \geq q_j + \text{LCE}(p_j, q_j)$ contradicts $q_j = \text{skip-lce}^*[j]$, which means that we do not use $\text{LCE}(p_j, q_j)$ to extend $\text{LCE}(i, j)$. ◀

We now state the main results, which are direct consequences of Lemma 14 and Lemma 8.

► **Theorem 15.** *Algorithm LCELYN using the modified extension technique computes the previous and next smaller suffix tables of a word over a general ordered alphabet. It does so in a back-to-front online manner, and in linear time with respect to the length of the word.*

► **Corollary 16.** *Theorem 15 also holds when computing the Lyndon table and forest.*

Proof. For the Lyndon table we output Lyn alongside nss , using Lemma 8. For the Lyndon forest, we additionally interleave the computation with Algorithm LYNDONFOREST. ◀

References

- 1 Golnaz Badkobeh and Maxime Crochemore. Left Lyndon tree construction. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2020, Prague, Czech Republic, August 31-September 2, 2020*, pages 84–95. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020. <https://arxiv.org/abs/2011.12742>.
- 2 Uwe Baier. Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, pages 23:1–23:12, 2016. doi:10.4230/LIPIcs.CPM.2016.23.
- 3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The “runs” theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

- 4 Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. LRM-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012. doi:10.1016/j.tcs.2012.08.010.
- 5 Frédérique Bassino, Julien Clément, and Cyril Nicaud. The standard factorization of Lyndon words: an average point of view. *Discret. Math.*, 290(1):1–25, 2005. doi:10.1016/j.disc.2004.11.002.
- 6 Nico Bertram, Jonas Ellert, and Johannes Fischer. Lyndon words accelerate suffix sorting. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPICs*, pages 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ESA.2021.15.
- 7 Philip Bille, Jonas Ellert, Johannes Fischer, Inge Li Gørtz, Florian Kurpicz, J. Ian Munro, and Eva Rotenberg. Space efficient construction of Lyndon arrays in linear time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPICs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ICALP.2020.14.
- 8 Dany Breslauer, Tao Jiang, and Zhigen Jiang. Rotations of periodic strings and short superstrings. *J. Algorithms*, 24(2):340–353, 1997. doi:10.1006/jagm.1997.0861.
- 9 Philippe Chassaing and Lucas Mercier. The height of the Lyndon tree. *Discrete Mathematics & Theoretical Computer Science*, 2013.
- 10 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.
- 11 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. 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 22–34, 2016. doi:10.1007/978-3-319-46049-9_3.
- 12 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. The maximal number of cubic runs in a word. *J. Comput. Syst. Sci.*, 78(6):1828–1836, 2012. doi:10.1016/j.jcss.2011.12.005.
- 13 Maxime Crochemore, Thierry Lecroq, and Wojciech Rytter. *125 Problems in Text Algorithms*. Cambridge University Press, 2021. 334 pages.
- 14 Maxime Crochemore and Dominique Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.
- 15 Maxime Crochemore and Luís M. S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 806:1–9, February 2020.
- 16 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.
- 17 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPICs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ICALP.2021.63.
- 18 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

- 19 Frantisek Franek, A. S. M. Sohiddul Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 172–184. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016. URL: <http://www.stringology.org/event/2016/p15.html>.
- 20 Frantisek Franek and Michael Liut. Algorithms to compute the Lyndon array revisited. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2019, Prague, Czech Republic, August 26-28, 2019*, pages 16–28. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019. URL: <http://www.stringology.org/event/2019/p03.html>.
- 21 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003. doi:10.1016/S0304-3975(03)00099-9.
- 22 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.
- 23 Dmitry Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.
- 24 M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983. Reprinted in 1997.
- 25 Felipe A. Louza, Sabrina Mantaci, Giovanni Manzini, Marinella Sciortino, and Guilherme P. Telles. Inducing the Lyndon array. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 138–151. Springer, 2019. doi:10.1007/978-3-030-32686-9_10.
- 26 R. C. Lyndon. On Burnside problem I. *Trans. Amer. Math. Soc.*, 77:202–215, 1954.
- 27 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327. SIAM, 1990. URL: <http://dl.acm.org/citation.cfm?id=320176.320218>.
- 28 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting suffixes of a text via its Lyndon factorization. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 119–127. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013. URL: <http://www.stringology.org/event/2013/p11.html>.
- 29 Joe Sawada and Frank Ruskey. Generating Lyndon brackets.: An addendum to: Fast algorithms to generate necklaces, unlabeled necklaces and irreducible polynomials over GF(2). *J. Algorithms*, 46(1):21–26, 2003. doi:10.1016/S0196-6774(02)00286-9.
- 30 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.

A Supplementary material for Section 3

The following inequalities (obtained by comparing sums and integrals) will be needed.

► **Lemma 17.** Let $H_n = \sum_{i=1}^n \frac{1}{i}$ and $L_n = \sum_{i=1}^n \log i$. For all $n \geq 1$, $H_n \leq \log n + 1$ and $L_n \geq n \log n - n$.

Proof. The mapping $x \mapsto \log x$ is increasing, so for any $i \geq 1$ and any $x \in [i, i+1]$, we have $\log x \leq \log(i+1)$. We integrate on the length-1 interval $[i, i+1]$ to get $\int_i^{i+1} \log x \, dx \leq \log(i+1)$. Summing for i ranging from 1 to $n-1$ gives

$$\int_1^n \log x \, dx \leq \sum_{i=2}^n \log i = L_n,$$

which concludes the proof since $\int_1^n \log x \, dx = n \log n - n + 1$. The proof is similar for H_n by considering the decreasing map $x \mapsto \frac{1}{x}$. ◀

A.1 Proof of Lemma 5

► **Lemma 5.** *Let Λ be an ordered alphabet with $|\Lambda| \geq 2$ letters, $\# \notin \Lambda$ be a new letter that is greater than every letter of Λ , and y be a word selected from Λ^t uniformly at random. Then there exists a constant $c \geq 1$ such that the probability that $y\#$ is a Lyndon word is at most $\frac{c}{t}$.*

Proof. Any Lyndon word w of length at least 2 can uniquely be written [16] as $w = u^e v b$ where u is a Lyndon word, e is a positive integer, va is a prefix of u , and a and b are letters such that $a < b$. Thus, if $y\#$ is a Lyndon word, then y can be uniquely written $y = ux$ where x is the longest border of y (or the empty word if y is a Lyndon word). As y is entirely defined by its associated u and its length t , this yields a bijection between the words y such that $y\# \in \mathcal{L}$ and the union of the Lyndon words of length i , for i ranging from 1 to t . Let $\kappa = |\Lambda|$. As we already established that $|\mathcal{L} \cap \Lambda^i| \leq \frac{\kappa^i}{i}$, we have

$$\mathbb{P}_t(y\# \in \mathcal{L}) \leq \frac{1}{\kappa^t} \sum_{i=1}^t \frac{\kappa^i}{i} = \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{t-i} = \frac{1}{t} \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{1-i/t}.$$

Observe that for any $x \in [0, \frac{t-1}{t}]$, we have $\frac{1}{1-x} \leq 1 + tx$, and therefore

$$\frac{1}{t} \sum_{i=0}^{t-1} \frac{\kappa^{-i}}{1-i/t} \leq \frac{1}{t} \sum_{i=0}^{t-1} (1+i)\kappa^{-i} \leq \frac{c}{t},$$

if we set $c = \sum_{i=0}^{\infty} (1+i)\kappa^{-i}$. This concludes the proof. ◀

A.2 Proof of Equation (2)

Recall that $\lambda = \lfloor \frac{j-i}{k} \rfloor$, so that there are k values of i for each λ . Hence

$$\sum_{i=0}^{j-k-1} (k+2) q \left(k, s, \left\lfloor \frac{j-i}{k} \right\rfloor \right) \leq k(k+2) \sum_{\lambda=1}^{\lceil j/k \rceil} q(k, s, \lambda).$$

We consider separately the two terms coming from Eq. (1):

$$\begin{aligned} \sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} k(k+2) \sum_{\lambda=1}^{\lceil j/k \rceil} \frac{s}{\sigma^{2k}} \left(\frac{\sigma^k - s}{\sigma^k} \right)^{\lambda-1} &\leq \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)s}{\sigma^{2k}} \sum_{\lambda=1}^{\infty} \left(1 - \frac{s}{\sigma^k} \right)^{\lambda-1} \\ &= \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)s}{\sigma^{2k}} \frac{1}{1 - (1 - s/\sigma^k)} \\ &= \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{k(k+2)}{\sigma^k} \\ &\leq \sum_{k=1}^{\infty} \frac{k(k+2)}{2^{k-1}} =: \nabla_2. \end{aligned}$$

13:20 Back-To-Front Online Lyndon Forest Construction

For the second part of Eq. (1), we have:

$$\begin{aligned}
\sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} k(k+2) \sum_{\lambda=1}^{\lfloor j/k \rfloor} \frac{c}{\lambda \sigma^k} \left(\frac{\sigma^k - s}{\sigma^k} \right)^\lambda &\leq \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \sum_{\lambda=1}^{\infty} \frac{1}{\lambda} \left(1 - \frac{s}{\sigma^k} \right)^\lambda \\
&= \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \log \left(\frac{1}{1 - (s/\sigma^k)} \right) \\
&\leq \sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \log \left(\frac{\sigma^k}{s} \right).
\end{aligned}$$

We have

$$\sum_{k=1}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \log \left(\frac{\sigma^k}{s} \right) = \sum_{s=1}^{\sigma-1} \frac{3c}{\sigma} \log \left(\frac{\sigma}{s} \right) + \sum_{k=2}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \log \left(\frac{\sigma^k}{s} \right).$$

We use Lemma 17 for the first sum:

$$\begin{aligned}
\sum_{s=1}^{\sigma-1} \log \left(\frac{\sigma}{s} \right) &= (\sigma - 1) \log \sigma - \sum_{s=1}^{\sigma-1} \log s \leq (\sigma - 1) \log \sigma - (\sigma - 1) \log(\sigma - 1) + \sigma - 1 \\
&= (\sigma - 1) \left(\log \left(\frac{\sigma}{\sigma - 1} \right) + 1 \right) \leq (1 + \log 2) \sigma.
\end{aligned}$$

Thus

$$\sum_{s=1}^{\sigma-1} \frac{3c}{\sigma} \log \left(\frac{\sigma}{s} \right) \leq 3c(1 + \log 2) =: \nabla_2.$$

Finally

$$\begin{aligned}
\sum_{k=2}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck(k+2)}{\sigma^k} \log \left(\frac{\sigma^k}{s} \right) &\leq \sum_{k=2}^{\infty} \sum_{s=1}^{\sigma-1} \frac{ck^2(k+2) \log \sigma}{\sigma^k} \\
&\leq \frac{\log \sigma}{\sigma} \sum_{k=2}^{\infty} \frac{ck^2(k+2)}{\sigma^{k-2}} \leq \sum_{k=2}^{\infty} \frac{ck^2(k+2)}{2^{k-2}} =: \nabla_3.
\end{aligned}$$

This concludes the proof by setting $\gamma_1 = \nabla_1 + \nabla_2 + \nabla_3$.

A.3 Proof of Equation (3)

Using Lemma 17 we have

$$\begin{aligned}
\sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \sum_{s=1}^{\sigma-1} \sum_{\ell=1}^{k-1} \frac{k+2}{\ell \sigma^k} &\leq \frac{\sigma-1}{\sigma} \sum_{k=1}^{\lfloor 3 \log_2 n \rfloor} \frac{(k+2)(\log k + 1)}{\sigma^{k-1}} \\
&\leq \sum_{k=0}^{\infty} \frac{(k+3)(\log(k+1) + 1)}{\sigma^k} \\
&\leq \sum_{k=0}^{\infty} \frac{(k+3)(\log(k+1) + 1)}{2^k} =: \gamma_2.
\end{aligned}$$

B Supplementary material for Section 4

In this section, we outline how to maintain *skip-lce* for all positions. Definition 11 depends solely on the values of *max-lce*, and thus updates to *skip-lce* are always accompanied by updates to *max-lce* and *max-left*. Assume that we just computed some $\text{LCE}(i, j)$, causing the updates $\text{max-lce}[j] \leftarrow \text{LCE}(i, j)$ and $\text{max-left}[j] \leftarrow i$. Consequently, we may have to update $\text{skip-lce}[j]$, and also assign $\text{skip-lce}[k] \leftarrow j$ for possibly multiple indices $k < j$.

B.1 Updating $\text{skip-lce}[j]$

We inherently discover the new value of $\text{skip-lce}[j]$ while computing $\text{LCE}(i, j)$. If we find that $\text{LCE}(i, j) = 0$ due to line 3 in Algorithm `LCELYN`, then we have to assign $\text{skip-lce}[j] \leftarrow j + 1$. If we obtain $\text{LCE}(i, j)$ from lines 4 or 5, we have to assign $\text{skip-lce}[j] \leftarrow j + \text{LCE}(i, j)$ (the correctness of this is relatively easy to see, since there is a run $y[j..j + \text{LCE}(i, j) - 1] = y[j]^{\text{LCE}(i, j)}$ of a single symbol). The only remaining situation in which an update to $\text{max-left}[j]$ may occur is after using the extension technique. Here, the new value of $\text{skip-lce}[j]$ depends on which case of Section 4.3 we terminate in (we use the same notation as in Section 4.3):

- We assign $\text{skip-lce}[j] \leftarrow \text{skip-lce}[q_j]$
 - if we terminate in case 1, or
 - if we terminate in case 2b with $\ell_i > \ell_j$, or
 - if we terminate in case 3 with $\ell_i \geq \ell_j$.

In all three cases, at termination time we have $j + \text{LCE}(i, j) = q_j + \text{max-lce}[q_j]$. This, together with $q_j = \text{skip-lce}^*[j]$ and Lemma 13, implies the correctness of assigning $\text{skip-lce}[j] \leftarrow \text{skip-lce}[q_j]$.

- We assign $\text{skip-lce}[j] \leftarrow q_j$, otherwise. Explicitly, this happens
 - if we terminate in case 2b with $\ell_i < \ell_j$, or
 - if we terminate in case 3 with $\ell_i < \ell_j$.

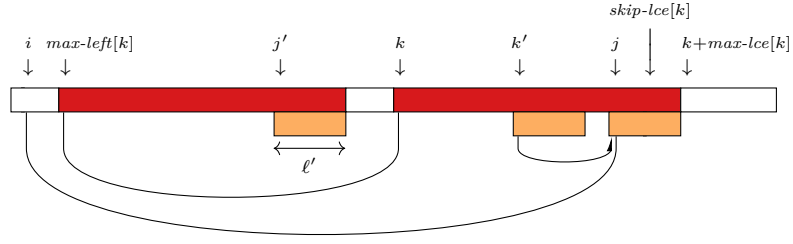
In both cases, at termination time we have $j + \text{LCE}(i, j) < q_j + \text{max-lce}[q_j]$. This, together with $q_j = \text{skip-lce}^*[j]$ and Lemma 13, implies the correctness of assigning $\text{skip-lce}[j] \leftarrow q_j$.

B.2 Updating $\text{skip-lce}[k] \leftarrow j$ for all matching k

The algorithm below describes the full process of maintaining the arrays *skip-lce*, *max-left*, and *max-lce*.³ We invoke this update procedure after every LCE computation. We will show the correctness of this approach in a moment. Before, we explain the total time needed for all invocations of the algorithm. Apart from the while-loop, each line takes constant time. The while-loop serves the purpose of finding the indices k for which we have to assign $\text{skip-lce}[k] \leftarrow j$. It does so by repeatedly applying *max-left* to j . This can be interpreted as following some right-to-left path of PSS and (reversed) NSS edges from j to i . Since we assign $\text{max-left}[j] \leftarrow i$ immediately afterwards, and due to Lemma 10, we will not follow any of these edges again during future invocations of the algorithm. Therefore, the total number of iterations of the while-loop and thus the total time spent for maintaining *skip-lce* is $\mathcal{O}(n)$.

³ The algorithm corresponds directly to the following part of our implementation:
<https://archive.softwareheritage.org/>
 swh:1:cnt:19a5c3e295db7241d03c1aafa396ba31057bbe60;
 origin=https://github.com/jonas-ellert/right-lyndon;
 visit=swh:1:snp:551e86cb1c7ff9452669f838323fad1aea23a6f2;
 anchor=swh:1:rev:f292fd218d0edb546530aaf517923eccb79b2736;
 path=/right-lyndon-extension-linear.hpp;lines=52-71

13:22 Back-To-Front Online Lyndon Forest Construction



■ **Figure 5** Supplementary drawing for Property 18.

UPDATE SKIP VALUES (after computing $LCE(i, j)$)

```

1  if  $LCE(i, j) > max-lce[j]$  then
2     $k \leftarrow max-left[j]$ 
3    while  $k > i$  do
4       $skip-lce[k] \leftarrow \min(skip-lce[k], j)$ 
5       $k \leftarrow max-left[k]$ 
6   $max-left[j] \leftarrow i$ 
7   $max-lce[j] \leftarrow LCE(i, j)$ 
8  update  $skip-lce[j]$  (depending on how we computed  $LCE(i, j)$ , as described above)

```

It remains to be shown that the algorithm maintains the arrays correctly. This is obvious for the arrays $max-lce$ and $max-left$. We already discussed how to correctly assign $skip-lce[j]$ in line 8 earlier. Thus it remains to be shown that we assign $skip-lce[k] \leftarrow j$ correctly and completely. We inductively assume that at the time we invoke the update algorithm for $LCE(i, j)$, all previous updates worked as intended, such that the arrays $skip-lce$, $max-lce$, and $max-left$ contain the correct values describing the current state of the main algorithm execution. We start the proof by showing the following auxiliary property:

► **Property 18.** *At a fixed point in time, let k be an arbitrary index with skip value $skip-lce[k]$, and let j be an arbitrary index from $\{k + 1, \dots, skip-lce[k] - 1\}$. If there is some index $i < k$ such that either $nss[i] = j$ or $i = pss[j]$, then there is some index $k' \in \{k, \dots, j - 1\}$ with $nss[k'] = j$ and $LCE(k', j) \geq k + max-lce[k] - j$.*

Proof. The property is illustrated in Figure 5.

■ Due to the assumptions about i, k, j , as well as Lemma 10 and Definition 11, it holds

$$i < max-left[k] < k < j < skip-lce[k] \leq k + max-lce[k]$$

- Let $\ell' = k + max-lce[k] - j$. The suffix $y[j'..n]$ with $j' = j - (k - max-left[k])$ has prefix $y[j'..j' + \ell' - 1] = y[j..j + \ell' - 1]$ (due to the LCE between $max-left[k]$ and k).
- Due to either $nss[i] = j$ or $i = pss[j]$, no suffix $y[k''..n]$ with $i < k'' < j$ can have a prefix $y[k''..k'' + \ell' - 1] < y[j..j + \ell' - 1]$ (since otherwise $y[k''..n] < y[j..n]$).
- Due to the previous two points, the lexicographically smallest suffix $y[k'..n]$ that satisfies $max-left[k] < k' < j$ also has prefix $y[j..j + \ell' - 1]$. Because of our choice of k' , all suffixes starting between k' and j are lexicographically larger than $y[k'..n]$, and it holds $nss[k'] = j$ ($nss[k'] > j$ would contradict Lemma 10). Due to $nss[k'] = j$ and Lemma 10, it holds $k' \geq k$. ◀

B.2.1 Correctness of the assignments

Now we show that the updates performed by the algorithm are correct.

- If the algorithm performs some update $skip-lce[k] \leftarrow j$ after computing $LCE(i, j)$, then clearly $i < k < j < skip-lce[k]$ (this follows readily from the pseudocode).
- Since either $nss[i] = j$ or $i = pss[j]$, Property 18 applies and there is some $k' \in \{k, \dots, j-1\}$ with $nss[k'] = j$ and $LCE(k', j) \geq k + max-lce[k] - j$. Due to the order in which we process the indices, we have already computed $LCE(k', j)$, and $max-lce[j] \geq k + max-lce[k] - j$ already holds.
- We only run the update algorithm if $LCE(i, j) > max-lce[j]$, thus $LCE(i, j) > k + max-lce[k] - j$.
- Equivalently, $j + LCE(i, j) > k + max-lce[k]$. Thus it is correct to assign $skip-lce[k] \leftarrow j$.

B.2.2 Completeness of the assignments

From now on, analogously to $skip-lce^*$ in Section 4.4, we use the notation $k = max-left^*[j]$ to denote that k can be obtained from j by repeatedly applying $max-left$. The update algorithm considers exactly the indices $k > i$ with $k = max-left^*[j]$. Now we show that the updates performed by the algorithm are complete, i.e. there is no index k for which we should assign $skip-lce[k] \leftarrow j$, but that does not satisfy $k = max-left^*[j]$.

- Consider an index k for which we have to assign $skip-lce[k] \leftarrow j$. If $k \neq max-left^*[j]$, then there must be indices $j'' = max-left^*[j]$ (possibly $j'' = j$) and $i'' = max-left[j'']$ with $i'' < k < j''$.
- Since we have to assign $skip-lce[k] \leftarrow j$, it currently holds $k < j'' \leq j < skip-lce[k]$. Therefore, $j'' + max-lce[j''] \leq k + max-lce[k]$ (otherwise, we would have assigned $skip-lce[k] \leftarrow j''$ earlier).
- Since either $nss[i''] = j''$ or $i'' = pss[j'']$, Property 18 applies and there is some $k' \in \{k, \dots, j''-1\}$ with $nss[k'] = j''$ and $LCE(k', j'') \geq k + max-lce[k] - j''$.
- Combining the previous two points, we have $j'' + max-lce[j''] = LCE(k', j'') = k + max-lce[k]$. This, however, means that $max-left[j''] \geq k'$, which contradicts the definition of i'' .

Cartesian Tree Subsequence Matching

Tsubasa Oizumi ✉

Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan

Takeshi Kai

Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan


Takuya Mieno^{1,2} ✉ 

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

Shunsuke Inenaga ✉ 

Department of Informatics, Kyushu University, Fukuoka, Japan

RESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Hiroki Arimura ✉ 

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

Abstract

Park et al. [TCS 2020] observed that the similarity between two (numerical) strings can be captured by the Cartesian trees: The Cartesian tree of a string is a binary tree recursively constructed by picking up the smallest value of the string as the root of the tree. Two strings of equal length are said to Cartesian-tree match if their Cartesian trees are isomorphic. Park et al. [TCS 2020] introduced the following *Cartesian tree substring matching* (*CTMStr*) problem: Given a text string T of length n and a pattern string of length m , find every consecutive substring $S = T[i..j]$ of a text string T such that S and P Cartesian-tree match. They showed how to solve this problem in $\tilde{O}(n+m)$ time. In this paper, we introduce the *Cartesian tree subsequence matching* (*CTMSeq*) problem, that asks to find every minimal substring $S = T[i..j]$ of T such that S contains a subsequence S' which Cartesian-tree matches P . We prove that the CTMSeq problem can be solved efficiently, in $O(mnp(n))$ time, where $p(n)$ denotes the update/query time for dynamic predecessor queries. By using a suitable dynamic predecessor data structure, we obtain $O(mn \log \log n)$ -time and $O(n \log m)$ -space solution for CTMSeq. This contrasts CTMSeq with closely related *order-preserving subsequence matching* (*OPMSeq*) which was shown to be NP-hard by Bose et al. [IPL 1998].

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, pattern matching, Cartesian tree subsequence matching, order preserving matching, episode matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.14

Funding *Takuya Mieno*: JSPS KAKENHI Grant Number JP20J11983.

Shunsuke Inenaga: JST PRESTO Grant Number JPMJPR1922.

Hiroki Arimura: JSPS KAKENHI Grant Number 20H00595, JST CREST Grant Number JP-MJCR18K3.

Acknowledgements The authors thank the anonymous referees for drawing our attention to reference [10].

¹ Corresponding author

² Current affiliation: University of Electro-Communications, Japan (tmieno@uec.ac.jp)



1 Introduction

A time series is a sequence of events which can be represented by symbols or numbers in many cases. An *episode* is a collection of events which occur in a short time period. The *episode matching* problem asks to find every *minimal* substring $S = T[i..j]$ of a text T such that a pattern P is a (non-consecutive) subsequence of S . Let n and m be the lengths of the text T and the pattern P , respectively. There exists a naïve $O(mn)$ -time $O(1)$ -space algorithm for episode matching, which scans the text back and forth. In 1997, Das et al. [7] presented a weakly subquadratic $O(mn/\log m)$ -time $O(m)$ -space algorithm for episode matching. Very recently, Bille et al. [3] showed that even a simpler version of episode matching, which computes the *shortest* substring containing P as a subsequence, cannot be solved in strongly subquadratic $O((mn)^{1-\epsilon})$ time for any constant $\epsilon > 0$, unless the Strong Exponential Time Hypothesis (SETH) fails.

In some applications, such as analysis of time series data of stock prices, one is often more interested in finding patterns of price fluctuations rather than the exact prices. The *order preserving matching* (OPM) model [16] is motivated for such purposes, where the task is to find consecutive substring S of a numeric text string T such that the relative orders of values in S are the same as that of a query numeric pattern string P . The order preserving *substring* matching problem (OPMStr) can be solved in $\tilde{O}(n+m)$ time [16, 17, 5, 6]. On the other hand, the order preserving *subsequence* matching problem (OPMSeq) is known to be NP-hard [4]. Another known model of pattern matching, called *parameterized matching* (PM), is able to capture structures of strings, namely, two strings are said to parameterized match if one string can be obtained by applying a character bijection to the other string [1]. Again, the parameterized *substring* matching problem (PMStr) can be solved in $\tilde{O}(n+m)$ time (see [1, 2, 14, 8, 19] and references therein), but the parameterized *subsequence* matching (PMSeq) is NP-hard [15]. We remark that both order preserving matching and parameterized matching belong to a general framework of pattern matching called the *substring-consistent equivalence relation* (SCER) [18]. Let \approx denote a string equivalence relation, and suppose that $X \approx Y$ holds for two strings X and Y of equal length n . We say that \approx is an SCER if $X[i..j] \approx Y[i..j]$ hold for any $1 \leq i \leq j \leq n$.

Cartesian tree matching (CTM), proposed by Park et al. [20], is a new class of SCER that is also motivated for numeric string processing. The Cartesian tree $CT(T)$ of a string T is a binary tree such that the root of $CT(T)$ is i if i is the leftmost occurrence of the smallest value in T , the left child of the root $T[i]$ is $CT(T[1..i-1])$, and the right child of the root $T[i]$ is $CT(T[i+1..n])$. We say that two strings Cartesian-tree match if the Cartesian trees of the two strings are isomorphic as ordered trees [13], i.e., preserving both the parent and sibling orders. Observe that CTM is similar to OPM. For instance, strings $(7, 2, 3, 1, 5)$ and $(9, 2, 4, 1, 6)$ both Cartesian-tree match and order-preserving match. It is easy to observe that if two strings order-preserving match, then they also Cartesian-tree match, but the opposite is not true in general. Thus CTM allows for more relaxed pattern matching than OPM. Indeed, the constraints for OPM that impose the relative order of all positions in the pattern can be too strict for some applications [20]. For example, two strings $(7, 2, 3, 1, 5)$ and $(6, 2, 4, 1, 9)$ both having a w-like shape do not order-preserving match. On the other hand, their similarity can be captured with CTM, since $(7, 2, 3, 1, 5)$ and $(6, 2, 4, 1, 9)$ Cartesian-tree match. This lead to the study of the Cartesian tree *substring* matching (CTMStr) problem, which asks to find every substring S of T such that S and P Cartesian-tree match. The CTMStr problem can be solved efficiently, in $\tilde{O}(n+m)$ time [20, 21].

On the other hand, since real-world numeric sequences contain errors and indeterminate values, patterns of interest may not always appear consecutively in the target data. Therefore numeric sequence pattern matching scheme, which allows for skipping some data and matching to non-consecutive subsequences, is desirable. However, such pattern matching is not supported by the CTMStr algorithms. Given the aforementioned background, this paper introduces Cartesian tree *subsequence* matching (CTMSeq), and further shows that this problem can be solved efficiently. Namely, we can find, in time polynomial in n and m , every minimal substring $S = T[i..j]$ of a text T such that there exists a subsequence S' of S where $CT(S')$ and $CT(P)$ are isomorphic. We remark that this is the CTM version of episode matching, which is also the first polynomial-time subsequence matching under SCER (except for exact matching, which is episode matching).

The contribution of this paper is the following:

- We first present a simple algorithm for solving CTMSeq in $O(mn^2)$ time and $O(mn)$ space based on dynamic programming (Section 3, Algorithm 1).
- We present a faster $O(mn \log \log n)$ -time $O(mn)$ -space algorithm for solving CTMSeq (Section 4, Algorithm 2). To achieve this speed-up, we exploit useful properties of our method that permits us to improve the $O(n^2)$ -time part of Algorithm 1 with $O(n)$ predecessor queries.
- We present space-efficient versions of the above algorithms that require only $O(n \log m)$ space, which are based on the idea from the *heavy-path decomposition* (Section 5).

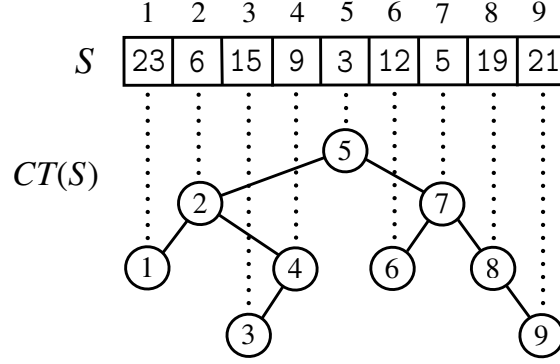
Technically speaking, our algorithms are related to the work by Gawrychowski et al. [10], who considered the problem of deciding whether two *indeterminate* strings of equal length n match under SCER. They showed that the CTM version of the problem can be solved in $O(n \log^2 n)$ time with $O(n \log n)$ space when the number r of uncertain characters in the strings is constant, using predecessor queries. They also proved that the OPM and PM versions of the problem are NP-hard for $r = 2$. NP-hardness for the OPM version in the case of $r = 3$ was previously shown in [12]. Our results on CTMSeq can be seen as yet another example that differentiates between CTM and OPM in terms of the time complexity class.

2 Preliminaries

2.1 Basic Notations and Assumptions

For any positive integers i, j with $1 \leq i \leq j$, we define a set $[i] = \{1, \dots, i\}$ of integers and a discrete interval $[i, j] = \{i, i + 1, \dots, j\}$. Let $\Sigma = \{1, \dots, \sigma\}$ be an *integer alphabet* of size σ . An element of Σ is called a *character*. A sequence of characters is called a *string*. The *length* of string S is denoted by $|S|$. The empty string ε is the string of length 0. For a string $S = (S[1], S[2], \dots, S[|S|])$, $S[i]$ denotes the i -th character of S for each i with $1 \leq i \leq |S|$. For each i, j with $1 \leq i \leq j \leq |S|$, $S[i..j]$ denotes the *substring* of S starting from i and ending at j . For convenience, let $S[i..j] = \varepsilon$ for $i > j$. We write $\min(S) := \min\{S[i] \mid i \in [n]\}$ for the minimum value contained in the string S . In this paper, all characters in the string S assume to be different from each other without loss of generality [16]³. Under the assumption, we denote by $\text{minidx}(S) := i$ the unique index satisfying the condition $S[i] = \min(S)$. For any $0 \leq m \leq n$, let \mathcal{I}_m^n be the set consisting of all *subscript sequence* $I = (i_1, \dots, i_m) \in [n]^m$ in ascending order satisfying $1 \leq i_1 < \dots < i_m \leq n$. Clearly, $|\mathcal{I}_m^n| = \binom{n}{m}$ holds. For a subscript

³ If the same character occurs more than once in S , the pair $ci = (c, i)$ of the original character c and index i can be extended as a new character to satisfy the assumption.



■ **Figure 1** Illustration for Cartesian tree $CT(S)$ of $S = (23, 6, 15, 9, 3, 12, 5, 19, 21)$. Since the minimum value among S is $S[5]$, node $v = 5$ is the root of $CT(S)$, $CT(S[1..4])$ is the left subtree of v , and $CT(S[6..9])$ is the right subtree of v . Then, $v.L = 2$, $v.R = 7$, $S_v = S[1..9] = S$, $S_{v.L} = S[1..4]$, and $S_{v.R} = S[6..9]$.

sequence $I = (i_1, \dots, i_m) \in \mathcal{I}_m^n$, we denote by $S_I := (S[i_1], \dots, S[i_m])$ the *subsequence* of S corresponding to I . Intuitively, a subsequence of S is a string obtained by removing zero or more characters from S and concatenating the remaining characters without changing the order. For a subscript sequence $I = (i_1, \dots, i_m) \in \mathcal{I}_m^n$ and its elements $i_s, i_t \in I$ with $i_s \leq i_t$, $I[i_s : i_t]$ denotes the substring of I that starts with i_s and ends with i_t . In this paper, we assume the standard *word RAM model* of word size $w = \Omega(\log n)$. Also we assume that $\sigma \leq 2^w$, i.e., any character in Σ fits within a single word.

2.2 Cartesian Tree

The Cartesian tree of string S , denoted by $CT(S)$, is the ordered binary tree recursively defined as follows: If $S = \varepsilon$, then $CT(S)$ is empty, and otherwise, $CT(S)$ is the tree rooted at v such that the left subtree of v is $CT(S[1..v-1])$, and the right subtree of v is $CT(S[v+1..|S|])$, where $v = \text{minidx}(S)$. For a node v , we denote by $v.L$ the left child of v if such a child exists and let $v.L = \text{nil}$ otherwise. Similarly, we use the notation $v.R$ for the right child of v . $CT(S)_v$ denotes the subtree of $CT(S)$ rooted at v . We say that two Cartesian trees $CT(S)$ and $CT(S')$ are *isomorphic* as ordered trees [13], denoted $CT(S) = CT(S')$.

There is an interplay between a sequence and its Cartesian tree as follows: We note that the indices of S identify the nodes of $CT(S)$, and *vice versa*. For any node v of $CT(S)$, we define the substring S_v of S recursively as follows:

- (i) If v is the root of $CT(S)$, then $S_v = S = S[1..|S|]$.
- (ii) If v is a node with substring $S_v = S[\ell..r]$, then $S[v]$ is the minimum value in $S[\ell..r]$, $S_{v.L} = S[\ell..v-1]$, and $S_{v.R} = S[v+1..r]$.

An example of a Cartesian tree is shown in Figure 1.

2.3 Cartesian Tree Subsequence Matching

Let T be a *text* string of length n and P be a *pattern* string of length $m \leq n$. We say that a pattern P *matches* text T , denoted by $P \sqsubseteq T$, if there exists a subscript sequence $I = (i_1, \dots, i_m) \in \mathcal{I}_m^n$ of T such that $CT(T_I) = CT(P)$ holds. Then, we refer to the subscript sequence I as a *trace*.

A possible choice of the notion of occurrences of a pattern P in T is to employ the traces of P as occurrences. However, it is not adequate since there can be exactly $\binom{n}{m}$ traces⁴ for a text and a pattern of lengths n and m . Instead, we employ *minimal occurrence intervals* as occurrences defined as follows.

► **Definition 1** (minimal occurrence interval). For a text $T[1..n]$ and $P[1..m]$, an interval $[\ell, r] \subseteq [n]$ is said to be an *occurrence interval for pattern P over text T* if $P \sqsubseteq T[\ell..r]$ holds. It is said to be *minimal* if there is no occurrence interval $[\ell', r']$ for P over T such that $[\ell', r'] \subsetneq [\ell, r]$.

► **Example 2.** Let text $T = (11, 3, 8, 6, 16, 19, 5, 15, 21, 24)$ and pattern $P = (9, 2, 17, 4, 13)$. The occurrence interval $[3, 9]$ for P over T is minimal since $I = (3, 4, 6, 8, 9)$ is a trace with $CT(T_I) = CT(P)$, and there is no other occurrence interval $[\ell, r] \subsetneq [3, 9]$ for P over T . The interval $[1, 8]$ is an occurrence interval, however, it is not minimal since there is another (minimal) occurrence interval $[1, 5] \subsetneq [1, 8]$ for P over T . Overall, all minimal occurrence intervals for P over T are $[1, 5]$ and $[3, 9]$.

From the definition, there are $O(n^2)$ occurrence intervals for P over T , while there are $O(n)$ minimal occurrence intervals. If we have the set of all minimal occurrence intervals, we can easily enumerate all occurrence intervals in constant time per occurrence interval. Thus, we focus on minimal occurrences in this paper. Now, the main problem of this paper is formalized as follows:

► **Definition 3** (Cartesian Tree Subsequence Matching (CTMSeq)). Given two strings $T[1..n]$ and $P[1..m]$, find all minimal occurrence intervals for P over T .

We can easily see that CTMSeq can be solved in $O(m\binom{n}{m})$ time by simply enumerating all possible subscript sequences. However, its time complexities are too large to apply to real-world data sets. Hence, our goal here is to devise efficient algorithms running in polynomial time.

In the rest of this paper, we fix text T of arbitrary length n and pattern P of arbitrary length m with $0 < m \leq n$.

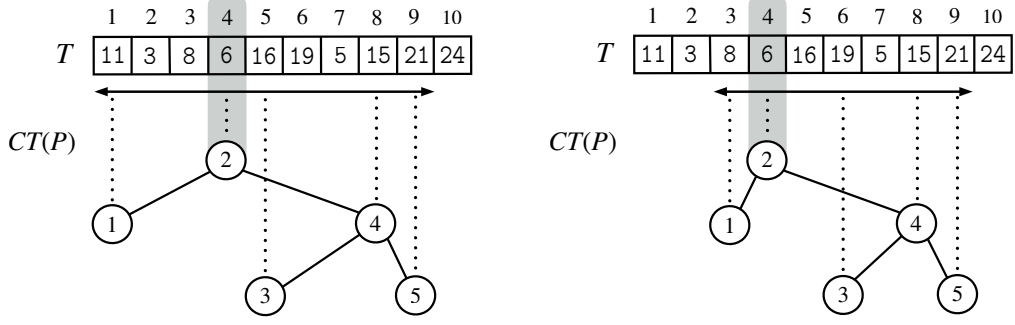
3 $O(mn^2)$ -time Dynamic Programming Algorithm

This section describes an algorithm based on dynamic programming which runs in time $O(mn^2)$. We later improve the running time to $O(mn \log \log n)$ in Section 4.

3.1 A Simple Algorithm

By dynamic programming approach, we can obtain a simple algorithm for CTMSeq with $O(mn^3)$ time and $O(mn^2)$ space complexities as follows. It recursively decides if the substring P_v matches in $T[\ell..r]$ for all indices v of P and all intervals $[\ell..r]$ in T from shorter to larger. These complexities mainly come from that it iterates the loop for $O(n^2)$ possible intervals in T . In the following section, we devise more efficient algorithms in time and space complexities by introducing the notion of *minimal fixed-intervals*.

⁴ which can be achieved by monotone sequences for P and T .



■ **Figure 2** Illustration for fixed-intervals for the pivot (v, i) , where $T = (11, 3, 8, 6, 16, 19, 5, 15, 21, 24)$, $P = (9, 2, 17, 4, 13)$, and $(v, i) = (2, 4)$. In the left figure, for the trace $I = (1, 4, 5, 8, 9)$ indicated by dotted lines, the interval $[1, 9]$ is a fixed-interval with the pivot (v, i) . In the right figure, $[3, 9]$ is a minimal fixed-interval with the pivot (v, i) since there is no fixed-interval $[\ell, r] \subsetneq [3, 9]$ with the pivot (v, i) .

3.2 Minimal Fixed-interval

To solve CTMSeq without iterating for all possible intervals, we focus on fixing the corresponding locations between node v of $CT(P)$ and index i of T . For a node $v \in [m]$ and index $i \in [n]$, we refer to a pair (v, i) as a *pivot*. Then, we define the minimal interval fixed with pivot (v, i) , called the *minimal fixed-interval*.

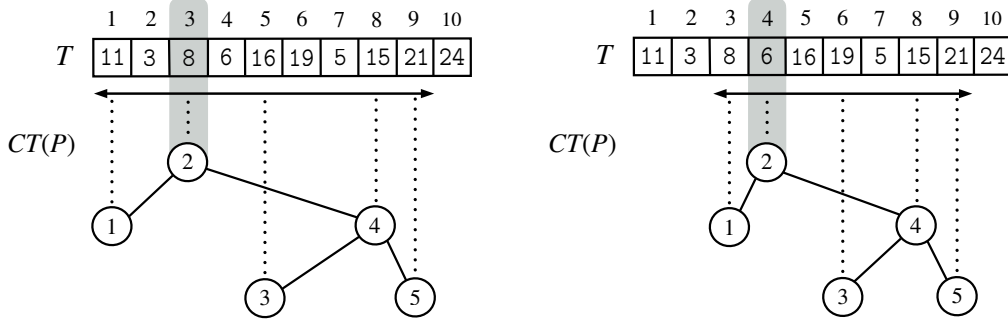
► **Definition 4** ((minimal) fixed-interval). For pivot $(v, i) \in [m] \times [n]$, interval $[\ell, r] \subseteq [n]$ is called a *fixed-interval with the pivot (v, i)* if there exists a trace $I = (i_1, \dots, i_{|P_v|}) \in \mathcal{I}_{|P_v|}^n$ satisfying the following conditions (i)–(iv): (i) i is an element of I , (ii) $[i_1, i_{|P_v|}] \subseteq [\ell, r]$ (iii) $CT(T_I) = CT(P_v)$ holds, and (iv) $T[i] = \min(T_I)$ holds. Furthermore, a fixed-interval $[\ell, r]$ with the pivot (v, i) is said to be *minimal* if there is no fixed-interval $[\ell', r'] \subsetneq [\ell, r]$ with the pivot (v, i) .

We show examples of (minimal) fixed-intervals on Figure 2. Here, we give an essential lemma concerning minimal fixed-intervals.

► **Lemma 5.** *For any pivot $(v, i) \in [m] \times [n]$, there exists at most one minimal fixed-interval with (v, i) .*

Proof. Assume that there are two minimal fixed-intervals with the pivot (v, i) . Let $[\ell, r]$ and $[\ell', r']$ be two such distinct intervals. Without loss of generality, assume $\ell \leq \ell'$. Then, by the minimalities of $[\ell, r]$ and $[\ell', r']$, $\ell < \ell'$ and $r < r'$ must hold. From Definition 4, there exist $I = (\ell, \dots, i, \dots, r)$ and $I' = (\ell', \dots, i, \dots, r')$ such that $CT(T_I) = CT(T_{I'}) = CT(P_v)$ and $T[i] = \min(T_I) = \min(T_{I'})$. Since $CT(T_I) = CT(T_{I'})$ and $T[i] = \min(T_I) = \min(T_{I'})$, the right subtree of i in $CT(T_I)$ is the same as that of $CT(T_{I'})$. Namely, $CT(T_{I_{[i+1:r]}}) = CT(T_{I'_{[i+1:r']}})$ holds. Thus, we have $CT(T_{I''}) = CT(P_v)$ where I'' is the subscript sequence of length $|I|$ that is the concatenation of $I[\ell' : i]$ and $I'[i+1 : r]$. Also, $i \in I''$ and $T[i] = \min(T_{I''})$ hold, and hence, $[\ell', r]$ is a fixed-interval with the pivot (v, i) . This contradicts that $[\ell', r']$ is a minimal fixed-interval. ◀

For convenience, we define the minimal fixed-interval with the pivot (v, i) as $[-\infty, \infty]$ if there is no fixed-interval with the pivot (v, i) . We denote by $\text{mfi}(v, i)$ the minimal fixed-interval with the pivot (v, i) . Let $\mathcal{M} = \{\text{mfi}(\text{minidx}(P), i) \mid i \in [n]\}$ be the set of all the minimal fixed-intervals for the root of $CT(P)$. By the definitions of minimal occurrence intervals and minimal fixed-intervals, the next corollary holds:



■ **Figure 3** Illustration for two minimal fixed-intervals, where T and P are the same as in Figure 2. From the figure, $\text{mfi}(2, 3) = [1, 9]$ and $\text{mfi}(2, 4) = [3, 9]$ hold. Note that $\text{mfi}(2, 3) = [1, 9]$ is not a solution of CTMSeq since $\text{mfi}(2, 4) \subsetneq \text{mfi}(2, 3)$ holds.

► **Corollary 6.** For any minimal occurrence interval $[\ell, r]$ for P over T , $[\ell, r] \in \mathcal{M}$ holds. Contrary, for any interval $[\ell, r] \in \mathcal{M}$, if there is no interval $[\ell', r'] \subsetneq [\ell, r]$ such that $[\ell', r'] \in \mathcal{M}$, $[\ell, r]$ is a minimal occurrence interval for P over T .

Note that not every intervals $[\ell, r] \in \mathcal{M}$ is a minimal occurrence interval for P over T . We show an example of a interval $[\ell, r] \in \mathcal{M}$ such that $[\ell, r]$ is not a solution of CTMSeq in Figure 3.

3.3 The Algorithm

From Corollary 6, once we compute the set \mathcal{M} of intervals, we can obtain the solution of CTMSeq by removing non-minimal intervals from \mathcal{M} . Since every interval in \mathcal{M} except $[-\infty, \infty]$ is a sub-interval of $[1, n]$, we can sort them in $O(n)$ time by using bucket sort, and thus, can also remove non-minimal intervals.

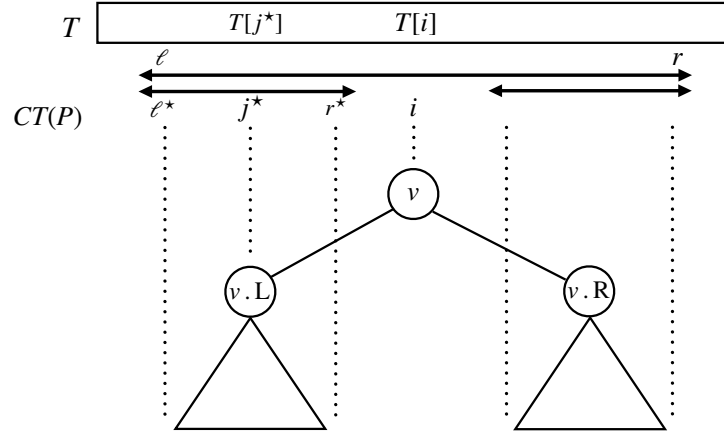
Thus, in what follows, we discuss how to efficiently compute \mathcal{M} , i.e., $\text{mfi}(\text{minidx}(P), i)$ for all $i \in [n]$. Now, we define two functions $L(v, i) = \ell$ and $R(v, i) = r$ for each node $v \in [m]$ in $CT(P)$ and each index $i \in [n]$, where $[\ell, r] = \text{mfi}(v, i)$. Then, our task is, to compute $L(\text{minidx}(P), i)$ and $R(\text{minidx}(P), i)$ for all $i \in [n]$. Regarding the two functions, we show the following lemma (see also Figure 4 for illustration):

► **Lemma 7.** For any pivot $(v, i) \in [m] \times [n]$, the following recurrence relations hold:

$$L(v, i) = \begin{cases} -\infty & \text{if } \text{mfi}(v, i) = [-\infty, \infty], \\ i & \text{if } \text{mfi}(v, i) \neq [-\infty, \infty] \\ & \text{and } v.L = \text{nil}, \\ \max_{1 \leq j \leq i-1} \{L(v.L, j) \mid T[i] < T[j], R(v.L, j) < i\} & \text{otherwise.} \end{cases}$$

$$R(v, i) = \begin{cases} \infty & \text{if } \text{mfi}(v, i) = [-\infty, \infty], \\ i & \text{if } \text{mfi}(v, i) \neq [-\infty, \infty] \\ & \text{and } v.R = \text{nil}, \\ \min_{i+1 \leq j \leq n} \{R(v.R, j) \mid T[i] < T[j], i < L(v.R, j)\} & \text{otherwise.} \end{cases}$$

Proof. We prove the validity of the first equation for $L(v, i)$. The second one can be proven by symmetric arguments. The first two cases are clearly correct by the definition of minimal fixed-intervals. We focus on the third case, when $\text{mfi}(v, i) \neq [-\infty, \infty]$ and $v.L \neq \text{nil}$.



■ **Figure 4** Illustration for intuitive understanding of recurrence relations in Lemma 7. The minimal fixed-interval $[\ell, r]$ with the pivot (v, i) can be obtained from $\text{mfi}(v.L, j)$ and $\text{mfi}(v.R, k)$ by choosing j and k appropriately. As for the left subtree of v , the candidates for such j must satisfy the conditions that the right-end of $\text{mfi}(v.L, j)$ does not exceed $i - 1$ and $T[j] > T[i]$. To minimize the width of fixed-intervals with (v, i) , we choose j^* that maximizes the left-end of $\text{mfi}(v.L, j^*)$ while satisfying the above conditions. Also, symmetric arguments can be applied to the right subtree of v .

Let $[\ell, r] = \text{mfi}(v, i)$. By Definition 4, there exists $I = (\ell, \dots, i, \dots, r)$ such that $CT(T_I) = CT(P_v)$ and $T[i] = \min(T_I)$. We notice that $CT(P_{v.L}) = CT(T_{I[\ell:pre_i]})$ holds where pre_i is the subscript preceding i in I . Thus, there exists k such that $\ell \leq k \leq i - 1$, $T[i] < T[k]$, $R(v.L, k) < i$, and $L(v.L, k) \geq \ell$. Now, let $j^* := \arg \max_{1 \leq j \leq i-1} \{L(v.L, j) \mid T[i] < T[j], R(v.L, j) < i\}$ and $[\ell^*, r^*] := \text{mfi}(v.L, j^*)$. Then, $\ell^* = L(v.L, j^*) \geq L(v.L, k) \geq \ell$ holds.

For the sake of contradiction, we assume $\ell < \ell^*$. By Definition 4, there exists $I^* = (\ell^*, \dots, j^*, \dots, r^*)$ such that $CT(T_{I^*}) = CT(P_{v.L})$ and $T[j^*] = \min(T_{I^*})$. Also, by the definition of j^* , $T[i] < T[j^*]$ and $r^* < i$ hold. Let I' be the concatenation of I^* and $I[i:r]$. Note that $I' \in \mathcal{I}_m^n$ since $r^* < i$. From the above discussions, $CT(T_{I'}) = CT(P_v)$ holds since $CT(T_{I^*}) = CT(P_{v.L})$ and $\min(T_{I'}) = T[j^*] > T[i]$. Also, $i \in I'$ and $T[i] = \min(T_{I'})$ clearly hold. Then, by Definition 4, $[\ell^*, r] \subsetneq [\ell, r]$ is a fixed-interval with (v, i) , however, this contradicts the minimality of $[\ell, r] = \text{mfi}(v, i)$. Therefore, $\ell = \ell^*$ holds. Namely, $L(v, i) = L(v.L, j^*) = \max_{1 \leq j \leq i-1} \{L(v.L, j) \mid T[i] < T[j], R(v.L, j) < i\}$ holds. ◀

Algorithm 1 is a pseudo code of our algorithm to solve CTMSeg using dynamic programming based on Lemma 7.

Correctness of Algorithm 1

Algorithm 1 computes tables $L[v][i] = L(v, i)$ and $R[v][i] = R(v, i)$ for all pivot $(v, i) \in [m] \times [n]$ in a *bottom-up* manner in $CT(P)$ (see Line 5). Since the recursion formulae of Lemma 7 hold for every node, Algorithm 1 correctly computes all the minimal fixed-intervals, and thus, all the minimal occurrence intervals for pattern P over text T .

Time and Space Complexities of Algorithm 1

At Line 4, we build the Cartesian tree C of a given pattern P . There is a linear-time algorithm to build a Cartesian tree [9], which takes $O(m)$ time here. In Lines 5–7, we call functions UPDATE-LEFT-MAX and UPDATE-RIGHT-MIN m times since C has m nodes.

■ **Algorithm 1** Algorithm for solving CTMSeq using dynamic programming.

```

1: procedure CARTESIAN-TREE-SUBSEQUENCE-MATCH( $T[1..n], P[1..m]$ )
2:    $L[v][i] \leftarrow -\infty$  for all  $v \in [m]$  and  $i \in [n]$ 
3:    $R[v][i] \leftarrow \infty$  for all  $v \in [m]$  and  $i \in [n]$ 
4:    $C \leftarrow CT(P)$ 
5:   for each  $v \in [m]$  in a bottom-up manner in  $C$  do
6:     call UPDATE-LEFT-MAX( $v, T, L, R$ )
7:     call UPDATE-RIGHT-MIN( $v, T, L, R$ )
8:   enumerate all minimal occurrence intervals for  $P$  over  $T$  by using  $L$  and  $R$ .
9: function UPDATE-LEFT-MAX( $v, T, L, R$ )
10:  if  $v.L = nil$  then
11:     $L[v][i] \leftarrow i$  for all  $i \in [n]$ 
12:  return
13:  for  $i \leftarrow 1$  to  $n$  do
14:    for  $j \leftarrow 1$  to  $i - 1$  do
15:      if  $T[i] < T[j]$  and  $R[v.L][j] < i$  then
16:         $L[v][i] \leftarrow \max(L[v][i], L[v.L][j])$ 
17: function UPDATE-RIGHT-MIN( $v, T, L, R$ )
18:  if  $v.R = nil$  then
19:     $R[v][i] \leftarrow i$  for all  $i \in [n]$ 
20:  return
21:  for  $i \leftarrow 1$  to  $n$  do
22:    for  $j \leftarrow i + 1$  to  $n$  do
23:      if  $T[i] < T[j]$  and  $i < L[v.R][j]$  then
24:         $R[v][i] \leftarrow \min(R[v][i], R[v.R][j])$ 

```

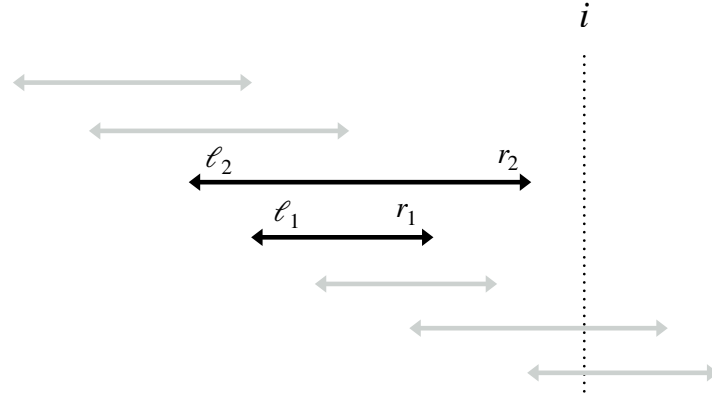
It is clear that the functions UPDATE-LEFT-MAX and UPDATE-RIGHT-MIN run in $O(n^2)$ time for each call. Thus, the total running time of Algorithm 1 is $O(mn^2)$. Also, the space complexity of Algorithm 1 is $O(mn)$, which is dominated by the size of tables L and R .

To summarize, we obtain the following theorem:

► **Theorem 8.** *The CTMSeq problem can be solved in $O(mn^2)$ time using $O(mn)$ space.*

With a few modifications, we can reconstruct a trace $I = (\ell, \dots, r) \in \mathcal{I}_m^n$ satisfying $CT(T_I) = CT(P)$ for each minimal occurrence interval $[\ell, r]$. Precisely, when we compute the minimal fixed-interval with each pivot (v, i) , we simultaneously compute and store which index will correspond to the root of the left subtree of v fixed at i . We do the same for the right subtree. Using the additional information, we can reconstruct a desired subscript sequence by tracing back from the root of $CT(P)$. The next corollary follows from the above discussion:

► **Corollary 9.** *Once we compute $L(v, i)$ and $R(v, i)$ extended with the information of tracing back for all pivots $(v, i) \in [m] \times [n]$, we can find a trace $I = (\ell, \dots, r)$ satisfying $CT(T_I) = CT(P)$ for each minimal occurrence interval $[\ell, r]$ for P over T in $O(m)$ time using $O(mn)$ space.*



■ **Figure 5** Illustration for the third observation for $LFI(v, i)$. The double-headed arrows represent the intervals in $LFI(v, i)$. The two intervals $[\ell_1, r_1]$ and $[\ell_2, r_2]$ are in $LFI(v, i)$ and $[\ell_1, r_1] \subsetneq [\ell_2, r_2]$ holds. It is clear that ℓ_2 is never chosen as $L(v, i)$ for any $i \in [n]$.

4 Reducing Time to $O(mn \log \log n)$ with Predecessor Dictionaries

This section describes how to improve the time complexity of Algorithm 1 to $O(mn \log \log n)$. In Algorithm 1, functions UPDATE-LEFT-MAX and UPDATE-RIGHT-MIN require $O(n^2)$ time for each call, which is a bottle-neck of Algorithm 1. By devising the update order of tables $L(v, i)$ and $R(v, i)$ and using a predecessor dictionary, we improve the running time of the above two functions to $O(n \log \log n)$.

4.1 Main Idea for Reducing Time

For any pivot $(v, i) \in [m] \times [n]$, let $LFI(v, i) = \{[L(v.L, j), R(v.L, j)] \mid 1 \leq j \leq n, T[i] < T[j]\}$ be a set of intervals which are candidates for a component of the minimal fixed-interval with (v, i) . By Lemma 7, $L(v, i) = \max(\{\ell \mid [\ell, r] \in LFI(v, i), r < i\} \cup \{-\infty\})$ holds if $v.L \neq nil$. Then, the next observations follow by the definitions:

- $LFI(v, i_1) \subseteq LFI(v, i_2)$ holds for any i_1, i_2 with $T[i_1] > T[i_2]$.
- If there are intervals $[\ell_1, r_1], [\ell_2, r_2] \in LFI(v, i)$ such that $\ell_2 = \ell_1 \leq r_1 < r_2$, then we can always choose ℓ_1 as $L(v, i)$.
- If there are intervals $[\ell_1, r_1], [\ell_2, r_2] \in LFI(v, i)$ such that $\ell_2 < \ell_1 \leq r_1 \leq r_2$, then ℓ_2 is never chosen as $L(v, i)$.

The intuitive explanation of the third observation is shown in Figure 5. From the third observation, we define a subset $LFI'(v, i)$ of $LFI(v, i)$, whose conditions are sufficient to our purpose: Let $LFI'(v, i)$ be the set of all intervals that are minimal within $LFI(v, i)$. Namely, $LFI'(v, i) = \{[\ell, r] \in LFI(v, i) \mid \text{there is no other interval } [\ell', r'] \in LFI(v, i) \text{ such that } [\ell', r'] \subsetneq [\ell, r]\}$. By the third observation,

$$L(v, i) = \max(\{\ell \mid [\ell, r] \in LFI'(v, i), r < i\} \cup \{-\infty\}) \quad (1)$$

holds if $v.L \neq nil$.

The main idea of our algorithm is to maintain a set \mathcal{S}_v of intervals so that it satisfies the invariant $\mathcal{S}_v = LFI'(v, i)$. To maintain \mathcal{S}_v efficiently, we utilize a data structure called *predecessor dictionary* for \mathcal{S}_v supporting the following operations:

- **insert**(\mathcal{S}_v, ℓ, r): insert interval $[\ell, r]$ into \mathcal{S}_v ,
- **delete**(\mathcal{S}_v, ℓ, r): delete interval $[\ell, r]$ from \mathcal{S}_v ,

■ **Algorithm 2** Faster algorithm for UPDATE-LEFT-MAX using van Emde Boas tree.

```

1: function UPDATE-LEFT-MAX( $v, T, L, R$ )
2:   if  $v.L = nil$  then
3:      $L[v][i] \leftarrow i$  for all  $i \in [n]$ 
4:   return
5:    $\mathcal{S}_v \leftarrow \emptyset$ .
6:   for each  $i \in [n]$  in the descending order of its value  $T[i]$  do
7:      $[\ell, r] \leftarrow \text{pred}(\mathcal{S}_v, i)$ 
8:     if  $[\ell, r] = nil$  then
9:        $L[v][i] \leftarrow -\infty$ 
10:    continue
11:     $L[v][i] \leftarrow \ell$ 
12:     $\ell_{new} \leftarrow L[v.L][i], r_{new} \leftarrow R[v.L][i]$ 
13:    loop ▷ delete all intervals that become non-minimal
14:       $[\ell_s, r_s] \leftarrow \text{succ}(\mathcal{S}_v, r_{new} - 1)$ 
15:      if  $[\ell_s, r_s] = nil$  or  $[\ell_{new}, r_{new}] \not\subseteq [\ell_s, r_s]$  then
16:        break
17:       $\text{delete}(\mathcal{S}_v, \ell_s, r_s)$ 
18:       $[\ell_p, r_p] \leftarrow \text{pred}(\mathcal{S}_v, r_{new} + 1)$ 
19:      if  $[\ell_p, r_p] = nil$  or  $[\ell_p, r_p] \not\subseteq [\ell_{new}, r_{new}]$  then ▷ insert new interval if it is minimal
20:         $\text{insert}(\mathcal{S}_v, \ell_{new}, r_{new})$ 

```

- $\text{pred}(\mathcal{S}_v, x)$: return the interval $[\ell, r] \in \mathcal{S}_v$ on which r is the largest among those satisfying $r < x$ (if it does not exist return *nil*), and
- $\text{succ}(\mathcal{S}_v, x)$: return the interval $[\ell, r] \in \mathcal{S}_v$ on which r is the smallest among those satisfying $x < r$ (if it does not exist return *nil*).

To implement a predecessor dictionary for \mathcal{S}_v , we use a famous data structure called *van Emde Boas tree* [22] that performs the operations as mentioned above in $O(\log \log n)$ time each⁵. In general, the space usage of van Emde Boas tree is $O(U)$, where U is the maximum of the integers to store. However, $U = n$ holds in our problem setting, and hence, the space complexity is $O(n)$.

4.2 Faster Algorithm

Algorithm 2 shows a function UPDATE-LEFT-MAX that computes $L(v, i)$ for all $i \in [n]$ based on the above idea. This function can be used to replace the function of the same name in Algorithm 1. The implementation of function UPDATE-RIGHT-MIN is symmetric.

Correctness of Algorithm 2

Remark that v is fixed in Algorithm 2. Let (i_1, \dots, i_n) be the permutation of $[n]$ that is sorted in the order in which they are picked up by the for-loop at Line 6. We assume that the invariant $\mathcal{S}_v = LFT'(v, i_j)$ holds at the beginning of the j -th step of the for-loop. The value of $L[v][i_j]$ is determined at either Line 3, 9, or 11. By Lemma 7, $L[v][i_j] = L(v, i_j)$ holds

⁵ The van Emde Boas tree is a data structure for the set of integers, however, it can be easily applied to the set of pairs of integers by associating the first element with the second element.

14:12 Cartesian Tree Subsequence Matching

if the value determined at Line 3 or 9. By the invariant $\mathcal{S}_v = LFI'(v, i_j)$ and Equation 1, $L[v][i_j] = L(v, i_j)$ also holds if the value determined at Line 11. Thus, $L(v, i_j)$ is computed correctly.

Next, let us consider the invariant for \mathcal{S}_v . At Line 12, we set $[\ell_{new}, r_{new}]$ the minimal fixed-interval with $(v.L, i_j)$. In the internal loop at Lines 13–17, we delete all intervals $[\ell_s, r_s]$ from \mathcal{S}_v such that $[\ell_s, r_s]$ becomes *non-minimal* within $\mathcal{S}_v \cup \{[\ell_{new}, r_{new}]\}$. To do so, we repeatedly query $\text{succ}(\mathcal{S}_v, r_{new} - 1)$ and check whether the obtained interval includes $[\ell_{new}, r_{new}]$. Finally, at the last two lines, we insert the new interval $[\ell_{new}, r_{new}]$ if it does not include any other interval in \mathcal{S}_v . Then, any intervals in \mathcal{S}_v are not nested each other, and thus, the invariant $\mathcal{S}_v = LFI'(v, i_{j+1})$ holds at the end of the j -th step.

Time and Space Complexities of Algorithm 2

We analyze the number of calls for each operation on a predecessor dictionary. Firstly, since `insert` is called only at Line 20, it is called at most n times throughout Algorithm 2. Similarly, `pred` at Line 7 and Line 18 is also called $O(n)$ times. From Line 13 to Line 17, `succ` and `delete` are called in the internal loop. The number of calls for `delete` is at most that of `insert`, and hence, `delete` is called at most n times, and `succ` as well. Thus, throughout Algorithm 2, the total number of calls for all queries is $O(n)$. Therefore, the running time of Algorithm 2 is $O(n \log \log n)$. Also, the space complexity of Algorithm 2 is $O(n)$.

To summarize this section, we obtain the following lemma:

► **Lemma 10.** *Algorithm 2 computes function UPDATE-LEFT-MAX in $O(n \log \log n)$ time using $O(n)$ space.*

5 Reducing Space to $O(n \log m)$

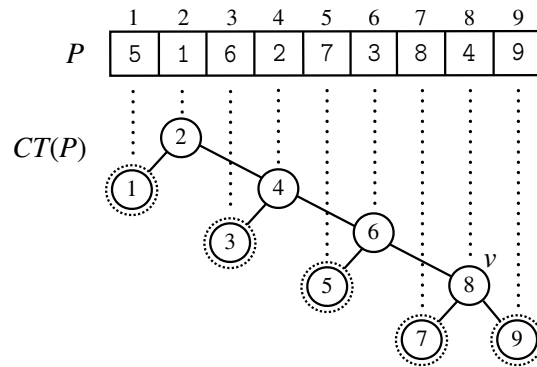
This section describes how to reduce the space complexity of our algorithm to $O(n \log m)$. Having the tables $L[v][i]$ and $R[v][i]$ for all pivot $(v, i) \in [m] \times [n]$ requires $\Theta(mn)$ space. By Lemma 7, to compute the table values for node $v \in CT(P)$, we only need the table values for $v.L$ and $v.R$. Thus, we can discard the remaining values no longer referenced. However, even if we discard such unnecessary ones, the space complexity will not be improved in the worst case if we fix the order in which subtree is visited first: Let us assume that the left subtree is always visited first, and consider pattern

$$P = (k + 1, 1, \dots, k + i, i, \dots, 2k, k, 2k + 1) \quad (2)$$

of length $m = 2k + 1$. It can be seen that every non-leaf node in $CT(P)$ has exactly two children, and the left child is a leaf (see also Figure 6 for a concrete example). Thus, when we process the node v numbered with $2k$, we need to store at least $k + 1$ tables since all tables for $k + 1$ leaves have been created and not been discarded yet, and it yields $\Theta(mn)$ space.

To avoid such a case, we add a new rule for which subtree is visited first; when we perform a depth-first traversal, we visit the *larger* subtree first if the current node v has two children. Specifically, we visit the left subtree first if $|CT(P)_{v.L}| > |CT(P)_{v.R}|$, and visit the right subtree first otherwise, where the cardinality of a tree means the number of nodes in the tree. Clearly, the correctness of the modified algorithm relies on the original one (i.e., Algorithm 1) since the only difference is the rule that decides the order to visit.

In the following, we show that the rule makes the space complexity $O(n \log m)$. We utilize a technique called *heavy-path decomposition* [11] (a.k.a. heavy-light decomposition). For each internal node $v \in [m]$ in $CT(P)$, we choose one of v 's children with the larger subtree



■ **Figure 6** Illustration for a worst case example of $CT(P)$ which causes the space complexity to be $\Theta(mn)$, where $P = (5, 1, 6, 2, 7, 3, 8, 4, 9)$. To compute the values of $L(v, i)$ and $R(v, i)$ for $v = 8$, we only need the values of $L(u, i)$ and $R(u, i)$ for node $u \in \{7, 9\}$. However, we have not finished computing $L(u, i)$ and $R(u, i)$ for node $u \in \{2, 4, 6\}$ yet, so we have to remember all of the values of $L(u, i)$ and $R(u, i)$ for node $u \in \{1, 3, 5, 7, 9\}$ simultaneously.

size and mark it as *heavy*, and we mark the other one as *light* if it exists. Exceptionally, we mark the root of $CT(P)$ as *heavy*. Then, it is known that the number of light nodes on any root-to-leaf path is $O(\log m)$ [11].

Now, we prove that the algorithm requires $O(n \log m)$ space at any step. Suppose we are now on node $u \in [m]$. Let \mathbf{p}_u be the path from the root to u in $CT(P)$. Note that each node v on \mathbf{p}_u is marked as either *heavy* or *light*. For each light node v_ℓ on \mathbf{p}_u , we have not discarded arrays L and R of size $O(n)$ associated with the sibling of v_ℓ to process the parent of v_ℓ in a later step. For each heavy node v_h on \mathbf{p}_u , we do not have to remember any array since we recurse on v_h first, and hence we require only $O(1)$ space for v_h . Since there are at most $O(\log m)$ light nodes on \mathbf{p}_u , the algorithm requires $O(n \log m)$ space at any step.

By combining these discussion with Theorem 8 and Lemma 10, we obtain our main theorem:

► **Theorem 11.** *The CTMSeg problem can be solved in $O(mn \log \log n)$ time using $O(n \log m)$ space.*

Note that the same method as for Corollary 9 can not be applied to the algorithm in this section since most tables are discarded to save space.

6 Preliminary Experiments

This section aims to investigate the behavior of each algorithm using artificial data. In the first experiment we use randomly generated strings to see how the algorithms would behave on average (Table 1). In the second experiment, we use the worst-case instance presented in Section 5 to check the worst-case behavior of the proposed algorithms (Table 2).

We conducted experiments on mac OS Mojave 10.14.6 with Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz. For each test, we use a single thread and limit the maximum run time by 60 minutes. All programs are implemented using C++ language compiled with Apple LLVM version 10.0.1 (clang-1001.0.46.4) with $-O3$ optimization option. We compared the running time and memory usage of our four proposed algorithms below by varying the length n of text and the length m of pattern:

■ **Table 1** Comparison of four algorithms for solving CTMSeq with randomly generated texts and patterns. The unit of time is second, and the unit of space is KB.

		basic		basic-HL		vEB		vEB-HL	
n	m	time	space	time	space	time	space	time	space
5000	50	2.03	1980	0.09	3148	0.03	2496	0.03	2124
5000	500	19.20	2788	19.86	2168	0.37	3272	0.37	2596
5000	1000	40.62	2932	40.34	2236	0.73	3520	0.73	2604
5000	2500	96.27	3124	96.23	2368	1.84	3532	1.84	2816
10000	50	7.77	2128	7.74	1804	0.07	2504	0.07	2188
10000	1000	159.82	2740	159.70	1960	1.38	3128	1.38	2352
10000	2000	321.07	2920	323.09	2068	3.08	3312	3.09	2452
10000	5000	841.85	3252	835.29	2212	7.22	3644	7.23	2592
50000	50	206.49	4976	211.24	3836	0.39	6076	0.40	4920
50000	5000	NA	NA	NA	NA	39.98	13040	39.70	6576
50000	10000	NA	NA	NA	NA	79.42	12684	80.20	7044
50000	25000	NA	NA	NA	NA	199.14	13900	197.71	7340

- **basic**: $O(mn^2)$ -time and $O(mn)$ -space algorithm (Algorithm 1) explained in Section 3,
- **basic-HL**: $O(mn^2)$ -time and $O(n \log m)$ -space algorithm obtained by applying the idea of memory reduction in Section 5 to **basic**.
- **vEB**⁶: $O(mn \log \log n)$ -time and $O(mn)$ -space algorithm obtained by combining Algorithm 1 in Section 3 with Algorithm 2 in Section 4, and
- **vEB-HL**: $O(mn \log \log n)$ -time and $O(n \log m)$ -space algorithm obtained by applying the idea of memory reduction in Section 5 to **vEB**.

Tables 1 and 2 show the comparison of the performance among four algorithms above. NA indicates that the measurement was terminated when the execution time exceeded 60 minutes. Common to both Table 1 and Table 2, we use a text T of length n that is a randomly chosen permutation of $(1, 2, \dots, n)$, and thus, T is a length- n string over the alphabet $\{1, 2, \dots, n\}$. In Table 1, we use a pattern P that is a randomly chosen subsequence of T , and thus, P is also a length- m string over the alphabet $\{1, 2, \dots, n\}$. In Table 2, we use the pattern $P = (k+1, 1, \dots, k+i, i, \dots, 2k, k)$ of length $m = 2k$ in Equation 2 (see also Figure 6), which requires $\Theta(mn)$ space when the idea of memory reduction in Section 5 is not applied.

Table 1 shows that the running time of **vEB** is faster than that of **basic** for all test cases, and the same result can be seen for **vEB-HL** and **basic-HL**. Comparing the memory usage of **vEB** with that of **basic**, it can be seen that the **vEB** uses more memory than **basic**, since the memory usage of the van Emde Boas tree is constant times larger than that of a basic array. The same is true for **vEB-HL** and **basic-HL**. The only difference between **basic** (**vEB**) and **basic-HL** (**vEB-HL**) is the search order of the tree traversal, so they have little difference in the running time for all test cases. Comparing these algorithms in terms of memory usage, it can be seen that the **basic-HL** (**vEB-HL**) uses less memory than **basic** (**vEB**), but the difference is not as pronounced as the theoretical difference in the space complexity. This is because P is generated at random, so there is not much bias in the size of the subtrees.

⁶ For the implementation of van Emde Boas trees, we used the following library: https://kopricky.github.io/code/Academic/van_emde_boas_tree.html

■ **Table 2** Comparison of four algorithms for solving CTMSeq with randomly generated texts and intentionally generated patterns of form $P = (k + 1, 1, \dots, k + i, i, \dots, 2k, k)$ in Equation 2. The unit of time is second, and the unit of space is KB.

		basic		basic-HL		vEB		vEB-HL	
n	m	time	space	time	space	time	space	time	space
5000	50	1.85	2572	1.86	1940	0.03	2920	0.03	2208
5000	500	18.01	11712	18.03	1912	0.23	12064	0.23	2372
5000	1000	37.65	21804	37.94	2028	0.41	22236	0.40	2516
5000	2500	92.58	52036	89.04	2220	0.96	52720	0.94	2960
10000	50	7.39	3444	7.45	1644	0.07	3748	0.07	2032
10000	1000	150.70	41632	153.18	1732	0.80	42192	0.79	2304
10000	2000	301.57	81856	303.77	1852	1.49	82584	1.46	2600
10000	5000	754.85	202408	759.71	2244	3.58	203656	3.49	3512
50000	50	186.05	12024	186.63	3048	0.37	13116	0.37	4140
50000	5000	NA	NA	NA	NA	18.36	650768	17.82	5616
50000	10000	NA	NA	NA	NA	35.42	963068	34.25	7112
50000	25000	NA	NA	NA	NA	87.28	998056	83.94	11600

On the other hand, the results in Table 2 show that **basic-HL** and **vEB-HL** are significantly more memory efficient than **basic** and **vEB** in the case where m is large. This is consistent with the theoretical difference in the amount of the space complexity.

We also conducted the additional experiments with other algorithms:

- **BST**: $O(mn \log n)$ -time and $O(mn)$ -space algorithm using the *binary search tree*⁷ instead of van Emde Boas tree in Section 4, and
- **BST-HL**: $O(mn \log n)$ -time and $O(mn)$ -space algorithm obtained by applying the idea of memory reduction in Section 5 to **BST**.

vEB outperformed **BST** in both time and space for all test cases, and so do **vEB-HL** and **BST-HL**, which we feel is of independent interest. The details of the results are shown in Appendix A.

7 Conclusions

This paper introduced the Cartesian tree subsequence matching (CTMSeq) problem: Given a text T of length n and a pattern P of length m , find every minimal substring S of T such that S contains a subsequence S' which Cartesian-tree matches P . This is the Cartesian-tree version of the episode matching [7]. We first presented a basic dynamic programming algorithm running in $O(mn^2)$ time, and then proposed a faster $O(mn \log \log n)$ -time solution to the problem. We showed how these algorithms can be performed with $O(n \log m)$ space. Our experiments showed that our $O(mn \log \log n)$ -time solution can be fast in practice.

An intriguing open problem is to show a non-trivial (conditional) lower bound for the CTMSeq problem. The episode matching (under the exact matching criterion) has $O((mn)^{1-\epsilon})$ -time conditional lower bound under SETH [3]. Although a solution to the CTMSeq problem that is significantly faster than $O(mn)$ seems unlikely, we have not found such a (conditional) lower bound yet. We remark that the episode matching problem is not readily reducible to the CTMSeq problem, since CTMSeq allows for more relaxed pattern matching and the reported intervals can be shorter than those found by episode matching.

⁷ For the implementation of binary search trees, we used `std::set` in C++.

References

- 1 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80. ACM, 1993. doi:10.1145/167088.167115.
- 2 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996. doi:10.1006/jcss.1996.0003.
- 3 Philip Bille, Inge Li Gørtz, Shay Mozes, Teresa Anna Steiner, and Oren Weimann. A conditional lower bound for episode matching. *CoRR*, abs/2108.08613, 2021.
- 4 Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998. doi:10.1016/S0020-0190(97)00209-3.
- 5 Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.*, 115(2):397–402, 2015. doi:10.1016/j.ipl.2014.10.018.
- 6 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.
- 7 Gautam Das, Rudolf Fleischer, Leszek Gasieniec, Dimitrios Gunopulos, and Juha Kärkkäinen. Episode matching. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching, 8th Annual Symposium, CPM 97, Aarhus, Denmark, June 30 - July 2, 1997, Proceedings*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 1997. doi:10.1007/3-540-63220-4_46.
- 8 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The parameterized suffix tray. In Tiziana Calamoneri and Federico Corò, editors, *Algorithms and Complexity - 12th International Conference, CIAC 2021, Virtual Event, May 10-12, 2021, Proceedings*, volume 12701 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2021. doi:10.1007/978-3-030-75242-2_18.
- 9 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In Richard A. DeMillo, editor, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*, pages 135–143. ACM, 1984. doi:10.1145/800057.808675.
- 10 Pawel Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate strings matching. In *Proc. 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *LIPICs*, pages 14:1–14:14, 2020.
- 11 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 12 Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai. Order-preserving pattern matching indeterminate strings. In *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *LIPICs*, pages 2:1–2:15, 2018.
- 13 Christoph M. Hoffmann and Michael J. O’Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982. doi:10.1145/322290.322295.
- 14 Ramana M. Idury and Alejandro A. Schäffer. Multiple matching of parametrized patterns. *Theor. Comput. Sci.*, 154(2):203–224, 1996. doi:10.1016/0304-3975(94)00270-3.
- 15 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. On the longest common parameterized subsequence. *Theor. Comput. Sci.*, 410(51):5347–5353, 2009. doi:10.1016/j.tcs.2009.09.011.
- 16 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.
- 17 Marcin Kubica, Tomasz Kulczynski, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013. doi:10.1016/j.ipl.2013.03.015.

- 18 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. doi:10.1016/j.tcs.2016.02.017.
- 19 Juan Mendivelso, Sharma V. Thankachan, and Yoan J. Pinzón. A brief history of parameterized matching problems. *Discret. Appl. Math.*, 274:103–115, 2020. doi:10.1016/j.dam.2018.07.017.
- 20 Sung Gwan Park, Magsarjav Bataa, Amihood Amir, Gad M. Landau, and Kunsoo Park. Finding patterns and periods in Cartesian tree matching. *Theor. Comput. Sci.*, 845:181–197, 2020. doi:10.1016/j.tcs.2020.09.014.
- 21 Siwoo Song, Geonmo Gu, Cheol Ryu, Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast algorithms for single and multiple pattern Cartesian tree matching. *Theor. Comput. Sci.*, 849:47–63, 2021. doi:10.1016/j.tcs.2020.10.009.
- 22 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

A

 Additional Table

■ **Table 3** Comparison of six algorithms with additional two algorithms for solving CTMSeq with randomly generated texts and patterns. The unit of time is second, and the unit of space is KB.

n	m	basic		basic-HL		BST		BST-HL		vEB		vEB-HL	
		time	space	time	space	time	space	time	space	time	space	time	space
5000	50	2.03	1980	2.03	2020	0.09	3284	0.09	3148	0.03	2496	0.03	2124
5000	500	19.20	2788	19.86	2168	0.85	3896	0.83	3240	0.37	3272	0.37	2596
5000	1000	40.62	2932	40.34	2236	1.68	4084	1.67	3348	0.73	3520	0.73	2604
5000	2500	96.27	3124	96.23	2368	4.21	4396	4.18	3480	1.84	3532	1.84	2816
10000	50	7.77	2128	7.74	1804	0.20	4076	0.19	3360	0.07	2504	0.07	2188
10000	1000	159.82	2740	159.70	1960	3.70	4724	3.64	3940	1.38	3128	1.38	2352
10000	2000	321.07	2920	323.09	2068	8.25	4912	8.22	4048	3.08	3312	3.09	2452
10000	5000	841.85	3252	835.29	2212	20.25	5232	19.69	4196	7.22	3644	7.23	2592
50000	50	206.49	4976	211.24	3836	1.46	10204	1.45	10004	0.39	6076	0.40	4920
50000	5000	NA	NA	NA	NA	141.22	17276	136.76	10868	39.98	13040	39.70	6576
50000	10000	NA	NA	NA	NA	271.18	16920	272.29	11440	79.42	12684	80.20	7044
50000	25000	NA	NA	NA	NA	691.63	18144	689.80	11780	199.14	13900	197.71	7340

Polynomial-Time Equivalences and Refined Algorithms for Longest Common Subsequence Variants

Yuichi Asahiro ✉

Kyushu Sangyo University, Fukuoka, Japan

Jesper Jansson ✉

Kyoto University, Japan

Guohui Lin ✉

University of Alberta, Edmonton, Canada

Eiji Miyano ✉

Kyushu Institute of Technology, Iizuka, Japan

Hiroataka Ono ✉

Nagoya University, Japan

Tadatoshi Utashima ✉

Kyushu Institute of Technology, Iizuka, Japan

Abstract

The problem of computing the longest common subsequence of two sequences (LCS for short) is a classical and fundamental problem in computer science. In this paper, we study four variants of LCS: the REPETITION-BOUNDED LONGEST COMMON SUBSEQUENCE problem (RBLCS) [2], the MULTISSET-RESTRICTED COMMON SUBSEQUENCE problem (MRCS) [11], the TWO-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (2FLCS), and the ONE-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (1FLCS) [5, 6]. Although the original LCS can be solved in polynomial time, all these four variants are known to be NP-hard. Recently, an exact, $O(1.44225^n)$ -time, dynamic programming (DP)-based algorithm for RBLCS was proposed [2], where the two input sequences have lengths n and $poly(n)$. We first establish that each of MRCS, 1FLCS, and 2FLCS is polynomially equivalent to RBLCS. Then, we design a refined DP-based algorithm for RBLCS that runs in $O(1.41422^n)$ time, which implies that MRCS, 1FLCS, and 2FLCS can also be solved in $O(1.41422^n)$ time. Finally, we give a polynomial-time 2-approximation algorithm for 2FLCS.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Repetition-bounded longest common subsequence problem, multiset restricted longest common subsequence problem, one-side-filled longest common subsequence problem, two-side-filled longest common subsequence problem, exact algorithms, and approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.15

Funding This work is partially supported by NSERC Canada, JST CREST JPMJR1402, and JSPS KAKENHI Grant Numbers JP20H05967, JP21K11755, JP21K19765, JP22H00513, and JP22K11915.

1 Introduction

1.1 Longest common subsequence problems with occurrence constraints

The problem of computing the longest common subsequence of two sequences (LCS for short) is a classical and fundamental problem in computer science [3, 4, 9, 16]. Indeed, many polynomial-time algorithms have been published for LCS [8, 9, 14, 15, 16]. A natural extension of LCS is to impose constraints on the occurrences of the symbols in the solution. It has been shown that even very simple constraints may make the problem computationally



© Yuichi Asahiro, Jesper Jansson, Guohui Lin, Eiji Miyano, Hiroataka Ono, and Tadatoshi Utashima; licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 15; pp. 15:1–15:17

Leibniz International Proceedings in Informatics



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

much harder. As an example, the REPETITION-FREE LONGEST COMMON SUBSEQUENCE problem (RFLCS), introduced by Adi et al. [1] is: Given two sequences X and Y over an alphabet Σ , the goal of RFLCS is to find a “*repetition-free*” longest common subsequence of X and Y , where each symbol appears at most once in the obtained subsequence. Adi et al. [1] proved that RFLCS is APX-hard even if each symbol appears at most twice in each of the given sequences. On the positive side, they showed that RFLCS admits a polynomial-time occ_{max} -approximation algorithm, where occ_{max} is defined as follows: Let $occ(W, \sigma)$ be the number of occurrences of a symbol σ in a sequence W . Then occ_{max} is the maximum of $\min\{occ(X, \sigma), occ(Y, \sigma)\}$ taken over all σ 's in two sequences X and Y .

Mincu and Popa [11] introduced a general form of RFLCS, called the MULTISSET RESTRICTED COMMON SUBSEQUENCE problem (MRCS): Given two sequences X and Y , and a multiset \mathcal{M} over the alphabet Σ , the goal of MRCS is to find a common subsequence $Z_{\mathcal{M}}$ of X and Y , that contains the maximum number of symbols from \mathcal{M} . If $\mathcal{M} = \Sigma$, then MRCS is essentially equivalent to RFLCS. Therefore, MRCS is also APX-hard. In [11], the authors showed that there exists an exact algorithm solving MRCS with running time $O(|X||Y|(t+1)^{|\Sigma|})$, where t is the maximum multiplicity of symbols in \mathcal{M} . Also, they provided a polynomial-time $2\sqrt{\min\{|X|, |Y|\}}$ -approximation algorithm for MRCS [11].

Recently, Asahiro et al. [2] introduced a slightly different generalization of RFLCS, called the REPETITION-BOUNDED LONGEST COMMON SUBSEQUENCE problem (RBLCS for short): Let $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ be an alphabet of k symbols and C_{occ} be an occurrence constraint $C_{occ} : \Sigma \rightarrow \mathbb{N}$, assigning an upper bound on the number of occurrences of each symbol in Σ . Given two sequences X and Y over the alphabet Σ and an occurrence constraint C_{occ} , the goal of RBLCS is to find a “*repetition-bounded*” longest common subsequence of X and Y , where each symbol σ_i appears at most $C_{occ}(\sigma_i)$ -times in the obtained subsequence for $i = 1, 2, \dots, k$. In [2], Asahiro et al. provided a dynamic programming (DP) based algorithm for RBLCS and proved that its running time is $O(1.44225^{|X|})$ for any occurrence constraint C_{occ} , assuming $|X| \leq |Y|$ and $|Y| = O(\text{poly}(|X|))$, and even less in certain special cases. In particular, for RFLCS, their DP-based algorithm runs in $O(1.41422^{|X|})$ time. NP-hardness and APX-hardness results for RBLCS on restricted instances were also shown in [2].

1.2 Longest common subsequence problems on incomplete sequences

The comparison of biological sequences is a widely investigated field of bioinformatics, in which the genomic features including DNA sequences and genes of different organisms are compared in order to identify biological differences and similarities. In genomic analyses, however, the considered genomes are usually not complete and thus there are cases where we have to reconstruct complete genomes from incomplete genomes (so-called *scaffolds*) by filling in missing genes. For this purpose, Muñoz et al. [13] formulated the following combinatorial optimization problem, called the ONE-SIDED SCAFFOLD FILLING problem (1SF): Given an incomplete genome Y , a multiset \mathcal{M} of missing genes, and a reference genome X , the goal of 1SF is to insert the missing genes into Y so that the number of common adjacencies between the resulting Y^* and X is maximized. Subsequently, Jiang et al. [10] proposed the *Two-Sided Scaffold Filling* problem (2SF): Given two scaffolds (incomplete genomes), the goal of 2SF is to fill the missing genes into those two scaffolds respectively to result in such two genomes that the number of common adjacencies between them is maximized.

Inspired by methods for genome comparison based on LCS and by 1SF/2SF, Castelli et al. [5] introduced a new variant of LCS, called the ONE-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (1FLCS), which aims to compare a complete sequence with an incomplete one, i.e., with some missing elements: Given a complete sequence X , an incomplete

sequence Y , and a multiset \mathcal{M}_Y of symbols missing in Y , 1FLCS asks for a sequence Y^+ obtained by inserting a subset of the symbols of \mathcal{M}_Y into Y so that Y^+ induces a common subsequence with X of maximum length. The authors proved the APX-hardness of 1FLCS and designed a polynomial-time $\frac{5}{3}$ -approximation algorithm for 1FLCS. They also presented an exponential-time exact algorithm for 1FLCS. (However, they did not analyze its time complexity in detail.) In [6], Castelli et al. showed that if the alphabet size $|\Sigma|$ is a constant, then there is a polynomial-time algorithm for 1FLCS, and concluded by introducing the TWO-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (2FLCS), i.e., LCS on two incomplete sequences and two multisets of missing symbols: Given two incomplete sequences X and Y , and two multisets \mathcal{M}_X and \mathcal{M}_Y , 2FLCS asks for two sequences X^+ and Y^+ obtained by inserting subsets of the symbols of \mathcal{M}_X and \mathcal{M}_Y into X and Y , respectively, so that X^+ and Y^+ induce a common subsequence of maximum length. They conjectured that 2FLCS can be approximated within a constant factor in polynomial time, and that the following simple method gives a 2-approximation: (1) First find a longest common subsequence Z_1 of input two sequences X and Y . Then, (2) obtain a sequence Z_2 that maximizes the number of symbols matched by inserting symbols of \mathcal{M}_X and \mathcal{M}_Y . Finally, (3) output the longest of Z_1 and Z_2 . Moreover, they conjectured that 2FLCS can be solved in polynomial time if the alphabet size is a constant.

1.3 Our contributions

Suppose that there exist an $O(T_A)$ -time algorithm for an optimization problem P_A and an $O(T_B)$ -time algorithm for another optimization problem P_B . In this paper, we say that two problems P_A and P_B are *polynomially equivalent*, or that *polynomial-time equivalence* between P_A and P_B holds, if an optimal solution for an instance I_A of P_A can be obtained in $O(T_B) + O(\text{poly}(|I_A|))$ time and an optimal solution for an instance I_B of P_B can be obtained in $O(T_A) + O(\text{poly}(|I_B|))$ time. Our contributions are:

1. We establish that MRCS is polynomially equivalent to RBLCS by showing the following:
 - (i) From an input (X, Y, \mathcal{M}) of MRCS, we construct an input (X, Y, C_{occ}) of RBLCS in $O(\text{poly}(|(X, Y, \mathcal{M})|))$ time. Then, from an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) , we construct an optimal solution Z_M of MRCS on (X, Y, \mathcal{M}) in $O(\text{poly}(|(X, Y, \mathcal{M})|))$ time. Conversely, (ii) from an input (X, Y, C_{occ}) of RBLCS, we construct an input (X, Y, \mathcal{M}) of MRCS in $O(\text{poly}(|(X, Y, C_{occ})|))$ time. Then, from an optimal solution Z_M of MRCS on (X, Y, \mathcal{M}) , we construct an optimal solution Z_R in $O(\text{poly}(|(X, Y, C_{occ})|))$ time. It is important to note that our constructions between two inputs are “input-sequences preserving reductions”, i.e., X and Y in (X, Y, \mathcal{M}) and (X, Y, C_{occ}) are identical.
2. Similarly to the above, we show the polynomial-time equivalence between 1FLCS and RBLCS: (i) From an input (X, Y, \mathcal{M}_Y) of 1FLCS, we construct an input (X, Y, C_{occ}) of RBLCS in $O(\text{poly}(|(X, Y, \mathcal{M}_Y)|))$ time. Then, from an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) , we construct an optimal solution Z_{1F} of 1FLCS on (X, Y, \mathcal{M}_Y) in $O(\text{poly}(|(X, Y, \mathcal{M}_Y)|))$ time. Conversely, (ii) from an input (X, Y, C_{occ}) of RBLCS, we construct an input (X, Y, \mathcal{M}_Y) of 1FLCS in $O(\text{poly}(|(X, Y, C_{occ})|))$ time. Then, from an optimal solution Z_{1F} of 1FLCS on (X, Y, \mathcal{M}_Y) , we construct an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) in $O(\text{poly}(|(X, Y, C_{occ})|))$ time.
3. We prove the polynomial-time equivalence between 2FLCS and RBLCS. Due to the second contribution and 1FLCS being a special case of 2FLCS, we only need to show one direction: (i) From an input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS, we construct an input (X, Y, C_{occ}) of RBLCS in $O(\text{poly}(|(X, Y, \mathcal{M}_X, \mathcal{M}_Y)|))$ time. Then, from an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) , we construct an optimal solution Z_{2F} of 2FLCS on $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ in $O(\text{poly}(|Z_R|))$ time.

4. We design a refined DP-based algorithm that runs in $O(1.41422^n)$ time for RBLCS on two sequences X of length n and Y of length m (assuming that $n \leq m$ and $m = O(\text{poly}(n))$), while the previously known running time was $O(1.44225^n)$ in [2].
5. We give a simple polynomial-time 2-approximation algorithm for 2FLCS, thus resolving one of the conjectures in [6].

► **Remark 1.** One sees that 1FLCS on (X, Y, \mathcal{M}_Y) is equivalent to 2FLCS on $(X, Y, \emptyset, \mathcal{M}_Y)$; 1FLCS can be solved by using an algorithm for 2FLCS. From (ii) in the second contribution, RBLCS can also be solved by using the algorithm for 2FLCS with some extra polynomial-time calculations. Therefore, the one-way equivalence in the third contribution demonstrates the “two-way” polynomial-time equivalence between 2FLCS and RBLCS. Furthermore, interestingly, an algorithm for 1FLCS can solve 2FLCS within an extra polynomial-time factor.

► **Remark 2.** None of the constructions between inputs described above change the sequences X and Y . In particular, $|X|$ and $|Y|$ remain the same, so the above polynomial-time equivalences imply that MRCS, 1FLCS, and 2FLCS can also be solved in $O(1.41422^n)$ time.

► **Remark 3.** We also remark that the polynomial-time equivalence between 1FLCS and 2FLCS gives an affirmative answer to the conjecture on the polynomial-time solvability of 2FLCS for a constant size alphabet in [6] since we do not change Σ .

2 Preliminaries

2.1 Notation

An *alphabet* $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ is a set of k *symbols*. Let X be a sequence over the alphabet Σ and $|X|$ be the length of the sequence X . Throughout the paper, a sequence X is often regarded as a *multiset* of the same symbols. For example, $X = \langle x_1, x_2, \dots, x_n \rangle$ is a sequence of length n , where $x_i \in \Sigma$ for $1 \leq i \leq n$, i.e., $|X| = n$. A *subsequence* of X is obtained by deleting zero or more symbols from X . Then, we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y . Given two sequences X and Y as input, the goal of the LONGEST COMMON SUBSEQUENCE problem (LCS) is to find a *longest* common subsequence of X and Y , which is denoted by $LCS(X, Y)$. Let $L(X, Y)$ denote the length of $LCS(X, Y)$.

For the sequence X , the *consecutive subsequence*, i.e., *substring* $\langle x_i, x_{i+1}, \dots, x_j \rangle$ is denoted by $X_{i..j}$. Then, we define the i th *prefix* of X , for $i = 1, \dots, n$, as $X_{1..i} = \langle x_1, x_2, \dots, x_i \rangle$. Also, we define the i th *suffix* of X , for $i = 1, \dots, n$, as $X_{i..n} = \langle x_i, x_{i+1}, \dots, x_n \rangle$. $X_{1..n}$ is X .

Let $X = \langle x_1, x_2, \dots, x_n \rangle$ and $Y = \langle y_1, y_2, \dots, y_m \rangle$ be the given two sequences of length n and length m , respectively. Assume that $n \leq m$ and $m = O(\text{poly}(n))$ throughout the paper. Suppose that $Z = \langle z_1, z_2, \dots, z_p \rangle$ is a common subsequence with length p of X and Y . Then, we can consider two strictly increasing sequences $I_X = \langle i_1, i_2, \dots, i_p \rangle$ of indices of X and $I_Y = \langle j_1, j_2, \dots, j_p \rangle$ of indices of Y such that $z_\ell = x_{i_\ell} = y_{j_\ell}$ holds for each $\ell = 1, 2, \dots, p$. We call the pair (I_X, I_Y) of such sequences an *index-expression* of the common sequence Z of X and Y . A pair (x_{i_ℓ}, y_{j_ℓ}) is called the ℓ th *match*. Also, we say that the ℓ th match is z_ℓ , x_{i_ℓ} , or y_{j_ℓ} .

For two sequences $A = \langle a_1, \dots, a_i \rangle$ of length i and $B = \langle b_1, \dots, b_j \rangle$ of length j , let $A \oplus B$ be the *concatenation* of A and B , i.e., the sequence $A \oplus B = \langle a_1, \dots, a_i, b_1, \dots, b_j \rangle$ of length $i+j$. For $X = \langle x_1, x_2, \dots, x_n \rangle$ of length n , let $X \setminus \langle i \rangle$ denote the sequence obtained by deleting

the i th symbol x_i from X , i.e., $X \setminus \langle i \rangle = X_{1..i-1} \oplus X_{i+1..n} = \langle x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n \rangle$. Similarly, for $1 \leq i_1 < i_2 < \dots < i_p \leq n$, let $X \setminus \langle i_1, i_2, \dots, i_p \rangle$ be the sequence obtained by deleting p symbols $x_{i_1}, x_{i_2}, \dots, x_{i_p}$ from X .

Let \mathcal{M} be a multiset of symbols in Σ and let $|\mathcal{M}|$ be the cardinality of \mathcal{M} . Let $occ(\mathcal{M}, \sigma)$ denote the occurrences (i.e., the *multiplicity*) of a symbol $\sigma \in \Sigma$ in a multiset \mathcal{M} . Let $\mathcal{M} \setminus \{\sigma^\ell\}$ be the multiset obtained by removing ℓ σ 's from a multiset \mathcal{M} . Let $\mathcal{M} \setminus \{\sigma^*\}$ be the multiset obtained by removing all σ 's from a multiset \mathcal{M} .

Consider a multiset \mathcal{M} of cardinality ℓ and obtain an arbitrarily fixed sequence $M = \langle \mu_1, \mu_2, \dots, \mu_\ell \rangle$ of ℓ symbols in \mathcal{M} , called a *sequence-expression* of the multiset \mathcal{M} . In the following, the multiset \mathcal{M} is often regarded as its sequence-expression M ; \mathcal{M} and M are used interchangeably. Similarly to the above, for $1 \leq i_1 < i_2 < \dots < i_p \leq \ell$, let $M \setminus \langle i_1, i_2, \dots, i_p \rangle$ be the sequence obtained by deleting p symbols $\mu_{i_1}, \mu_{i_2}, \dots, \mu_{i_p}$ from M .

An algorithm **ALG** is called an α -approximation algorithm and **ALG**'s approximation ratio is α if $OPT(x)/ALG(x) \leq \alpha$ holds for every input x of an LCS-type problem, where $ALG(x)$ and $OPT(x)$ are the length of solutions obtained by **ALG** and an optimal algorithm in polynomial-time.

2.2 Repetition-bounded longest common subsequence

Recall that $occ(W, \sigma)$ is the number of occurrences of $\sigma \in \Sigma$ in a sequence W . Without loss of generality, we assume that two input sequences X and Y have all k symbols in Σ , and thus $occ(X, \sigma_i) \geq 1$ and $occ(Y, \sigma_i) \geq 1$ for every symbol σ_i . Let C_{occ} be an occurrence constraint, i.e., a function $C_{occ} : \Sigma \rightarrow \mathbb{N}$ assigning an upper bound on the number of occurrences of each symbol in Σ . The REPETITION-BOUNDED LONGEST COMMON SUBSEQUENCE problem (RBLCS) can be formally defined as follows [2]:

REPETITION-BOUNDED LONGEST COMMON SUBSEQUENCE problem (RBLCS)

Input: A pair of sequences X and Y , and an occurrence constraint C_{occ} .

Goal: Find a longest common subsequence Z of X and Y such that $occ(Z, \sigma) \leq C_{occ}(\sigma)$ is satisfied for every $\sigma \in \Sigma$.

We call Z a *repetition-bounded* longest common subsequence. Let $LCS(X, Y, C_{occ})$ denote the repetition-bounded longest common subsequence for the input triple (X, Y, C_{occ}) . Also, $L(X, Y, C_{occ})$ denotes the length of $LCS(X, Y, C_{occ})$.

► **Example 4.** Let (X, Y, C_{occ}) be an instance of RBLCS defined by:

$$X = \langle t, g, t, c, a, c, g, t, g, a, a, g \rangle, \quad Y = \langle a, t, g, c, a, t, g, g, a, c, a, g, c \rangle; \text{ and}$$

$$C_{occ}(a) = 1, C_{occ}(c) = 1, C_{occ}(g) = 2, C_{occ}(t) = 1.$$

$Z = \langle g, c, t, g, a \rangle$ of length five is an optimal solution of RBLCS since $occ(Z, a) = 1$, $occ(Z, c) = 1$, $occ(Z, g) = 2$, $occ(Z, t) = 1$, and $\sum_{\sigma \in \{a, c, g, t\}} C_{occ}(\sigma) = 5$, i.e., $L(X, Y, C_{occ}) = 5$. As a side note, $\langle t, g, c, a, t, g, a, a, g \rangle$ of length nine is an optimal solution of the original LCS.

Consider an input triple (X, Y, C_{occ}) of RBLCS and a feasible solution Z_R for (X, Y, C_{occ}) . Then, for every $\sigma \in \Sigma$, the number of occurrences $occ(Z_R, \sigma)$ of σ must be bounded above by $C_{occ}(\sigma)$. If $C_{occ}(\sigma') > \min\{occ(X, \sigma'), occ(Y, \sigma')\}$ for some σ' , then the constraint C_{occ} is somewhere redundant. Therefore, if the input (X, Y, C_{occ}) of RBLCS satisfies $C_{occ}(\sigma) \leq \min\{occ(X, \sigma), occ(Y, \sigma)\}$ for every $\sigma \in \Sigma$, then we call (X, Y, C_{occ}) the *standard* input. Without loss of generality, we assume that every input of RBLCS is standard in the following.

2.3 Multiset restricted common subsequence

The formal definition of the MULTISSET RESTRICTED COMMON SUBSEQUENCE problem (MRCS) is as follows [11]:

MULTISSET RESTRICTED COMMON SUBSEQUENCE problem (MRCS)

Input: A pair of sequences X and Y , and a multiset \mathcal{M} .

Goal: Find a common subsequence Z of X and Y such that Z contains the maximum number of symbols from \mathcal{M} .

That is, the goal of MRCS is to maximize $|\mathcal{M} \cap Z|$ as a multiset intersection or, equivalently, to minimize $|\mathcal{M} \setminus Z|$ as a multiset difference (if Z is regarded as the corresponding multiset). The optimal solution Z is denoted by $LCS(X, Y, \mathcal{M})$ in the following. The length of $LCS(X, Y, \mathcal{M})$ is denoted by $L(X, Y, \mathcal{M})$.

► **Example 5.** Consider the following input triple (X, Y, \mathcal{M}) of MRCS:

$$X = \langle t, g, t, c, a, c, g, t, g, a, a, g \rangle, \quad Y = \langle a, t, g, c, a, t, g, g, a, c, a, g, c \rangle, \quad \mathcal{M} = \{a, c, g, g, t\}$$

One sees that a common subsequence $\langle g, c, t, g, a \rangle$ of X and Y is an optimal solution of MRCS since $|\mathcal{M}| = 5$ and solutions of length five with all the symbols in \mathcal{M} are equally as good as longer solutions. For example, the objective function value of a longer common subsequence $Z = \langle g, c, t, g, a, a, g \rangle$ is also five since $|\mathcal{M} \cap Z| = 5$.

2.4 Filled longest common subsequence

Let \mathcal{M}_X (\mathcal{M}_Y , resp.) be a multiset of symbols in Σ . Then, we denote the cardinality of the multiset \mathcal{M}_X (\mathcal{M}_Y , resp.) by $|\mathcal{M}_X|$ ($|\mathcal{M}_Y|$, resp.), i.e., $\sum_{\sigma \in \mathcal{M}_X} occ(\mathcal{M}_X, \sigma)$ ($\sum_{\sigma \in \mathcal{M}_Y} occ(\mathcal{M}_Y, \sigma)$, resp.). A *filling* X^+ (Y^+ , resp.) of the sequence X (Y , resp.) is defined as a sequence obtained from X (Y , resp.) by inserting a subset of the symbols from \mathcal{M}_X (\mathcal{M}_Y , resp.) into X (Y , resp.). That is, for some $0 \leq p \leq |\mathcal{M}_X|$ and $\mathcal{M}'_X = \{\chi_1, \dots, \chi_p\} \subseteq \mathcal{M}_X$, the filling X^+ obtained by inserting \mathcal{M}'_X into X is the following concatenation of $2p + 1$ subsequences (some might be a *null* sequence):

$$X^+ = X_{1..j_1} \oplus \langle \chi_{i_1} \rangle \oplus X_{j_1+1..j_2} \oplus \langle \chi_{i_2} \rangle \oplus \dots \oplus \langle \chi_{i_p} \rangle \oplus X_{j_{p+1}..n},$$

where $X = X_{1..j_1} \oplus X_{j_1+1..j_2} \oplus \dots \oplus X_{j_{p+1}..n}$ and $\{i_1, \dots, i_p\} = \{1, \dots, p\}$. For some $0 \leq q \leq |\mathcal{M}_Y|$ and $\mathcal{M}'_Y = \{\psi_1, \dots, \psi_q\} \subseteq \mathcal{M}_Y$, the filling Y^+ obtained by inserting \mathcal{M}'_Y into Y is similarly defined. Let X^* and Y^* be fillings such that the length of $LCS(X^*, Y^*)$ is the longest among the length of $LCS(X^+, Y^+)$ over all pairs of X^+ and Y^+ . The TWO-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (2FLCS) is defined as follows [6]:

TWO-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (2FLCS)

Input: A pair of sequences X and Y , and a pair of multisets \mathcal{M}_X and \mathcal{M}_Y .

Goal: Find two fillings X^* and Y^* such that the length of $LCS(X^*, Y^*)$ is the longest among the lengths of $LCS(X^+, Y^+)$ over all pairs of X^+ and Y^+ .

In the following, the longest common subsequence $LCS(X^*, Y^*)$ of two fillings X^* and Y^* is written as $LCS(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$. The length of $LCS(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ is denoted by $L(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$. As a special case, if $\mathcal{M}_X = \emptyset$, then the problem is called the ONE-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (1FLCS) [6]:

ONE-SIDE-FILLED LONGEST COMMON SUBSEQUENCE problem (1FLCS)

Input: A pair of sequences X and Y , and a multiset \mathcal{M}_Y .

Goal: Find a filling Y^* such that the length of $LCS(X, Y^*)$ is the longest among the length of $LCS(X, Y^+)$ over all fillings Y^+ .

Let $LCS(X, Y, \mathcal{M}_Y)$ and $L(X, Y, \mathcal{M}_Y)$ be the longest common subsequence $LCS(X, Y^*)$ and its length, respectively.

► **Example 6.** Now we consider the following example, two sequences X and Y , and two multisets \mathcal{M}_X and \mathcal{M}_Y , as input to 2FLCS:

$$X = \langle g, t, c, a, c, t, g, a \rangle, Y = \langle g, a, t, c, c, g, t, g \rangle, \mathcal{M}_X = \{g, t\}, \text{ and } \mathcal{M}_Y = \{c, t, t\}$$

Here, for example, $occ(X, c) = 2$ and $occ(\mathcal{M}_Y, c) = 1$. One sees that for the input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$, an optimal pair of fillings is as follows:

$$X^* = \langle \underline{t}, g, t, c, a, c, \underline{g}, t, g, a \rangle \quad \text{and} \quad Y^* = \langle \underline{t}, g, \underline{t}, \underline{c}, a, t, c, c, g, t, g \rangle.$$

That is, the leftmost \underline{t} and the seventh \underline{g} in X^* are inserted into the original X from \mathcal{M}_X . For Y^* , the first, third, and fourth symbols (\underline{t} , \underline{t} , and \underline{c} , respectively) are inserted into Y from \mathcal{M}_Y . Then, the longest common subsequence $LCS(X^*, Y^*)$ of those fillings X^* and Y^* is $\langle \underline{t}, g, t, c, a, c, \underline{g}, t, g \rangle$. Note that $I_{X^*} = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle$ and $I_{Y^*} = \langle 1, 2, 3, 4, 5, 7, 9, 10, 11 \rangle$. One can verify that, for example, the first symbol t in $LCS(X^*, Y^*)$ originally comes from \mathcal{M}_X and \mathcal{M}_Y , but the second symbol g comes from X and Y .

Now let $X^+ = \langle x_1, x_2, \dots, x_n \rangle$ and $Y^+ = \langle y_1, y_2, \dots, y_m \rangle$ be two fillings of X and Y , respectively. Let (I_{X^+}, I_{Y^+}) be an index-expression of a common subsequence of two fillings X^+ and Y^+ . Then, the ℓ th match (x_{i_ℓ}, y_{j_ℓ}) is one of the following four types of matches:

- $\mathcal{M}_X\mathcal{M}_Y$ -match: x_{i_ℓ} and y_{j_ℓ} are inserted from \mathcal{M}_X and \mathcal{M}_Y , respectively.
- \mathcal{M}_XY -match: x_{i_ℓ} is inserted from \mathcal{M}_X but y_{j_ℓ} is originally in Y .
- $X\mathcal{M}_Y$ -match: x_{i_ℓ} is originally in X but y_{j_ℓ} is inserted from \mathcal{M}_Y .
- XY -match: x_{i_ℓ} and y_{j_ℓ} are originally in X and Y , respectively.

Let X^* and Y^* denote optimal fillings for the quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS. If there exists at least one symbol, say, σ , in \mathcal{M}_Y that does not appear in an optimal filling Y^* , then the length of $LCS(X^*, Y^* \oplus \langle \sigma \rangle)$ is equal to one of $LCS(X^*, Y^*)$, which implies that $Y^* \oplus \langle \sigma \rangle$ is another optimal filling. Similarly, if $\sigma' \in \mathcal{M}_X$ does not appear in X^* , then $X^* \oplus \langle \sigma' \rangle$ is another optimal filling. Therefore, without loss of generality, we assume that all the symbols in \mathcal{M}_X and \mathcal{M}_Y are inserted to the optimal fillings.

2.5 Known results on exact/approximation algorithms

Here, we summarize the previously known results on exact and approximation algorithms. For RBLCS, the following exact exponential-time algorithm is known:

► **Proposition 7** ([2]). *There is an $O(1.44225^n)$ -time algorithm for RBLCS on two sequences X and Y , where $|X| = n$, $|Y| = m$, and $n \leq m$, assuming that $m = O(\text{poly}(n))$.*

If $C_{occ}(\sigma) = 1$ for every symbol $\sigma \in \Sigma$, then we can design a faster exact algorithm:

► **Proposition 8** ([2]). *There is an $O(1.41422^n)$ -time algorithm for RFLCS on two sequences X and Y , where $|X| = n$, $|Y| = m$, and $n \leq m$, assuming that $m = O(\text{poly}(n))$.*

Furthermore, the following approximation algorithm is known for RFLCS:

► **Proposition 9** ([1]). *There is a polynomial-time occ_{max} -approximation algorithm for RFLCS on two sequences X and Y , where $occ_{max} = \max_{\sigma \in \Sigma} \{\min\{occ(X, \sigma), occ(Y, \sigma)\}\}$.*

For MRCS, the following exact exponential-time algorithm and the polynomial-time approximation algorithm are proposed in [11]:

► **Proposition 10** ([11]). *There is an $O(nm(t+1)^k)$ -time algorithm for MRCS on two sequences X and Y , and a multiset \mathcal{M} , where t and k are the maximum multiplicity of \mathcal{M} and the alphabet size $|\Sigma|$, respectively¹.*

► **Proposition 11** ([11]). *There is a polynomial-time $2\sqrt{\min\{n, m\}}$ -approximation algorithm for MRCS on two sequences X and Y , and a multiset \mathcal{M} , where $|X| = n$ and $|Y| = m$.*

For 1FLCS, an FPT-algorithm parameterized by the number k of $X\mathcal{M}_Y$ -matches in the optimal subsequence is known [6]. Note that k may be as large as the length of X , i.e., n .

► **Proposition 12** ([6]). *There is an $O(2^{O(k)} \text{poly}(n+m+|\mathcal{M}_Y|))$ -time algorithm for 1FLCS on an input triple (X, Y, \mathcal{M}_Y) if the number of $X\mathcal{M}_Y$ -matches in $LCS(X, Y^*)$ is k .*

The following algorithm for 1FLCS runs in polynomial time if $|\Sigma|$ is a constant [6]:

► **Proposition 13** ([6]). *There is an $O(n^{|\Sigma|+2}m)$ -time algorithm for 1FLCS on (X, Y, \mathcal{M}_Y) .*

The following approximability result is also known for 1FLCS:

► **Proposition 14** ([6]). *There is a polynomial-time $\frac{5}{3}$ -approximation algorithm for 1FLCS.*

3 Polynomial-time equivalence of RBLCS and MRCS

In this section we show the polynomial-time equivalence between RBLCS and MRCS. First consider any optimal solution $Z_{\mathcal{M}}$ for an input (X, Y, \mathcal{M}) of MRCS. Recall that the objective function value of MRCS is $|\mathcal{M} \cap Z_{\mathcal{M}}|$. Hence, $|\mathcal{M} \cap Z_{\mathcal{M}}|$ can be regarded as the summation of occurrences of all the symbols in the solution. Furthermore, intuitively, the number $occ(\mathcal{M}, \sigma)$ of occurrences of every symbol $\sigma \in \mathcal{M}$ can be regarded as the occurrence constraint $C_{occ}(\sigma)$ of the solution for RBLCS, and vice versa. One sees that we can transform from/to a multiset \mathcal{M} of symbols in Σ to/from an occurrence constraint C_{occ} of symbols in Σ such that $C_{occ}(\sigma) = occ(\mathcal{M}, \sigma)$ for every $\sigma \in \Sigma$ clearly in polynomial time; all we have to do is count the multiplicity/occurrences of every symbol in \mathcal{M} . Then, we can obtain the following theorem (see the journal version of this paper for its proof):

► **Theorem 15.** *Consider a pair of a multiset \mathcal{M} in an input for MRCS and an occurrence constraint C_{occ} of symbols in Σ in an input for RBLCS such that $C_{occ}(\sigma) = occ(\mathcal{M}, \sigma)$ for every $\sigma \in \Sigma$. Then, the followings hold: (1) Given an optimal solution Z_R for an input (X, Y, C_{occ}) of RBLCS, we can obtain an optimal solution for an input (X, Y, \mathcal{M}) of MRCS in polynomial time. (2) Given an optimal solution $Z_{\mathcal{M}}$ for an input (X, Y, \mathcal{M}) of MRCS, we can obtain an optimal solution for an input (X, Y, C_{occ}) of RBLCS in polynomial time.*

¹ We remark that the time complexity shown in Theorem 3 of [11] is $O(nmt^k)$, but the correct one must be $O(nm(t+1)^k)$ because the algorithm has to store $t+1$ values from 0 through t for the maximum multiplicity. As described before, if $\mathcal{M} = \Sigma$, i.e., $t = 1$, then MRCS is essentially equivalent to RFLCS and thus MRCS is NP-hard. If we can solve MRCS with $t = 1$ in $O(nmt^k) = O(nm)$ time, then we can obtain $P = NP$.

4 Polynomial-time equivalence of RBLCS, 1FLCS, and 2FLCS

4.1 Proof tools

In this subsection we give some proof tools. The first tool reduces the numbers of XY -matches and $\mathcal{M}_X\mathcal{M}_Y$ -matches in an output subsequence (see the journal version of this paper for its proof):

► **Lemma 16.** *Suppose that $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ is an input for 2FLCS, and X^* and Y^* are optimal fillings of $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$. Also, suppose that the numbers of XY -matches, $\mathcal{M}_X\mathcal{M}_Y$ -matches, $X\mathcal{M}_Y$ -matches, and \mathcal{M}_XY -matches of some σ in the index-expression (I_{X^*}, I_{Y^*}) of X^* and Y^* are $\alpha > 0$, $\beta > 0$, $\zeta \geq 0$, and $\eta \geq 0$, respectively. Then, we can obtain in polynomial time another pair of optimal fillings X^{**} and Y^{**} such that (i) the numbers of XY -matches, $\mathcal{M}_X\mathcal{M}_Y$ -matches, $X\mathcal{M}_Y$ -matches, and \mathcal{M}_XY -matches of σ in the index-expression $(I_{X^{**}}, I_{Y^{**}})$ of X^{**} and Y^{**} are $\alpha - 1$, $\beta - 1$, $\zeta + 1$, and $\eta + 1$, respectively, and (ii) all the matches of any different symbol $\sigma' \neq \sigma$ do not change.*

If we use the above tool iteratively α -times for $\alpha \leq \beta$ (β -times for $\beta \leq \alpha$, resp.), then we can obtain so-called an “ XY -match-free” (“ $\mathcal{M}_X\mathcal{M}_Y$ -match-free”, resp.) output subsequence.

► **Lemma 17.** *Suppose that an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ satisfies $\text{occ}(X, \sigma) > 0$ and $\text{occ}(\mathcal{M}_Y, \sigma) > 0$ for some $\sigma \in \Sigma$. Let $X = \langle x_1, \dots, x_n \rangle$ and $\mathcal{M}_Y = \langle \psi_1, \dots, \psi_\ell \rangle$. Then,*

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) = \max_{\sigma=x_i=\psi_j} L(X \setminus \langle i \rangle, Y, \mathcal{M}_X, \mathcal{M}_Y \setminus \langle j \rangle) + 1.$$

We can apply very similar arguments to the pair Y and \mathcal{M}_X , which gives:

► **Corollary 18.** *Suppose that an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ satisfies $\text{occ}(Y, \sigma) > 0$ and $\text{occ}(\mathcal{M}_X, \sigma) > 0$ for some $\sigma \in \Sigma$. Let $Y = \langle y_1, \dots, y_m \rangle$ and $\mathcal{M}_X = \langle \chi_1, \dots, \chi_\ell \rangle$. Then,*

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) = \max_{\sigma=y_i=\chi_j} L(X, Y \setminus \langle i \rangle, \mathcal{M}_X \setminus \langle j \rangle, \mathcal{M}_Y) + 1.$$

The following lemma and corollary deal with the symbol additions to multisets:

► **Lemma 19.** *Let X^+ be a filling of X and \mathcal{M}_X , and let Y^+ be a filling of Y and \mathcal{M}_Y . Suppose that a common subsequence Z of X^+ and Y^+ satisfies $\text{occ}(Z, \sigma) < \text{occ}(Y^+, \sigma)$ for some symbol $\sigma \in \Sigma$. Then, we can find in polynomial time a new filling X^{++} of X and $\mathcal{M}_X \cup \{\sigma\}$ and a common subsequence Z' of X^{++} and Y^+ satisfying the following conditions: (1) $\text{occ}(Z, \sigma) + 1 = \text{occ}(Z', \sigma)$, and (2) for every σ' except for σ $\text{occ}(Z, \sigma') = \text{occ}(Z', \sigma')$.*

► **Corollary 20.** *Let X^+ be a filling of X and \mathcal{M}_X , and let Y^+ be a filling of Y and \mathcal{M}_Y . Suppose that a common subsequence Z of X^+ and Y^+ satisfies $\text{occ}(Z, \sigma) < \text{occ}(X^+, \sigma)$ for some symbol $\sigma \in \Sigma$. Then, we can find in polynomial time a new filling Y^{++} of Y and $\mathcal{M}_Y \cup \{\sigma\}$ and a common subsequence Z' of Y^{++} and X^+ satisfying the following conditions: (1) $\text{occ}(Z, \sigma) + 1 = \text{occ}(Z', \sigma)$, and (2) for every σ' except for σ , $\text{occ}(Z, \sigma') = \text{occ}(Z', \sigma')$.*

4.2 RBLCS and 1FLCS

In this subsection we show that 1FLCS is polynomially equivalent to RBLCS. Consider an input triple (X, Y, \mathcal{M}_Y) of 1FLCS. In [12], Mincu and Popa observed that a filling-procedure of a symbol $\sigma \in \mathcal{M}_Y$ into Y to match some σ in X can be seen as a deleting-procedure of the matched σ from X [12]. Our basic ideas are based on their observation: Every symbol

15:10 Polynomial-Time Equivalences Among LCS Variants

$\sigma \in \mathcal{M}_Y$ can be matched to σ at any position in X without restrictions. After all σ 's in \mathcal{M}_Y are matched, the number of remaining unmatched σ 's in X is $occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$, which can be seen as the occurrence constraint $C_{occ}(\sigma)$ of the input (X, Y, C_{occ}) for RBLCS. In the following, we show that (i) from the input (X, Y, \mathcal{M}_Y) for 1FLCS, we can construct the input (X, Y, C_{occ}) for RBLCS such that $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$ in polynomial time, and vice versa; (ii) from an optimal solution of the former problem, we can construct an optimal solution of the latter problem in polynomial time, and vice versa.

Consider an input triple (X, Y, \mathcal{M}_Y) of 1FLCS and a feasible solution Z_{1F} . Then, for every symbol σ , $occ(Z_{1F}, \sigma) \leq occ(X, \sigma)$ holds. If $occ(X, \sigma) < occ(\mathcal{M}_Y, \sigma)$, then $occ(\mathcal{M}_Y, \sigma) - occ(X, \sigma)$ σ 's are clearly redundant. If the input (X, Y, \mathcal{M}_Y) of 1FLCS satisfies $occ(X, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$, then we call (X, Y, \mathcal{M}_Y) the *standard* input. Without loss of generality, we assume that every input of 1FLCS is standard.

► **Lemma 21.** *Suppose that a triple (X, Y, \mathcal{M}_Y) is a standard input for 1FLCS, Y^* is an optimal filling, and Z is the longest common subsequence of X and Y^* . Then, for every σ in Σ , $occ(Z, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ is satisfied.*

Proof. Let $X = \langle x_1, \dots, x_n \rangle$, $Y^* = \langle y_1^*, \dots, y_m^* \rangle$, and $Z = \langle z_1, \dots, z_\ell \rangle = \langle x_{i_1}, \dots, x_{i_\ell} \rangle = \langle y_{j_1}^*, \dots, y_{j_\ell}^* \rangle$ (i.e., $i_p < i_{p+1}$ and $j_p < j_{p+1}$ hold for every $1 \leq p \leq \ell - 1$). Since the input is standard, for every σ , $occ(\mathcal{M}_Y, \sigma) \leq occ(X, \sigma)$ holds.

Now suppose for the purpose of obtaining a contradiction that there exists at least one symbol, say, σ' , $occ(Z, \sigma') < occ(\mathcal{M}_Y, \sigma') \leq occ(X, \sigma')$ holds. Since $occ(Z, \sigma') < occ(X, \sigma')$ holds, we can find an index q such that the q th symbol x_q in X is σ' but q is not in $I_X = \langle i_1, i_2, \dots, i_\ell \rangle$. First, we assume that $i_p < q < i_{p+1}$ holds for some p where $1 \leq p \leq \ell - 1$. Then, we construct a new sequence $Z' = \langle x_{i_1}, \dots, x_{i_p} \rangle \oplus \langle \sigma' \rangle \oplus \langle x_{i_{p+1}}, \dots, x_{i_\ell} \rangle$ of length $\ell + 1$. If $q < i_1$ ($i_\ell < q$, resp.), then we insert σ' to the head position, i.e., $Z' = \langle \sigma' \rangle \oplus \langle x_{i_1}, \dots, x_{i_\ell} \rangle$ (to the tail position, i.e., $Z' = \langle x_{i_1}, \dots, x_{i_\ell} \rangle \oplus \langle \sigma' \rangle$, resp.). Moreover, since $occ(Z, \sigma') < occ(\mathcal{M}_Y, \sigma')$, we can find an index q' such that the q' th symbol $y_{q'}$ inserted into Y^* is σ' but q' is not in $I_{Y^*} = \langle j_1, j_2, \dots, j_\ell \rangle$. Then we construct a new filling Y^{**} as follows: (1) First remove the q' th symbol $y_{q'}$ ($= \sigma'$) from Y^* , and then (2) insert $y_{q'}$ right after y_{j_p} of Y^* . Note that the $(p + 1)$ st symbol in the new sequence Z' is σ' . It follows that $LCS(X, Y^{**}) = Z'$ and thus we can obtain the sequence of length $\ell + 1$ from (X, Y, \mathcal{M}_Y) , which is a contradiction. Therefore, for all σ in Σ , $occ(Z, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ holds. ◀

Consider an input triple (X, Y, \mathcal{M}_Y) of 1FLCS and its optimal solution Z_{1F} . Suppose that there is a symbol σ such that $occ(X, \sigma) > occ(Y, \sigma) + occ(\mathcal{M}_Y, \sigma)$. Let $\ell = occ(X, \sigma) - (occ(Y, \sigma) + occ(\mathcal{M}_Y, \sigma)) \geq 0$. Then, at least ℓ σ 's in X do not appear in Z_{1F} . Let \mathcal{S}_σ be a multiset of ℓ σ 's. Now, suppose that for a new triple $(X, Y, \mathcal{M}_Y \cup \mathcal{S}_\sigma)$, we can obtain an optimal solution Z . Then, the length of Z must be equal to $|Z_{1F}| + \ell$. Moreover, by removing ℓ σ 's in \mathcal{S}_σ from Z , we can easily find the original optimal solution Z_{1F} for (X, Y, \mathcal{M}_Y) . For every symbol σ' in Σ satisfying $occ(X, \sigma') > occ(Y, \sigma') + occ(\mathcal{M}_Y, \sigma')$, the similar discussion as the above can be applied. Let $\mathcal{S} = \bigcup_{\sigma': occ(X, \sigma') > occ(Y, \sigma') + occ(\mathcal{M}_Y, \sigma')} \mathcal{S}_{\sigma'}$. If we are given the triple $(X, Y, \mathcal{M}_Y \cup \mathcal{S})$, then by finding its optimal solution Z' first, and then removing all the symbols in \mathcal{S} from Z' , we obtain Z_{1F} . In the following we call the triple $(X, Y, \mathcal{M}_Y \cup \mathcal{S})$ by merging \mathcal{S} to \mathcal{M}_Y an *extended triple*. If the triple (X, Y, \mathcal{M}_Y) of 1FLCS is extended and satisfies $occ(X, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$ then it is called *ex-standard*. To simplify the discussion, we assume that every input triple (X, Y, \mathcal{M}_Y) of 1FLCS is always ex-standard.

The following lemma is quite trivial but plays an important role:

► **Lemma 22.** (1) Suppose that an input triple (X, Y, \mathcal{M}_Y) for 1FLCS is ex-standard. Then, we can construct a standard input triple (X, Y, C_{occ}) for RBLCS satisfying $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$ in polynomial time. (2) Suppose that an input triple (X, Y, C_{occ}) for RBLCS is standard. Then, we can construct an ex-standard input triple (X, Y, \mathcal{M}_Y) for 1FLCS satisfying $occ(\mathcal{M}_Y, \sigma) = occ(X, \sigma) - C_{occ}(\sigma)$ for every $\sigma \in \Sigma$ in polynomial time.

Proof. (1) Since the triple (X, Y, \mathcal{M}_Y) is ex-standard, $occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma) \geq 0$ for every σ . Therefore, we can always obtain the valid occurrence constraint such that $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ for every σ . Furthermore, since (X, Y, \mathcal{M}_Y) is ex-standard, $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma) \leq occ(Y, \sigma)$. It follows that $C_{occ}(\sigma) \leq \min\{occ(X, \sigma), occ(Y, \sigma)\}$. Hence, the triple (X, Y, C_{occ}) must be standard for RBLCS. (2) Since the triple (X, Y, C_{occ}) is standard, $C_{occ}(\sigma) \leq \min\{occ(X, \sigma), occ(Y, \sigma)\}$. Therefore, we can always obtain the valid multiset \mathcal{M}_Y such that $occ(\mathcal{M}_Y, \sigma) = occ(X, \sigma) - C_{occ}(\sigma) \geq 0$ for every σ . ◀

► **Lemma 23.** Consider an ex-standard input (X, Y, \mathcal{M}_Y) for 1FLCS and a standard input (X, Y, C_{occ}) for RBLCS such that $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ holds for every $\sigma \in \Sigma$. Let $Z_F = LCS(X, Y, \mathcal{M}_Y)$ and Y^* be an optimal filling for 1FLCS. Also, let $Z_R = LCS(X, Y, C_{occ})$ be an optimal solution for RBLCS. Then, $|Z_R| + |\mathcal{M}_Y| = |Z_F|$ holds.

Proof. First, from Lemma 22, we always find a pair of triples (X, Y, \mathcal{M}_Y) and (X, Y, C_{occ}) such that the former and the latter are the ex-standard input for 1FLCS and the standard input for RBLCS satisfying $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$, respectively.

(1) We first show that $|Z_F| \leq |Z_R| + |\mathcal{M}_Y|$ holds. Let $X = \langle x_1, \dots, x_n \rangle$, $Y = \langle y_1, \dots, y_m \rangle$, and $\mathcal{M}_Y = \langle \psi_1, \dots, \psi_\ell \rangle$, where \mathcal{M}_Y is the sequence-expression of \mathcal{M}_Y . By the assumption that (X, Y, \mathcal{M}_Y) is ex-standard, there exists a sequence $\langle i_1, i_2, \dots, i_\ell \rangle$ of indices of X satisfying $L(X, Y, \mathcal{M}_Y) = L(X \setminus \langle i_1, \dots, i_\ell \rangle, Y, \emptyset) + \ell$, by regarding $L(X, Y, \emptyset, \mathcal{M}_Y)$ as $L(X, Y, \mathcal{M}_Y)$, and by using the formula in Lemma 17 recursively. Since $\mathcal{M}_Y = \emptyset$, $L(X \setminus \langle i_1, \dots, i_\ell \rangle, Y, \emptyset)$ is clearly equal to the length of the longest common subsequence Z' of $X \setminus \langle i_1, \dots, i_\ell \rangle$ and Y . Therefore, $|Z_F| = |Z'| + |\mathcal{M}_Y|$. Note that Z' is a common subsequence of the original X and Y and satisfies the following for every σ :

$$C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma) = occ(X \setminus \langle i_1, \dots, i_\ell \rangle, \sigma) \geq occ(Z', \sigma).$$

That is, every symbol in Z' satisfies the occurrence constraint C_{occ} of RBLCS, which implies that $|Z'| \leq |Z_R|$. As a result, $|Z_F| = |Z'| + |\mathcal{M}_Y| \leq |Z_R| + |\mathcal{M}_Y|$ holds.

(2) Next, we show that $|Z_R| + |\mathcal{M}_Y| \leq |Z_F|$. Recall that for every σ , $occ(Z_R, \sigma) \leq C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ is satisfied. Here, from the viewpoint of 1FLCS, we can obtain a longer sequence than Z_R by filling symbols of \mathcal{M}_Y into Y . Suppose that Z_R is a common subsequence for RBLCS on (X, Y, C_{occ}) and (X, Y, \emptyset) is an input triple for 1FLCS. From Lemma 19, by setting a multiset $\mathcal{M}'_Y = \{\sigma\}$ and filling σ into Y as matched with some σ in X , we can obtain a common subsequence Z_1 such that $|Z_1| = |Z_R| + 1$, $occ(Z_1, \sigma) = occ(Z_R, \sigma) + 1$, and $occ(Z_1, \sigma') = occ(Z_R, \sigma')$ for every σ' except for σ . By repeating the merge $\mathcal{M}'_Y \cup \{\sigma\}$ and the filling of σ $occ(\mathcal{M}_Y, \sigma)$ -times for every $\sigma \in \Sigma$, we can eventually obtain \mathcal{M}_Y , the filling of Y and \mathcal{M}_Y , and a common subsequence Z satisfying $|Z| = |Z_R| + \sum_{\sigma \in \Sigma} occ(\mathcal{M}_Y, \sigma) = |Z_R| + |\mathcal{M}_Y|$. Since Z_F is the longest, $|Z| \leq |Z_F|$. Hence, $|Z_R| + |\mathcal{M}_Y| = |Z| \leq |Z_F|$ holds.

From (1) and (2), $|Z_R| + |\mathcal{M}_Y| = |Z_F|$. This completes the proof. ◀

15:12 Polynomial-Time Equivalences Among LCS Variants

► **Theorem 24.** *Consider an ex-standard input (X, Y, \mathcal{M}_Y) for 1FLCS and a standard input (X, Y, C_{occ}) for RBLCS such that $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ holds for every $\sigma \in \Sigma$. Let $Z_F = LCS(X, Y, \mathcal{M}_Y)$ and Y^* be an optimal filling for 1FLCS. Also, let $Z_R = LCS(X, Y, C_{occ})$ be an optimal solution for RBLCS. Then, the followings hold: (1) Given an optimal solution Z_R for RBLCS, we can obtain an optimal solution for 1FLCS in polynomial time. (2) Given an optimal filling Y^* for 1FLCS, we can obtain an optimal solution for RBLCS in polynomial time.*

Proof. Consider two sequences X and Y , a multiset \mathcal{M}_Y , and an occurrence constraint C_{occ} such that $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma)$ holds for every $\sigma \in \Sigma$.

(1) Suppose that the optimal solution Z_R for RBLCS is now given. From Lemma 23, every optimal solution for 1FLCS is of length $|Z_R| + |\mathcal{M}_Y|$. Hence, it is enough to prove that we can obtain an optimal filling Y^* of Y and \mathcal{M}_Y from Z_R and a common subsequence Z_F of X and Y^* such that $|Z_R| + |\mathcal{M}_Y| = |Z_F|$ in polynomial time. As seen in the proof of Lemma 23, by repeating the merge $\mathcal{M}'_Y = \mathcal{M}_Y \cup \{\sigma\}$ and the filling of σ $occ(\mathcal{M}_Y, \sigma)$ -times for every $\sigma \in \Sigma$, we eventually obtain Y^* and Z_F satisfying $|Z_F| = |Z_R| + \sum_{\sigma \in \Sigma} occ(\mathcal{M}_Y, \sigma) = |Z_R| + |\mathcal{M}_Y|$. The total number of iterations is $|\mathcal{M}_Y|$. Since each iteration works in polynomial time as shown in Lemma 23, Y^* and Z_F of 1FLCS can be obtained in polynomial time.

(2) Suppose that the optimal filling Y^* is now given. The longest common subsequence Z_F of X and Y^* , and its index-expression (I_X, I_{Y^*}) can be obtained in polynomial time. From Lemma 21, $occ(Z_F, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ holds for every $\sigma \in \Sigma$. Therefore, we can find $|Z_F| - |\mathcal{M}_Y|$ XY -matches in (I_X, I_{Y^*}) . Letting z_ℓ be the symbol of the ℓ th XY -match ($1 \leq \ell \leq |Z_F| - |\mathcal{M}_Y|$), we construct the sequence $Z_F^- = \langle z_1, z_2, \dots, z_{|Z_F| - |\mathcal{M}_Y|} \rangle$ of length $|Z_F| - |\mathcal{M}_Y|$. Note that Z_F^- must be a common subsequence of X and Y . Moreover, Z_F^- satisfies the occurrence constraint $C_{occ}(\sigma) = occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma) \geq occ(Z_F, \sigma) - occ(\mathcal{M}_Y, \sigma)$ for every $\sigma \in \Sigma$. Since $|Z_F^-| = |Z_F| - |\mathcal{M}_Y|$, Z_F^- is an optimal solution for RBLCS from Lemma 23. The construction of Z_F^- can be easily executed by scanning the index-expression (I_X, I_{Y^*}) and thus it can be done in polynomial time. ◀

4.3 RBLCS and 2FLCS

In this subsection we consider the polynomial-time equivalence between 2FLCS and RBLCS. Since 1FLCS on (X, Y, \mathcal{M}_Y) is equivalent to 2FLCS on $(X, Y, \emptyset, \mathcal{M}_Y)$, 1FLCS can be solved by using any algorithm for 2FLCS. From the polynomial-time equivalence between 1FLCS and RBLCS in the previous subsection, RBLCS can also be solved by the same algorithm with some extra polynomial-time calculations. Therefore, to establish the equivalence between RBLCS and 2FLCS, only one direction remains to be proved. To do so, we first give a pair of two inputs $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ for 2FLCS and (X, Y, C_{occ}) for RBLCS. Then, we show that given an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) , we can obtain optimal fillings X^* and Y^* of 2FLCS on $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ in polynomial time.

► **Lemma 25.** *Suppose that an input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS satisfies $occ(X, \sigma) = p < occ(\mathcal{M}_Y, \sigma) = q$ and $\min\{occ(\mathcal{M}_X, \sigma), occ(Y, \sigma) + q - p\} = \lambda \geq 0$ for some positive integers p and q . Then the following holds:*

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) \leq L(X, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \{\sigma^{q-p}\}) + \lambda$$

Proof. Suppose that an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS satisfies $\text{occ}(X, \sigma) = p < \text{occ}(\mathcal{M}_Y, \sigma) = q$. If we set $X = \langle x_1, \dots, x_n \rangle$ and $\mathcal{M}_Y = \langle \psi_1, \dots, \psi_\ell \rangle$, and apply the recursive formula in Lemma 17 recursively, then there exist two sequences of $\langle i_1, \dots, i_p \rangle$ and $\langle j_1, \dots, j_p \rangle$ of indices such that $\sigma = x_{i_r} = \psi_{j_r}$ for every $1 \leq r \leq p$. Therefore, we obtain

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) = L(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X, \mathcal{M}_Y \setminus \langle j_1, \dots, j_p \rangle) + p.$$

Suppose that X^+ and Y^+ are optimal fillings of $(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X, \mathcal{M}_Y \setminus \langle j_1, \dots, j_p \rangle)$. Then, $\text{occ}(X^+, \sigma) \leq \text{occ}(\mathcal{M}_X, \sigma)$ since $\sigma \notin X \setminus \langle i_1, \dots, i_p \rangle$ and $\text{occ}(Y^+, \sigma) \leq \text{occ}(Y, \sigma) + q - p$. Therefore, we obtain $\text{occ}(Z, \sigma) \leq \min \{ \text{occ}(\mathcal{M}_X, \sigma), \text{occ}(Y, \sigma) + q - p \}$ for $Z = \text{LCS}(X^+, Y^+)$. Now, we set $\min \{ \text{occ}(\mathcal{M}_X, \sigma), \text{occ}(Y, \sigma) + q - p \} = \lambda$. Then, we have:

$$\begin{aligned} L(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \{\sigma^*\}) + \lambda &\geq \\ L(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X, \mathcal{M}_Y \setminus \langle j_1, \dots, j_p \rangle) & . \end{aligned}$$

Suppose that a sequence $J^+ = \langle j_1, \dots, j_q \rangle$ of indices satisfies that $\psi_{j_{r'}} = \sigma$ for every $1 \leq r' \leq q$. Then, we obtain:

$$\begin{aligned} L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) &= L(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X, \mathcal{M}_Y \setminus \langle j_1, \dots, j_p \rangle) + p \\ &\leq L(X \setminus \langle i_1, \dots, i_p \rangle, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \{\sigma^*\}) + \lambda + p \\ &= L(X, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \langle j_{p+1}, \dots, j_q \rangle) + \lambda. \end{aligned}$$

This completes the proof. ◀

► **Theorem 26.** *Suppose that an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS satisfies $\text{occ}(X, \sigma) = p < \text{occ}(\mathcal{M}_Y, \sigma) = q$ for some positive integers p and q , and optimal fillings X_1^+ and Y_1^+ of $(X, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \{\sigma^{q-p}\})$ are given. Then, optimal fillings X_2^+ and Y_2^+ of an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ can be obtained in polynomial time.*

Proof. Suppose that Z_1 is the longest common subsequence of X_1^+ and Y_1^+ such that the index-expression of Z_1 is (I, J) , where $I = \langle i_1, \dots, i_k \rangle$ and $J = \langle j_1, \dots, j_k \rangle$. Also suppose that Z_2 is the longest common subsequence of X_2^+ and Y_2^+ . From Lemma 25, $|Z_1| + \lambda \geq |Z_2|$ holds, where $\min \{ \text{occ}(\mathcal{M}_X, \sigma), \text{occ}(Y, \sigma) + q - p \} = \lambda$.

Now suppose that $X_1^+ = \langle x_1, \dots, x_n \rangle$ and $Y_1^+ = \langle y_1, \dots, y_m \rangle$. Also suppose that $Y_2^+ = Y_1^+ \oplus \overbrace{\langle \sigma, \dots, \sigma \rangle}^{q-p}$. One can see that $\text{occ}(Y_2^+, \sigma) = \text{occ}(Y, \sigma) + q$, $\text{occ}(Z_1, \sigma) \leq p < \text{occ}(\mathcal{M}_Y, \sigma) = q$, and Z_1 is a common subsequence of X_1^+ and Y_2^+ . Therefore, by applying the formula in (1) of Lemma 19 $\min \{ \text{occ}(\mathcal{M}_X, \sigma), \text{occ}(Y, \sigma) + q - p \}$ -times, we can get the target sequence X_2^+ in polynomial time. ◀

It is important to note that $(X, Y, \mathcal{M}_X \setminus \{\sigma^*\}, \mathcal{M}_Y \setminus \{\sigma^{q-p}\})$ does not satisfy both $\text{occ}(X, \sigma) < \text{occ}(\mathcal{M}_Y, \sigma)$ and $\text{occ}(Y, \sigma) < \text{occ}(\mathcal{M}_X, \sigma)$. For Y and \mathcal{M}_X , we have:

► **Corollary 27.** *Suppose that an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS satisfies $\text{occ}(Y, \sigma) = p < \text{occ}(\mathcal{M}_X, \sigma) = q$ for some positive integers p and q , and optimal fillings X_1^+ and Y_1^+ of $(X, Y, \mathcal{M}_X \setminus \{\sigma^{q-p}\}, \mathcal{M}_Y \setminus \{\sigma^*\})$ are given. Then, optimal fillings X_2^+ and Y_2^+ of an input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ can be obtained in polynomial time.*

From Theorem 26 and Corollary 27, any input can be reduced to the quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ such that for every σ , both $\text{occ}(X, \sigma) \geq \text{occ}(\mathcal{M}_Y, \sigma)$ and $\text{occ}(Y, \sigma) \geq \text{occ}(\mathcal{M}_X, \sigma)$ are satisfied. Therefore, if the input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS satisfies both $\text{occ}(X, \sigma) \geq \text{occ}(\mathcal{M}_Y, \sigma)$ and $\text{occ}(Y, \sigma) \geq \text{occ}(\mathcal{M}_X, \sigma)$ for every $\sigma \in \Sigma$, then we call $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ the *standard input*.

► **Theorem 28.** *For a standard input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$, consider an occurrence constraint C_{occ} such that $C_{occ}(\sigma) = \min \{occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma), occ(Y, \sigma) - occ(\mathcal{M}_X, \sigma)\}$ holds for every $\sigma \in \Sigma$. Then, the triple (X, Y, C_{occ}) must be standard for RBLCS. If an optimal solution Z_R of RBLCS on (X, Y, C_{occ}) is given, then we can obtain optimal fillings X^* and Y^* of 2FLCS on a standard input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ in polynomial time.*

Proof. Suppose that the input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS is standard, $|\mathcal{M}_X| = p$, and $|\mathcal{M}_Y| = q$. Let $X = \langle x_1, \dots, x_n \rangle$ and $Y = \langle y_1, \dots, y_m \rangle$. Then, by applying the arguments of Lemma 17 and Corollary 18 to all the symbols recursively, we can obtain the sequences $\langle i_1, \dots, i_q \rangle$ and $\langle j_1, \dots, j_p \rangle$ of different indices that satisfy the following:

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) = L(X \setminus \langle i_1, \dots, i_q \rangle, Y \setminus \langle j_1, \dots, j_p \rangle, \emptyset, \emptyset) + p + q.$$

One can verify that for the input $(X \setminus \langle i_1, \dots, i_q \rangle, Y \setminus \langle j_1, \dots, j_p \rangle, \emptyset, \emptyset)$ of 2FLCS, the longest common subsequence of $X \setminus \langle i_1, \dots, i_q \rangle$ and $Y \setminus \langle j_1, \dots, j_p \rangle$ is clearly an optimal solution of the classical LCS. Let Z' be such a sequence. Here, note that for every $\sigma \in \Sigma$, we can obtain:

$$\begin{aligned} occ(X \setminus \langle i_1, \dots, i_q \rangle, \sigma) &= occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma), \text{ and} \\ occ(Y \setminus \langle j_1, \dots, j_p \rangle, \sigma) &= occ(Y, \sigma) - occ(\mathcal{M}_X, \sigma). \end{aligned}$$

Therefore, we have:

$$occ(Z', \sigma) \leq \min \{occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma), occ(Y, \sigma) - occ(\mathcal{M}_X, \sigma)\}.$$

Since Z' is a common subsequence of X and Y , Z' is a feasible solution of RBLCS on (X, Y, C_{occ}) . Therefore, $|Z_R| \geq |Z'|$ holds. It follows that $|Z_R| + |\mathcal{M}_X| + |\mathcal{M}_Y| \geq |Z'| + |\mathcal{M}_X| + |\mathcal{M}_Y| = L(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$.

As for Z_R , $occ(Z_R, \sigma) \leq C_{occ}(\sigma) = \min \{occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma), occ(Y, \sigma) - occ(\mathcal{M}_X, \sigma)\}$ holds for every σ . Therefore, by applying Lemma 19 $occ(\mathcal{M}_X, \sigma)$ -times for every symbol $\sigma \in \Sigma$, we can construct in polynomial time the filling X^+ of X and \mathcal{M}_X , and a common subsequence Z_1 of X^+ and Y such that $|Z_1| = |Z_R| + |\mathcal{M}_X|$ and $occ(Z_1, \sigma) \leq C_{occ}(\sigma) + |\mathcal{M}_X|$.

Note that for every σ , $occ(X^+, \sigma) = occ(X, \sigma) + occ(\mathcal{M}_X, \sigma)$ and $occ(Z_1, \sigma) \leq occ(X, \sigma) - occ(\mathcal{M}_Y, \sigma) + occ(\mathcal{M}_X, \sigma)$ hold. Hence, by applying Corollary 20 $occ(\mathcal{M}_Y, \sigma)$ -times for every symbol σ , we can construct in polynomial time the filling Y^+ of Y and \mathcal{M}_Y , and a common subsequence Z_2 of X^+ and Y^+ such that $|Z_2| = |Z_1| + |\mathcal{M}_Y| = |Z_R| + |\mathcal{M}_X| + |\mathcal{M}_Y|$. Recall that $|Z_R| + |\mathcal{M}_X| + |\mathcal{M}_Y| \geq L(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$. Therefore, $|Z_2| \geq L(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ holds.

As a result, X^+ and Y^+ are optimal fillings of 2FLCS on $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ and those can be obtained in polynomial time if Z_R is given. This completes the proof. ◀

5 $O(1.41422^n)$ -time exact algorithm for RBLCS

In [2], a dynamic programming (DP) based algorithm for RBLCS was provided and it was explicitly proved that its running time is $O(1.44255^n)$. In this section we improve the running time from $O(1.44255^n)$ to $O(1.41422^n)$, but give only the basic ideas here. Further details can be found in the journal version of this paper.

Now, let us consider the original LCS and its typical DP-based algorithm. Let $L(i, j)$ denote the length of a longest common subsequence of the i th prefix $X_{1..i}$ of X and the j th prefix $Y_{1..j}$ of Y . In the process of execution, each value of $L(i, j)$ is computed and is stored into a two-dimensional DP-table L_0 of size $(n+1) \times (m+1)$. For more details, e.g., see [7].

For RBLCS, the previous DP-based algorithm proposed in [2] has to store not only the length of the subsequence Z , but also the occurrence $occ(Z, \sigma)$ of every σ in Z not to break the occurrence constraint $C_{occ}(\sigma)$. To store the occurrences, the algorithm introduces an

occurrence vector \mathbf{v} . Let $L(i, j, \mathbf{v})$ be the length of a repetition-bounded longest common subsequence of $X_{1..i}$ and $Y_{1..j}$ satisfying the occurrence vector \mathbf{v} , i.e., the length of the longest subsequence which does not break the occurrence constraint. Then, each value of $L(i, j, \mathbf{v})$ is stored into a three-dimensional DP-table L_1 of size $(n+1) \times (m+1) \times \prod_{\sigma} (C_{occ}(\sigma) + 1)$. In [2], the authors showed that the table size of L_1 is bounded above by $O(1.44255^n)$.

Our new DP-based algorithm prepares a smaller DP-table of size $(n+1) \times (m+1) \times \prod_{\sigma} (\min\{C_{occ}(\sigma), occ(X, \sigma) - C_{occ}(\sigma)\} + 1)$. One can show that the DP-table size is reduced to $O(1.41422^n)$:

► **Theorem 29.** *There is an $O(1.41422^n)$ -time DP-based algorithm to solve RBLCS for two input sequences X and Y , where $|X| = n$, $|Y| = m = O(\text{poly}(n))$, and $|X| \leq |Y|$.*

Recall that all reductions in the previous sections preserve X and Y . By our polynomial-time equivalences, we obtain the following corollary:

► **Corollary 30.** *MRCS, 1FLCS, and 2FLCS can be solved in $O(1.41422^n)$ time.*

6 A polynomial-time 2-approximation algorithm for 2FLCS

In this section, we give a polynomial-time algorithm for 2FLCS and show that its approximation ratio is bounded above by two by using the proof tools introduced in Section 4.1.

Algorithm. Suppose that a standard input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ is given, i.e., $occ(X, \sigma) \geq occ(\mathcal{M}_Y, \sigma)$ and $occ(Y, \sigma) \geq occ(\mathcal{M}_X, \sigma)$ are satisfied. Let $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_m\}$. Here is an outline of our algorithm ALG: (Step 1) Let $X_b = \varepsilon$ and $Y_f = \varepsilon$ be two empty sequences. (1-1) While scanning from x_1 to x_n of X , if the i th symbol x_i in X matches a symbol, say, σ_y , in \mathcal{M}_Y , then $x_i (= \sigma_y)$ is concatenated to Y_f , i.e., $Y_f = Y_f \oplus \langle \sigma_y \rangle$ and removed from \mathcal{M}_Y . Then, obtain a filling $Y_2 = Y_f \oplus Y$ of Y and \mathcal{M}_Y . Similarly, (1-2) while scanning from y_1 to y_m of Y , if the i th symbol y_i in Y matches a symbol, say, σ_x , in \mathcal{M}_X , then $y_i (= \sigma_x)$ is concatenated to X_b , i.e., $X_b = X_b \oplus \langle \sigma_x \rangle$ and removed from \mathcal{M}_X . Then, obtain a filling $X_2 = X \oplus X_b$ of X and \mathcal{M}_X (n.b., not $X_b \oplus X$). (Step 2) Obtain a longest common subsequence Z of two fillings X^+ and Y^+ . (Step 3) Output a solution triple (X^+, Y^+, Z) . See Algorithm 1 for the detailed description of ALG.

► **Theorem 31.** *Algorithm ALG is a polynomial-time 2-approximation algorithm for 2FLCS on a standard input quadruple $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$.*

Proof. Suppose that the input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$ of 2FLCS is standard. Let $X = \langle x_1, \dots, x_n \rangle$ and $Y = \langle y_1, \dots, y_m \rangle$. Then, by applying the arguments of Lemma 17 and Corollary 18 to all the symbols recursively, we can obtain the sequences $\langle i_1, \dots, i_{|\mathcal{M}_Y|} \rangle$ and $\langle j_1, \dots, j_{|\mathcal{M}_X|} \rangle$ of different indices that satisfy the following:

$$L(X, Y, \mathcal{M}_X, \mathcal{M}_Y) = L(X \setminus \langle i_1, \dots, i_{|\mathcal{M}_Y|} \rangle, Y \setminus \langle j_1, \dots, j_{|\mathcal{M}_X|} \rangle, \emptyset, \emptyset) + |\mathcal{M}_Y| + |\mathcal{M}_X|.$$

Clearly, the first term $L(X \setminus \langle i_1, \dots, i_{|\mathcal{M}_Y|} \rangle, Y \setminus \langle j_1, \dots, j_{|\mathcal{M}_X|} \rangle, \emptyset, \emptyset)$ of the right-hand side is at most $L(X, Y)$ since $X \setminus \langle i_1, \dots, i_{|\mathcal{M}_Y|} \rangle$ and $Y \setminus \langle j_1, \dots, j_{|\mathcal{M}_X|} \rangle$ are subsequences of X and Y , respectively. Therefore, the longest length OPT of 2FLCS is at most $L(X, Y) + |\mathcal{M}_X| + |\mathcal{M}_Y|$.

Algorithm 1 ALG.

Input: Two sequences $X = \langle x_1, \dots, x_n \rangle$ and $Y = \langle y_1, \dots, y_m \rangle$; and two multisets \mathcal{M}_X and \mathcal{M}_Y

Output: Two fillings X^+ of X and \mathcal{M}_X and Y^+ of Y and \mathcal{M}_Y ; and a common subsequence Z of X^+ and Y^+

```

1  $X_b := \varepsilon, Y_f := \varepsilon;$ 
2 for  $i = 1$  to  $n$  do
3   | if  $x_i = \sigma_y$  for  $\sigma_y \in \mathcal{M}_Y$  then
4   | |  $Y_f := Y_f \oplus \langle \sigma_y \rangle, \mathcal{M}_Y := \mathcal{M}_Y \setminus \{\sigma_y\};$ 
5  $Y^+ := Y_f \oplus Y;$ 
6 for  $i = 1$  to  $m$  do
7   | if  $y_i = \sigma_x$  for  $\sigma_x \in \mathcal{M}_X$  then
8   | |  $X_b := X_b \oplus \langle \sigma_x \rangle, \mathcal{M}_X := \mathcal{M}_X \setminus \{\sigma_x\};$ 
9  $X^+ := X \oplus X_b;$ 
10 Find a longest common subsequence  $Z$  of the two sequences  $X^+$  and  $Y^+;$ 
11 return  $(X^+, Y^+, Z);$ 

```

Let $ALG = |Z|$ be the length obtained by our algorithm ALG for the input $(X, Y, \mathcal{M}_X, \mathcal{M}_Y)$, i.e., $ALG = L(X^+, Y^+)$. Since a longest common subsequence of X and Y is a common subsequence of X^+ and Y^+ , $ALG \geq L(X, Y)$ holds. Furthermore, since $LCS(X, Y_f) \oplus LCS(X_b, Y)$ is another common subsequence of X^+ and Y^+ , $ALG \geq L(X, Y_f) + L(X_b, Y) = |\mathcal{M}_Y| + |\mathcal{M}_X|$ holds. As a result, the approximation ratio of ALG is bounded as follows:

$$\begin{aligned}
\frac{OPT}{ALG} &\leq \frac{L(X, Y) + |\mathcal{M}_X| + |\mathcal{M}_Y|}{\max\{L(X, Y), |\mathcal{M}_X| + |\mathcal{M}_Y|\}} \\
&= \frac{2(L(X, Y) + |\mathcal{M}_X| + |\mathcal{M}_Y|)}{2(\max\{L(X, Y), |\mathcal{M}_X| + |\mathcal{M}_Y|\})} \\
&\leq \frac{2(L(X, Y) + |\mathcal{M}_X| + |\mathcal{M}_Y|)}{L(X, Y) + |\mathcal{M}_X| + |\mathcal{M}_Y|} \\
&= 2.
\end{aligned}$$

Clearly, ALG runs in polynomial time. This completes the proof. \blacktriangleleft

For non-standard inputs, we can also obtain a 2-approximation algorithm by slightly modifying ALG. All we have to do is to add $\mathcal{M}_X \mathcal{M}_Y$ -matches of redundant symbols. If the sequence of length ℓ is concatenated, then we get $\frac{OPT+\ell}{ALG+\ell} \leq 2$. Further details can be found in the journal version of this paper.

References

- 1 Said Sadique Adi, Marília D. V. Braga, Cristina G. Fernandes, Carlos Eduardo Ferreira, Fábio Viduani Martinez, Marie-France Sagot, Marco Aurelio Stefanos, Christian Tjandraatmadja, and Yoshiko Wakabayashi. Repetition-free longest common subsequence. *Electron. Notes Discret. Math.*, 30:243–248, 2008. doi:10.1016/j.endm.2008.01.042.
- 2 Yuichi Asahiro, Jesper Jansson, Guohui Lin, Eiji Miyano, Hirotaka Ono, and Tadatoshi Utashima. Exact algorithms for the repetition-bounded longest common subsequence problem. *Theor. Comput. Sci.*, 838:238–249, 2020. doi:10.1016/j.tcs.2020.07.042.

- 3 Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In Pablo de la Fuente, editor, *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, pages 39–48. IEEE Computer Society, 2000. doi:10.1109/SPIRE.2000.878178.
- 4 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for np-hard string problems: The algorithmics column by Gerhard J. Woeginger. *Bull. EATCS*, 114, 2014. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/view/310>.
- 5 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.
- 6 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.
- 7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: <http://mitpress.mit.edu/books/introduction-algorithms-fourth-edition>.
- 8 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.
- 9 Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977. doi:10.1145/322033.322044.
- 10 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.
- 11 Radu Stefan Mincu and Alexandru Popa. Better heuristic algorithms for the repetition free LCS and other variants. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 297–310. Springer, 2018. doi:10.1007/978-3-030-00479-8_24.
- 12 Radu Stefan Mincu and Alexandru Popa. Heuristic algorithms for the longest filled common subsequence problem. In *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*, pages 449–453. IEEE, 2018. doi:10.1109/SYNASC.2018.00075.
- 13 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.
- 14 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.
- 15 David Sankoff. Matching sequences under deletion/insertion constraints. In *Proc. National Academy of Science USA*, volume 69, pages 4–6, 1972. doi:10.1073/pnas.69.1.4.
- 16 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.

On Strings Having the Same Length- k Substrings

Giulia Bernardini ✉ 

University of Trieste, Italy
CWI, Amsterdam, The Netherlands

Alessio Conte ✉

University of Pisa, Italy

Esteban Gabory ✉

CWI, Amsterdam, The Netherlands

Roberto Grossi ✉

University of Pisa, Italy

Grigorios Loukides ✉ 

King's College London, UK

Solon P. Pissis ✉ 

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Giulia Punzi ✉

University of Pisa, Italy

Michelle Sweering ✉

CWI, Amsterdam, The Netherlands

Abstract

Let $\text{Substr}_k(X)$ denote the set of length- k substrings of a given string X for a given integer $k > 0$. We study the following basic string problem, called z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS: Given a set \mathcal{S}_k of n length- k strings and an integer $z > 0$, list z shortest distinct strings T_1, \dots, T_z such that $\text{Substr}_k(T_i) = \mathcal{S}_k$, for all $i \in [1, z]$. The z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS problem arises naturally as an encoding problem in many real-world applications; e.g., in data privacy, in data compression, and in bioinformatics. The 1-SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS, referred to as SHORTEST \mathcal{S}_k -EQUIVALENT STRING, asks for a shortest string X such that $\text{Substr}_k(X) = \mathcal{S}_k$.

Our main contributions are summarized below:

- Given a directed graph $G(V, E)$, the DIRECTED CHINESE POSTMAN (DCP) problem asks for a shortest closed walk that visits every edge of G at least once. DCP can be solved in $\tilde{O}(|E||V|)$ time using an algorithm for min-cost flow. We show, via a non-trivial reduction, that if SHORTEST \mathcal{S}_k -EQUIVALENT STRING over a *binary alphabet* has a near-linear-time solution then so does DCP.
- We show that the length of a shortest string output by SHORTEST \mathcal{S}_k -EQUIVALENT STRING is in $\mathcal{O}(k + n^2)$. We generalize this bound by showing that the total length of z shortest strings is in $\mathcal{O}(zk + zn^2 + z^2n)$. We derive these upper bounds by showing (asymptotically tight) bounds on the total length of z shortest Eulerian walks in general directed graphs.
- We present an algorithm for solving z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS in $\mathcal{O}(nk + n^2 \log^2 n + zn^2 \log n + |\text{output}|)$ time. If $z = 1$, the time becomes $\mathcal{O}(nk + n^2 \log^2 n)$ by the fact that the size of the input is $\Theta(nk)$ and the size of the output is $\mathcal{O}(k + n^2)$.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases string algorithms, combinatorics on words, de Bruijn graph, Chinese Postman

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.16

Funding The work in this paper is supported in part by: the MIUR project Algorithms for HARnessing networked Data (AHEAD); and 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.

Giulia Bernardini: Supported by the Netherlands Organisation for Scientific Research (NWO) under project OCENW.GROOT.2019.015 “Optimization for and with Machine Learning (OPTIMAL)”.

Grigorios Loukides: Supported by the Leverhulme Trust RPG-2019-399 project.

Michelle Sweering: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.



© Giulia Bernardini, Alessio Conte, Esteban Gabory, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, Giulia Punzi, and Michelle Sweering;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 16; pp. 16:1–16:17



Leibniz International Proceedings in Informatics

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

1 Introduction

We start with some basic definitions and notation on strings from [6]. Let $X = X[0] \cdots X[n-1]$ be a *string* of length $|X| = n$ over an alphabet Σ whose elements are called *letters*. For any two positions i and $j \geq i$ of X , $X[i..j]$ is the *fragment* of X starting at position i and ending at position j . The fragment $X[i..j]$ is an *occurrence* of the underlying *substring* $P = X[i] \cdots X[j]$; we say that P occurs at *position* i in X . A *prefix* of X is a fragment of the form $X[0..j]$ and a *suffix* of X is a fragment of the form $X[i..n-1]$. By XY or $X \cdot Y$ we denote the *concatenation* of two strings X and Y , i.e., $XY = X[0] \cdots X[|X| - 1]Y[0] \cdots Y[|Y| - 1]$.

Let $\text{Substr}_k(X)$ denote the set of length- k substrings of a finite string X . We consider the following basic problem on strings.

z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS

Input: A set \mathcal{S}_k of n length- k strings over an integer alphabet $\Sigma = [0, nk)$ and an integer $z > 0$.

Output: A list $\mathcal{T}_z = T_1, \dots, T_z$ of z distinct strings over Σ , such that for all $i \in [1, z]$, $\text{Substr}_k(T_i) = \mathcal{S}_k$ and for every string T' not in \mathcal{T}_z with $\text{Substr}_k(T') = \mathcal{S}_k$, $|T'| \geq |T_i|$, for all $i \in [1, z]$; or **FAIL** if that is not possible.

In particular, if $z = 1$ the problem consists in finding a shortest string X such that $\text{Substr}_k(X) = \mathcal{S}_k$. In this case, we call the problem **SHORTEST \mathcal{S}_k -EQUIVALENT STRING**. We solve z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS by considering the de Bruijn graph of order k of \mathcal{S}_k and reducing this problem to the problem of listing Eulerian walks on directed graphs. Let us first recall a few basic definitions before formally defining the problem in scope. Given \mathcal{S}_k , the *de Bruijn graph* (DBG) of order k of \mathcal{S}_k is a directed multigraph $G_{\mathcal{S}_k} = (V, E)$, where V is the set of length- $(k-1)$ substrings of the strings in \mathcal{S}_k , and $G_{\mathcal{S}_k}$ contains an edge (u, v) if and only if the string $S = u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and $S \in \mathcal{S}_k$. A *walk* in a directed graph $G(V, E)$ is a sequence of edges from E which joins a sequence of nodes from V . An *Eulerian walk* in G is a walk which visits all edges in E at least once. Any string X such that $\text{Substr}_k(X) = \mathcal{S}_k$ corresponds to an Eulerian walk W in the DBG of order k of \mathcal{S}_k and vice-versa. We formally define the problem of listing Eulerian walks in directed graphs.

z -SHORTEST EULERIAN WALKS

Input: A directed graph $G(V, E)$ and an integer $z > 0$.

Output: A list $\mathcal{W}_z = W_1, \dots, W_z$ of z distinct Eulerian walks of G , such that for every Eulerian walk W' of G not in \mathcal{W}_z , $|W'| \geq |W_i|$, for all $i \in [1, z]$; or **FAIL** if that is not possible.

If $z = 1$, we call the problem **SHORTEST EULERIAN WALK**. Let us denote the total size of \mathcal{W}_z (that is, the total length of the walks) by $||\mathcal{W}_z||$. We show the following result¹.

► **Theorem 1.** *The z -SHORTEST EULERIAN WALKS problem can be solved in:*

- $\mathcal{O}(|E||V| \log^2 |V| + z|V|^3 + ||\mathcal{W}_z||)$ time;
- or $\mathcal{O}(|E||V| \log^2 |V| + z(|E||V| + |V|^2 \log |V|) + ||\mathcal{W}_z||)$ time.

We also investigate the combinatorial bounds on the total length of z shortest Eulerian walks in directed graphs. We show the following result.

► **Theorem 2.** $||\mathcal{W}_1|| \leq |V||E|$ and this bound is asymptotically tight. Moreover, $||\mathcal{W}_z|| \leq z|V||E| + z^2|V|$ and this bound is asymptotically tight.

¹ We assume that basic arithmetic operations take constant time, which is the case when $z = \text{poly}(|E|)$.

By employing Theorem 2 we show the following result, where $||\mathcal{T}_z||$ denotes the total length of the strings output for z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS.

► **Theorem 3.** $||\mathcal{T}_1|| = \mathcal{O}(k + n^2)$. Moreover, $||\mathcal{T}_z|| = \mathcal{O}(zk + zn^2 + z^2n)$.

By employing Theorem 1 and Theorem 3 we show the following result (recalling that the size of the input in z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS is $\Theta(nk)$).

► **Theorem 4.** *The z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS problem can be solved in $\mathcal{O}(nk + n^2 \log^2 n + zn^2 \log n + ||\mathcal{T}_z||)$ time. If $z = 1$ this becomes $\mathcal{O}(nk + n^2 \log^2 n)$.*

We complement these results with the following reduction of independent interest. Given a directed graph $G(V, E)$, the DIRECTED CHINESE POSTMAN (DCP) problem (also known as the ROUTE INSPECTION problem) asks for a shortest closed walk that visits every edge of G at least once. DCP can be solved using an algorithm for computing a min-cost flow [8]. To this end, we can use the *network simplex algorithm* to solve DCP in $\tilde{\mathcal{O}}(|E||V|)$ time [28, 32]. We show here, via a non-trivial reduction, that if the SHORTEST \mathcal{S}_k -EQUIVALENT STRING problem over a *binary alphabet* has a near-linear-time solution then so does DCP.

Motivation and Related Work. The main theoretical motivation for this work comes from the following “gap” in the literature. Let \mathcal{M}_k be a *multiset* (rather than a set) of length- k strings. Counting (resp. listing) all distinct strings whose multiset of length- k strings is \mathcal{M}_k corresponds to counting (resp. listing) all node-distinct Eulerian trails (i.e., walks that do not repeat any edges) in the de Bruijn multigraph of order k of \mathcal{M}_k [16, 17, 22, 3]. Counting all node-distinct Eulerian trails can be done in polynomial time by employing the well-known BEST theorem [34] (see also [21] for the analogous result on strings). Efficient algorithms for listing z node-distinct Eulerian trails are also known [5, 24]. However, the analogous, perhaps more basic, results for counting or listing all strings whose *set* of length- k strings is \mathcal{S}_k are, to the best of our knowledge, unknown. Here we focus on listing by observing that strings whose set of length- k strings is \mathcal{S}_k correspond to Eulerian walks in the DBG of order k of \mathcal{S}_k . Counting remains wide open, as an analogous to BEST theorem for Eulerian walks is unknown. Indeed, a fundamental difference is that \mathcal{M}_k , by definition, gives the exact length of all strings whose multiset of length- k substrings is \mathcal{M}_k , while \mathcal{S}_k gives only a lower bound.

The practical motivation for this work comes from the fact that the z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS problem arises naturally as an encoding problem in many real-world applications. In data privacy, the output strings can be used to construct reverse-safe data structures for pattern matching when \mathcal{S}_k comes from a private string [2, 3]. In data compression, the output strings can be used to represent compactly a set \mathcal{S}_k of length- k strings [26]. In bioinformatics, the output strings correspond to different possible genome reconstructions [29] when \mathcal{S}_k is a set of sequences generated by a sequencing experiment.

Our work is in some sense related to Simon’s congruence [31]. Two strings are \sim_k -congruent if they have the same set of *subsequences* of length at most k . For details on the combinatorial properties of the congruence see [31, 25, 19, 18, 20, 1] and for some algorithmic works see [15, 11, 33, 7, 10, 1, 12]. A long-standing open problem was to design an algorithm which, given two strings S and T , computes the largest k for which $S \sim_k T$. Gawrychowski et al. [12] have recently settled the problem optimally by showing a linear-time algorithm.

Paper Organization. In Section 2 we provide some basic definitions and notation. In Section 3 we present the reduction from DCP to the SHORTEST \mathcal{S}_k -EQUIVALENT STRING problem. In Section 4 we show the combinatorial bounds on the length of a shortest Eulerian walk and on the total length of z shortest Eulerian walks. From these bounds, we infer the

16:4 On Strings Having the Same Length- k Substrings

bounds on the length of the strings output for z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS. In Section 5 we present our algorithm for solving the z -SHORTEST EULERIAN WALKS problem. From this algorithm, we infer our solution to z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS.

2 Preliminaries

We consider directed graphs $G(V, E)$ such that there is at most one directed edge (u, v) for any $u, v \in V$ (that is, all edges in E have multiplicity 1). For a graph $G(V, E)$, we call *multiplicity function* on G a mapping $E \rightarrow \mathbb{N}$. We denote by $G_\mu(V, E)$ the *multigraph* with *underlying graph* G and multiplicity function μ : G_μ is a version of G having $\mu(e)$ copies of each edge $e \in E$. We call G_μ an *extension* of G if $\mu > 0$. We call v the *head* of an edge (u, v) , and we call u the edge *tail*.

A *walk* in G is any sequence $W = e_1 e_2 \dots e_{|W|}$ of edges in E such that the head of e_i is equal to the tail of e_{i+1} , for all $i \in [1, |W| - 1]$; $|W|$ is the *length* of W . A walk may traverse any edge multiple times. We denote by $I(W)$ the tail of e_1 , by $L(W)$ the head of $e_{|W|}$ and by $\text{mult}_e(W)$ the number of times W visits e , for any $e \in E$. A walk W is *closed* (and in this case we call it a *cycle*) if $I(W) = L(W)$, that is, if it starts and ends at the same node. A walk W is *Eulerian* if it traverses all the edges of G at least once, that is, if the set $\{e_i\}_{i \in [1, |W|]} = E$. A walk that does not traverse any edge twice is a *trail*.

Given a multigraph G_μ , a walk W is Eulerian on G_μ if $\text{mult}_e(W) \geq \mu(e)$ for every $e \in E$ and it is an Eulerian trail on G_μ if $\text{mult}_e(W) = \mu(e)$ for every $e \in E$. A (multi-)graph $G(V, E)$ is *semi-Eulerian* if it admits an Eulerian trail, *Eulerian* if it admits an Eulerian cycle, and *strictly semi-Eulerian* if it is semi-Eulerian but not Eulerian. We denote by $EW(G)$ and $ET(G)$ the set of Eulerian walks and the set of Eulerian trails on G , respectively.

A graph $G(V, E)$ is *strongly connected* if, for any two nodes $u, v \in V$ with $u \neq v$, there exist both a walk from u to v and from v to u . The *strongly connected components* (SCCs in short) of G are its inclusion-maximal strongly connected subgraphs. A graph is *weakly connected* if replacing all of its directed edges with undirected edges yields a connected graph.

► **Definition 5 (Flow).** Let $G = (V, E)$ be a directed graph and $\delta : V \rightarrow \mathbb{Z}$ be a function called the *supply*. Let m (resp. M) be a function $E \rightarrow \mathbb{N} \cup \{+\infty\}$ called *minimal* (resp. *maximal*) *capacity*. A *flow* on G with supply δ , minimal capacity m and maximal capacity M is a function $f : E \rightarrow \mathbb{N}$ such that:

$$\begin{aligned} \forall e \in E, \quad m(e) &\leq f(e) \leq M(e) \\ \forall v \in V, \quad \sum_{e=(v,w) \in E} f(e) - \sum_{e=(w,v) \in E} f(e) &= \delta(v) \end{aligned}$$

We denote this set of flows by $\mathcal{F}(G, \delta, m, M)$.

3 Reducing Directed Chinese Postman to Shortest Equivalent String

DIRECTED CHINESE POSTMAN (DCP)

Input: A directed graph $G(V, E)$.

Output: A shortest closed Eulerian walk, or FAIL if that is not possible.

The main goal of this section is to reduce DCP to SHORTEST \mathcal{S}_k -EQUIVALENT STRING. We first show a simple linear-time reduction that uses an alphabet of size $\mathcal{O}(|V|)$. We then show a more involved near-linear-time reduction that uses a binary alphabet. The latter reduction has the following important implication: if SHORTEST \mathcal{S}_k -EQUIVALENT STRING over a binary alphabet has a near-linear-time solution, then so does DCP.

3.1 Large Alphabet

► **Lemma 6.** *Given a directed graph $G(V, E)$, we define $\tilde{G}(\tilde{V}, \tilde{E})$ such that $\tilde{V} = V \cup \{a, b\}$, where a, b are two bogus nodes, and $\tilde{E} = E \cup \{(a, u), (u, b)\}$ for an arbitrary node $u \in V$.*

Then from any shortest Eulerian walk on \tilde{G} , we can compute a shortest Eulerian closed walk on G in constant time.

Proof. Let \tilde{W} be a shortest Eulerian walk on \tilde{G} . By definition, the bogus node a does not have any ingoing edge, and then since \tilde{W} must traverse edge (a, u) , it has to start with a . By a symmetrical argument, it must end with b . The rest of the walk is on G , and since a and b are connected only to u , the latter is the second and second-to-last node crossed by \tilde{W} . The edges traversed by \tilde{W} between these two visits of u form a closed Eulerian walk W on G .

If a shorter Eulerian cycle on G would exist, we could, by a rotation, make it start and end at u . Then, we could add edges (a, u) and (u, b) at the beginning and at the end, obtaining an Eulerian walk on \tilde{G} shorter than \tilde{W} , a contradiction. Thus W is a shortest closed Eulerian walk on G , and we can compute it from \tilde{W} in constant time by removing the first and last edge traversed by \tilde{W} . ◀

► **Theorem 7.** *Any instance of DCP can be reduced to an instance of the SHORTEST \mathcal{S}_k -EQUIVALENT STRING problem in linear time.*

Proof. Let $\mathcal{I}_{\text{DCP}} = G(V, E)$ be an instance of DCP. We define $\tilde{G}(\tilde{V}, \tilde{E})$ as in Lemma 6. A walk W in \tilde{G} can be expressed as a sequence $v_0, \dots, v_{|W|}$ of nodes from \tilde{V} such that $(v_i, v_{i+1}) \in \tilde{E}$ for every $i = 0, \dots, |W| - 1$. Equivalently, such a walk W can be seen as a string $v_0 \dots v_{|W|}$ over \tilde{V} such that its set of length-2 substrings $\mathcal{S}_2(W)$ is included in \tilde{E} . By definition, W is Eulerian if and only if $\mathcal{S}_2(W)$ is exactly \tilde{E} , so finding a shortest Eulerian walk W in \tilde{G} corresponds to finding a shortest string over \tilde{V} such that $\mathcal{S}_2(W) = \tilde{E}$, which is an instance of the SHORTEST \mathcal{S}_2 -EQUIVALENT STRING problem. Finally, by Lemma 6, a solution to DCP on G can be obtained from an Eulerian walk W on \tilde{G} in constant time. ◀

3.2 Binary Alphabet

We reduce any instance $G(V, E)$ of the DIRECTED CHINESE POSTMAN problem to an instance of SHORTEST k -EQUIVALENT STRINGS over a binary alphabet $\Sigma = \{0, 1\}$; we assume that G is weakly connected, otherwise the problem has a trivial solution FAIL.

For a given integer ℓ , we denote by $\mathbf{0}_\ell$ the constant string consisting of ℓ zeros. We assign to every $v \in V$ two binary strings v_A, v_B over $\{0, 1\}$ such that:

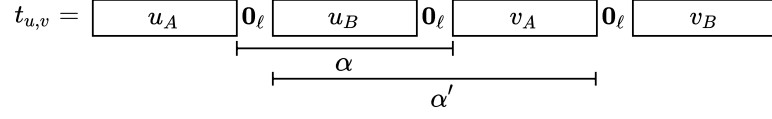
- For every $v \in V$, v_A and v_B are different from each other and from any w_A and w_B for $w \neq v$ in V , so that one can identify v by knowing only v_A or v_B .
- All strings v_A and v_B start and end with a 1, and they all have the same length ℓ .

► **Observation 8.** *The length ℓ can be chosen to be in $\mathcal{O}(\log |V|)$.*

Proof. We need two unique binary strings v_A, v_B for every $v \in V$. Since there exist 2^α distinct binary strings of length α , we seek the minimum length ℓ such that $2^{\ell-2} \geq 2|V|$, because we restrict to strings starting and ending with 1. The observation follows. ◀

Let $k = 3\ell$. Our reduction models both nodes and edges of G with appropriate string gadgets. The node gadgets are defined as length- k strings $s(v) := v_A \cdot \mathbf{0}_\ell \cdot v_B$, for every node $v \in V$. Let $S_0 := \{s(v) \mid v \in V\}$ be the set of such gadgets. We model each edge $(u, v) \in E$ as a length- (7ℓ) string gadget $s(u) \cdot \mathbf{0}_\ell \cdot s(v)$. We define the set S of length- k

16:6 On Strings Having the Same Length- k Substrings



■ **Figure 1** The configuration described in the proof of Lemma 10.

strings input to SHORTEST \mathcal{S}_k -EQUIVALENT STRING as the set of length- k substrings of all the edge gadgets: $S := \cup_{(u,v) \in E} \text{Substr}_k(s(u) \cdot \mathbf{0}_\ell \cdot s(v))$. Note that $S_0 \subseteq S$. Since $\ell = \mathcal{O}(\log |V|)$, each edge gadget has length $7\ell = \mathcal{O}(\log |V|)$ so it has $\mathcal{O}(\log |V|)$ substrings of length $k = 3\ell = \mathcal{O}(\log |V|)$. Therefore S contains $\mathcal{O}(|E| \log |V|)$ strings of length $\mathcal{O}(\log |V|)$.

► **Observation 9.** *Every occurrence of $\mathbf{0}_\ell$ in an edge gadget starts at position ℓ , 3ℓ , or 5ℓ of the edge gadget (i.e., it is one of the occurrences explicitly used for the construction). It follows that there are no spurious (i.e., not coming from one of the described occurrences in the edge gadgets) occurrences of $\mathbf{0}_\ell$ in the strings in S .*

Proof. Every string v_A or v_B starts and ends with a 1, so they cannot overlap an occurrence of $\mathbf{0}_\ell$. Since we never concatenate two copies of $\mathbf{0}_\ell$, the result follows. ◀

We now want to prove the correspondence between walks on G and strings having their length- k substrings in S . The following lemma is a first step in that direction: we show that, given a string T with $\text{Substr}_k(T) \subseteq S$, the node gadgets never overlap in T , and that their succession precisely corresponds to adjacency relations between nodes in G .

► **Lemma 10.** *Let $G(V, E)$ be a directed graph and let S be the set of length- k strings defined above. Let T be a string with $\text{Substr}_k(T) \subseteq S$. If, for some $u, v \in V$, T has a substring $t_{u,v}$ such that (i) $t_{u,v}$ has $s(u)$ as a prefix, (ii) $t_{u,v}$ has $s(v)$ as a suffix, and (iii) $t_{u,v}$ has no other substrings of the form $s(w)$ for $w \in V$, then $t_{u,v} = s(u) \cdot \mathbf{0}_\ell \cdot s(v)$ and $(u, v) \in E$.*

Proof. Let $t_{u,v}$ be a substring of T satisfying conditions (i)-(iii), so that $t_{u,v}[0..k-1] = s(u) = u_A \cdot \mathbf{0}_\ell \cdot u_B$ for some $u \in V$. Note that the length- k substring α starting at position ℓ of $t_{u,v}$ has $\mathbf{0}_\ell$ as a prefix. By the definition of S and Observation 9, every length- k string in S with $\mathbf{0}_\ell$ as a prefix has it also as a suffix, thus $\mathbf{0}_\ell$ is also a suffix of α (inspect Figure 1). Let α' be the length- k substring starting at position 2ℓ of $t_{u,v}$. String α' has u_B as a prefix, and it has an occurrence of $\mathbf{0}_\ell$ at position ℓ (the suffix of α). Since $\alpha' \in S$, and since it has a $\mathbf{0}_\ell$ in a central position, we know that α' is equal to $u_B \cdot \mathbf{0}_\ell \cdot w_A$ for some $w \in V$ such that $(u, w) \in E$. Let now β be the length- k substring of $t_{u,v}$ starting at position 3ℓ . We know that, since $\mathbf{0}_\ell$ is a prefix of β , it is also its length- ℓ suffix. Thus, the length- k substring β' starting at position 4ℓ of $t_{u,v}$, has $w_A \cdot \mathbf{0}_\ell$ as a prefix; by looking at S we find that $\beta' = s(w)$. Now, by definition of $t_{u,v}$, the string β' has to be $s(v)$. Thus, $w = v$, $(u, v) \in E$, and $t_{u,v} = s(u) \cdot \mathbf{0}_\ell \cdot s(v)$. ◀

► **Proposition 11.** *Let $G(V, E)$ be a directed graph, let S and S_0 be the sets of strings defined above, and let \mathcal{T} be the set of strings of length at least $k+1$, having prefix and suffix in S_0 and such that $\text{Substr}_k(T) \subseteq S$ for all $T \in \mathcal{T}$. The mapping*

$$\varphi : W = ((v_0, v_1), (v_1, v_2), \dots, (v_{R-1}, v_R)) \mapsto T = s(v_0) \cdot \mathbf{0}_\ell \cdot s(v_1) \cdot \dots \cdot \mathbf{0}_\ell \cdot s(v_R)$$

defines a bijection between the set of walks on G and \mathcal{T} . From any $T \in \mathcal{T}$, $\varphi^{-1}(T)$ can be computed in $\mathcal{O}(|T|)$ time. Moreover, given a walk W , the set of edges traversed by W is exactly the set of $(u, v) \in E$ such that $u_B \cdot \mathbf{0}_\ell \cdot v_A$ are substrings of $\varphi(W)$.

Proof. The mapping is well defined between the given sets and it is injective by the definition of the string gadgets. Consider an arbitrary string $T \in \mathcal{T}$. Let $u, v \in V$ be nodes such that $s(u) \in S_0$ is a prefix of T and $s(v) \in S_0$ is a suffix of T . Let t be the second leftmost occurrence of any string from S_0 in T (the prefix and suffix of T are in S_0 , and since the length of T is at least $k + 1$, they do not coincide, thus there are at least two occurrences of strings from S_0). We have $t = s(w)$ for some $w \in V$, and then T has a prefix $t_{u,w}$ satisfying conditions (i)-(iii) of Lemma 10. Then $(u, w) \in E$, and $t_{u,w} = s(u) \cdot \mathbf{0}_\ell \cdot s(w)$. By induction, we can find a sequence of nodes $u_0 = u, u_1 = w, \dots, u_R = v$ such that $T = s(u_0) \cdot \mathbf{0}_\ell \cdot \dots \cdot \mathbf{0}_\ell \cdot s(u_R)$, and such that $(u_i, u_{i+1}) \in E$ for every $i = 0, \dots, R - 1$. This gives us a walk $W = u_0, u_1, \dots, u_R$ on G with $\varphi(W) = T$. The mapping φ is therefore surjective, hence it is a bijection. Its inverse map can be computed in $\mathcal{O}(|T|)$ time, by storing the encodings $s(v)$ for $v \in V$ in a dictionary, and linearly constructing the list of v from the occurrences of $s(v)$ in T . ◀

We are now ready to prove the main result of this section.

► **Theorem 12.** *Any instance $G(V, E)$ of DIRECTED CHINESE POSTMAN with output W can be reduced in $\mathcal{O}(|E| \log |V| + |W|)$ time to an instance of SHORTEST \mathcal{S}_k -EQUIVALENT STRING with $n = |\mathcal{S}_k| = \mathcal{O}(|E| \log |V|)$ strings over the binary alphabet and $k = \mathcal{O}(\log |V|)$.*

Proof. Let $G(V, E)$ be a directed graph. We construct a graph \tilde{G} from G with bogus nodes a and b as in Lemma 6 and we construct sets S_0 and S on \tilde{G} as described at the beginning of this section. S contains $\mathcal{O}(|E| \log |V|)$ strings, each of them having length $\mathcal{O}(\log |V|)$ bits, and it can be constructed in $\mathcal{O}(|E| \log |V|)$ time by listing the binary encoding of the integers we assign to the nodes. We now prove that we can compute a shortest Eulerian walk in \tilde{G} from the output of SHORTEST \mathcal{S}_k -EQUIVALENT STRING, and Lemma 6 concludes the proof.

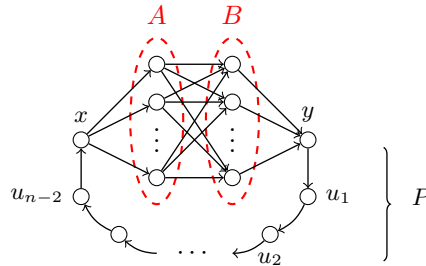
Indeed, from the bijection in Proposition 11, every string in \mathcal{T} gives rise to a unique walk in \tilde{G} . Let T be a solution to SHORTEST \mathcal{S}_k -EQUIVALENT STRING with $\mathcal{S}_k = S$. Since node a does not have any ingoing edge in \tilde{G} , by the definition of S , the length- $(k - 1)$ prefix of $s(a)$ cannot be a suffix of any string in S . Since T contains every string in S , the string $s(a)$ must be a prefix of T . By symmetry, the string $s(b)$ is a suffix of T . Finally, since $s(a) \neq s(b)$ by definition of the node gadgets, T has length at least $k + 1$ and therefore $T \in \mathcal{T}$.

Now, by Proposition 11 we can get in $\mathcal{O}(|T|)$ time a unique walk $W = v_0, \dots, v_R$ such that $T = s(v_0) \cdot \mathbf{0}_\ell \cdot s(v_1) \cdot \dots \cdot s(v_R)$. The edges traversed by W are exactly edges $(u, v) \in \tilde{E}$ such that $u_B \cdot \mathbf{0}_\ell \cdot v_A$ is a substring of T . But since T contains every string from S , and since $u_B \cdot \mathbf{0}_\ell \cdot v_A \in S$ for every $(u, v) \in \tilde{E}$, the walk W traverses every edge in \tilde{G} and it is therefore Eulerian. Furthermore, it is minimal, as otherwise some shorter Eulerian walk W' would give rise to $T' = \varphi(W')$ shorter than T . But then T' would be a shorter string with $\text{Substr}_k(T') = S$. It follows that $W = \varphi^{-1}(T)$ is a shortest Eulerian walk of \tilde{G} . By looking at the bijection of Proposition 11, we get $|T| = \mathcal{O}(|W|k) = \mathcal{O}(|W| \log |V|)$ bits, so the total reduction time is $\mathcal{O}(|E| \log |V| + |W|)$. ◀

4 Combinatorial Bounds

The main goal of this section is to prove Theorem 2; in particular, to provide bounds on the quantities $\|\mathcal{W}_1\|$ and $\|\mathcal{W}_z\|$ for a directed graph $G(V, E)$. In Section 4.1, we prove the upper bound $\|\mathcal{W}_1\| \leq |V| \cdot |E|$, and show that it is asymptotically tight. In Section 4.2, we prove the bound $\|\mathcal{W}_z\| \leq z|V| \cdot |E| + z^2|V|$, and show that it is asymptotically tight as well. Finally, in Section 4.3, we employ Theorem 2 to prove Theorem 3.

Firstly, let us observe that there are three types of graphs to consider.



■ **Figure 2** A directed graph where the shortest Eulerian walk has length $\Omega(|V||E|) = \Omega(|V|^3)$.

► **Observation 13.** *Every directed graph has either 0, 1, or infinitely many distinct Eulerian walks.*

Proof. Any directed path has exactly 1 Eulerian walk. It is easily seen that any other directed acyclic graph has none, and that any directed graph with 2 or more Eulerian walks must have a directed cycle:² This cycle may be repeated any number of times, yielding infinitely many distinct Eulerian walks. ◀

As the first two types of graphs are trivial with respect to this analysis, in the rest of the section we focus on graphs having infinitely many distinct Eulerian walks.

4.1 Length of a Shortest Eulerian Walk

► **Lemma 14.** *For any directed graph $G(V, E)$, we have $||\mathcal{W}_1|| \leq |V| \cdot |E|$.*

Proof. Let us first assume that G is strongly connected. Letting $e_1, \dots, e_{|E|}$ be the edges of E , we inductively build a walk which traverses all of these edges in the given order, and which has length at most $|V| \cdot |E|$. The walk starts as the single edge e_1 . Inductively, given a walk W' that crosses e_1, \dots, e_{h-1} and traverses at most $|V| \cdot (h-1)$ edges for any $h \in [2, |E|]$, there is a walk W connecting the head of e_{h-1} to the tail of e_h , since G is strongly connected. Moreover, W traverses at most $|V| - 1$ edges: indeed, if W goes through the same node twice, we can remove the cycle that it forms doing so. As a result, $W'W e_h$ is a walk that crosses e_1, \dots, e_h and traverses at most $|V| \cdot h$ edges. The claim follows when $h = |E|$, noting that a shortest walk cannot be longer than the one that we have just described inductively.

Consider the general case of G and $\mathcal{W}_1 = \{W\}$. Walk W induces a topological order on the SCCs of G , as otherwise the walk cannot traverse all the edges, and these SCCs form a chain graph of the form $G_1 \rightarrow G_2 \rightarrow \dots \rightarrow G_k$ (see [5]). Since each $G_i(V_i, E_i)$ is strongly connected, we can apply the above argument, so that W traverses at most $|V_i| \cdot |E_i|$ edges there. Overall, since $|E| = k - 1 + \sum_{i=1}^k |E_i|$, W traverses at most $k - 1 + \sum_{i=1}^k |V_i||E_i| \leq k - 1 + |V| \cdot \sum_{i=1}^k |E_i| \leq |V| \cdot |E|$ edges. ◀

We now show in Lemma 15 that the bound in Lemma 14 is asymptotically tight by providing an example of a graph G with $||\mathcal{W}_1|| = \Omega(|V| \cdot |E|) = \Omega(|V|^3)$.

► **Lemma 15.** *There is an infinite family of directed graphs, such that each graph $G(V, E)$ satisfies $||\mathcal{W}_1|| = \Omega(|V| \cdot |E|)$.*

² Note that some directed graphs with cycles may still allow 0 Eulerian walks.

Proof. We construct G as follows for any given $n \geq 3$ (inspect Figure 2): we start from two sets A, B of n nodes each, and the set of n^2 edges $F = \{(a, b) \mid a \in A, b \in B\}$ (i.e., A, B induce a complete directed bipartite graph with edges directed from A to B). We then add two more nodes x, y and add edges (x, a) for all $a \in A$, and edges (b, y) for all $b \in B$. We further connect y to x by adding $n - 2$ new nodes u_1, \dots, u_{n-2} and all edges of $P = \{(y, u_1)\} \cup \{(u_{n-2}, x)\} \cup \{(u_i, u_{i+1}) \mid i = 1, \dots, n - 3\}$. In total, this graph has $|V| = 3n$ nodes and $|E| = n^2 + 3n - 1 = \Theta(|V|^2)$ edges.

Let us now see why the length of a shortest Eulerian walk W of G is $\Omega(|V| \cdot |E|)$. By definition, \mathcal{W}_1 must traverse all the n^2 edges in the set of edges F . Without loss of generality, let $(a_1, b_1), \dots, (a_{n^2}, b_{n^2})$ denote the order in which W traverses the edges in F . Let $i < n^2$, and let us consider the point when W has just traversed (a_i, b_i) for the first time: to be able to reach the next (a_{i+1}, b_{i+1}) , the walk necessarily needs to traverse the whole of P . This means that W needs to traverse all $n - 1$ edges of P once for each $i = 1, \dots, n^2 - 1$. Furthermore, it must also traverse at least one time all the remaining $2n$ edges of the form (x, a) for all $a \in A$, and (y, b) for all $b \in B$. This leads to a total length of at least $(n - 1)(n^2 - 1) + 2n = n^3 - n^2 + n + 1 = \Omega(|V| \cdot |E|)$, proving the claim. ◀

4.2 Total Length of z Shortest Eulerian Walks

► **Lemma 16.** *For any directed graph $G(V, E)$, if $c_1 \leq \dots \leq c_z$ are the lengths of the walks in \mathcal{W}_z , then $c_i \leq |V|(i - 1 + |E|)$, for all $i \in [1, z]$, and we have $|\mathcal{W}_z| \leq z|V| \cdot |E| + z^2|V|$.*

Proof. First observe that if G is acyclic, then it has at most one Eulerian walk, and by Lemma 14 its length is at most $|V| \cdot |E|$, so the claim holds for $z = 1$.

We can then assume that G has a cycle, let C be the shortest one. Given an Eulerian walk W , the length of the next shortest walk is always bounded by $|W| + |C|$. In fact, let us consider a walk W' defined starting from W , to which we add a tour of cycle C when W traverses for the first time a node of C . Walk W' is Eulerian as it contains W , and it has length exactly $|W| + |C|$. The next shortest walk with respect to W cannot be longer than W' , and so its length must be bounded by $|W| + |C| \leq |W| + |V|$.

Now we can prove by a simple induction that the i -th shortest length for an Eulerian walk is at most $|\mathcal{W}_1| + (i - 1)|V| \leq |V|(i - 1 + |E|)$, where the base case $i = 1$ is trivial (and bounded by Lemma 14) and the inductive step holds as the length of the shortest longer walk increases by at most $|V|$. Hence, $|\mathcal{W}_z| \leq \sum_{i=1}^z (|\mathcal{W}_1| + (i - 1)|V|) \leq z|V||E| + z^2|V|$. ◀

We now show in Lemma 17 that the bound in Lemma 16 is asymptotically tight.

► **Lemma 17.** *There are infinite families of directed graphs, such that each graph $G(V, E)$ satisfies either $|\mathcal{W}_z| = \Omega(z|V| \cdot |E|)$ or $|\mathcal{W}_z| = \Omega(z^2|V|)$, respectively.*

Proof. As clearly the i -th shortest Eulerian walk, for any i , is not shorter than the shortest one, we have $|\mathcal{W}_z| \geq z \cdot |\mathcal{W}_1|$, so the first family follows from Lemma 15 (shown in Figure 2). As for the second, simply consider a directed cycle with an incoming pendant edge and an outgoing one: more formally, the directed cycle $(v_1, v_2), \dots, (v_{|V|-2}, v_1)$ of length $|V| - 2$, plus two nodes x, y and the edges (x, v_1) and (v_1, y) . Any Eulerian walk must start from the source node x , traverse the whole cycle at least once (say, $i > 0$ times), and then follow the edge (v_1, y) . As different walks must traverse the cycle a different number of times, it follows that the i -th shortest Eulerian walk has length $2 + i(|V| - 2)$, so $|\mathcal{W}_z| = \sum_{i=\{1, \dots, z\}} (2 + i(|V| - 2)) = \Omega(z^2|V|)$. ◀

We can conclude from Lemma 17 that a better worst-case bound than $\Omega(z|V| \cdot |E| + z^2|V|)$ is not possible, so Lemmas 14, 15, 16, 17 yield directly Theorem 2.

4.3 Total Length of z Shortest Equivalent Strings

Let us now prove Theorem 3, which infers upper bounds on the total length of a solution to z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS. We use the following observation:

► **Observation 18.** *Let \mathcal{S}_k be a set of n strings each of length k , such that there exists a string T with $\text{Substr}_k(T) = \mathcal{S}_k$. Let $G(V, E)$ be the dBG of order k of \mathcal{S}_k . It holds (i) $|E| = n$; and (ii) $|V| \leq n + 1$.*

Proof. (i) It follows from the definition of dBG: each edge corresponds to a string from \mathcal{S}_k and reciprocally, if $S \in \mathcal{S}_k$ there is an edge between $S[0..k-2]$ and $S[1..k-1]$, which corresponds to S . Therefore $|E| = |\mathcal{S}_k| = n$.

(ii) Since there exists T with $\text{Substr}_k(T) = \mathcal{S}_k$, we know that there is a walk W traversing every edge in G . It follows that all $e \in E$ (except possibly for the first and last edges traversed by W) are such that the head of e coincides with the tail of some $e' \in E$, and symmetrically, the tail of e coincides with the head of some $e'' \in E$. We conclude that there cannot be more than $|E| + 1$ distinct nodes, thus $|V| \leq |E| + 1 = n + 1$. ◀

Let \mathcal{S}_k be a set of n strings each of length k ; we know that if $G(V, E)$ is the dBG of order k of \mathcal{S}_k , a (shortest) string T such that $\text{Substr}_k(T) = \mathcal{S}_k$ has length $k - 1 + |W|$, where W is a (shortest) Eulerian walk on G . From Theorem 2 and Observation 18 we get that $\|\mathcal{W}_1\| \leq |V| \cdot |E| \leq n^2 + n$ and so $\|\mathcal{T}_1\| \leq k - 1 + n^2 + n = \mathcal{O}(k + n^2)$. Using again Theorem 2 and Observation 18, we get that $\|\mathcal{W}_z\| \leq z|V| \cdot |E| + z^2|V|$ and so $\|\mathcal{T}_z\| \leq z(k - 1 + n^2 + n) + z^2(n + 1) = \mathcal{O}(zk + zn^2 + z^2n)$. We have arrived at Theorem 3.

5 Listing z Shortest Eulerian Walks

The main goal of this section is to prove Theorem 1; in particular, to provide an efficient algorithm for solving z -SHORTEST EULERIAN WALKS. In Section 5.1, we start by providing the state-of-the-art results for listing z best flows in a directed graph; the underlying algorithms form the main computational routine of our algorithm, which is provided in detail next. Finally, in Section 5.2, we employ Theorem 1 and Theorem 3 to prove Theorem 4; namely, to provide an efficient algorithm for solving z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS.

5.1 Main Algorithm

In general, we can equip a graph $G(V, E)$ with a *cost* function $C : E \rightarrow \mathbb{N}$. Given a flow $f \in \mathcal{F}(G, \delta, m, M)$ for some supply function δ , and minimal (resp. maximal) capacities function m (resp. M), the *cost of f* is $C(f) = \sum_{e \in E} C(e)f(e)$. Let us now formally define the problem of listing z best flows (i.e., flows of minimal cost) from the set $\mathcal{F}(G, \delta, m, M)$ of all feasible flows (with respect to the given conditions) in a directed graph.

z -BEST FLOWS

Input: A directed graph $G = (V, E)$, a supply function δ , a minimal capacity function m , a maximal capacity function M , a cost function C (all taking finite values), a min-cost flow f , and an integer $z > 0$.

Output: A list F of z flows in $\mathcal{F}(G, \delta, m, M)$, such that for every flow f' not in F , $C(f') \geq C(F[i])$, for all $i \in [0, z - 1]$, and ordered such that $C(F[0]) \leq \dots \leq C(F[z - 1])$; or FAIL if that is not possible.

A min-cost flow f can be computed for any G using one of the following results.

► **Lemma 19** ([13, 27, 28, 32]). *Given any directed graph $G(V, E)$, computing a min-cost flow takes $\mathcal{O}(|E| \log |V|)(|E| + |V| \log |V|)$ time or $\mathcal{O}(|E||V| \log |V| \log K|V|)$ time, where K is the maximum cost of any edge of G .*

The following recent result for the z -BEST FLOWS problem is known [23]; see also [14, 30].

► **Lemma 20** ([23]). *The z -BEST FLOWS problem can be solved in $\mathcal{O}(z|V|^3)$ time or in $\mathcal{O}(z(|E||V| + |V|^2 \log |V|))$ time.*

Main Idea. We list Eulerian walks as follows: we construct semi-Eulerian multigraphs G_μ , which are extensions of G obtained by duplicating some edges. Recall that each Eulerian walk in G can be seen as a trail in the semi-Eulerian extension G_μ for some multiplicity function μ such that, for every edge e , $\mu(e) = \text{mult}_e(W)$. We will therefore treat Eulerian walks on G and Eulerian trails on G_μ as if they were the same objects, and use the fact that these semi-Eulerian extensions of G form a partition of the sets of Eulerian walks on G . Our algorithm has two steps: first, listing the extensions, and then, listing the trails in each extension, using the algorithm given in [5]. For the first part, we need the following:

► **Proposition 21.** *Let $G = (V, E)$ be a directed graph and let $M = \mathbb{N}^E$ be the set of all possible multiplicity functions on G . We have the following set equality:³*

$$EW(G) = \bigsqcup_{\mu \in M} ET(G_\mu) \quad (1)$$

Proof. An Eulerian walk W on G is an Eulerian trail on the semi-Eulerian multigraph obtained by taking as many copies of each edge as the number of times W traverses it. ◀

To list semi-Eulerian extensions, we will use flows. For Eulerian extensions, the balancing conditions are precisely a flow problem, as specified in the following proposition.

► **Proposition 22.** *Let G be a directed weakly connected graph, and μ be a multiplicity function on G . The multigraph G_μ is Eulerian if and only if the balancing conditions hold:*

$$\sum_{(u,w) \in E} \mu(u, w) - \sum_{(w,u) \in E} \mu(w, u) = 0 \quad \text{for any fixed } u \in V$$

and is semi-Eulerian if the equality above holds or if it exists $(a, b) \in V$ such that:

$$\begin{aligned} \sum_{(u,w) \in E} \mu(u, w) - \sum_{(w,u) \in E} \mu(w, u) &= 0 && \text{for any fixed } u \in V \setminus \{a, b\} \\ \sum_{(a,w) \in E} \mu(a, w) - \sum_{(w,a) \in E} \mu(w, a) &= 1 && \sum_{(b,w) \in E} \mu(b, w) - \sum_{(w,b) \in E} \mu(w, b) = -1 \end{aligned}$$

Proof. This is a reformulation of the well-known Euler's theorem [9] in the directed case with multiplicities. ◀

For any directed graph $G(V, E)$ and for $v \in V$, we write $\mathbb{1}_v$ for the indicator function on V of v ; namely, for every $u \in V$, $\mathbb{1}_v(u) = 1 \iff u = v$. We define:

- For every $v, w \in V$, a supply function $\delta_{v,w} : u \mapsto \mathbb{1}_v(u) - \mathbb{1}_w(u)$. If $v = w$, this is the null function, and if $v \neq w$, the function has value 1 on v , -1 on w , and 0 on any other node.
- For every $n \in \mathbb{N} \cup \{\infty\}$, a function c_n on E constantly equal to n .

³ We use squared cup to underline that the sets $ET(G_\mu)$ are pairwise disjoint.

16:12 On Strings Having the Same Length- k Substrings

► **Observation 23.** An extension G_μ of a directed weakly connected graph G with multiplicities μ is semi-Eulerian if and only if

$$\mu \in \bigcup_{(v,w) \in V^2} \mathcal{F}(G, \delta_{v,w}, c_1, c_\infty)$$

Proof. This is a reformulation of the balancing conditions (Proposition 22). ◀

The semi-Eulerian extensions of G exactly correspond to flows on G with supply function of the form $\delta_{v,w}$, with $(v,w) \in V^2$. However, we would like to treat all cases by solving a single flow problem in order to avoid solving $\mathcal{O}(|V|^2)$ of them separately.

Let $G = (V, E)$ be a directed graph, and $G_{s,t} = (V_{s,t}, E_{s,t})$ be an extension of G with $V_{s,t} = V \cup \{s, t\}$ for some $s, t \notin V$ and $E_{s,t} = E \cup (\bigcup_{v \in V} (s, v)) \cup (\bigcup_{v \in V} (v, t))$. We will compute flows on $G_{s,t}$ by defining a maximal capacity function equal to c_∞ on every edge, a minimal capacity function m with $m|_E = c_1$ and $m|_{(E_{s,t} \setminus E)} = c_0$, and a supply function $\delta_{s,t}$.

Because we set the *minimal* capacity to be 1 on every edge in E , and since the flow can enter and exit G from any node via the edges from s to every $v \in V$, there is almost an equivalence between flows in $\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ and semi-Eulerian extensions of G , except for the following detail: a flow also contains information about the starting and ending point of some trail in the extension, as it specifies which nodes are connected to the bogus nodes. So multiple flows that differ only in the edges they use to connect to the bogus nodes correspond to the same extension of G : this happens when the extension is in fact Eulerian.

► **Proposition 24.** Let $G = (V, E)$ be a directed graph and let $f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$. There is a unique $i \in V$ such that $f(s, i) = 1$, and a unique $o \in V$ such that $f(o, t) = 1$. The flow f takes value 0 on every other edge with tail s or head t .

Proof. The node s has supply 1. Since there are no ingoing edges for s , the outgoing flow has to be 1, so only one node i has $f(s, i) = 1$. The node t has supply -1 . Since there are no outgoing edges for t , the ingoing flow has to be 1, so only one node o has $f(o, t) = 1$. ◀

We call $i = \mathbf{en}(f)$ the *entrance* of f and $o = \mathbf{ex}(f)$ its *exit*. Note that $\mathbf{en}(f)$ and $\mathbf{ex}(f)$ are not necessarily distinct. We can now formally describe the relation between the flows in $\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ and the walks in G . We will make use of a function `WalkToFlow` that takes a walk on G and returns a specific flow on $G_{s,t}$, as defined in the following proposition.

► **Proposition 25.** Let $G = (V, E)$ be a directed weakly connected graph and $G_{s,t}$ the graph extended with the additional nodes s, t , as defined above. For each Eulerian walk W on G with multiplicity function μ , there is a unique flow $\text{WalkToFlow}(W) \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ such that (i) $\text{WalkToFlow}(W)|_E = \mu$, (ii) $\mathbf{en}(\text{WalkToFlow}(W)) = I(W)$ and (iii) $\mathbf{ex}(\text{WalkToFlow}(W)) = L(W)$. For every multiplicity function μ , we then have the following partition:

$$ET(G_\mu) = \bigsqcup_{\substack{f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty) \\ f|_E = \mu}} \text{WalkToFlow}^{-1}(f)$$

In particular, the sets $\text{WalkToFlow}^{-1}(f)$ for $f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ are nonempty and pairwise disjoint.

Proof. Let μ be a multiplicity function such that an Eulerian walk W with multiplicity μ exists on G , or in other terms, that G_μ is semi-Eulerian. From Observation 23 we know that μ is a flow in $\mathcal{F}(G, \delta_{v,w}, c_1, c_\infty)$ for some nodes $v, w \in V$, namely $v = I(W)$, $w = L(W)$. It is easy to verify that conditions (i)-(iii) uniquely define the following flow:

$$\text{WalkToFlow}(W) : e \mapsto \begin{cases} \mu(e) & \text{if } e \in E \\ 1 & \text{if } e = (s, I(W)) \text{ or } e = (L(W), t) \\ 0 & \text{otherwise} \end{cases}$$

Now, for every flow $f_0 \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$, the set $\text{WalkToFlow}^{-1}(f_0)$ is the set of Eulerian trails in $G_{(f_0)|_E}$ starting at $\mathbf{en}(f_0)$ and ending at $\mathbf{ex}(f_0)$. To get all the Eulerian trails having a given multiplicity μ , we need to consider all the flows agreeing with μ on G , regardless of their entrance or exit. Hence we have:

$$ET(G_\mu) = \bigsqcup_{\substack{f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty) \\ f|_E = \mu}} \text{WalkToFlow}^{-1}(f)$$

The sets $\text{WalkToFlow}^{-1}(f)$ for $f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ are pairwise disjoint because they are defined as reciprocal sets for the mapping WalkToFlow , and are nonempty. Indeed, for a flow $f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$, the balancing conditions hold in $G_{f|_E}$ (Observation 23). One can then choose an Eulerian trail W in $G_{f|_E}$ starting at $\mathbf{en}(f)$ and ending at $\mathbf{ex}(f)$. We then obtain a partition of $ET(G_\mu)$. ◀

To compute shortest walks, we need to assign costs to the flows, which are equal to the lengths of the corresponding walks. This is achieved by defining a cost function C_{walks} on $G_{s,t}$ such that $C_{\text{walks}}|_E = c_1$ (the function constantly equal to 1), and $C_{\text{walks}}|_{E_{s,t} \setminus E} = c_0$ (the function constantly equal to 0). Note that this definition coincides with the definition of the minimal capacity function m that we use, but we distinguish the two functions for clarity.

► **Observation 26.** *Let us equip graph $G_{s,t}$ with the cost function C_{walks} . For any Eulerian walk W on G , the total cost of the flow $\text{WalkToFlow}(W)$ on $G_{s,t}$ is the length of W .*

Proof. By the definition of WalkToFlow (Proposition 25), for a walk W of length ℓ we have

$$\sum_{e \in E_{s,t}} \text{WalkToFlow}(W)(e)C(e) = \sum_{e \in E} \text{WalkToFlow}(W)(e) = \ell \quad \blacktriangleleft$$

► **Proposition 27.** *Let $G(V, E)$ be a directed weakly connected graph. It holds*

$$EW(G) = \bigsqcup_{f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)} \text{WalkToFlow}^{-1}(f)$$

Proof. It follows directly from Proposition 21 and Proposition 25. ◀

Let $G(V, E)$ be a directed graph, $z \geq 1$ be an integer, and $A \in \mathbb{N} \cup \{\infty\}$. By $\mathcal{F}_{z,A}$ we denote the $\min(z, |\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_A)|)$ minimal cost flows (with cost function C_{walks} and an arbitrary but fixed order on flows having the same total cost) in $\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_A)$.

We now prove that we can list z shortest Eulerian walks in a directed graph G by computing only $\mathcal{F}_{z,\infty}$.

16:14 On Strings Having the Same Length- k Substrings

► **Proposition 28.** *Let $G = (V, E)$ be a directed weakly connected graph and $z > 0$ be an integer. We equip the extension $G_{s,t}$ with the cost function C_{walks} . Consider the set of flows $\mathcal{F}_{z,\infty}$ and a subset F_s of $\mathcal{F}_{z,\infty}$ containing exactly one representative of each class in $\mathcal{F}_{z,\infty}$ under the relation $f \sim f' \iff f|_E = f'|_E$.*

Then, each of the $\min(z, |EW(G)|)$ shortest Eulerian walks on G has multiplicities $f|_E$ for some $f \in F_s$, i.e., each of them is an Eulerian trail in some extension $G_{f|_E}$ with $f \in F_s$.

Proof. We prove that each of the $\min(z, |EW(G)|)$ shortest Eulerian walks on G corresponds to trails in the following set:

$$\mathbf{B} = \bigsqcup_{f \in F_s} ET(G_{f|_E})$$

From the definition of F and Observation 26, the walks in \mathbf{B} are minimal, in the sense that any walk in $EW(G) \setminus \mathbf{B}$ is at least as long as any walk in \mathbf{B} . This implies the result if $|\mathbf{B}| \geq \min(z, |EW(G)|)$. To prove that $|\mathbf{B}| \geq \min(z, |EW(G)|)$, we will show that if $|\mathbf{B}| < z$, then $\mathbf{B} = EW(G)$, and this will complete the proof. If $|\mathbf{B}| < z$, we have:

$$\begin{aligned} \mathbf{B} &= \bigsqcup_{f \in F_s} ET(G_{f|_E}) = \bigsqcup_{f \in F_s} \bigsqcup_{f' \sim f} \text{WalkToFlow}^{-1}(f') && \text{(by Proposition 27)} \\ &\supseteq \bigsqcup_{f \in F_s} \bigsqcup_{\substack{f' \in \mathcal{F}_{z,\infty} \\ f' \sim f}} \text{WalkToFlow}^{-1}(f') = \bigsqcup_{f \in \mathcal{F}_{z,\infty}} \text{WalkToFlow}^{-1}(f), \text{ so} \\ |\mathbf{B}| &\geq \sum_{f \in \mathcal{F}_{z,\infty}} |\text{WalkToFlow}^{-1}(f)| \geq |\mathcal{F}_{z,\infty}| \end{aligned}$$

where the last inequality follows from the fact that we are considering the disjoint union of nonempty sets (Proposition 25). Recall that $|\mathcal{F}_{z,\infty}| = \min(z, |\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)|)$, so if $|\mathbf{B}| < z$, then $|\mathcal{F}_{z,\infty}| < z$ and $\mathcal{F}_{z,\infty} = \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$. By Proposition 27, it follows that

$$|EW(G)| = \left| \bigsqcup_{f \in \mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)} \text{WalkToFlow}^{-1}(f) \right| \leq |\mathbf{B}|,$$

so $|\mathbf{B}| \geq |EW(G)|$, and in fact $\mathbf{B} = EW(G)$ from the trivial inclusion. ◀

In order to compute flows in $\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ by means of existing algorithms, we need to define an equivalent problem having finite maximal capacity on each edge, as the known algorithms are not constructed for infinite maximal capacities.

► **Lemma 29.** *Let $G(V, E)$ be a directed graph and $z \geq 1$ be an integer. We have that $\mathcal{F}_{z,\infty} = \mathcal{F}_{z, |V|(z-1+|E|)}$.*

Proof. Let $F = f_0, \dots$ (resp. $F' = f'_0, \dots, f'_N$) be the list of feasible flows for the maximal capacity function c_∞ (resp. $c_{|V|(z-1+|E|)}$), ordered by increasing cost and such that the order on flows having the same total cost is arbitrary but fixed. Note that the list F may be infinite. Each flow in the list F' is trivially in the list F . If $F = F'$ the lemma holds. Otherwise, let f_i be the minimal cost flow in F which is not in F' , so that $f_j = f'_j$ for every $j < i$. If $i \leq z$, then by Proposition 27 there is at least one Eulerian walk W in $\text{WalkToFlow}^{-1}(f_i)$, and this walk has length $C_{\text{walks}}(f_i)$. But from Lemma 16, the length of W is also at most $|V|(z-1+|E|)$, so $f_i \in F'$ (since the flows which appear strictly before in the list F' are less than z), which gives a contradiction. Therefore the first mismatch (if any) between F and F' has to be after the z first elements, and $\mathcal{F}_{z,\infty} = \mathcal{F}_{z, |V|(z-1+|E|)}$. ◀

Algorithm 1 EW-List($G(V, E), z$).

```

1: if  $G$  is not weakly connected then
2:   return FAIL
3:  $s \leftarrow |V| + 1; t \leftarrow |V| + 2; V_{s,t} \leftarrow V \cup \{s, t\}$  ▷ Add two bogus nodes
4:  $E_{s,t} \leftarrow E \cup \{(s, v), (v, t) \mid v \in V\}$  ▷ Add  $2|V|$  bogus edges
5: Construct  $G_{s,t}, \delta_{s,t}, m, c_{|V|(z-1+|E|)}$ , and  $C_{\text{walks}}$  accordingly
6:  $f \leftarrow \text{Best-Flow}(G_{s,t}, \delta_{s,t}, m, c_{|V|(z-1+|E|)}, C_{\text{walks}})$  ▷ Lemma 19
7:  $F \leftarrow z\text{-Best-Flows}(G_{s,t}, \delta_{s,t}, m, c_{|V|(z-1+|E|)}, C_{\text{walks}}, f, z)$  ▷ Lemma 20
8:  $F_s \leftarrow \text{list}(f|_E \text{ for } f \in F)$  ▷ We keep only the values flows take on the initial graph
9: Remove duplicates in  $F_s$  if any ▷ Duplicates might arise from the loss of information
10:  $\text{EW} \leftarrow \emptyset; i \leftarrow 0;$ 
11: while  $|\text{EW}| < z$  do
12:    $\text{EW} \leftarrow \text{EW} \cup \{\text{ET-List}(G_{F_s[i]}, z - |\text{EW}|)\}$  ▷ List Eulerian trails on extended  $G$  [5, 24]
13:    $i \leftarrow i + 1;$  ▷ Retrieve the next best flow
14:   if  $i = |F_s|$  then ▷ We have consumed all best flows
15:     break;
16: if  $|\text{EW}| \geq z$  then
17:   return EW
18: else
19:   return FAIL

```

► **Lemma 30.** *Given a directed graph $G = (V, E)$ and an integer z , Algorithm 1 terminates and solves the z -SHORTEST EULERIAN WALKS problem on G .*

Proof. Algorithm 1 computes z flows with minimal cost in $\mathcal{F}(G_{s,t}, \delta_{s,t}, m, c_\infty)$ (for the cost function C_{walks}) as defined in Proposition 28, or all of them if there are less than z (Line 7), from Lemma 29. The set F_s is defined in Lines 8 and 9 as in Proposition 28; and in Line 12, the function ET-List computes, if they exist, the $z - |\text{EW}|$ shortest elements from $\text{EW}(G_{f|_E})$ for every $f \in F_s$ (where $|\text{EW}|$ is, at each step, the number of Eulerian walks already computed), so we know that no more than z Eulerian walks are computed in the end. The function is implemented by means of the algorithm provided in [5] or the one in [24]. The correctness then directly follows from the equivalence proved in Proposition 28. ◀

► **Lemma 31.** *Given a directed graph $G = (V, E)$ and an integer z , Algorithm 1 requires:*

- $\mathcal{O}(|E||V| \log^2 |V| + z|V|^3 + ||\mathcal{W}_z||)$ time;
- or $\mathcal{O}(|E||V| \log^2 |V| + z(|E||V| + |V|^2 \log |V|) + ||\mathcal{W}_z||)$ time.

Proof. Computing a min-cost flow takes $\mathcal{O}(|E||V| \log^2 |V|)$ time by Lemma 19 because the maximum cost of any edge is 1. Finding z flows with minimal cost (or all of them if there are less) takes $\mathcal{O}(z|V|^3)$ time or $\mathcal{O}(z(|E||V| + |V|^2 \log |V|))$ time by Lemma 20. Listing z Eulerian trails takes $\mathcal{O}(|\mathcal{W}_z|)$ time by applying the algorithm of [5] or the one of [24]. ◀

Lemmas 30 and 31 imply Theorem 1.

5.2 Listing z Shortest Equivalent Strings

Any instance of the z -SHORTEST \mathcal{S}_k -EQUIVALENT STRINGS problem can be reduced to some instance of the z -SHORTEST EULERIAN WALKS problem in linear time. In particular, this reduction consists in constructing the dBG of order k of \mathcal{S}_k in $\mathcal{O}(nk)$ time [4]. The resultant

DBG is the directed graph $G(V, E)$ given as input to z -SHORTEST EULERIAN WALKS. By Observation 18, the total number of nodes in V is $\mathcal{O}(n)$ and the total number of edges in E is also $\mathcal{O}(n)$. Any Eulerian walk W in G corresponds to a string of length $k - 1 + |W|$: $k - 1$ is the length of the first node of the walk to which we concatenate one letter for each of the $|W|$ edges of the walk. Thus by employing Theorem 1 and Theorem 3 we obtain Theorem 4.

References

- 1 Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. In *Developments in Language Theory – 24th International Conference*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2020. doi:10.1007/978-3-030-48516-0_2.
- 2 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe data structures for text indexing. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX*, pages 199–213, 2020. doi:10.1137/1.9781611976007.16.
- 3 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe text indexing. *ACM J. Exp. Algorithmics*, 26, 2021. doi:10.1145/3461698.
- 4 Bastien Cazaux, Thierry Lecroq, and Eric Rivals. Linking indexing data structures to de Bruijn graphs: Construction and update. *J. Comput. Syst. Sci.*, 104:165–183, 2019. doi:10.1016/j.jcss.2016.06.008.
- 5 Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giulia Punzi. Beyond the BEST theorem: Fast assessment of Eulerian trails. In *Fundamentals of Computation Theory – 23rd International Symposium*, volume 12867 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2021. doi:10.1007/978-3-030-86593-1_11.
- 6 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 7 Maxime Crochemore, Borivoj Melichar, and Zdenek Troníček. Directed acyclic subsequence graph – overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003. doi:10.1016/S1570-8667(03)00029-7.
- 8 Jack R. Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese Postman. *Math. Program.*, 5(1):88–124, 1973. doi:10.1007/BF01580113.
- 9 Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Euler Archive – All Works*, 53:15, 1741.
- 10 Lukas Fleischer and Manfred Kufleitner. Testing Simon’s congruence. In *43rd International Symposium on Mathematical Foundations of Computer Science*, volume 117 of *LIPICs*, pages 62:1–62:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.MFCS.2018.62.
- 11 Emmanuelle Garel. Minimal separators of two words. In *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 1993. doi:10.1007/BFb0029795.
- 12 Pawel Gawrychowski, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Efficiently testing Simon’s congruence. In *38th International Symposium on Theoretical Aspects of Computer Science*, volume 187 of *LIPICs*, pages 34:1–34:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.STACS.2021.34.
- 13 Andrew V. Goldberg and Robert Endre Tarjan. Solving minimum-cost flow problems by successive approximation. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 7–18. ACM, 1987. doi:10.1145/28395.28397.
- 14 Horst W. Hamacher. A note on K best network flows. *Ann. Oper. Res.*, 57(1):65–72, 1995. doi:10.1007/BF02099691.
- 15 Jean-Jacques Hébrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theor. Comput. Sci.*, 82(1):35–49, 1991. doi:10.1016/0304-3975(91)90170-7.

- 16 Joan P. Hutchinson. On words with prescribed overlapping subsequences. *Utilitas Mathematica*, 7:241–250, 1975.
- 17 Joan P. Hutchinson and Herbert S. Wilf. On Eulerian circuits and words with prescribed adjacency patterns. *J. Comb. Theory, Ser. A*, 18(1):80–87, 1975. doi:10.1016/0097-3165(75)90068-0.
- 18 Prateek Karandikar, Manfred Kufleitner, and Philippe Schnoebelen. On the index of Simon’s congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015. doi:10.1016/j.ipl.2014.11.008.
- 19 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *25th EACSL Annual Conference on Computer Science Logic*, volume 62 of *LIPICs*, pages 37:1–37:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.CSL.2016.37.
- 20 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 15(2), 2019. doi:10.23638/LMCS-15(2:6)2019.
- 21 Juhani Karhumäki, Svetlana Puzynina, Michaël Rao, and Markus A. Whiteland. On cardinalities of k-abelian equivalence classes. *Theor. Comput. Sci.*, 658:190–204, 2017. doi:10.1016/j.tcs.2016.06.010.
- 22 Carl Kingsford, Michael C. Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinform.*, 11:21, 2010. doi:10.1186/1471-2105-11-21.
- 23 David Könen, Daniel R. Schmidt, and Christiane Spisla. Finding all minimum cost flows and a faster algorithm for the K best flow problem. *CoRR*, abs/2105.10225, 2021. arXiv:2105.10225.
- 24 Kazuhiro Kurita and Kunihiro Wasa. Constant amortized time enumeration of Eulerian trails. *CoRR*, abs/2101.10473, 2021. arXiv:2101.10473.
- 25 M. Lothaire. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, 2nd edition, 1997. doi:10.1017/CB09780511566097.
- 26 Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. *Algorithmica*, 74(1):65–90, 2016. doi:10.1007/s00453-014-9936-y.
- 27 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Oper. Res.*, 41(2):338–350, 1993. doi:10.1287/opre.41.2.338.
- 28 James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Program.*, 77:109–129, 1997. doi:10.1007/BF02614365.
- 29 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.
- 30 Antonio Sedeño-Noda and Juan José Espino-Martín. On the K best integer network flows. *Comput. Oper. Res.*, 40(2):616–626, 2013. doi:10.1016/j.cor.2012.08.014.
- 31 Imre Simon. Piecewise testable events. In *Automata Theory and Formal Languages, 2nd GI Conference*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1975. doi:10.1007/3-540-07407-4_23.
- 32 Robert Endre Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Math. Program.*, 77:169–177, 1997. doi:10.1007/BF02614369.
- 33 Zdenek Troníček. Common subsequence automaton. In *Implementation and Application of Automata, 7th International Conference*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275. Springer, 2002. doi:10.1007/3-540-44977-9_28.
- 34 Tatyana van Aardenne-Ehrenfest and Nicolaas G. de Bruijn. Circuits and trees in oriented linear graphs. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, pages 149–163. Birkhäuser Boston, Boston, MA, 1987. doi:10.1007/978-0-8176-4842-8_12.

The Normalized Edit Distance with Uniform Operation Costs Is a Metric

Dana Fisman ✉

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Joshua Grogin ✉

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Oded Margalit ✉

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Gera Weiss ✉

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Abstract

We prove that the normalized edit distance proposed in [Marzal and Vidal 1993] is a metric when the cost of all the edit operations are the same. This closes a long standing gap in the literature where several authors noted that this distance does not satisfy the triangle inequality in the general case, and that it was not known whether it is satisfied in the uniform case – where all the edit costs are equal. We compare this metric to two normalized metrics proposed as alternatives in the literature, when people thought that Marzal’s and Vidal’s distance is not a metric, and identify key properties that explain why the original distance, now known to also be a metric, is better for some applications. Our examination is from a point of view of formal verification, but the properties and their significance are stated in an application agnostic way.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases edit distance, normalized distance, triangle inequality, metric

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.17

Related Version *Previous Version*: <https://arxiv.org/abs/2201.06115>

Funding This work was supported in part by ISF grants 2714/19 and 2507/21.

1 Introduction

The *edit distance* [5], also called *Levenshtein distance*, is the minimal number of insertions, deletions or substitutions of characters needed to edit one word into another. This is a commonly used measure of the distance between strings. It is used in error correction, pattern recognition, computational biology, and other fields where the data is represented by strings.

One limitation of the edit distance is that it does not contain a normalization with respect to the lengths of the compared strings. This limits its use because, in many applications, having many edit operations when comparing short strings is more significant than having the same number of edit operations in a comparison of longer strings, i.e., some applications require a measure that captures the “average” number of operations per letter, in some sort.

There are several approaches in the literature to add a normalization factor to the edit distance, as follows. The simplest idea that comes to mind is, of course, to divide the edit distance by the sum of lengths of the strings. However, Vidal and Marzal [8] showed that this function, termed *post-normalized edit distance* in [8], does not satisfy the triangle inequality, and thus is not a metric. Dividing by the length of the minimal or maximal among the strings also breaks the triangle inequality [2]. The fact that a distance measure is (or is not) a metric allows (resp. prevents) optimizations in many applications. For example, many efficient algorithms for searching shortest paths in graphs, such as Dijkstra’s algorithm, make use of the fact that the underlying distance is a metric.



© Dana Fisman, Joshua Grogin, Oded Margalit, and Gera Weiss;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 17; pp. 17:1–17:17

Leibniz International Proceedings in Informatics



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

Vidal and Marzal propose thus another function, that we will focus on in this paper, that they term *the normalized edit distance* (NED) and say that this function, “*seems more likely to fulfill the triangle inequality*”. They however, show that when the sum of the costs of deleting and inserting a particular symbol is much smaller than any other elemental edit cost the function that they suggest is also non-triangular. The question of whether this distance is triangular in less contrived situations is given only an empirical answer – “*triangular behavior has actually been observed in practice for the normalized edit distance*”. This state of affairs opened the way for attempts to define edit distance functions that are normalized and satisfy the triangle inequality, as discussed in the following two paragraphs.¹

Li and Liu [6] proposed an alternative normalization method. They open their paper by saying that “*Although a number of normalized edit distances presented so far may offer good performance in some applications, none of them can be regarded as a genuine metric between strings because they do not satisfy the triangle inequality*”. They, then, define a new distance, *the generalized edit distance* (GED), that is a simple function of the lengths of the compared strings and the edit distance between them and show that it is a metric.

De la Higuera and Mičo [2] propose the *contextual normalised edit distance* (CED). Their normalization goes by dividing each edit operation locally by the length of the string on which it is applied. Specifically, instead of dividing the total edit costs by the length of the edit path, they propose to divide the cost of each edit operation by the length of the string at the time of edit. They prove that this is a metric, provide an efficient approximation procedure for it, and demonstrate its performance in several application domains.

In this paper we prove that NED, the original edit normalization approach proposed by Vidal and Marzal [8] does satisfy the triangle inequality when the cost of all the edit operations are the same. Since this setup is very common in many applications of the edit distance, our result gives a simple normalization technique that satisfies the triangle inequality. While there are other normalized edit distance functions that are a metric, in particular the two mentioned above (GED and CED), their definition is more complicated and they capture a different notion of distance than that of NED.

The motivation that led us to engage in distances between words came from the field of formal methods; specifically, for software verification. In this field, it is customary to represent runs of a system using words and analyze the relationship between the set of words that satisfy a given specification and the set of words that the system under examination produces. Naturally, the main question asked is whether there is a word that the system produces that does not satisfy the requirement, but an appropriate concept of distance opens up the possibility of asking further questions. For example, for systems that meet the specifications, the robustness question would be, “is there a run that is closer than a given threshold to not meeting the requirements?”. In this context, we would like the distance to measure how much “disturbance” in a word we can afford without risking non-compliance. Naturally, since editing model symmetric disturbances, we use uniform weights. As we will explain in Subsection 3.2 below, the NED distance satisfies certain properties required for use in formal the field of formal methods that other metrics do not. Another advantage of NED in the context of formal methods is that its definition allows direct use of a PTIME algorithm proposed by Filliot et al. [3] for computing the distance between regular sets of

¹ The complexity of computing NED was first shown to be $O(mn^2)$ with experimental data that suggested that it is actually $O(mn)$ [9]. It was later proven to be $O(mn \log n)$ in the uniform case [1]. Here, $n \geq m$ are the lengths of the compared words.

words represented using finite automata. This is useful since verification tools work with automata to represent the specification and the program runs, and verification questions are usually reduced to questions on automata.

2 Preliminaries

Let Σ be a finite alphabet and Σ^* the set of all finite strings over Σ . The length of string $w = \sigma_1\sigma_2\dots\sigma_n$, denoted $|w|$, is n . We use $w[i]$ for the i -th letter of w , and $w[i..]$ for the suffix of w starting at i , namely $w[i..] = \sigma_i\sigma_{i+1}\dots\sigma_n$.

Basic and extended edit letters. The literature on defining distance between words over Σ uses the notion of *edit paths*, which are strings over *edit letters* defining how to transform a given string s_1 to another string s_2 . The standard operations are *deleting* a letter, *inserting* a letter, or *swapping* one letter with another letter. Formally, the *basic edit letters alphabet* Γ is defined as $\Gamma = \{\mathbf{n}, \mathbf{c}, \mathbf{v}, \mathbf{x}\}$ where:

- \mathbf{c} stands for *change*: the relevant letter in the source string is replaced with another letter.
- \mathbf{v} stands for *insert*: a new letter is added to the destination string.
- \mathbf{x} stands for *delete*: the current letter from the source string is deleted and not copied to the destination string.
- \mathbf{n} stands for *no-change*: the current letter is copied as is from the source string to the destination string.

The edit letters in Γ do not carry enough information to transform a string w over Σ to an unknown string over Σ , since for instance the letter \mathbf{v} does not provide information on which letter $\sigma \in \Sigma$ should be inserted. To this aim we define the alphabet Γ_Σ that provides all the information required. Formally, $\Gamma_\Sigma = \{\mathbf{1}_\sigma \mid \sigma \in \Sigma, \mathbf{1} \in \{\mathbf{n}, \mathbf{v}, \mathbf{x}\}\} \cup \{\mathbf{c}_{(\sigma_1, \sigma_2)} \mid \sigma_1, \sigma_2 \in \Sigma\}$. We call strings over Γ_Σ *edit paths*. Throughout this document we use w, w_1, w_2, w', \dots and s, s_1, s_2, s', \dots for strings over Σ and p, p_1, p_2, p', \dots for edit paths.

Weights and length of edit paths. Given a function $wgt: \Gamma_\Sigma \rightarrow \mathbb{N}$, that defines a weight to each edit letter, we define the weight of an edit path $wgt: \Gamma_\Sigma^* \rightarrow \mathbb{N}$ as the sum of weights of the letter it is composed from, namely for an edit path $p = \gamma_1\gamma_2\dots\gamma_m \in \Gamma_\Sigma^*$, $wgt(\gamma_1\dots\gamma_m) = \sum_{i=1}^m wgt(\gamma_i)$.

In our case we are interested in *uniform* costs where the weight of \mathbf{n} is 0 and the weight of all other operations is the same. For simplicity we can assume that the weight of all other operations is 1. Thus, we can define the weight over Γ instead of Γ_Σ simply as $wgt: \Gamma \rightarrow \mathbb{N}$ where $wgt(\gamma) = 0$ if $\gamma = \mathbf{n}$ and $wgt(\gamma) = 1$ otherwise, namely if $\gamma \in \{\mathbf{c}, \mathbf{v}, \mathbf{x}\}$. We also define the function $len: \Gamma_\Sigma \rightarrow \mathbb{N}$ as $len(\gamma) = 1$ and $len: \Gamma_\Sigma^* \rightarrow \mathbb{N}$ as $len(\gamma_1\dots\gamma_m) = \sum_{i=1}^m len(\gamma_i)$. Clearly here we have $len(p) = |p|$. Later on we will introduce new edit letters whose length is different from 1, thus the need for a definition of len that is not just the count of letters.

► **Example 1.** Let $w_1 = abcd$ and $w_2 = badee$. Then $p = \mathbf{x}_a \cdot \mathbf{n}_b \cdot \mathbf{c}_{c,a} \cdot \mathbf{n}_d \cdot \mathbf{v}_e \cdot \mathbf{v}_e$ is an edit path transforming w_1 to w_2 . We have that $wgt(p) = wgt(\mathbf{xncnvv}) = 4$ and $len(p) = 6$.

Applying an edit path to a string. Given a string w over Σ , and an edit path p over Γ_Σ we can now define the result of applying p to w .

17:4 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

► **Definition 2.** We define a function $apply: \Sigma^* \times \Gamma_\Sigma^* \rightarrow (\Sigma \cup \{\perp\})^*$ that given a string w over Σ , and an edit path p over Γ_Σ returns a new string w' over $\Sigma \cup \{\perp\}$. If p is a valid edit path for w it returns a string over Σ , otherwise a string that contains \perp .

$$apply(p, w) = \begin{cases} \varepsilon & \text{if } p = w = \varepsilon \\ \sigma' \cdot apply(p[2..], w) & \text{if } p[1] = v_{\sigma'} \\ \sigma' \cdot apply(p[2..], w[2..]) & \text{if } p[1] = c_{(\sigma, \sigma')} \text{ and } w[1] = \sigma \\ \sigma \cdot apply(p[2..], w[2..]) & \text{if } p[1] = n_\sigma \text{ and } w[1] = \sigma \\ apply(p[2..], w[2..]) & \text{if } p[1] = x_\sigma \text{ and } w[1] = \sigma \\ \perp & \text{otherwise} \end{cases}$$

We say that a string p_{ij} over Γ_Σ is an *edit path from* string s_i to string s_j over Σ if $apply(p_{ij}, s_i) = s_j$. With a bit of overriding, we say that a string p_{ij} over Γ is an *edit path from* strings s_i to s_j over Σ if there exists an extension of p_{ij} with subscripts from Σ that results in an edit path from s_i to s_j .

► **Example 3.** Following on Ex. 1, we have that $apply(x_a n_b c_{c,a} n_d v_e v_e, abcd) = badee$, and that $xncnv$ is an edit path from $abcd$ to $badee$.

The normalized edit distance. Let p be an edit path. The *cost* of p , denoted $cost(p)$ is defined to be the weight of p divided by the length of p , if the length is not zero, and zero otherwise. That is, $cost(p) = 0$ if $|p| = 0$ and $cost(p) = \frac{wgt(p)}{len(p)}$ otherwise.

Using the definition of *cost* we can define the notion we study in this paper, namely the *normalized edit distance*, NED, of Marzal and Vidal [8].

► **Definition 4** (The normalized edit distance, NED [8]). *The normalized edit distance between s_i and s_j , denoted $NED(s_i, s_j)$ is the minimal cost of an edit path p_{ij} from s_i to s_j . That is,*

$$NED(s_i, s_j) = \min \{ cost(p_{ij}) \mid p_{ij} \in \Gamma_\Sigma^* \text{ and } apply(p_{ij}, s_i) = s_j \}$$

Note that while, in general, *wgt* may assign arbitrary weights to edit letters, in this paper we assume the uniform weights as defined above.

► **Example 5.** Let $\Sigma = \{a, b, c\}$, $s_1 = acbb$ and $s_2 = cc$. Then the string $xnxc$ denotes an edit path taking s_1 , deleting the first letter (a), copying the second letter (c), deleting the third letter (b), and replacing the fourth letter (b) by c . This edit path indeed transforms s_1 to s_2 . Its cost is $\frac{1+0+1+1}{4} = \frac{3}{4}$. It is not hard to verify that this cost is minimal, therefore $NED(s_1, s_2) = \frac{3}{4}$.

The alignment view. Recall that distance functions defined by dividing the weight by the sum, max or min of the given strings does not yield a metric [2, 8]. The main contribution of the paper is to show that the choice to use the length of the edit path in the denominator, makes the resulting definition, NED, a metric. To understand the motivation behind dividing by the length of the edit path, note that an edit path can be thought of as defining an alignment between the given words s_1 and s_2 by padding the first string with some blank symbol, denote it $_$, whenever an insert operation is conducted, and padding the second string with $_$ symbols whenever a delete operation is conducted. The resulting words s'_1 and s'_2 would thus be of the same length, and the weight of the edit path would correspond to the Hamming distance between the words. (The Hamming distance applies only to words of same length and counts the number of positions i in which the two words differ.) When dealing with words of the same length it makes sense to normalize them by dividing by their length, and the length of the padded words equals the length of the edit paths.

► **Example 6.** In Ex. 5 we used $s_1 = acbb$, $s_2 = cc$. The edit path $xnxc$ corresponds to the alignment $s'_1 = acbb$ and $s'_2 = _c_c$, and since the length of s'_1 and s'_2 is 4 and they differ in all positions but one the corresponding cost is $3/4$.

In Ex. 1, we used $w_1 = abcd$ and $w_2 = badee$ and considered the edit path $xncnvv$. This path correspond to the alignment $w'_1 = abcd_ _$ and $w'_2 = _badee$. Since w'_1 and w'_2 differ in four out of the six positions, we have that the cost of this path is $4/6$.

A metric space. A metric space is an ordered pair (\mathbb{M}, d) where \mathbb{M} is a set and $d: \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{R}$ is a *metric*, i.e., it satisfies the following for all $m_1, m_2, m_3 \in \mathbb{M}$:

1. $d(m_1, m_2) = 0$ iff $m_1 = m_2$;
2. $d(m_1, m_2) = d(m_2, m_1)$;
3. $d(m_1, m_3) \leq d(m_1, m_2) + d(m_2, m_3)$.

The first condition is referred to as *identity of indiscernibles*, the second as *symmetry*, the third as the *triangle inequality*.

Basic properties of NED. It is not hard to see that NED satisfies the first and second condition of being a metric. The following proposition establishes that the distance of a string to itself, according to NED, is zero, and that the distance between two strings is symmetric.

► **Proposition 7.** *Let $s, s_1, s_2 \in \Sigma^*$. Then*

1. $NED(s, s) = 0$
2. *if $s_1 \neq s_2$ then $NED(s_1, s_2) > 0$*
3. $NED(s_1, s_2) = NED(s_2, s_1)$

Its straight forward proof can be found in the archived version [4].

The challenge is proving that NED satisfies the third condition, the triangle inequality. We do this in Section 4. Before that we investigate some properties of NED and other edit distance functions.

3 Properties of the various normalized edit distance functions

3.1 Other edit distance functions

In the introduction we mentioned several edit distance functions known to be a metric. We use the term *edit distance* for functions between words to values that are based on *delete*, *insert* and *swaps*. In general these definition may allow arbitrary weight assignment to edit letters, but we consider the case of uniform weights. We start by introducing the edit distance functions, ED, GED, and CED, and then turn to compare their properties, with those of NED.

We start with the commonly used *edit distance*, introduced by Levenstein [5].

► **Definition 8** (The edit (Levenstein) distance, ED). *The edit distance between s_i and s_j , denoted $ED(s_i, s_j)$, is the minimal weight of a path p_{ij} from s_i to s_j . That is,*

$$ED(s_i, s_j) = \min \{wgt(p_{ij}) \mid p_{ij} \in \Gamma_{\Sigma}^* \text{ and } apply(p_{ij}, s_i) = s_j\}$$

This function is a metric, but it completely ignores the lengths of the words, thus it is not normalized.

We turn to introduce the *generalized normalized edit distance* proposed and proven to be a metric by Li and Liu [6].

► **Definition 9** (The generalized edit distance). $GED(s_i, s_j) = \frac{2 \cdot ED(s_i, s_j)}{|s_i| + |s_j| + ED(s_i, s_j)}$.

17:6 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

Last, we define the *contextual edit distance*, proposed and proven to be a metric by de la Higuera and Micó [2]. It starts with a definition of distance between two strings whose Levenstein distance is 1, from which it builds the distance for an arbitrary set of words, by looking at a sequence of intermediate transformations.

► **Definition 10** (The contextual edit distance). *Let s, s' be such that $ED(s, s') = 1$ their contextual edit distance is defined by $CED(s, s') = \frac{1}{\max(|s|, |s'|)}$. Note that given $ED(s, s') = 1$ the difference between the lengths of s and s' is at most one, thus $\max(|s|, |s'|) \leq \min(|s|, |s'|) + 1$.*

Given a sequence of strings $\alpha = (s_0, s_1, \dots, s_k)$ such that $ED(s_i, s_{i+1}) = 1$ for all $0 \leq i < k$, one can define $CED(\alpha) = \sum_{i=1}^k CED(s_{i-1}, s_i)$. To define the contextual edit distance between arbitrary strings s_x and s_y one considers the minimum of $CED(\alpha)$ among all sequence of strings $\alpha = s_0, s_1, \dots, s_k$ as above such that $s_0 = s_x, s_k = s_y$. That is, $CED(s_x, s_y) = \min \{ CED(\alpha) \mid \alpha = (s_0, s_1, \dots, s_k), s_0 = s_x, s_k = s_y, ED(s_i, s_{i+1}) = 1 \}$.

3.2 Comparison to other edit distance functions

Comparing NED and ED is easy. The NED distance (like CED and GED) measures the average number edits, not just the total count. To see why this is needed, consider two short words x_1, x_2 that differ in k letters and two long word y_1, y_2 that also differ in k letters. In the context of software verification, for example, the latter represent runs that are more similar to one another than the former. We thus, expect the distance between the y s to be less than the distances between the x s but this is not the case in ED, as can be observed by inspecting the following words.

$$\begin{array}{ll} ED(abcde, abpcg) = 4 & NED(abcde, abpcg) = 4/7 \\ ED(a^{96}b^4, a^{100}) = 4 & NED(a^{96}b^4, a^{100}) = 4/100 \end{array}$$

We turn to a comparisons of NED with the other normalized edit distances, GED and CED. Usually, being normalized means that the values of the distance functions are bounded within a given range, but this is not always the case. The lower bound is clearly 0 for NED, GED, and CED, since they are metric. The upper value of NED and GED is 1 but the values for CED are not bounded:

► **Claim 11.** *The values of NED and GED cannot exceed 1 and may reach 1, the values of CED are unbounded.*

Proof. For NED the numerator is the weight of an edit path, which is always smaller than the denominator which is the length of the edit path, thus $NED(w_1, w_2) \leq 1$ for all $w_1, w_2 \in \Sigma^*$. Since $NED(\varepsilon, a) = 1$ the upper bound is 1.

For GED the numerator is twice the weight of the edit path, and the denominator is once the weight of the edit path, plus the sum of length of the strings which is at least the size of the edit path, thus clearly at least the weight of the edit path. This shows GED cannot exceed 1. The fact that $GED(\varepsilon, a) = 1$ shows that 1 is the upper bound.

To see why CED is not bounded consider the sequence of words $\{a^i\}_{i \in \mathbb{N}}$. That is, the sequence $\varepsilon, a, aa, aaa, \dots$. We have that $CED(\varepsilon, a^i) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{i}$. Thus $CED(\varepsilon, a^i)$ is the sum of the Harmonic sequence up to the i th element, and since the Harmonic sequence diverges, CED is unbounded. ◀

Towards the second property of metrics that we consider, recall that the first requirements of a metric, *identity of indiscernibles*, is that $d(s_1, s_2) = 0$ if and only if $s_1 = s_2$. That is, the distance between two strings (in our case) is zero if and only if it is the exact same string. In the case of strings, when working with a normalized distance with an upper bound 1, we

expect the distance to be 1, the maximal possible, if the strings are completely different, namely they do not have any letter in common, that is, for all $\sigma \in \Sigma$ if σ appears in s_1 it does not appear in s_2 and vice versa. In software verification, for example, this means that the system produced a run that is completely unrelated to the specification, thus we expect the distance to be 1, indicating it is as far away as possible from the specification.

Since CED is unbounded, we consider for the purpose of the next property, a slightly different version, that we call CED', defined as $\text{CED}'(s_1, s_2) = \min(1, \text{CED}(s_1, s_2))$.²

► **Property 12** (max variance of antitheticals). *Let $d: \Sigma^* \times \Sigma^* \rightarrow [0, 1]$ be an edit distance function. We say that d has the property of max variance of antitheticals if $d(s_1, s_2) = 1$ if and only if s_1 and s_2 have no letter in common.*

We show that NED has this property while GED and CED' do not.³

► **Claim 13.** *The property of max variance of antitheticals holds for NED, but does not hold for GED and CED'.*

Proof. Consider aa and bb . Since they have no common letter, we expect their distance to be 1. The fact that $\text{GED}(aa, bb) = 2/3$ shows that GED violates the property of max variance of antitheticals.⁴ Consider a and $aaaa$. Since they do have a common letter, we expect their distance to be strictly less than 1. The fact that $\text{CED}'(a, aaaa) = 1$ shows that CED' violates the property of max variance of antitheticals.

To see that NED has this property, note that it results in a value of 1 iff the numerator equals the denominator, i.e., the weight of the edit path is the same as its length; which holds iff there are no edit letters with weight zero. Since the only zero weight edit letter is no-change, n , the value of NED is 1 if and only if the words have no common letter. ◀

For the third metric comparison property, consider two words u and v and suppose $d(u, v) = c$ for the concerned edit distance function d . When considering normalized edit distance, we expect that $d(u^i, v^i)$ will not exceed c since by repeating i times the edit operations for transforming u into v we should be able to transform u^i into v^i and the “average” number of edits will not change. It could be that when considering the longer words u^i and v^i there is a better sequence of edits, thus we do not expect equality. As before, our motivation for requiring this property comes from software verification. Specifically, when considering periodic runs, generated, e.g., by code with loops, one would expect that the distance between the periodic runs is not larger than the distance between the periods because an error that repeats regularly should only be counted once in a normalized measure that models average error rate.

► **Property 14** (Non escalation of repetitions). *Let d be an edit distance function. Let $u, v \in \Sigma^*$. If $d(u^k, v^k) \leq d(u, v)$ for any $k > 1$ we say that d does not escalate repetitions.*

► **Claim 15.** *The NED and GED distances satisfy the property of non escalation of repetitions. The CED and CED' distances do not.*

² This is inspired by [7] that explains this choice as follows: “This measure is not normalized to a particular range. Indeed, for a string of infinite length and a string of 0 length, the contextual normalized edit distance would be infinity. But so long as the relative difference in string lengths is not too great, the distance will generally remain below 1.0”.

³ Note that extending this property to require that $d(s_1, s_2)$ equals the maximal value (be it 1 or more) only for antitheticals, so that it can be applied to the original CED, would not make CED satisfy it since $\text{CED}(\varepsilon, a) = 1 < \infty$.

⁴ We note that, moreover, $\text{GED}(aab, b)$ is also $2/3$ though we expect $\text{GED}(aab, b) < \text{GED}(aa, bb)$ since the average number of edits is smaller in the first case.

17:8 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

Proof. Consider $u = aab$ and $v = aaab$. The following shows that CED and CED' escalate repetitions.

$$\begin{aligned} \text{CED}((aab)^1, (aaab)^1) &= \frac{1}{4} = 0.25 \\ \text{CED}((aab)^2, (aaab)^2) &= \frac{1}{7} + \frac{1}{8} = \frac{15}{56} = 0.2678 \\ \text{CED}((aab)^3, (aaab)^3) &= \frac{1}{10} + \frac{1}{11} + \frac{1}{12} = \frac{181}{660} = 0.2742 \end{aligned}$$

To see that NED does not escalate repetitions, assume p_{uv} is an optimal edit path transforming u to v . Since $(p_{uv})^k$, the edit path obtained by repeating k times p_{uv} , is an edit path transforming u^k to v^k :

$$\text{NED}(u^k, v^k) \leq \frac{k \cdot \text{wt}(p_{uv})}{k \cdot \text{len}(p_{uv})} = \frac{\text{wt}(p_{uv})}{\text{len}(p_{uv})} = \text{NED}(u, v).$$

The same reasoning shows that GED does not escalate repetitions.

$$\text{GED}(u^k, v^k) \leq \frac{2k \cdot \text{ED}(u, v)}{k(|u|+|v|)+k \cdot \text{ED}(u, v)} = \frac{2 \cdot \text{ED}(u, v)}{|u|+|v|+\text{ED}(u, v)} = \text{GED}(u, v). \quad \blacktriangleleft$$

The last property we consider is referred to as *pure uniformity of operations*. While we assume the weights of delete, insert and substitution are uniform, the resulting edit distance function may not be purely uniform, in the following sense. Consider two strings s_1 and s_2 such that s_1 is shorter than s_2 . Then to transform s_1 to s_2 we would need some insertion operations. Consider now a word s'_1 that is longer than s_1 but not longer than s_2 and is obtained by padding s_1 with some new letter σ_{new} in some arbitrary set of positions. Since insert and substitution weigh the same, we expect $d(s_1, s_2)$ to be equal to $d(s'_1, s_2)$.

To define this formally we use the following notations. Let $\Sigma' \subseteq \Sigma$ and $s \in \Sigma^*$ we use $\pi_{\Sigma'}(s)$ for the string obtained from s by leaving only letters in Σ' . For instance, if $\Sigma = \{a, b, c\}$ and $s = abcacc$ then $\pi_{\{a,b\}} = abba$.

► **Property 16** (pure uniformity). *Let $\Sigma, \Sigma_1, \Sigma_2$ be disjoint alphabets, and let $s_1, s_2 \in \Sigma^*$. We call d purely uniform if $d(s_1, s_2) = \min\{d(s'_1, s_2) \mid s'_i \in (\Sigma \uplus \Sigma_i)^* \text{ and } \pi_{\Sigma}(s'_i) = s_i \text{ for } i \in \{1, 2\}\}$.*

We can now show that NED satisfies this property while GED and CED do not.

► **Claim 17.** *The NED distance is purely uniform. The GED and CED distances are not.*

Proof. To see why GED and CED are not purely uniform consider the words $s_1 = a^{50}$, $s_2 = a^{100}$ and $s'_1 = a^{50}c^{50}$ and note that $\pi_{\{a,b\}}(s'_1) = s_1$. We have that $\text{GED}(a^{50}, a^{100}) = 2 \cdot 50/(150+50) = 1/2$ whereas $\text{GED}(a^{50}c^{50}, a^{100}) = 100/(200+100) = 1/3$. Considering CED, we have that $\text{CED}(a^{50}, a^{100}) = \sum_{i=51}^{100} \frac{1}{i} \approx 0.68817$ whereas $\text{CED}(a^{50}c^{50}, a^{100}) = \sum_{i=51}^{100} \frac{1}{100} = 0.5$. Since all values are below 1, the same is true for CED'.

To show that NED is purely uniform we first note that $s_1, s_2 \in \Sigma^*$ implies s_1, s_2 are in $(\Sigma \uplus \Sigma_1)^*$ and $(\Sigma \uplus \Sigma_2)^*$, respectively, thus the \geq direction of the equality in Property 16 clearly holds. For the \leq direction, we turn to Claim 18 below, which essentially formalized the intuition provided regarding the *alignment view* of NED. Thus, given s'_1 and s'_2 establishing the min in the RHS of Property 16, and $p' \in \Gamma^*$ an edit path transforming s'_1 into s'_2 , we can build an edit path $p \in \Gamma^*$ transforming $\pi_{\Sigma}(s'_1)$ into $\pi_{\Sigma}(s'_2)$ such that $\text{cost}(p) \leq \text{cost}(p')$. This shows that $\text{NED}(s_1, s_2) \leq \text{NED}(s'_1, s'_2)$ for every such s'_1, s'_2 . Thus NED satisfies the pure uniformity property. \blacktriangleleft

► **Claim 18.** *Let $\Sigma, \Sigma_1, \Sigma_2$ be disjoint nonempty alphabets. Let $s'_1 \in \Sigma \uplus \Sigma_1$ and $s'_2 \in \Sigma \uplus \Sigma_2$ and p' an edit path transforming s'_1 to s'_2 . There exists an edit path p transforming $\pi_{\Sigma}(s'_1)$ to $\pi_{\Sigma}(s'_2)$ such that $\text{cost}(p) \leq \text{cost}(p')$.*

4 A Proof of the Triangle Inequality

This section is the main contribution of the paper – showing that NED with uniform costs satisfies the triangle inequality.

Let $s_1, s_2, s_3 \in \Sigma^*$ and p_{12}, p_{23} be edit paths, such that $apply(p_{12}, s_1) = s_2$, $apply(p_{23}, s_2) = s_3$. We would like to define a method $cmps: \Gamma_\Sigma^* \times \Gamma_\Sigma^* \rightarrow \Gamma_\Sigma^*$ that given the two edit paths p_{12}, p_{23} returns an edit path p_{13} from s_1 to s_3 . In addition, using the notations $d_* = wgt(p_*)$ and $l_* = len(p_*)$ for $* \in \{12, 23, 13\}$, we would like to show that both of the following hold:

$$d_{13} \leq d_{12} + d_{23} \quad (1) \qquad l_{13} \geq \max\{l_{12}, l_{23}\} \quad (2)$$

From these two equations we can deduce that the cost of the resulting path p_{13} is at most the sum of costs of the given paths p_{12} and p_{23} proving that NED satisfies the triangle inequality.

Introducing a new edit letter. To do this we need, for technical reasons, to introduce a new edit letter, which we denote \mathbf{b} (for *blank*). This is actually an abbreviation of \mathbf{vx} , that is, it signifies that a new letter is added and immediately deleted. We enhance the weight and length definition from Γ to $\Gamma \cup \{\mathbf{b}\}$ as follows.

$$wgt(\gamma) = \begin{cases} 0 & \text{if } \gamma = \mathbf{n} \\ 1 & \text{if } \gamma \in \{\mathbf{c}, \mathbf{v}, \mathbf{x}\} \\ 2 & \text{if } \gamma = \mathbf{b} \end{cases} \qquad len(\gamma) = \begin{cases} 1 & \text{if } \gamma \in \{\mathbf{n}, \mathbf{c}, \mathbf{v}, \mathbf{x}\} \\ 2 & \text{if } \gamma = \mathbf{b} \end{cases}$$

As before we use the natural extensions of wgt and len from letters to strings and define $cost(p)$ to be $wgt(p)/len(p)$.

The compose method. We define a helper function $cmps_h$ that produces a string over $(\Gamma_\Sigma \cup \{\mathbf{b}\})^*$ (rather than over Γ_Σ^*). Given such a sequence we can convert it into a sequence over Γ_Σ by deleting all \mathbf{b} symbols. The method $cmps_h: \Gamma_\Sigma^* \times \Gamma_\Sigma^* \rightarrow (\Gamma_\Sigma \cup \{\mathbf{b}\})^* \cup \{\perp\}$ is defined inductively, in Def. 19, by scanning the letters of the given edit paths p_{12}, p_{23} . We say that $cmps_h$ is well defined if it does not return \perp . We show that, when applied on edit paths p_{12} and p_{23} transforming some s_1 into s_2 and s_2 into s_3 , respectively, $cmps_h$ is well defined.

► **Definition 19.** Let p_{12}, p_{23} be edit paths over Γ_Σ . We define $cmps_h(p_{12}, p_{23})$ inductively as follows.

$$cmps_h(p_{12}, p_{23}) = \begin{cases} \varepsilon & \text{if } p_{12} = p_{23} = \varepsilon & (0) \\ \mathbf{x}_\sigma \cdot cmps_h(p_{12}[2..], p_{23}) & \text{if } p_{12}[1] = \mathbf{x}_\sigma & (1) \\ \mathbf{v}_\sigma \cdot cmps_h(p_{12}, p_{23}[2..]) & \text{if } p_{23}[1] = \mathbf{v}_\sigma & (2) \\ \mathbf{n}_\sigma \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{n}_\sigma, \mathbf{n}_\sigma) & (3) \\ \mathbf{c}_{(\sigma', \sigma)} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{n}_{\sigma'}, \mathbf{c}_{(\sigma', \sigma)}) & (4) \\ \mathbf{x}_\sigma \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{n}_\sigma, \mathbf{x}_\sigma) & (5) \\ \mathbf{c}_{(\sigma_1, \sigma_3)} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{c}_{(\sigma_1, \sigma_2)}, \mathbf{c}_{(\sigma_2, \sigma_3)}) & (6) \\ \mathbf{x}_{\sigma_1} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{c}_{(\sigma_1, \sigma_2)}, \mathbf{x}_{\sigma_2}) & (7) \\ \mathbf{c}_{(\sigma', \sigma)} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{c}_{(\sigma', \sigma)}, \mathbf{n}_\sigma) & (8) \\ \mathbf{v}_\sigma \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{v}_\sigma, \mathbf{n}_\sigma) & (9) \\ \mathbf{v}_{\sigma_2} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{v}_{\sigma_1}, \mathbf{c}_{(\sigma_1, \sigma_2)}) & (10) \\ \mathbf{b} \cdot cmps_h(p_{12}[2..], p_{23}[2..]) & \text{if } (p_{12}[1], p_{23}[1]) = (\mathbf{v}_\sigma, \mathbf{x}_\sigma) & (11) \\ \perp & \text{otherwise} & (12) \end{cases}$$

17:10 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

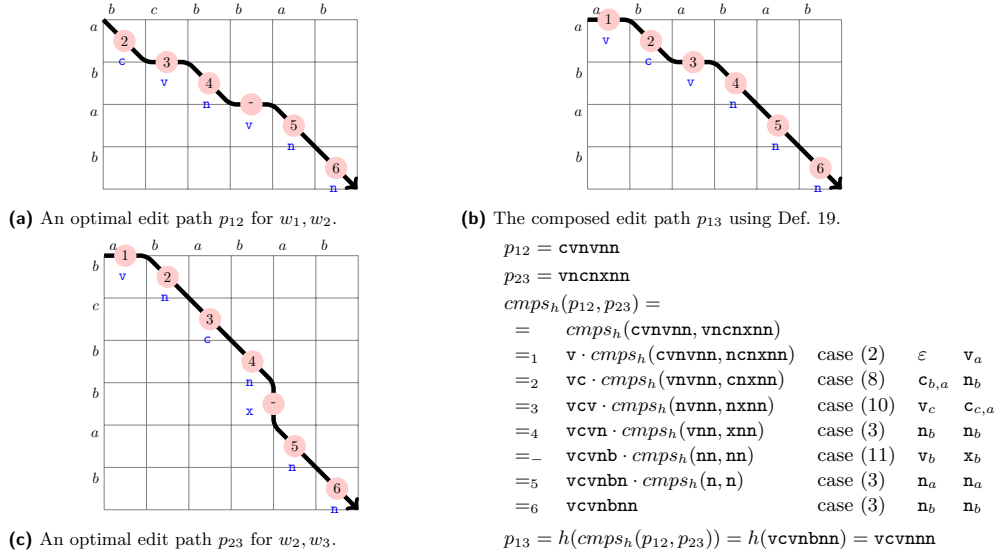


Figure 1 Let $w_1 = abab$, $w_2 = bcbbab$, $w_3 = ababab$. Figure 1a shows an optimal edit path p_{12} between w_1 to w_2 , Figure 1c shows an optimal edit path p_{23} between w_2 to w_3 . Figure 1b shows the edit path p_{13} composed from p_{12} and p_{23} using Def. 19. The edit operations in Figure 1b are marked with numbers 1 to 6. A number n in between 1 and 6 in Figure 1a and Figure 1c signifies that the corresponding edge contributed to the construction of the edge marked n in Figure 1b (thus for the operations corresponding to cases (1) and (2) of Def. 19, there is one corresponding marking in Figure 1a and Figure 1c and for the others there are two). The labels $-$ in Figure 1a and Figure 1c correspond to case (11) dealing with adding a letter when going from s_1 to s_2 and deleting it when going from s_2 to s_3 , which yields the edit symbol b . Note that p_{13} is not optimal; still its cost is better than the sum of the costs of p_{12} and p_{23} .

We further show that if the resulting string is p_{13} then applying the function $apply$ to s_1 and the edit path obtained from p_{13} by deleting all b results in the string s_3 . Figure 1 shows an example of the application of cmph_h on two given edit paths. In the sequel we will further show that the desired equations (Equation 1) and (Equation 2) hold.

Note that if we reach case (12) then we cannot claim that the result is an edit path. We thus first show that if cmph_h is applied to two edit paths p_{12}, p_{23} such that $apply(p_{12}, s_1) = s_2$, and $apply(p_{23}, s_2) = s_3$, then the recursive application of $\text{cmph}_h(p_{12}, p_{23})$ will never reach the (12) case. That is, $\text{cmph}_h(p_{12}, p_{23})$ is well defined.

► **Lemma 20.** Let $s_1, s_2, s_3 \in \Sigma^*$ and $p_{12}, p_{23} \in \Gamma_\Sigma^*$ be edit paths, such that $apply(p_{12}, s_1) = s_2$ and $apply(p_{23}, s_2) = s_3$. Then $p_{13} = \text{cmph}_h(p_{12}, p_{23})$ is well-defined.

Proof. The proof is by structural induction on cmph_h . For the **base case**, we have that $p_{12} = p_{23} = \varepsilon$. Then $p_{13} = \varepsilon$. Thus cmph_h reaches case (0) and is well defined.

For the **induction step** we have $p_{12} \neq \varepsilon$ or $p_{23} \neq \varepsilon$. If $p_{12} = \varepsilon$ then it follows from the definition of $apply$ that $s_1 = s_2 = \varepsilon$. Given that $apply(p_{23}, \varepsilon)$ is defined we get that $p_{23}[1] = \text{v}_\sigma$. From the definition of $apply$ we have $s_3 = \sigma \cdot apply(p_{23}[2..], s_2)$. Hence $s_3[2..] = apply(p_{23}[2..], s_2)$. Therefore, cmph_h reaches case (2) and will never reach case (12) since from the induction hypothesis it follows that $\text{cmph}_h(p_{12}, p_{23}[2..])$ is well defined.

If $p_{23} = \varepsilon$ we get $s_2 = s_3 = \varepsilon$ and $p_{12}[1] = \text{x}_\sigma$. Hence cmph_h reaches case (1) and similar reasoning shows that the induction hypothesis holds for the recursive application, and thus the result is well defined.

Otherwise the first character of p_{12} is not x and the first character of p_{23} is not v . We consider the remaining cases, by examining first the first letter of p_{12} .

1. **Case** $p_{12}[1] = v_{\sigma_1}$.
From the definition of *apply* we get that $s_2 = \sigma_1 \cdot s_2[2..]$ and $s_2[2..] = \text{apply}(p_{12}[2..], s_1)$.
 - a. **Subcase** $p_{23}[1] = c_{(\sigma_2, \sigma_3)}$.
From the definition of *apply* it follows that $\sigma_1 = \sigma_2$, $s_3 = \sigma_3 \cdot s_3[2..]$ and $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (10) and the induction hypothesis holds for the recursive application.
 - b. **Subcase** $p_{23}[1] = n_{\sigma_2}$.
Similarly, from the definition of *apply* we get that $\sigma_1 = \sigma_2$, $s_3 = \sigma_2 \cdot s_3[2..]$ and furthermore $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (9) and the induction hypothesis holds for the recursive application.
 - c. **Subcase** $p_{23}[1] = x_{\sigma_2}$.
Similarly, from the definition of *apply* we get that $\sigma_1 = \sigma_2$ and $s_3 = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (11) and the induction hypothesis holds for the recursive application.
2. **Case** $p_{12}[1] = c_{(\sigma_1, \sigma_2)}$.
From the definition of *apply* we get that $s_1 = \sigma_1 \cdot s_1[2..]$, $s_2 = \sigma_2 \cdot s_2[2..]$ and furthermore $s_2[2..] = \text{apply}(p_{12}[2..], s_1[2..])$.
 - a. **Subcase** $p_{23}[1] = c_{(\sigma_3, \sigma_4)}$.
From the definition of *apply* we get that $\sigma_2 = \sigma_3$, $s_3 = \sigma_4 \cdot s_3[2..]$ and $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (6) and the induction hypothesis holds for the recursive application.
 - b. **Subcase** $p_{23}[1] = n_{\sigma_3}$.
Similarly, from the definition of *apply* it follows that $\sigma_2 = \sigma_3$, $s_3 = \sigma_3 \cdot s_3[2..]$ and $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (8) and the induction hypothesis holds for the recursive application.
 - c. **Subcase** $p_{23}[1] = x_{\sigma_3}$.
Similarly, from the definition of *apply* we get that $\sigma_2 = \sigma_3$ and $s_3 = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (7) and the induction hypothesis holds for the recursive application.
3. **Case** $p_{12}[1] = n_{\sigma}$.
From the definition of *apply* we get that $s_1 = \sigma \cdot s_1[2..]$, $s_2 = \sigma \cdot s_2[2..]$ and $s_2[2..] = \text{apply}(p_{12}[2..], s_1[2..])$.
 - a. **Subcase** $p_{23}[1] = c_{(\sigma_1, \sigma_2)}$.
From the definition of *apply* it follows that $\sigma = \sigma_1$, $s_3 = \sigma_2 \cdot s_3[2..]$ and $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (4) and the induction hypothesis holds for the recursive application.
 - b. **Subcase** $p_{23}[1] = n_{\sigma_2}$.
Similarly, from the definition of *apply* it follows that $\sigma = \sigma_2$, $s_3 = \sigma_2 \cdot s_3[2..]$ and furthermore $s_3[2..] = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (3) and the induction hypothesis holds for the recursive application.
 - c. **Subcase** $p_{23}[1] = x_{\sigma_2}$.
Similarly, from the definition of *apply* we get that $\sigma = \sigma_2$ and $s_3 = \text{apply}(p_{23}[2..], s_2[2..])$. Thus *cmps_h* reaches case (5) and the induction hypothesis holds for the recursive application. ◀

17:12 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

Recall that the $cmps_h$ returns a string over $\Gamma_\Sigma \cup \{\mathbf{b}\}$ while $apply$ first argument is expected to be a string over Γ_Σ . We can convert the string returned by $cmps_h$ to a string over Γ_Σ by simply removing the \mathbf{b} symbols. To make this precise we introduce the function $h: \Gamma_\Sigma \cup \{\mathbf{b}\} \rightarrow \Gamma_\Sigma$ defined as follows $h(\gamma) = \varepsilon$ if $\gamma = \mathbf{b}$ and $h(\gamma) = \gamma$ otherwise; and its natural extension $h: (\Gamma_\Sigma \cup \{\mathbf{b}\})^* \rightarrow \Gamma_\Sigma^*$ defined as $h(\gamma_1\gamma_2 \cdots \gamma_n) = h(\gamma_1)h(\gamma_2) \cdots h(\gamma_n)$.

We are now ready to state that $cmps_h$ fulfills its task, namely if it returns p_{13} then $h(p_{13})$ is an edit path from s_1 to s_3 and its weight and length satisfy Equation 1 and Equation 2. Note that even if p_{12} and p_{23} are optimal, $h(p_{13})$ is not necessarily an optimal path from s_1 to s_3 . Since the optimal path is no worse than $h(p_{13})$, it is enough for our purpose that $h(p_{13})$ is better than going through s_2 .

► **Proposition 21.** *Let $s_1, s_2, s_3 \in \Sigma^*$ and p_{12}, p_{23} be edit paths, such that $apply(p_{12}, s_1) = s_2$, $apply(p_{23}, s_2) = s_3$. Let $p_{13} = cmps_h(p_{12}, p_{23})$. Let $d_* = wgt(p_*)$ and $l_* = len(p_*)$ for $* \in \{12, 23, 13\}$. Then the following holds*

1. $apply(h(p_{13}), s_1) = s_3$
2. $d_{13} \leq d_{12} + d_{23}$
3. $l_{13} \geq \max\{l_{12}, l_{23}\}$

Proof. The proof is by structural induction on $cmps_h$. For the **base case**, we have that $p_{12} = p_{23} = \varepsilon$. Then $p_{13} = \varepsilon$, by definition of $apply$ we get that $s_1 = s_2 = s_3 = \varepsilon$. Thus

1. $apply(h(p_{13}), s_1) = apply(\varepsilon, \varepsilon) = \varepsilon = s_1 = s_3$
2. and 3. we have that $d_{13} = 0 \leq d_{12} + d_{23} = 0$ and $l_{13} = 0 \geq \max\{l_{12}, l_{23}\} = 0$

For the **induction steps**, we have $p_{12} \neq \varepsilon$ or $p_{23} \neq \varepsilon$. Recall that $p_{13} = cmps_h(p_{12}, p_{23})$. Thus, from Lem. 20 we can conclude p_{13} is a string over $\Gamma_\Sigma \cup \{\mathbf{b}\}$. Let $s'_* = s_*[2..]$, $p'_* = p_*[2..]$, $d'_* = wgt(p'_*)$, $l'_* = len(p'_*)$ for $* \in \{12, 23, 13\}$. The proof proceeds with the case analysis of $cmps_h$, going over cases (1)-(11) of Def. 19.

(1) Here $p_{12}[1] = \mathbf{x}_\sigma$.

Then from $apply$ we have $s_1 = \sigma \cdot s'_1$, from definition of $cmps_h$ we have $p_{13} = \mathbf{x}_\sigma \cdot p'_{13}$. Since $s_2 = apply(p_{12}, s_1) = apply(\mathbf{x}_\sigma \cdot p'_{12}, \sigma \cdot s'_1) = apply(p'_{12}, s'_1)$ and $apply(p_{23}, s_2) = s_3$, by applying the induction hypotheses on s'_1, s_2, s_3 we get

1. $apply(h(p'_{13}), s'_1) = s_3$
2. $d'_{13} \leq d'_{12} + d_{23}$
3. $l'_{13} \geq \max\{l'_{12}, l_{23}\}$

Therefore

1. $apply(h(p_{13}), s_1) = apply(\mathbf{x}_\sigma \cdot h(p'_{13}), \sigma \cdot s'_1) = apply(h(p'_{13}), s'_1) = s_3$
2. $d_{13} = 1 + d'_{13} \leq 1 + d'_{12} + d_{23} = d_{12} + d_{23}$
3. $l_{13} = 1 + l'_{13} \geq 1 + \max\{l'_{12}, l_{23}\} \geq \max\{1 + l'_{12}, l_{23}\} = \max\{l_{12}, l_{23}\}$.

(2) Here $p_{23}[1] = \mathbf{v}_\sigma$.

Then from $apply$ we have $s_3 = \sigma \cdot s'_3$, from definition of $cmps_h$ we have $p_{13} = \mathbf{v}_\sigma \cdot p'_{13}$. Since $apply(p_{23}, s_2) = apply(\mathbf{v}_\sigma \cdot p'_{23}, s_2) = \sigma \cdot apply(p'_{23}, s_2) = s_3 = \sigma \cdot s'_3$ we get $apply(p'_{23}, s_2) = s'_3$ and $apply(p_{12}, s_1) = s_2$, by applying the induction hypotheses on s_1, s_2, s'_3 we get

1. $apply(h(p'_{13}), s_1) = s'_3$
2. $d'_{13} \leq d_{12} + d'_{23}$
3. $l'_{13} \geq \max\{l_{12}, l'_{23}\}$.

Therefore

1. $apply(h(p_{13}), s_1) = apply(\mathbf{v}_\sigma \cdot h(p'_{13}), s_1) = \sigma \cdot apply(h(p'_{13}), s_1) = \sigma \cdot s'_3 = s_3$
2. $d_{13} = 1 + d'_{13} \leq d_{12} + 1 + d'_{23} = d_{12} + d_{23}$
3. $l_{13} = 1 + l'_{13} \geq 1 + \max\{l_{12}, l'_{23}\} \geq \max\{l_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}$.

(3) Here $(p_{12}[1], p_{23}[1]) = (\mathbf{n}_\sigma, \mathbf{n}_\sigma)$.

From the definition of $cmps_h$ we have $p_{13} = \mathbf{n}_\sigma \cdot p'_{13}$ and from $apply$ we have $apply(p_{12}, s_1) = apply(\mathbf{n}_\sigma \cdot p'_{12}, \sigma \cdot s'_1) = \sigma \cdot apply(p'_{12}, s'_1) = \sigma \cdot s'_2 = s_2$ and $apply(p_{23}, s_2) = apply(\mathbf{n}_\sigma \cdot p'_{23}, \sigma \cdot s'_2) = \sigma \cdot apply(p'_{23}, s'_2) = \sigma \cdot s'_3 = s_3$.

Since $\text{apply}(p'_{12}, s'_1) = s'_2$ and $\text{apply}(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s'_1, s'_2, s'_3 we get

$$1. \text{apply}(h(p'_{13}), s'_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. \text{apply}(h(p_{13}), s_1) &= \text{apply}(\mathbf{n}_{\sigma'} \cdot h(p'_{13}), \sigma \cdot s'_1) = \sigma \cdot \text{apply}(h(p'_{13}), s'_1) = \sigma \cdot s'_3 = s_3 \\ 2. d_{13} &= d'_{13} \leq d'_{12} + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(4) Here $(p_{12}[1], p_{23}[1]) = (\mathbf{n}_{\sigma'}, \mathbf{c}_{(\sigma', \sigma)})$.

By definition of compose we get $p_{13} = \mathbf{c}_{(\sigma', \sigma)} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} \text{apply}(p_{12}, s_1) &= \text{apply}(\mathbf{n}_{\sigma'} \cdot p'_{12}, \sigma' \cdot s'_1) = \sigma' \cdot \text{apply}(p'_{12}, s'_1) = \sigma' \cdot s'_2 = s_2 \text{ and} \\ \text{apply}(p_{23}, s_2) &= \text{apply}(\mathbf{c}_{(\sigma', \sigma)} \cdot p'_{23}, \sigma' \cdot s'_2) = \sigma \cdot \text{apply}(p'_{23}, s'_2) = \sigma \cdot s'_3 = s_3. \end{aligned}$$

Since $\text{apply}(p'_{12}, s'_1) = s'_2$ and $\text{apply}(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s'_1, s'_2, s'_3 we get

$$1. \text{apply}(h(p'_{13}), s'_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. \text{apply}(h(p_{13}), s_1) &= \text{apply}(\mathbf{c}_{(\sigma', \sigma)} \cdot h(p'_{13}), \sigma' \cdot s'_1) = \sigma \cdot \text{apply}(h(p'_{13}), s'_1) = \sigma \cdot s'_3 = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq d'_{12} + 1 + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(5) Here $(p_{12}[1], p_{23}[1]) = (\mathbf{n}_{\sigma}, \mathbf{x}_{\sigma})$.

By definition of compose we get that $p_{13} = \mathbf{x}_{\sigma} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} \text{apply}(p_{12}, s_1) &= \text{apply}(\mathbf{n}_{\sigma} \cdot p'_{12}, \sigma \cdot s'_1) = \sigma \cdot \text{apply}(p'_{12}, s'_1) = \sigma \cdot s'_2 = s_2 \text{ and} \\ \text{apply}(p_{23}, s_2) &= \text{apply}(\mathbf{x}_{\sigma} \cdot p'_{23}, \sigma \cdot s'_2) = \text{apply}(p'_{23}, s'_2) = s_3. \end{aligned}$$

Since $\text{apply}(p'_{12}, s'_1) = s'_2$ and $\text{apply}(p'_{23}, s'_2) = s_3$, by applying the induction hypotheses on s'_1, s'_2, s_3 we get

$$1. \text{apply}(h(p'_{13}), s'_1) = s_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. \text{apply}(h(p_{13}), s_1) &= \text{apply}(\mathbf{x}_{\sigma} \cdot h(p'_{13}), \sigma \cdot s'_1) = \text{apply}(h(p'_{13}), s'_1) = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq d'_{12} + 1 + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(6) Here $(p_{12}[1], p_{23}[1]) = (\mathbf{c}_{(\sigma_1, \sigma_2)}, \mathbf{c}_{(\sigma_2, \sigma_3)})$.

By definition of compose we get that $p_{13} = \mathbf{c}_{(\sigma_1, \sigma_3)} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} \text{apply}(p_{12}, s_1) &= \text{apply}(\mathbf{c}_{(\sigma_1, \sigma_2)} \cdot p'_{12}, \sigma_1 \cdot s'_1) = \sigma_2 \cdot \text{apply}(p'_{12}, s'_1) = \sigma_2 \cdot s'_2 = s_2 \text{ and} \\ \text{apply}(p_{23}, s_2) &= \text{apply}(\mathbf{c}_{(\sigma_2, \sigma_3)} \cdot p'_{23}, \sigma_2 \cdot s'_2) = \sigma_3 \cdot \text{apply}(p'_{23}, s'_2) = \sigma_3 \cdot s'_3 = s_3. \end{aligned}$$

Since $\text{apply}(p'_{12}, s'_1) = s'_2$ and $\text{apply}(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s'_1, s'_2, s'_3 we get

$$1. \text{apply}(h(p'_{13}), s'_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. \text{apply}(h(p_{13}), s_1) &= \text{apply}(\mathbf{c}_{(\sigma_1, \sigma_3)} \cdot h(p'_{13}), \sigma_1 \cdot s'_1) = \sigma_3 \cdot \text{apply}(h(p'_{13}), s'_1) = \sigma_3 s'_3 = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq 1 + d'_{12} + d'_{23} < 1 + d'_{12} + 1 + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(7) Here $(p_{12}[1], p_{23}[1]) = (\mathbf{c}_{(\sigma_1, \sigma_2)}, \mathbf{x}_{\sigma_2})$.

By definition of compose we get that $p_{13} = \mathbf{x}_{\sigma_1} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} \text{apply}(p_{12}, s_1) &= \text{apply}(\mathbf{c}_{(\sigma_1, \sigma_2)} \cdot p'_{12}, \sigma_1 \cdot s'_1) = \sigma_2 \cdot \text{apply}(p'_{12}, s'_1) = \sigma_2 \cdot s'_2 = s_2 \text{ and} \\ \text{apply}(p_{23}, s_2) &= \text{apply}(\mathbf{x}_{\sigma_2} \cdot p'_{23}, \sigma_2 \cdot s'_2) = \text{apply}(p'_{23}, s'_2) = s_3. \end{aligned}$$

17:14 The Normalized Edit Distance with Uniform Operation Costs Is a Metric

Since $apply(p'_{12}, s'_1) = s'_2$ and $apply(p'_{23}, s'_2) = s_3$, by applying the induction hypotheses on s'_1, s'_2, s_3 we get

$$1. apply(h(p'_{13}), s'_1) = s_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. apply(h(p_{13}), s_1) &= apply(x_{\sigma_1} \cdot h(p'_{13}), \sigma_1 \cdot s'_1) = apply(h(p'_{13}), s'_1) = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq 1 + d'_{12} + d'_{23} < 1 + d'_{12} + 1 + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(8) Here $(p_{12}[1], p_{23}[1]) = (c_{(\sigma', \sigma)}, n_\sigma)$.

By definition of compose we get that $p_{13} = c_{(\sigma', \sigma)} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} apply(p_{12}, s_1) &= apply(c_{(\sigma', \sigma)} \cdot p'_{12}, \sigma' \cdot s'_1) = \sigma \cdot apply(p'_{12}, s'_1) = \sigma \cdot s'_2 = s_2 \text{ and} \\ apply(p_{23}, s_2) &= apply(n_\sigma \cdot p'_{23}, \sigma \cdot s'_2) = \sigma \cdot apply(p'_{23}, s'_2) = \sigma \cdot s'_3 = s_3. \end{aligned}$$

Since $apply(p'_{12}, s'_1) = s'_2$ and $apply(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s'_1, s'_2, s'_3 we get

$$1. apply(h(p'_{13}), s'_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. apply(h(p_{13}), s_1) &= apply(c_{(\sigma', \sigma)} \cdot h(p'_{13}), \sigma' \cdot s'_1) = \sigma \cdot apply(h(p'_{13}), s'_1) = \sigma \cdot s'_3 = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq 1 + d'_{12} + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(9) Here $(p_{12}[1], p_{23}[1]) = (v_\sigma, n_\sigma)$.

By definition of compose we get that $p_{13} = v_\sigma \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} apply(p_{12}, s_1) &= apply(v_\sigma \cdot p'_{12}, s_1) = \sigma \cdot apply(p'_{12}, s_1) = \sigma \cdot s'_2 = s_2 \text{ and} \\ apply(p_{23}, s_2) &= apply(n_\sigma \cdot p'_{23}, \sigma \cdot s'_2) = \sigma \cdot apply(p'_{23}, s'_2) = \sigma \cdot s'_3 = s_3. \end{aligned}$$

Since $apply(p'_{12}, s_1) = s'_2$ and $apply(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s_1, s'_2, s'_3 we get

$$1. apply(h(p'_{13}), s_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

$$\begin{aligned} 1. apply(h(p_{13}), s_1) &= apply(v_\sigma \cdot h(p'_{13}), s_1) = \sigma \cdot apply(h(p'_{13}), s_1) = \sigma \cdot s'_3 = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq 1 + d'_{12} + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(10) Here $(p_{12}[1], p_{23}[1]) = (v_{\sigma_1}, c_{(\sigma_1, \sigma_2)})$.

By definition of compose we get that $p_{13} = v_{\sigma_2} \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} apply(p_{12}, s_1) &= apply(v_{\sigma_1} \cdot p'_{12}, s_1) = \sigma_1 \cdot apply(p'_{12}, s_1) = \sigma_1 \cdot s'_2 = s_2 \text{ and} \\ apply(p_{23}, s_2) &= apply(c_{(\sigma_1, \sigma_2)} \cdot p'_{23}, \sigma_1 \cdot s'_2) = \sigma_2 \cdot apply(p'_{23}, s'_2) = \sigma_2 \cdot s'_3 = s_3. \end{aligned}$$

Since $apply(p'_{12}, s_1) = s'_2$ and $apply(p'_{23}, s'_2) = s'_3$, by applying the induction hypotheses on s_1, s'_2, s'_3 we get

$$1. apply(h(p'_{13}), s_1) = s'_3 \quad 2. d'_{13} \leq d'_{12} + d'_{23} \quad 3. l'_{13} \geq \max\{l'_{12}, l'_{23}\}.$$

Therefore

$$\begin{aligned} 1. apply(h(p_{13}), s_1) &= apply(v_{\sigma_2} \cdot h(p'_{13}), s_1) = \sigma_2 \cdot apply(h(p'_{13}), s_1) = \sigma_2 \cdot s'_3 = s_3 \\ 2. d_{13} &= 1 + d'_{13} \leq 1 + d'_{12} + d'_{23} < 1 + d'_{12} + 1 + d'_{23} = d_{12} + d_{23} \\ 3. l_{13} &= 1 + l'_{13} \geq 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}. \end{aligned}$$

(11) Here $(p_{12}[1], p_{23}[1]) = (v_\sigma, x_\sigma)$.

By definition of compose we get that $p_{13} = b \cdot p'_{13}$. From *apply* we have

$$\begin{aligned} apply(p_{12}, s_1) &= apply(v_\sigma \cdot p'_{12}, s_1) = \sigma \cdot apply(p'_{12}, s_1) = \sigma \cdot s'_2 = s_2 \text{ and} \\ apply(p_{23}, s_2) &= apply(x_\sigma \cdot p'_{23}, \sigma \cdot s'_2) = apply(p'_{23}, s'_2) = s_3. \end{aligned}$$

Since $apply(p'_{12}, s_1) = s'_2$ and $apply(p'_{23}, s'_2) = s_3$, by applying the induction hypotheses on s_1, s'_2, s_3 we get

1. $apply(h(p'_{13}), s_1) = s_3$
2. $d'_{13} \leq d'_{12} + d'_{23}$
3. $l'_{13} \geq \max\{l'_{12}, l'_{23}\}$.

Therefore

1. $apply(h(p_{13}), s_1) = apply(h(p'_{13}), s_1) = s_3$
2. $d_{13} = 2 + d'_{13} \leq 1 + d'_{12} + 1 + d'_{23} = d_{12} + d_{23}$
3. $l_{13} = 2 + l'_{13} > 1 + \max\{l'_{12}, l'_{23}\} = \max\{1 + l'_{12}, 1 + l'_{23}\} = \max\{l_{12}, l_{23}\}$. ◀

The sequel makes use of the following lemmas regarding non-negative integers d and l .

► **Lemma 22.** *If $d \leq l$ then $\frac{d+1}{l+1} \geq \frac{d}{l}$*

Proof. $\frac{d+1}{l+1} = \frac{l(d+1)}{l(l+1)} \geq \frac{d(l+1)}{l(l+1)} = \frac{d}{l}$. ◀

► **Lemma 23.** *If $d_{13} \leq d_{12} + d_{23}$ and $l_{13} \geq \max\{l_{12}, l_{23}\}$ then $\frac{d_{12}}{l_{12}} + \frac{d_{23}}{l_{23}} \geq \frac{d_{13}}{l_{13}}$.*

Proof. $\frac{d_{13}}{l_{13}} \leq \frac{d_{12} + d_{23}}{l_{13}} = \frac{d_{12}}{l_{13}} + \frac{d_{23}}{l_{13}} \leq \frac{d_{12}}{l_{12}} + \frac{d_{23}}{l_{23}}$. ◀

Recall that $cost$ is defined as wgt divided by len . Let p_{13} be the string obtained by compose in Prop. 21. Then by items 2 and 3 we know that

$$wgt(p_{13}) \leq wgt(p_{12}) + wgt(p_{23}) \quad (3) \quad len(p_{13}) \geq \max\{len(p_{12}), len(p_{23})\} \quad (4)$$

We can thus conclude from Lem. 23 that the cost of the path obtained by $cmps_h$ is at most the sum of the costs of the edit paths from which it was obtained, as stated in the following corollary.

► **Corollary 24.** *Let $s_1, s_2, s_3 \in \Sigma^*$ and p_{12}, p_{23} be edit paths, such that $apply(p_{12}, s_1) = s_2$, $apply(p_{23}, s_2) = s_3$. Let $p_{13} = cmps_h(p_{12}, p_{23})$. Then $cost(p_{13}) \leq cost(p_{12}) + cost(p_{23})$.*

We are not done yet, since p_{13} contains \mathbf{b} symbols, and thus it is not really an edit path. Let k be the number of \mathbf{b} 's in p_{13} . Then $wgt(p_{13}) = 2k + wgt(h(p_{13}))$ and $len(p_{13}) = 2k + len(h(p_{13}))$, applying $2k$ times Lem. 22, we conclude that $\frac{wgt(p_{13})}{len(p_{13})} \geq \frac{wgt(h(p_{13}))}{len(h(p_{13}))}$.

► **Corollary 25.** $cost(p) \geq cost(h(p))$

► **Proposition 26.** *The normalized edit distance obeys the triangle inequality.*

Proof. Let $s_1, s_2, s_3 \in \Sigma^*$ and p_{12}, p_{23} be optimal edit paths. That is, $apply(p_{12}, s_1) = s_2$ and $apply(p_{23}, s_2) = s_3$ and $NED(s_1, s_2) = cost(p_{12})$ and $NED(s_2, s_3) = cost(p_{23})$. Let $p_{13} = cmps_h(p_{12}, p_{23})$. From Cor. 24 we get that $cost(p_{13}) \leq cost(p_{12}) + cost(p_{23})$. From Prop. 21 it holds that $h(p_{13})$ is a valid edit path over Γ_Σ . From Cor. 25 we get that $cost(h(p_{13})) \leq cost(p_{13})$. By definition of NED as it chooses the minimal cost of an edit path, $NED(s_1, s_3) \leq cost(h(p_{13}))$. To conclude, we get $NED(s_1, s_3) \leq NED(s_1, s_2) + NED(s_2, s_3)$. ◀

► **Theorem 27.** *The Normalized Levenshtein Distance NED (provided in Def. 4) with uniform costs (i.e., where the cost of all inserts, deletes and swaps are some constant c) is a metric on the space Σ^* .*

Proof. The first two conditions of being a metric follow from Prop. 7. The third condition, namely triangle inequality, follows from Prop. 26. ◀

5 Conclusions

We closed a gap regarding the normalized version of the editing distance proposed by Marzal and Vidal, denoted here as NED. Marzal and Vidal noted that NED is not a metric in general and left open the question of whether it is a metric in case all weights are equal. This open point, spawned two versions of a normalized editing distance that have been proven to be metrics – GED and CED. We proved that, with uniform weights, NED is also a metric. To pinpoint the benefits of NED over the other distances we have defined a number of properties that NED maintains and CED and/or GED do not. The motivation for formulating the properties as we did comes from formal verification, so is our interest in uniform weights.

References

- 1 Abdullah N Arslan and Omer Egecioglu. Efficient algorithms for normalized edit distance. *Journal of Discrete Algorithms*, 1(1):3–20, 2000.
- 2 Colin de la Higuera and Luisa Micó. A contextual normalised edit distance. In *Proceedings of the 24th International Conference on Data Engineering Workshops, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 354–361. IEEE Computer Society, 2008.
- 3 Emmanuel Filiot, Nicolas Mazzocchi, Jean-François Raskin, Sriram Sankaranarayanan, and Ashutosh Trivedi. Weighted transducers for robustness verification. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, pages 17:1–17:21, 2020.
- 4 Dana Fisman, Joshua Grogin, Oded Margalit, and Gera Weiss. The normalized edit distance with uniform operation costs is a metric. *CoRR*, abs/2201.06115, 2022. [arXiv:2201.06115](https://arxiv.org/abs/2201.06115).
- 5 Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.
- 6 Yujian Li and Bi Liu. A normalized levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1091–1095, 2007.
- 7 Christopher C. Little. <https://abydos.readthedocs.io/en/latest/abydos.distance.html#abydos.distance.HigueraMico>.
- 8 Andrés Marzal and Enrique Vidal. Computation of normalized edit distance and applications. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(9):926–932, 1993.
- 9 Enrique Vidal, Andrés Marzal, and Pablo Aibar. Fast computation of normalized edit distances. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(9):899–902, 1995. doi:10.1109/34.406656.

A Appendix

We provide here two proofs that we could not fit in the body of the paper.

► **Proposition 7 (restated).** *Let $s, s_1, s_2 \in \Sigma^*$. Then*

1. $NED(s, s) = 0$
2. if $s_1 \neq s_2$ then $NED(s_1, s_2) > 0$
3. $NED(s_1, s_2) = NED(s_2, s_1)$

Proof. First clearly, if $s \neq \varepsilon$ then $\mathbf{n}^{|s|}$ is an edit path from s to s , and thus $NED(s, s) = \frac{0}{|s|} = 0$. Second, if $s_1 \neq s_2$ then any edit path from s_1 to s_2 must contain at least one non- \mathbf{n} character. Thus, its cost is $\frac{d}{l}$ for some $d > 0$, implying $NED(s_1, s_2) > 0$. Third, assume $p_{12} = \gamma_1 \gamma_2 \dots \gamma_k$ is an edit path from s_1 to s_2 . Define $\bar{p}_{12} = \bar{\gamma}_1 \bar{\gamma}_2 \dots \bar{\gamma}_k$ where

$$\bar{\gamma} = \begin{cases} \mathbf{n}_\sigma & \text{if } \gamma = \mathbf{n}_\sigma \\ \mathbf{c}_{(\sigma_2, \sigma_1)} & \text{if } \gamma = \mathbf{c}_{(\sigma_1, \sigma_2)} \\ \mathbf{x}_\sigma & \text{if } \gamma = \mathbf{v}_\sigma \\ \mathbf{v}_\sigma & \text{if } \gamma = \mathbf{x}_\sigma \end{cases}$$

Then $\overline{p_{12}}$ is an edit path from s_2 to s_1 and the cost they induce is the same. Hence, if p_{12} is a minimal edit path from s_1 to s_2 then $\overline{p_{12}}$ is a minimal edit path from s_2 to s_1 implying $\text{NED}(s_1, s_2) = \text{NED}(s_2, s_1)$. ◀

► **Claim 18** (restated). *Let $\Sigma, \Sigma_1, \Sigma_2$ be disjoint nonempty alphabets. Let $s'_1 \in \Sigma \uplus \Sigma_1$ and $s'_2 \in \Sigma \uplus \Sigma_2$ and p' an edit path transforming s'_1 to s'_2 . There exists an edit path p transforming $\pi_\Sigma(s'_1)$ to $\pi_\Sigma(s'_2)$ such that $\text{cost}(p) \leq \text{cost}(p')$.*

Proof. Let $\gamma \in \Gamma$, $p' \in \Gamma_{\Sigma \uplus \Sigma_1 \uplus \Sigma_2}^*$. We define $f : \Gamma_{\Sigma \uplus \Sigma_1 \uplus \Sigma_2} \rightarrow \Gamma_\Sigma$ as follows

$$f(\gamma) = \begin{cases} 1_\sigma & \text{if } \gamma = 1_\sigma \text{ for some } 1 \in \{\mathbf{v}, \mathbf{x}, \mathbf{n}\} \text{ and } \sigma \in \Sigma \\ c_{\sigma, \sigma'} & \text{if } \gamma = c_{\sigma, \sigma'} \text{ and } \sigma, \sigma' \in \Sigma \\ \mathbf{v}_\sigma & \text{if } \gamma = c_{\sigma_1, \sigma} \text{ and } \sigma_1 \in \Sigma_1, \sigma \in \Sigma \\ \mathbf{x}_\sigma & \text{if } \gamma = c_{\sigma, \sigma_2} \text{ and } \sigma \in \Sigma, \sigma_2 \in \Sigma_2 \\ \varepsilon & \text{otherwise} \end{cases}$$

Let $p = f(p')$ where $f : \Gamma_{\Sigma \uplus \Sigma_1 \uplus \Sigma_2}^* \rightarrow \Gamma_\Sigma^*$ is the natural extension of f defined by $f(\gamma_1 \dots \gamma_m) = f(\gamma_1) \dots f(\gamma_m)$.

It is not hard to see that p is an edit path from $\pi_\Sigma(s'_1)$ to $\pi_\Sigma(s'_2)$. Since all removed edit operations have cost 1 we get from Lem. 22 that $\text{cost}(p) \leq \text{cost}(p')$ ◀


The Dynamic k -Mismatch Problem

Raphaël Clifford ✉ 

Department of Computer Science, University of Bristol, UK

Paweł Gawrychowski ✉ 

Institute of Computer Science, University of Wrocław, Poland

Tomasz Kociumaka ✉ 

University of California, Berkeley, CA, USA

Daniel P. Martin ✉ 

The Alan Turing Institute, British Library, London, UK

Przemysław Uznański ✉ 

Institute of Computer Science, University of Wrocław, Poland

Abstract

The text-to-pattern Hamming distances problem asks to compute the Hamming distances between a given pattern of length m and all length- m substrings of a given text of length $n \geq m$. We focus on the well-studied k -mismatch version of the problem, where a distance needs to be returned only if it does not exceed a threshold k . Moreover, we assume $n \leq 2m$ (in general, one can partition the text into overlapping blocks). In this work, we develop data structures for the dynamic version of the k -mismatch problem supporting two operations: An update performs a single-letter substitution in the pattern or the text, whereas a query, given an index i , returns the Hamming distance between the pattern and the text substring starting at position i , or reports that the distance exceeds k .

First, we describe a simple data structure with $\tilde{O}(1)$ update time and $\tilde{O}(k)$ query time. Through considerably more sophisticated techniques, we show that $\tilde{O}(k)$ update time and $\tilde{O}(1)$ query time is also achievable. These two solutions likely provide an essentially optimal trade-off for the dynamic k -mismatch problem with $m^{\Omega(1)} \leq k \leq \sqrt{m}$: we prove that, in that case, conditioned on the 3SUM conjecture, one cannot simultaneously achieve $k^{1-\Omega(1)}$ time for all operations (updates and queries) after $n^{\mathcal{O}(1)}$ -time initialization. For $k \geq \sqrt{m}$, the same lower bound excludes achieving $m^{1/2-\Omega(1)}$ time per operation. This is known to be essentially tight for constant-sized alphabets: already Clifford et al. (STACS 2018) achieved $\tilde{O}(\sqrt{m})$ time per operation in that case, but their solution for large alphabets costs $\tilde{O}(m^{3/4})$ time per operation. We improve and extend the latter result by developing a trade-off algorithm that, given a parameter $1 \leq x \leq k$, achieves update time $\tilde{O}(\frac{m}{k} + \sqrt{\frac{mk}{x}})$ and query time $\tilde{O}(x)$. In particular, for $k \geq \sqrt{m}$, an appropriate choice of x yields $\tilde{O}(\sqrt[3]{mk})$ time per operation, which is $\tilde{O}(m^{2/3})$ when only the trivial threshold $k = m$ is provided.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern matching, Hamming distance, dynamic algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.18

Funding *Tomasz Kociumaka*: Partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

Przemysław Uznański: Supported by Polish National Science Centre grant 2019/33/B/ST6/00298.

Acknowledgements We are grateful to Ely Porat and Shay Golan for insightful conversations about the dynamic k -mismatch problem at an early stage of this work.



© Raphaël Clifford, Paweł Gawrychowski, Tomasz Kociumaka, Daniel P. Martin, and Przemysław Uznański;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 18; pp. 18:1–18:15



Leibniz International Proceedings in Informatics

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

1 Introduction

The development of dynamic data structures for string problems has become a topic of renewed interest in recent years (see, for example, [2, 3, 4, 5, 10, 11, 13, 15, 20] and references therein). Our focus will be on approximate pattern matching, where the general problem is as follows: Given a pattern of length m and a longer text of length n , return the value of a distance function between the pattern and substrings of the text.

We develop new dynamic data structures for a thresholded version of the Hamming distance function, known as the k -mismatch function. In this setting, we only need to report the Hamming distance if does not exceed k . The k -mismatch problem is well studied in the offline setting, where all alignments of the pattern with the text substring that meet this threshold must be found. In 1980s, an $\mathcal{O}(nk)$ -time algorithm was given [21], and this stood as the record for a over a decade. However, in the last twenty years, significant progress has been made. In a breakthrough result, Amir et al. [6] gave $\mathcal{O}(n\sqrt{k\log k})$ -time and $\mathcal{O}(n + \frac{k^3 \log k}{m})$ -time algorithms, which were subsequently improved to $\mathcal{O}(n \log^{\mathcal{O}(1)} m + \frac{nk^2 \log k}{m})$ time [12], $\mathcal{O}(n \log^2 m \log \sigma + \frac{nk\sqrt{\log m}}{\sqrt{m}})$ time [16], and finally to $\mathcal{O}(n + \min(\frac{nk\sqrt{\log m}}{\sqrt{m}}, \frac{nk^2}{m}))$ time [9].

In the dynamic k -mismatch problem, there are two input strings: a pattern P of length m and a text T of length $n \geq m$. For a query at index i , the data structure must return the Hamming distance between P and $T[i \dots i + m]$ if the Hamming distance is less than k , and ∞ otherwise. The queries can be interspersed with updates of the form $\text{Update}(S, i, x)$, which assign $S[i] := x$, where S can be either the pattern or the text. There are two naive approaches for solving the dynamic problem. The first is to rerun a static offline algorithm after each update, and then have constant-time queries. The second is to simply modify the input at each update and compute the Hamming distance naively for each query. Our goal is to perform better than these naive solutions.

We primarily focus on the case when $n = \Theta(m)$ (in general, one can partition the text into $\Theta(\frac{n}{m})$ overlapping blocks of length $\Theta(m)$). When $k = m$ and $\sigma = n^{\mathcal{O}(1)}$, known upper bounds and conditional lower bounds match up to a subpolynomial factor: There exists a dynamic data structure with an $\mathcal{O}(\sqrt{n \log n} \cdot \sigma)$ upper bound for both updates and queries and an almost matching $n^{1/2 - \Omega(1)}$ lower bound [13] conditioned on the hardness of the online matrix-vector multiplication problem. Although there is no existing work directly on the dynamic k -mismatch problem we consider, it was shown very recently that a compact representation of all k -mismatch occurrences can be reported in $\tilde{\mathcal{O}}(k^2)$ time¹ after each $\mathcal{O}(\log n)$ -time update [11].

We give three data structures for the dynamic k -mismatch problem. The first has update time of $\tilde{\mathcal{O}}(1)$ and a query time of $\tilde{\mathcal{O}}(k)$. The main tool we use is the dynamic strings data structure [15] which allows enumerating mismatches in $\mathcal{O}(\log n)$ time each. The second has update time $\tilde{\mathcal{O}}(k)$ and a query time of $\tilde{\mathcal{O}}(1)$. Here, we build on the newly developed generic solution for the static k -mismatch problem from [11]. The third data structure, optimized for $k \geq \sqrt{n}$, gives a trade-off between update and query times. The overall approach is a lazy rebuilding scheme using the state-of-the-art offline k -mismatch algorithm. In order to achieve a fast solution, we handle instances with many and few $2k$ -mismatch occurrences differently. Basing on combinatorial insights developed in the sequence of papers on the offline and streaming versions of the k -mismatch problem [9, 12, 14, 16, 17], we are able to achieve update time $\tilde{\mathcal{O}}(\frac{n}{k} + \sqrt{\frac{nk}{x}})$ and query time $\tilde{\mathcal{O}}(x)$ for any trade-off parameter

¹ The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors.

$x \in [1..k]$ provided at initialization.² To put the trade-off complexity in context, we note that, e.g., when $k = m$, this allows achieving $U(n, k) = Q(n, k) = \tilde{O}(n^{2/3})$, which improves upon an $\tilde{O}(n^{3/4})$ bound presented in [13] (where only the case of $k = m$ is considered).

We also show conditional lower bounds which are in most cases within subpolynomial factors of our upper bounds. For the case where the text length is linear in the length of the pattern, we do this by reducing from the 3SUM conjecture [23]. However, in the case that the text is much longer than the pattern, our reduction requires the Online Matrix vector conjecture [18]. Interestingly the lower bound for the superlinear case is asymmetric between the query and update time.

2 Preliminaries

In this section, we provide the required basic definitions. We begin with the string distance metric which will be used throughout.

► **Definition 1** (Hamming Distance). *The Hamming distance between two strings S, R of the same length is defined as $\text{HD}(S, R) = |\{i : S[i] \neq R[i]\}|$.*

From this point forward, for simplicity of exposition, we will assume that the pattern is half the length of the text. All our upper bounds are straightforward to generalise to a text whose length is linear in the length of the pattern. In Theorem 27, we show higher lower bounds for the case where the text is much longer than the pattern.

We can now define the central dynamic data structure problem we consider in this paper.

► **Definition 2** (Dynamic k -Mismatch Problem). *Let P be a pattern of length m and T be a text of length $n \leq 2m$. For $i \in [0..n-m]$, a query $\text{Query}(i)$ must return $\text{HD}(T[i..i+m], P)$ if $\text{HD}(T[i..i+m], P) \leq k$, and ∞ otherwise. The queries can be interspersed with updates of the form $\text{Update}(S, i, x)$ which assign $S[i] := x$, where S can be the pattern or the text.*

For the remainder of the paper, we use $Q(n, k)$ and $U(n, k)$ to be the time complexity of Query and Update, respectively. If $n > 2m$, then a standard reduction yields $\mathcal{O}(Q(m, k))$ -time queries, $\mathcal{O}(U(m, k))$ -time updates in T , and $\mathcal{O}(\frac{n}{m}Q(m, k))$ -time updates in P .

3 Upper Bounds

In this section, we provide three solutions of the dynamic k -mismatch problem. We start with a simple application of dynamic strings resulting in $\tilde{O}(k)$ query time and $\tilde{O}(1)$ update time.

The data structure of Gawrychowski et al. [15] maintains a dynamic family \mathcal{X} of strings of total length N supporting the following updates:³

- Insert to \mathcal{X} a given string S (in time $\mathcal{O}(|S| + \log N)$).
- Insert to \mathcal{X} the concatenation of two strings already in \mathcal{X} (in time $\mathcal{O}(\log N)$).
- Insert to \mathcal{X} an arbitrary prefix or suffix of a string already in \mathcal{X} (in time $\mathcal{O}(\log N)$).

Queries include $\mathcal{O}(1)$ -time computation of the longest common prefix of two strings in \mathcal{X} .

² Throughout this paper, we denote $[a..b] = \{i \in \mathbb{Z} : a \leq i \leq b\}$ and $[a..b) = \{i \in \mathbb{Z} : a \leq i < b\}$.

³ This data structure is Las-Vegas randomized, and the running times are valid with high probability with respect to N . A deterministic version, using [1] and deterministic dynamic dictionaries, has an $\mathcal{O}(\log N)$ -factor overhead in the running times, which translates to an $\mathcal{O}(\log n)$ -factor overhead in the query and update times of all our randomized algorithms for the dynamic k -mismatch problem.

► **Theorem 3.** *There exists a Las-Vegas randomized algorithm for the dynamic k -mismatch problem satisfying $U(n, k) = \mathcal{O}(\log n)$ and $Q(n, k) = \mathcal{O}(k \log n)$ with high probability.*

Proof. We maintain a dynamic string collection \mathcal{X} of [15] containing P and T . Given that a string S' resulting from setting $S[i] := x$ in a string $S \in \mathcal{X}$ is the concatenation of a prefix $S[0..i)$, the new character x , and a suffix $S[i+1..|S|)$, it is straightforward to construct S' with $\mathcal{O}(1)$ auxiliary strings added to \mathcal{X} . Hence, we implement an update in $\mathcal{O}(\log n)$ time.

Armed with this tool, we perform dynamic k -mismatch queries by so-called “kangaroo jumps” [21]. That is, we align the pattern with $T[i..i+m)$, where i is the query position in the text T , and we repeatedly extend the match we have found so far until we reach a fresh mismatch. Each longest common extension query can be implemented in $\mathcal{O}(\log n)$ time. For this, we extract the relevant suffixes of P and T (we insert them to \mathcal{X} in $\mathcal{O}(\log n)$ time each) and ask for their longest common prefix (which costs $\mathcal{O}(1)$ time). As we stop once $k+1$ mismatches have been found or once we have reached the end of the text or pattern, the total query time is $\mathcal{O}(k \log n)$. ◀

3.1 Faster Queries, Slower Updates

Given the result above, a natural question is whether there exists an approach with an efficient query algorithm, in return for a slower update algorithm. We answer affirmatively in this section based on a recent work of Charalampopoulos et al. [11].

3.1.1 The PILLAR model

Charalampopoulos et al. [11] developed a generic static algorithm for the k -mismatch problem. They formalized their solution using an abstract interface, called the **PILLAR model**, which captures certain primitive operations that can be implemented efficiently in all settings considered in [11]. Thus, we bound the running times in terms of **PILLAR** operations – if the algorithm uses more time than **PILLAR** operations, we also specify the extra running time.

In the **PILLAR** model, we are given a family of strings \mathcal{X} for preprocessing. The elementary objects are fragments $X[\ell..r)$ of strings $X \in \mathcal{X}$. Initially, the model provides access to each $X \in \mathcal{X}$ interpreted as $X[0..|X|)$. Other fragments can be obtained through an **Extract** operation.

- **Extract**(S, ℓ, r): Given a fragment S and positions $0 \leq \ell \leq r \leq |S|$, extract the (sub)fragment $S[\ell..r)$, which is defined as $X[\ell'+\ell.. \ell'+r)$ if $S = X[\ell'..r')$ for $X \in \mathcal{X}$. Furthermore, the following primitive operations are supported in the **PILLAR** model:
- **LCP**(S, T): Compute the length of the longest common prefix of S and T .
- **LCP^R**(S, T): Compute the length of the longest common suffix of S and T .
- **IPM**(P, T): Assuming that $|T| \leq 2|P|$, compute the occurrences of P in T , i.e., $\text{Occ}(P, T) = \{i \in [0..|T| - |P|] : P = T[i..i + |P|)\}$ represented as an arithmetic progression.
- **Access**(S, i): Retrieve the character $S[i]$.
- **Length**(S): Compute the length $|S|$ of the string S .

Among several instantiations of the model, Charalampopoulos et al. [11, Section 7.3] showed that the primitive **PILLAR** operations can be implemented in $\mathcal{O}(\log^2 N)$ time on top of the data structure for dynamic strings [15], which we recalled above. Consequently, we are able to maintain two dynamic strings P and T subject to character substitutions, achieving $\mathcal{O}(\log^2 n)$ -time elementary **PILLAR** operations and $\mathcal{O}(\log n)$ -time updates.

► **Corollary 4.** *Let T be a dynamic string of length n and P be a dynamic string of length $m \leq n$, both of which can be updated via substitutions of single characters. There exists a Las-Vegas randomized data structure supporting the PILLAR operations on $\mathcal{X} = \{T, P\}$ in $\mathcal{O}(\log^2 n)$ time w.h.p. and updates in $\mathcal{O}(\log n)$ time w.h.p.*

3.1.2 The Static k -Mismatch Problem

The (static) k -mismatch problem consists in computing $\text{Occ}_k(P, T) = \{i \in [0..n - m] : \text{HD}(P, T[i..i + m]) \leq k\}$, with each position $i \in \text{Occ}_k(P, T)$ reported along with the corresponding Hamming distance $d_i := \text{HD}(P, T[i..i + m])$. Charalampopoulos et al. [11, Theorem 3.1 and Corollary 3.5] proved that $\text{Occ}_k(P, T)$ admits a compact representation: this set can be decomposed into $\mathcal{O}(\frac{n}{m} \cdot k^2)$ disjoint arithmetic progressions so that occurrences in a single progression share the same Hamming distance d_i . Moreover, all the non-trivial progressions (i.e., progressions with two or more terms) share the same difference. The following algorithm gives this compact representation on the output.

► **Theorem 5** ([11, Main Theorem 8]). *There exists a PILLAR-model algorithm that, given a pattern P of length m , a text T of length $n \geq m$, and a positive integer $k \leq m$, solves the k -mismatches problem in $\mathcal{O}(\frac{n}{m} \cdot k^2 \log \log k)$ time using $\mathcal{O}(\frac{n}{m} \cdot k^2)$ PILLAR operations.*

3.1.3 Warm-Up Algorithm

Intuitively, the algorithm of Theorem 5 precomputes the answers to all queries $\text{Query}(i)$ with $i \in [0..n - m]$. Hence, a straightforward solution to the dynamic k -mismatch problem would be to maintain the data structure of Corollary 4, use the algorithm of Theorem 5 after each update, and then retrieve the precomputed answers for each query asked. The data structure described below follows this strategy, making sure that the compact representation of $\text{Occ}_k(P, T)$ is augmented with infrastructure for efficient random access.

► **Proposition 6.** *There exists a Las-Vegas algorithm for the dynamic k -mismatch problem satisfying $U(n, k) = \mathcal{O}(k^2 \log^2 n)$ and $Q(n, k) = \mathcal{O}(\log \log n)$ with high probability.*

Proof. We maintain a PILLAR-model implementation of $\mathcal{X} = \{P, T\}$ using Corollary 4; this costs $\mathcal{O}(\log n)$ time per update and provides $\mathcal{O}(\log^2 n)$ -time primitive PILLAR operations.

Following each update, we use Theorem 5 so that a space-efficient representation of $\text{Occ}_k(P, T)$ is computed in $\mathcal{O}(k^2 \log^2 n)$ time (recall that $m = \Theta(n)$). This output is then post-processed as described below. Let q be the common difference of non-trivial arithmetic progression forming $\text{Occ}_k(P, T)$; we set $q = 1$ if all progressions are trivial. Consider the indices $i \in [0..n - m]$ ordered by $(i \bmod q, i)$, that is, first by the remainder modulo q and then by the index itself. In this ordering, each arithmetic progression contained in the output $\text{Occ}_k(P, T)$ yields a contiguous block of indices i with a common finite answer to queries $\text{Query}(i)$. The goal of post-processing is to store the sequence of answers using run-length encoding (with run boundaries kept in a predecessor data structure). This way, for each of the $\mathcal{O}(k^2)$ arithmetic progressions in $\text{Occ}_k(P, T)$, the corresponding answers $\text{Query}(i)$ can be set in $\mathcal{O}(\log \log n)$ time to the common value d_i reported along with the progression. In total, the post-processing time is therefore $\mathcal{O}(k^2 \log \log n)$.

At query time, any requested value $\text{Query}(i)$ can be retrieved in $\mathcal{O}(\log \log n)$ time. ◀

3.1.4 Structural Insight

In order to improve the update time, we bring some of the combinatorial insight from [11].

A string is *primitive* if it is not a string power with an integer exponent strictly greater than 1. For a non-empty string Q , we denote by Q^∞ an infinite string obtained by concatenating infinitely many copies of Q . For an arbitrary string S , we further set $\text{HD}(S, Q^*) = \text{HD}(S, Q^\infty[0..|S|])$. In other words, the $\text{HD}(\cdot, \cdot^*)$ function generalizes $\text{HD}(\cdot, \cdot)$ in that the second string is cyclically extended to match the length of the first one. We use the same convention to define $M(S, Q^*) = \{i : S[i] \neq Q^\infty[i]\} = \{i : S[i] \neq Q[i \bmod |Q|]\}$.

► **Proposition 7** ([11, Theorems 3.1 and 3.2]). *Let P be a pattern of length m , let T be a text of length $n \leq \frac{3}{2}m$, and let $k \leq m$ be a positive integer. At least one of the following holds:*

1. *The number of k -mismatch occurrences of P in T is $|\text{Occ}_k(P, T)| \leq 864k$.*
2. *There is a primitive string Q of length $|Q| \leq \frac{m}{128k}$ such that $\text{HD}(P, Q^*) < 2k$. Moreover, if $\text{Occ}_k(P, T) \neq \emptyset$ and (2) holds, then a fragment $T' = T[\min \text{Occ}_k(P, T) .. m + \max \text{Occ}_k(P, T)]$ satisfies $\text{HD}(T', Q^*) < 6k$ and every position in $\text{Occ}_k(P, T')$ is a multiple of $|Q|$.*

We also need a characterization of the values $\text{HD}(P, T'[j|Q| .. m + j|Q|])$.

► **Proposition 8** ([11, Lemma 3.3 and Claim 3.4]). *Let P be a pattern of length m , let T be a text of length n , and let $k \leq m$ be a positive integer. For any non-empty string Q and non-negative integer $j \leq \frac{n-m}{|Q|}$, we have*

$$\text{HD}(P, T'[j|Q| .. m + j|Q|]) = |M(P, Q^*)| + |M(T, Q^*) \cap [j|Q| .. m + j|Q|]| - \mu_j,$$

where

$$\mu_j = \sum_{\rho \in M(P, Q^*), \tau \in M(T, Q^*) : \tau = j|Q| + \rho} 2 - \text{HD}(T[\tau], P[\rho]).$$

3.1.5 Improved Solution

The idea behind achieving $\mathcal{O}(k \log^2 n)$ update time is to run Theorem 5 once every k updates, but with a doubled threshold $2k$ instead of k . The motivation behind this choice of parameters is that if the current instance P, T is obtained by up to k substitutions from a past instance \bar{P}, \bar{T} , then $\text{HD}(P, \bar{P}) + \text{HD}(T, \bar{T}) \leq k$ yields $\text{Occ}_k(P, T) \subseteq \text{Occ}_{2k}(\bar{P}, \bar{T})$. Consequently, the algorithm may safely return ∞ while answering $\text{Query}(i)$ for any position $i \notin \text{Occ}_{2k}(\bar{P}, \bar{T})$.

If the application of Theorem 5 identifies few $2k$ -mismatch occurrences, then we maintain the Hamming distances d_i at these positions throughout the k subsequent updates. Otherwise, we identify Q and T' , as defined in Proposition 7, as well as the sets $M(P, Q^*)$, $M(T', Q^*)$, and the values μ_j of Proposition 8 so that the distances $\text{HD}(P, T'[j|Q| .. m + j|Q|])$ can be retrieved efficiently.

The latter task requires extending Theorem 5 so that the string Q and the sets $M(P, Q^*)$, $M(T', Q^*)$ can be constructed whenever there are many k -mismatch occurrences.

► **Lemma 9.** *There exists a PILLAR-model algorithm that, given a pattern P of length m , a text T of length $n \leq \frac{3}{2}m$, and a positive integer $k \leq m$, returns $\text{Occ}_k(P, T)$ along with the corresponding Hamming distances provided that $|\text{Occ}_k(P, T)| \leq 864k$, or, otherwise, returns the fragment $T' = T[\min \text{Occ}_k(P, T) .. m + \max \text{Occ}_k(P, T)]$, a string Q such that $\text{HD}(P, Q^*) < 2k$, $\text{HD}(T', Q^*) < 6k$, and $\text{Occ}_k(P, T')$ consists of multiples of $|Q|$, and sets $M(P, Q^*)$, $M(T', Q^*)$. The algorithm takes $\mathcal{O}(k^2 \log \log k)$ time plus $\mathcal{O}(k^2)$ PILLAR operations.*

Proof. First, we use Theorem 5 in order to construct $\text{Occ}_k(P, T)$ in a compact representation as $\mathcal{O}(k^2)$ arithmetic progressions. Based on this representation, both $|\text{Occ}_k(P, T)|$ and T' can be computed in $\mathcal{O}(k^2)$ time. If $|\text{Occ}_k(P, T)| \leq 864k$, then $\text{Occ}_k(P, T)$ is converted to a plain representation (with each position reported explicitly along with the corresponding Hamming distance). Otherwise, we use the $\text{Analyze}(P, k)$ procedure of [11, Lemma 4.4]. This procedure costs $\mathcal{O}(k)$ time in the PILLAR model, and it detects a structure within the pattern P that can be of one of three types. A possible outcome includes a primitive string Q such that $|Q| \leq \frac{m}{128k}$ and $\text{HD}(P, Q^*) < 8k$. Moreover, the existence of a structure of either of the other two types contradicts $|\text{Occ}_k(P, T)| \leq 864k$ (due to [11, Lemmas 3.8 and 3.11]), and so does $2k \leq \text{HD}(P, Q^*) < 8k$ (due to [11, Lemma 3.14]). Consequently, we are guaranteed to obtain a primitive string Q such that $|Q| \leq \frac{m}{128k}$ and $\text{HD}(P, Q^*) < 2k$, which are precisely the conditions in the second case of Proposition 7. Thus, we conclude that $\text{HD}(T', Q^*) < 6k$ and that $\text{Occ}_k(P, T')$ consists of multiples of $|Q|$. It remains to report $M(P, Q^*)$ and $M(T', Q^*)$. For this task, we employ [11, Corollary 4.2], whose time cost in the PILLAR model is proportional to the output size, i.e., $\mathcal{O}(k)$ for both instances. ◀

We are now ready to describe the dynamic algorithm based on the intuition above. Initially, we only improve the *amortized* query time from $\mathcal{O}(k^2 \log^2 n)$ to $\mathcal{O}(k \log^2 n)$.

► **Proposition 10.** *There exists a Las-Vegas randomized algorithm for the dynamic k -mismatch problem satisfying $Q(n, k) = \mathcal{O}(\log \log n)$ and $U(n, k) = \mathcal{O}(k + \log n)$ with high probability, except that every k th update costs $\mathcal{O}(k^2 \log^2 n)$ time w.h.p.*

Proof. The algorithm logically partitions its runtime into epochs, with k updates in each epoch. The first update in every epoch costs $\mathcal{O}(k^2 \log^2 n)$ time, and the remaining updates cost $\mathcal{O}(k + \log n)$ time. A representation of $\mathcal{X} = \{P, T\}$ supporting the PILLAR operations (Corollary 4) is maintained throughout the execution of the algorithm, while the remaining data is destroyed after each epoch.

Once the arrival of an update marks the beginning of a new epoch, we run the algorithm of Lemma 9 with a doubled threshold $2k$. This procedure costs $\mathcal{O}(k^2 \log^2 n)$ time, and it may have one of two types of outcome.

The first possibility is that it returns a set $O := \text{Occ}_{2k}(P, T)$ of up to $1728k$ positions, with the Hamming distance $d_i := \text{HD}(P, T[i..i+m])$ reported along with each position $i \in O$. Since $d_i > 2k$ for $i \notin O$ and any update may decrease d_i by at most one, we are guaranteed that $\text{Query}(i) = \infty$ can be returned for $i \notin O$ for the duration of the epoch. Consequently, the algorithm only maintains d_i for $i \in O$. For each of the subsequent updates, the algorithm iterates over $i \in O$ and checks if d_i needs to be changed: If the update involves $P[j]$, then both the old and the new value of $P[j]$ are compared against $T[i+j]$. Similarly, if the update involves $T[j]$ and $j \in [i..i+m)$, then both the old and the new value of $T[j]$ are compared against $P[i-j]$. Thus, the update time is $\mathcal{O}(k)$ and the query time is $\mathcal{O}(1)$.

The second possibility is that the algorithm of Lemma 9 results in a fragment $T' = T[\ell..r)$, a string Q , and the mismatching positions $M(P, Q^*)$ and $M(T', Q^*)$. We are then guaranteed that each $2k$ -mismatch occurrence of P in T starts at a position $i \in [\ell..r-m]$ congruent to ℓ modulo $|Q|$. We call these positions *relevant*. As in the previous case, $\text{Query}(i) = \infty$ can be returned for irrelevant i for the duration of the epoch. The Hamming distances d_i at relevant positions are computed using Proposition 8. For this, we maintain $M(P, Q^*)$, $M(T', Q^*)$, and all non-zero values μ_j for $j \in [0.. \lfloor \frac{r-\ell-m}{|Q|} \rfloor]$. Moreover, $M(T', Q^*)$ is stored in a predecessor data structure, and each element of $M(T', Q^*)$ maintains its rank in this set. Every subsequent update affects at most one element of $M(P, Q^*)$ or $M(T', Q^*)$, so these sets can be updated in $\mathcal{O}(1)$ time. Maintaining the predecessor data structure costs further

$\mathcal{O}(\log \log n)$ time, and maintaining the ranks costs up to $\mathcal{O}(\text{HD}(T', Q^*))$ time. In order to update the values μ_j , we proceed as follows. If the update involves a character $P[\rho]$, we iterate over $\tau \in M(T', Q^*)$. If $j = \tau - \rho|Q|$ is an integer between 0 and $\frac{r-\ell-m}{|Q|}$, we may need to update the entry μ_j (which costs constant time). An update involving $T[\ell + \tau]$ is processed in a similar way. Overall, the update time is $\mathcal{O}(\log n + \text{HD}(T', Q^*) + \text{HD}(P, Q^*)) = \mathcal{O}(\log n + k)$ because $\text{HD}(P, Q^*) + \text{HD}(T', Q^*) < 2k + 6k + k = 9k$ holds for the duration of the epoch.

As for the query $\text{Query}(i)$, we return ∞ if i is irrelevant, i.e., $i < \ell$, $i > r - m$, or $i \not\equiv \ell \pmod{|Q|}$. Otherwise, we set $j = \frac{i-\ell}{|Q|}$ and, according to Proposition 8, return $|M(P, Q^*)| + |M(T', Q^*) \cap [j|Q|..j|Q| + m]| - \mu_j$. The second term is determined in $\mathcal{O}(\log \log n)$ time using the predecessor data structure on top of $M(T', Q^*)$ as well as the rank stored for each element of this set. \blacktriangleleft

Finally, we show how to achieve *worst-case* $\mathcal{O}(k \log^2 n)$ update time.

► **Theorem 11.** *There exists a Las-Vegas randomized algorithm for the dynamic k -mismatch problem satisfying $Q(n, k) = \mathcal{O}(\log \log n)$ and $U(n, k) = \mathcal{O}(k \log^2 n)$ with high probability.*

Proof. We maintain two instances of the algorithm of Proposition 10, with updates forwarded to both instances, but queries forwarded to a single instance that is currently *active*.

The algorithm logically partitions its runtime into epochs, with $\frac{1}{2}k$ updates in each epoch. For the two instances, the time-consuming updates are chosen to be the first updates of every even and odd epoch, respectively. Once an instance has to perform a time-consuming update, it becomes inactive (it buffers the subsequent updates and cannot be used for answering queries) and stays inactive for the duration of the epoch. The work needed to perform the time-consuming update is spread across the time allowance for the first half of the epoch, with the time allowance for the second half of the epoch used in order to clear the accumulated backlog of updates (by processing updates at a doubled rate). During this epoch, the other (active) instance processes updates and queries as they arrive in $\mathcal{O}(k \log^2 n)$ and $\mathcal{O}(\log \log n)$ worst-case time, respectively. \blacktriangleleft

3.2 Trade-off between Update Time and Query Time

The next natural question is the existence of a trade-off between the run-times of Theorems 3 and 11. Due to Theorem 23 (in Section 4), the answer is likely negative for $k \ll \sqrt{n}$. Nevertheless, for $k \gg \sqrt{n}$, the trade-off presented below simultaneously achieves $Q(n, k), U(n, k) = k^{1-\Omega(1)}$.

We first recall some combinatorial properties originating from previous work on the k -mismatch problem [9, 12, 14, 17]. The description below mostly follows [17, Section 3].

► **Definition 12** ([12]). *Let X be a string and let d be a non-negative integer. A positive integer $\rho \leq |X|$ is a d -period of X if $\text{HD}(X[\rho..|X|], X[0..|X| - \rho]) \leq d$.*

Recall that $\text{Occ}_k(P, T) = \{i : \text{HD}(P, T[i..i + m]) \leq k\}$ for a pattern P and text T .

► **Lemma 13** ([12]). *If $i, i' \in \text{Occ}_k(P, T)$ are distinct, then $\rho := |i' - i|$ is a $2k$ -period of P . Moreover, if $n \leq 2m$, then ρ is a $(8k + \rho)$ -period of $T[\min \text{Occ}_k(P)..m + \max \text{Occ}_k(P)]$.*

Recall that the L_0 -norm of a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ defined as $\|f\|_0 = |\{x : f(x) \neq 0\}|$. The *convolution* of two functions $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ with finite L_0 -norms is a function $f * g : \mathbb{Z} \rightarrow \mathbb{Z}$ such that

$$[f * g](i) = \sum_{j \in \mathbb{Z}} f(j) \cdot g(i - j).$$

For a string X over Σ and a symbol $c \in \Sigma$, the *characteristic function* of X and c is $X_c : \mathbb{Z} \rightarrow \{0, 1\}$ such that $X_c(i) = 1$ if and only if $X[i] = c$. For a string X , let X^R denote X reversed. The *cross-correlation* of strings X and Y over Σ is a function $X \otimes Y : \mathbb{Z} \rightarrow \mathbb{Z}$ such that

$$X \otimes Y = \sum_{c \in \Sigma} X_c * Y_c^R.$$

► **Fact 14** ([14, Fact 7.1]). *For $i \in [m-1..n]$, we have $[T \otimes P](i) = |P| - \text{HD}(P, T(i-m..i))$. For $i < 0$ and for $i \geq m+n$, we have $[T \otimes P](i) = 0$.*

By Fact 14, $[T \otimes P](i+m-1)$ suffices to compute $\text{HD}(P, T[i..i+m])$ for $i \in [0..n-m]$. The *backward difference* of a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ due to $\rho \in \mathbb{Z}_+$ is $\Delta_\rho[f](i) = f(i) - f(i-\rho)$.

► **Observation 15** ([14, Observation 7.2]). *If a string X has a d -period ρ , then*

$$\sum_{c \in \Sigma} \|\Delta_\rho[X_c]\|_0 \leq 2(d + \rho).$$

Our computation of $T \otimes P$ is based on the following lemma:

► **Lemma 16** (See [17, Lemma 6]). *For every pattern P , text T , and positive integer ρ , we have $\Delta_\rho[\Delta_\rho[T \otimes P]] = \sum_{c \in \Sigma} \Delta_\rho[T_c] * \Delta_\rho[P_c^R]$. Consequently, for every $i \in \mathbb{Z}$,*

$$[T \otimes P](i) = \sum_{j=0}^{\infty} (j+1) \cdot \left[\sum_{c \in \Sigma} \Delta_\rho[T_c] * \Delta_\rho[P_c^R] \right] (i - j\rho).$$

► **Theorem 17.** *There exists a deterministic algorithm for the dynamic k -mismatch problem with $U(n, k) = \mathcal{O}\left(\sqrt{\frac{nk}{x}} + \frac{n}{k}\right)$ and $Q(n, k) = \tilde{\mathcal{O}}(x)$, where x is a trade-off parameter that can be set in $[1..k]$.*

Proof. We solve the problem using a lazy rebuilding scheme similar to that in the proof of Theorem 11. Hence, we can afford update time $\tilde{\mathcal{O}}(n + k\sqrt{n})$ every k updates. Thus, if an incoming update marks the beginning of a new epoch (lasting for k updates), we run a (static) $2k$ -mismatch algorithm [9, 16], resulting in $O := \text{Occ}_{2k}(P, T)$ and the Hamming distances $d_i = \text{HD}(P, T[i..i+m])$ for each $i \in O$. This takes $\tilde{\mathcal{O}}(n + k\sqrt{n})$ time. As in the proof of Proposition 10, since $\text{Occ}_k(P, Q) \subseteq O$ holds for the duration of the epoch, we can safely return ∞ for $\text{Query}(i)$ with $i \notin O$. We distinguish two cases.

$|O| \leq \frac{n}{k}$: We maintain the distances d_i for $i \in O$. As noted above, $\text{Occ}_k(P, T) \subseteq O$ even after k updates. We now observe that any update requires only updating the mismatches for every element of O , with $\mathcal{O}(1)$ cost per element and $\mathcal{O}(\frac{n}{k})$ total; the queries are handled by finding the answer stored for $i \in O$, at $\tilde{\mathcal{O}}(1)$ cost.

$|O| > \frac{n}{k}$: We set ρ to be the distance between two closest elements of O ; we have $\rho \leq k$ due to $|O| > \frac{n}{k}$. By Lemma 13, ρ is a $4k$ -period of P and a $17k$ -period of $T' := T[\min O..m + \max O]$. Moreover, $\text{Occ}_k(P, T) \subseteq O \subseteq [\min O..m + \max O]$ holds for the duration of the epoch, so all k -mismatch occurrences of P in T remain contained in T' .

We have thus reduced our problem to answering queries and performing updates for P and T' . Moreover, we have a positive integer $\rho \leq k$ which is initially a $4k$ -period of P and a $17k$ -period of T' , and, after k updates, it remains a $6k$ -period of P and $19k$ -period of T' . Let us define the *weight* of $c \in \Sigma$ as $\|\Delta_\rho[P_c^R]\|_0 + \|\Delta_\rho[T'_c]\|_0$; by Observation 15, the total weight across $c \in \Sigma$ remains $\mathcal{O}(k)$.

18:10 The Dynamic k -Mismatch Problem

We proceed as follows. We maintain $\Delta_\rho(P_c^R) * \Delta_\rho(T'_c)$ for each letter $c \in \Sigma$ separately, and the sum $\Delta_\rho[\Delta_\rho[T \otimes P]] = \sum_{c \in \Sigma} \Delta_\rho(P_c^R) * \Delta_\rho(T'_c)$. For each remainder $i \bmod \rho$, the values $\Delta_\rho[\Delta_\rho[T \otimes P]](i)$ are stored in a data structure that allows queries for prefix sums (both unweighted and weighted by $\lfloor i/\rho \rfloor$) so that $[T \otimes P](i)$ can be retrieved efficiently using Lemma 16. Every update to P or T incurs updates to $\Delta_\rho(P_c^R)$ or $\Delta_\rho(T'_c)$, in $\mathcal{O}(1)$ places in total (two for each letter involved in the substitution). We buffer the updates to those convolutions of (potentially) sparse functions during subepochs of x updates, and then we recompute values of $\Delta_\rho(P_c^R) * \Delta_\rho(T'_c)$ amortized during the next x updates. We fix a threshold value t (specified later), and iterate through letters $c \in \Sigma$.

- If a letter c had weight at least t or accumulated at least t updates, we recompute the corresponding convolution from scratch, at the cost of $\tilde{\mathcal{O}}(n)$ time per each such *heavy* letter.
- Otherwise, updates are processed one by one, at the cost of $\tilde{\mathcal{O}}(t)$ time per update.

There are $\mathcal{O}(\frac{k+x}{t})$ heavy letters, which is $\tilde{\mathcal{O}}(\frac{k}{t})$ since $x \leq k$. Thus, the total cost $\tilde{\mathcal{O}}(\frac{nk}{t} + xt)$ is minimized when $t = \sqrt{\frac{nk}{x}}$ and gives $\tilde{\mathcal{O}}(\sqrt{nkx})$ time per subepoch, or $\tilde{\mathcal{O}}(\sqrt{\frac{nk}{x}})$ time per update.

To perform queries, we retrieve $[T \otimes P](i + m - 1)$ using Lemma 16 and the data structure maintaining $\Delta_\rho[\Delta_\rho[T \otimes P]]$ to recover the number of matches last time we stored the convolutions. Next, we scan through the list of at most $2x$ updates to potentially update the answer. ◀

To put the trade-off complexity in context, we note that e.g., when $k = m$, it is possible to achieve $U(n, k), Q(n, k) = \tilde{\mathcal{O}}(n^{2/3})$. This improves over $\tilde{\mathcal{O}}(n^{3/4})$ presented in [13].

4 Lower Bounds

In this section, we give conditional lower bounds for the dynamic k -mismatch problem based on the 3SUM conjecture [23]. For the 3SUM problem, we use the following definition.

► **Definition 18 (3SUM Problem).** *Given three sets $A, B, C \subseteq [-N..N]$ of total size $|A| + |B| + |C| = n$, decide whether there exist $a \in A, b \in B, c \in C$ such that $a + b + c = 0$.*

Henceforth, we consider algorithms for the word RAM model with w -bit machine words, where $w = \Omega(\log N)$. In this model, there is a simple $\mathcal{O}(n^2)$ -time solution for the 3SUM problem. This can be improved by log factors [7], with the current record being $\mathcal{O}((n^2/\log^2 n)(\log \log n)^{\mathcal{O}(1)})$ time [8].

► **Conjecture 19 (3SUM Conjecture).** *For every constant $\varepsilon > 0$, there is no Las-Vegas randomized algorithm solving the 3SUM problem in $\mathcal{O}(n^{2-\varepsilon})$ expected time.*

As a first step, we note that the 3SUM problem remains hard even if we allow for polynomial-time preprocessing of A . The following reduction is based on [22, Theorem 13].

► **Lemma 20.** *Suppose that, for some constants $d \geq 2$ and $\varepsilon > 0$, there exists an algorithm that, after preprocessing integers $n, N \in \mathbb{Z}_+$ and a set $A \subseteq [-N..N]$ in $\mathcal{O}(n^d)$ expected time, given sets $B, C \subseteq [-N..N]$ of total size $|A| + |B| + |C| \leq n$, solves the underlying instance of the 3SUM problem in expected $\mathcal{O}(n^{2-\varepsilon})$ time. Then, the 3SUM conjecture fails.*

Proof. We shall demonstrate an algorithm solving the 3SUM problem in $\mathcal{O}(n^{2-\hat{\varepsilon}})$ time, where $\hat{\varepsilon} = \min(\frac{1}{2}, \frac{\varepsilon}{2(d-1)}) > 0$. Let $g = \lceil n^{\frac{d-1.5}{d-1}} \rceil$. We construct a decomposition $A = \bigcup_{i=1}^g A_i$ into disjoint subsets such that $|A_i| \leq \lceil \frac{1}{g}|A| \rceil$ and $\max A_i < \min A_{i'}$ hold for $i, i' \in [1..g]$ with $i < i'$. Similarly, we also decompose $B = \bigcup_{j=1}^g B_j$ and $C = \bigcup_{k=1}^g C_k$.

Next, we construct $T = \{(i, j, k) \in [1..g]^3 : \min A_i + \min B_j + \min C_k \leq 0 \leq \max A_i + \max B_j + \max C_k\}$. Observe that if $a + b + c = 0$ for $(a, b, c) \in A \times B \times C$, then the triple $(i, j, k) \in [1..g]^3$ satisfying $(a, b, c) \in A_i \times B_j \times C_k$ clearly belongs to T . Moreover, T can be constructed in $\mathcal{O}(g^2 \log g + |T|)$ time by performing a binary search over $k \in [1..g]$ for all $(i, j) \in [1..g]^2$. To provide a worst-case bound on this running time, we shall prove that $|T| = \mathcal{O}(g^2)$. For this, let us define the *domination* order \prec on $[1..g]^3$ so that $(i, j, k) \prec (i', j', k')$ if and only if $i < i'$, $j < j'$, and $k < k'$. Observe that T is an \prec -antichain and that $[1..g]^3$ can be covered with $\mathcal{O}(g^2)$ \prec -chains. Hence, $|T| = \mathcal{O}(g^2)$ holds as claimed.

Let $\hat{n} := \left\lceil \frac{1}{g}|A| \right\rceil + \left\lceil \frac{1}{g}|B| \right\rceil + \left\lceil \frac{1}{g}|C| \right\rceil = \mathcal{O}\left(\frac{n}{g}\right)$. We preprocess (\hat{n}, N, A_i) for each $i \in [1..g]$, at the cost of $\mathcal{O}(g\hat{n}^d) = \mathcal{O}\left(\frac{n^d}{g^{d-1}}\right) = \mathcal{O}\left(\frac{n^d}{n^{d-1.5}}\right) = \mathcal{O}(n^{1.5})$ time. Then, for each triple $(i, j, k) \in T$, we solve the underlying instance of the 3SUM problem, at the cost of $\mathcal{O}(g^2 \hat{n}^{2-\varepsilon}) = \mathcal{O}\left(g^2 \frac{n^{2-\varepsilon}}{g^{2-\varepsilon}}\right) = \mathcal{O}(n^{2-\varepsilon} g^\varepsilon) = \mathcal{O}\left(n^{2-\varepsilon+\varepsilon \frac{d-1.5}{d-1}}\right) = \mathcal{O}\left(n^{2-\frac{\varepsilon}{2(d-1)}}\right)$ expected time in total. As noted above, it suffices to return YES if and only if at least one of these calls returns YES. \blacktriangleleft

Our lower bounds also rely on the following variant of the 3SUM problem.

► **Definition 21 (3SUM⁺ Problem).** *Given three sets $A, B, C \subseteq [-N..N]$ of total size $|A| + |B| + |C| = n$, report all $c \in C$ such that $a + b + c = 0$ for some $a \in A$ and $b \in B$.*

The benefit of using 3SUM⁺ is that it remains hard for $N \geq n^{2+\Omega(1)}$ (as shown in [19]); in comparison, regular 3SUM is known to be hard only for $N \geq n^3$. The following proposition generalizes the results of [19] (allowing for preprocessing of A); its proof relies on the techniques of [7].

► **Proposition 22.** *Suppose that, for some constants $d \geq 2$ and $\varepsilon, \delta > 0$, there exists an algorithm that, after $\mathcal{O}(n^d)$ -time preprocessing of integers $n, N \in \mathbb{Z}_+$, with $N \leq n^{2+\delta}$, and a set $A \subseteq [-N..N]$, given sets $B, C \subseteq [-N..N]$ of total size $|B| + |C| \leq n$, solves the underlying 3SUM⁺ instance in expected $\mathcal{O}(n^{2-\varepsilon})$ time. Then, the 3SUM conjecture fails.*

Proof. We shall demonstrate an algorithm violating the 3SUM conjecture via Lemma 20. If the input instance already satisfies $N \leq n^{2+\delta}$, there is nothing to do. Thus, we henceforth assume $N > n^{2+\delta}$. Let $v = \lceil \log 3N \rceil$ and $u = \lfloor \log n^{2+\delta} \rfloor$. In the preprocessing, we draw a uniformly random *odd* integer $\alpha \in [0..2^v)$, which defines a hash function $h : \mathbb{Z} \rightarrow [0..2^u)$ with $h(x) = \lfloor \frac{\alpha x \bmod 2^v}{2^v - u} \rfloor$ for $x \in \mathbb{Z}$. The key property of this function is that $(h(a) + h(b) + h(c) - h(a + b + c)) \bmod 2^u \in \{0, -1, -2\}$ holds for all $a, b, c \in \mathbb{Z}$. At the preprocessing stage, we also preprocess $(n, 2^u, h(A))$ for the hypothetical 3SUM⁺ algorithm (note that $2^u \leq n^{2+\delta}$). Overall, the preprocessing stage costs $\mathcal{O}(n^d)$ time.

In the main phase, we solve the following 3SUM⁺ instances, each of size at most n and over universe $[-2^u..2^u)$, denoting $X + y := \{x + y : x \in X\}$:

- $(h(A), h(B), h(C))$,
- $(h(A), h(B), h(C) - 2^u + 2)$,
- $(h(A), h(B), h(C) - 2^u + 1)$,
- $(h(A), h(B), h(C) - 2^u)$,
- $(h(A), h(B) - 2^u, h(C) - 2^u + 2)$,
- $(h(A), h(B) - 2^u, h(C) - 2^u + 1)$,
- $(h(A), h(B) - 2^u, h(C) - 2^u)$;

this step costs $\mathcal{O}(n^{2-\varepsilon})$ time. Combining the results of these calls, in $\mathcal{O}(n)$ time we derive

$$S := \{c \in C : h(a) + h(b) + h(c) \in \{0, 2^u - 2, 2^u - 1, 2^u, 2 \cdot 2^u - 2, 2 \cdot 2^u - 1, 2 \cdot 2^u\}\}.$$

18:12 The Dynamic k -Mismatch Problem

Finally, for each $c \in S$, we check in $\mathcal{O}(n)$ time whether $a + b + c = 0$ holds for some $a \in A$ and $b \in B$. Upon encountering the first witness $c \in S$, we return YES. If no witness is found, we return NO.

Let us analyze the correctness of this reduction. If we return YES, then clearly $a + b + c = 0$ holds for some $a \in A$, $b \in B$, and $c \in C$. For the converse implication, suppose that $a + b + c = 0$ holds for some $a \in A$, $b \in B$, and $c \in C$. Then, $(h(a) + h(b) + h(c) - h(a + b + c)) \bmod 2^u = (h(a) + h(b) + h(c)) \bmod 2^u \in \{0, -1, -2\}$. Given that $h(a), h(b), h(c) \in [0..2^u)$, this means that $h(a) + h(b) + h(c) \in \{0, 2^u - 2, 2^u - 1, 2^u, 2 \cdot 2^u - 2, 2 \cdot 2^u - 1, 2 \cdot 2^u\}$, i.e., $c \in S$. Consequently, we are guaranteed to return YES while processing $c \in S$ at the latest.

It remains to bound the expected running time. For this, it suffices to prove that there are, in expectation, $\mathcal{O}(n^{1-\delta})$ triples $(a, b, c) \in A \times B \times C$ such that $a + b + c \neq 0$ yet $h(a) + h(b) + h(c) \in \{0, 2^u - 2, 2^u - 1, 2^u, 2 \cdot 2^u - 2, 2 \cdot 2^u - 1, 2 \cdot 2^u\}$ (in particular, this means that, in expectation, S contains at most $\mathcal{O}(n^{1-\delta})$ non-witnesses; verifying all of them costs $\mathcal{O}(n^{2-\delta})$ expected time in total). Specifically, we shall prove that each triple satisfies the aforementioned condition with probability $\mathcal{O}(n^{-2-\delta})$.

Due to the fact that $(h(a) + h(b) + h(c) - h(a + b + c)) \bmod 2^u \in \{0, -1, -2\}$, the bad event holds only if $a + b + c \neq 0$ yet $h(a + b + c) \in \{0, 1, 2, 2^u - 2, 2^u - 1\}$. Let $a + b + c = 2^t \beta$ for an integer $t \in \mathbb{Z}_{\geq 0}$ and odd integer $\beta \in \mathbb{Z}$. Due to $|a + b + c| \leq 3N \leq 2^v$, we must have $t \in [0..v)$.

- If $t > v - u + 1$, then $h(a + b + c)$ is uniformly random odd multiple of 2^{t-v+u} within $[0..2^u)$. Hence, $\Pr[h(a + b + c) \in \{0, 1, 2, 2^u - 2, 2^u - 1\}] = 0$.
- If $t = v - u + 1$, then $h(a + b + c)$ is a uniformly random odd multiple of 2 within $[0..2^u)$. Hence, $\Pr[h(a + b + c) \in \{0, 1, 2, 2^u - 2, 2^u - 1\}] \leq \frac{2}{2^{v-2}} = \frac{8}{2^v}$.
- If $t = v - u$, then $h(a + b + c)$ is a uniformly random odd multiple of 1 within $[0..2^u)$. Hence, $\Pr[h(a + b + c) \in \{0, 1, 2, 2^u - 2, 2^u - 1\}] \leq \frac{2}{2^{v-1}} = \frac{4}{2^v}$.
- If $t < v - u$, then $h(a + b + c)$ is a uniformly random element of $[0..2^u)$. Hence, $\Pr[h(a + b + c) \in \{0, 1, 2, 2^u - 2, 2^u - 1\}] \leq \frac{5}{2^v}$.

Overall, the probability is bounded by $\frac{8}{2^v} = \mathcal{O}(n^{-2-\delta})$. ◀

We are now in a position to give the lower bound for the dynamic k -mismatch problem.

► **Theorem 23.** *Suppose that, for some constants $p > 0$, $\varepsilon > 0$, and $0 < c < \frac{1}{2}$, there exists a dynamic k -mismatch algorithm that solves instances satisfying $k = \lceil m^c \rceil$ using initialization in $\mathcal{O}(n^p)$ expected time, updates in $\mathcal{O}(k^{1-\varepsilon})$ expected time, and queries in $\mathcal{O}(k^{1-\varepsilon})$ expected time. Then, the 3SUM conjecture fails. This statement remains true when updates are allowed in either the pattern or the text (but not both).*

Proof. We shall provide an algorithm contradicting Proposition 22 for $\delta = \frac{1-2c}{2c}$ and $d = \frac{p}{c}$. Suppose that the task is to solve a size- \hat{n} instance of the 3SUM⁺ problem with $A, B, C \subseteq [-N..N)$. We set $m = \lceil \hat{n}^{1/c} \rceil$ (so that $k = \lceil m^c \rceil \geq \hat{n}$), and $n = 2m$, and we initialize a pattern to $P = 0^m$ and a text to $T = 0^n$. Observe that $m \geq \hat{n}^{1/c} \geq N^{\frac{1}{c(2+\delta)}} = N^{\frac{2}{1+2c}}$. If $N^{\frac{2}{1+2c}} < 2N$, then $N = \mathcal{O}(1)$, and we can afford to solve the 3SUM⁺ instance naively. Otherwise, we are guaranteed that $m \geq 2N$, and we proceed as follows:

- we set $P[a + N] := 1$ for each $a \in A$;
- we set $T[2N - b] := 1$ for each $b \in B$.

Finally, for each element $c \in C$, we perform a query at position $c + N$, and report c if and only if $\text{HD}(P, T[c + N..c + N + m]) < \text{HD}(P, 0^m) + \text{HD}(T[c + N..c + N + m], 0^m)$. Due to the fact that $\text{HD}(P, 0^m) + \text{HD}(T[c + N..c + N + m], 0^m) \leq |A| + |B| \leq \hat{n} = k$, this can be decided based on the answer to the query. Equivalently, we report $c \in C$ if and only if

$P[i] = T[c + N + i] = 1$ holds for some $i \in [0..N)$, i.e., $T[a + c + 2N] = 1$ for some $a \in A$, or, equivalently, $-a - c \in B$, i.e., $a + b + c = 0$ for some $b \in B$. This proves the correctness of the algorithm.

As for the running time, note that the preprocessing phase costs $\mathcal{O}(n^p) = \mathcal{O}(m^p) = \mathcal{O}(\hat{n}^{\frac{p}{c}}) = \mathcal{O}(\hat{n}^d)$ expected time. The main phase, on the other hand, involves $\mathcal{O}(\hat{n})$ updates and queries, which cost $\mathcal{O}(\hat{n} \cdot k^{1-\varepsilon}) = \mathcal{O}(\hat{n}^{2-\varepsilon})$ expected time in total. By Proposition 22, this algorithm for 3SUM⁺ would violate the 3SUM conjecture.

If the updates are allowed in the text only, we set up the pattern during the preprocessing phase based on the fact that the target value of P depends on A only. If the updates are allowed in the pattern only, we exchange the roles of A and B and set up the text during the preprocessing phase. ◀

Next, we note that the lower bound can be naturally extended to $c \geq \frac{1}{2}$.

► **Corollary 24.** *Suppose that, for some constants $p > 0$, $\varepsilon > 0$, and $0 < c \leq 1$, there exists a dynamic k -mismatch algorithm that solves instances satisfying $k = \lceil m^c \rceil$ using initialization in $\mathcal{O}(n^p)$ expected time, updates in $\mathcal{O}(\min(\sqrt{m}, k)^{1-\varepsilon})$ expected time, and queries in $\mathcal{O}(\min(\sqrt{m}, k)^{1-\varepsilon})$ expected time. Then, the 3SUM conjecture fails. This statement remains true when updates are allowed in either the pattern or the text (but not both).*

Proof. When $c < \frac{1}{2}$, the result holds directly due to Theorem 23. When $c \geq \frac{1}{2}$, we prove that the 3SUM conjecture would be violated through Theorem 23 with $\hat{c} = \frac{1-\varepsilon}{2-\varepsilon}$ and $\hat{\varepsilon} = \frac{\varepsilon}{2}$. Since the \hat{k} -mismatch problem with $\hat{k} = \lceil m \rceil^{\hat{c}}$ can be simulated using an instance of the k -mismatch problem with $k = \lceil m \rceil^c$, we note that, in the former setting, the queries and updates can be hypothetically implemented in $\mathcal{O}((\sqrt{m})^{1-\varepsilon}) = \mathcal{O}(\hat{k}^{\frac{1-\varepsilon}{2-\varepsilon}}) = \mathcal{O}(\hat{k}^{\frac{2-\varepsilon}{2}}) = \mathcal{O}(\hat{k}^{1-\hat{\varepsilon}})$ expected time, violating the 3SUM conjecture via Theorem 23. ◀

4.1 Lower Bound for $m \ll n$

While most of the work in this paper focuses on the case where the length of the pattern is linear in the length of the text, for completeness, we provide a lower bound that is only of interest when the pattern is considerably shorter. Our lower bound is conditioned on the Online Matrix-Vector Multiplication conjecture [18], which is often used in the context of dynamic algorithms.

In the Online Boolean Matrix-Vector Multiplication (OMv) problem, we are given as input a Boolean matrix $M \in \{0, 1\}^{n \times n}$. Then, a sequence of n vectors $v_1, \dots, v_n \in \{0, 1\}^n$ arrives in an online fashion. For each such vector v_i , we are required to output Mv_i before receiving v_{i+1} .

► **Conjecture 25** (OMv Conjecture [18]). *For any constant $\epsilon > 0$, there is no $\mathcal{O}(n^{3-\epsilon})$ -time algorithm that solves OMv correctly with probability at least $\frac{2}{3}$.*

We use the following simplified version of [18, Theorem 2.2].

► **Theorem 26** ([18]). *Suppose that, for some constants $\gamma, \varepsilon > 0$, there is an algorithm that, given as input a matrix $M \in \{0, 1\}^{p \times q}$, with $q = \lfloor p^\gamma \rfloor$, preprocesses M in time polynomial in $p \cdot q$, and then, presented with a vector $v \in \{0, 1\}^q$, computes Mv in time $\mathcal{O}(p^{1+\gamma-\varepsilon})$ correctly with probability at least $\frac{2}{3}$. Then, the OMv conjecture fails.*

Theorem 26 lets us derive our lower bound for the dynamic k -mismatch problem.

► **Theorem 27.** *Suppose that, for some constants $\gamma, \varepsilon > 0$, there is a dynamic k -mismatch algorithm that solves instances satisfying $k = 2 \lfloor (\frac{n}{m})^\gamma \rfloor$, with preprocessing in $\mathcal{O}(n^{\mathcal{O}(1)})$ time, updates of the pattern in $\mathcal{O}((\frac{n}{m})^{1-\varepsilon})$ time, and queries in $\mathcal{O}(k^{1-\varepsilon})$ time, providing correct answers with high probability. Then, the OMv conjecture fails.*

Proof. Given a matrix $M \in \{0, 1\}^{p \times q}$, we set $m = 3q$, $n = 3pq$, and $k = 2q$ (so that $k = 2q = 2 \lfloor p^\gamma \rfloor = 2 \lfloor (\frac{n}{m})^\gamma \rfloor$ holds). At the preprocessing phase, we initially set $P = 0^m$ and $T = 0^n$. As for the text, for each $i \in [0..p)$ and $j \in [0..q)$, we set $T[3iq + 3j..3iq + 3j + 3)$ to 100 if $M[i, j] = 0$ and to 111 if $M[i, j] = 1$.

When a vector v arrives, the pattern is set as follows: for each $j \in [0..q)$, we set $P[3j..3j + 3)$ to 001 if $v[j] = 0$ and to 111 if $v[j] = 1$. This requires $\mathcal{O}(q)$ update calls to our dynamic data structure. Queries are then made at position im for all $i \in [0..p)$. By definition of the Hamming distance, $\text{HD}(100, 001) = \text{HD}(100, 111) = \text{HD}(111, 001) = 2$, whereas $\text{HD}(111, 111) = 0$; therefore, the only time that the returned Hamming distance will be less than $k = 2q$ is when $M[i, j] = v[j] = 1$ for some $j \in [1..q]$, i.e., $(Mv)[i] = 1$. Therefore, the OMv product can be computed by making $\mathcal{O}(p)$ queries and $\mathcal{O}(q)$ updates to the dynamic k -mismatch data structure as before. The total cost of these operations is $\mathcal{O}(pq^{1-\varepsilon} + qp^{1-\varepsilon}) = \mathcal{O}(p^{1+\gamma(1-\varepsilon)} + p^{1+\gamma-\varepsilon}) = \mathcal{O}(p^{1+\gamma-\min(1,\gamma)\varepsilon})$. By Theorem 26 with $\hat{\varepsilon} = \min(1, \gamma)\varepsilon$, this would violate the OMv conjecture. ◀

References



- 1 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828, 2000. URL: <http://dl.acm.org/citation.cfm?id=338219.338645>.
- 2 Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020*, volume 181 of *LIPICs*, pages 63:1–63:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ISAAC.2020.63.
- 3 Amihood Amir and Itai Boneh. Dynamic suffix array with sub-linear update time and poly-logarithmic lookup time. *CoRR*, 2021. arXiv:2112.12678.
- 4 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Konradovsky. Repetition detection in a dynamic string. In *ESA*, volume 144 of *LIPICs*, pages 5:1–5:18, 2019. doi:10.4230/LIPICs.ESA.2019.5.
- 5 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common substring made fully dynamic. In *ESA*, volume 144 of *LIPICs*, pages 6:1–6:17, 2019. doi:10.4230/LIPICs.ESA.2019.6.
- 6 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.
- 7 Ilya Baran, Erik D. Demaine, and Mihai Patrascu. Subquadratic algorithms for 3sum. *Algorithmica*, 50(4):584–596, 2008. doi:10.1007/s00453-007-9036-3.
- 8 Timothy M. Chan. More logarithmic-factor speedups for 3SUM, (median, +)-convolution, and some geometric 3SUM-hard problems. *ACM Transactions on Algorithms*, 16(1):7:1–7:23, 2020. doi:10.1145/3363541.
- 9 Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern Hamming distances. In *STOC*, pages 643–656. ACM, 2020. doi:10.1145/3357713.3384266.
- 10 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *ICALP*, volume 168 of *LIPICs*, pages 27:1–27:19, 2020. doi:10.4230/LIPICs.ICALP.2020.27.

- 11 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *FOCS*, pages 978–989, 2020. doi:10.1109/FOCS46700.2020.00095.
- 12 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k -mismatch problem revisited. In *SODA*, pages 2039–2052. SIAM, 2016. doi:10.1137/1.9781611974331.ch142.
- 13 Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *STACS*, volume 96 of *LIPIcs*, pages 22:1–22:14, 2018. doi:10.4230/LIPIcs.STACS.2018.22.
- 14 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k -mismatch problem. In *SODA*, pages 1106–1125. SIAM, 2019. doi:10.1137/1.9781611975482.68.
- 15 Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *SODA*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.
- 16 Paweł Gawrychowski and Przemysław Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In *ICALP*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.
- 17 Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. The streaming k -mismatch problem: Tradeoffs between space and total time. In *CPM*, volume 161 of *LIPIcs*, pages 15:1–15:15, 2020. doi:10.4230/LIPIcs.CPM.2020.15.
- 18 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.
- 19 Chloe Ching-Yun Hsu and Chris Umans. On multidimensional and monotone k -sum. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017*, volume 83 of *LIPIcs*, pages 50:1–50:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.50.
- 20 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. *CoRR*, 2022. arXiv:2201.01285.
- 21 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.
- 22 Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic time-space trade-offs for k -sum. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *LIPIcs*, pages 58:1–58:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.58.
- 23 Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *STOC*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.

Indexable Elastic Founder Graphs of Minimum Height

Nicola Rizzo  

Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen  

Department of Computer Science, University of Helsinki, Finland

Abstract

Indexable elastic founder graphs have been recently proposed as a data structure for genomics applications supporting fast pattern matching queries. Consider segmenting a multiple sequence alignment $MSA[1..m, 1..n]$ into b blocks $MSA[1..m, 1..j_1]$, $MSA[1..m, j_1 + 1..j_2]$, \dots , $MSA[1..m, j_{b-1} + 1..n]$. The resulting *elastic founder graph* (EFG) is obtained by merging in each block the strings that are equivalent after the removal of gap symbols, taking the strings as the nodes of the block and the original MSA connections as edges. We call an elastic founder graph *indexable* if a node label occurs as a prefix of only those paths that start from a node of the same block. Equi et al. (ISAAC 2021) showed that such EFGs support fast pattern matching and studied their construction maximizing the number of blocks and minimizing the maximum length of a block, but left open the case of minimizing the maximum number of distinct strings in a block that we call *graph height*. For the simplified gapless setting, we give an $O(mn)$ time algorithm to find a segmentation of an MSA minimizing the height of the resulting indexable founder graph, by combining previous results in segmentation algorithms and founder graphs. For the general setting, the known techniques yield a linear-time parameterized solution on constant alphabet Σ , taking time $O(mn^2 \log|\Sigma|)$ in the worst case, so we study the refined measure of *prefix-aware height*, that omits counting strings that are prefixes of another considered string. The indexable EFG minimizing the maximum prefix-aware height provides a lower bound for the original height: by exploiting suffix trees built from the MSA rows and the data structure answering weighted ancestor queries in constant time of Belazzougui et al. (CPM 2021), we give an $O(mn)$ -time algorithm for the optimal EFG under this alternative height.

2012 ACM Subject Classification Theory of computation \rightarrow Graph algorithms analysis; Theory of computation \rightarrow Pattern matching; Theory of computation \rightarrow Sorting and searching; Theory of computation \rightarrow Dynamic programming; Applied computing \rightarrow Genomics

Keywords and phrases multiple sequence alignment, pattern matching, data structures, segmentation algorithms, dynamic programming, suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.19

Funding This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956229.

1 Introduction

String matching in a text and its variants are classic problems in computer science, with a myriad of applications such as biological sequence analysis. The generalization of the string matching problem concerned with searching strings in a labeled graph has gained more and more importance in computational biology, and for a good reason: the rapidly increasing number of sequenced data makes it possible to capture the variation of many species, populations, and cancer genomes, for example forming the so-called *pangenome* of a species [5]. A central challenge of pangenomics is then to provide the computational tools to swap a single reference genome with a pangenomic representation of hundreds – if not thousands – of genomes in the established analysis tasks [19, 24, 25, 13, 17, 7, 20].



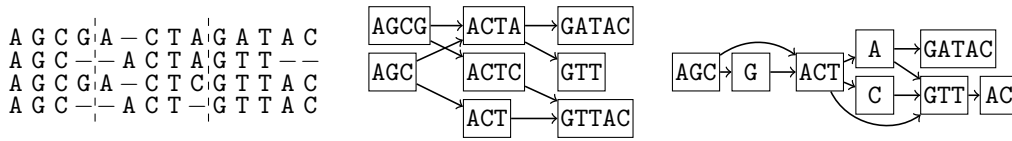
© Nicola Rizzo and Veli Mäkinen;
licensed under Creative Commons License CC-BY 4.0
33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 19; pp. 19:1–19:19

Leibniz International Proceedings in Informatics



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



■ **Figure 1** An elastic founder graph (middle) induced from a MSA segmentation (left). This EFG is *semi-repeat-free*, meaning that each node label appears only as prefix of paths starting from the same block. Thus, we say it is indexable since it supports fast pattern matching. On the right, the modification of the EFG suggested by the prefix-aware height, compacting labels that are prefixes of others in the same block: we reserve its study as future work.

The most popular representation for a pangenome is a graph whose paths spell the input genomes, and the basic primitive required on such pangenome graphs is to be able to search occurrences of query strings (short reads) as subpaths of the graph. On this front, research efforts have been met with theoretical roadblocks: string matching in labeled graphs cannot be solved in sub-quadratic time even for simple graph classes, unless the Orthogonal Vectors Hypothesis (OVH) is false [8]; under OVH, no polynomial-time indexing scheme of a graph can support sub-quadratic time queries [9]; identifying if a graph belongs to the class of Wheeler graphs, that are easy to index, is NP-complete [14]. Therefore, practical tools deploy various heuristics or use other pangenome representations as a basis.

The pangenomic representation at the heart of the *elastic founder graph* (EFG), proposed by Mäkinen et al. [18] and Equi et al. [10], is then to assume as input a *multiple sequence alignment*, a matrix $\text{MSA}[1..m, 1..n]$ composed of m rows that are strings of length n , drawn from an alphabet Σ plus a special “gap” symbol where each column represents an aligned position of the characters of the rows. As seen in Figure 1, the EFG is then created by choosing a segmentation of the MSA, that is, a partition of the columns in consecutive blocks: the strings in each block become the nodes of the block, and the edges are defined by the original row connections. If the node labels do not appear as prefix of any other path than those starting at the same block, then the so-called *semi-repeat-free property* holds and the graph supports an index structure for fast pattern matching [18, 10]. Equi et al. [10] also show that an indexability property like this one is required to have pattern matching in sub-quadratic time, since the OVH-based lower bound holds even when restricted to EFGs induced by MSA segmentations. Mäkinen et al. [18] gave an $O(mn)$ time algorithm constructing an indexable EFG minimizing the maximum block length, given a gapless $\text{MSA}[1..m, 1..n]$. Equi et al. [10] extended the result to general MSAs, obtaining $O(mn \log m)$ time algorithms for the same optimization and for maximizing the number of blocks, and we recently improved these two results to $O(mn)$ time [22]. We refer the reader to the aforementioned papers for the connections of the approach to Elastic Degenerate Strings and Wheeler graphs.

In this paper, we continue the study of indexable EFGs and focus on optimizing the height of the resulting graph. This measure is defined as the maximum number of distinct strings (i.e. nodes) in a block, and in the example of Figure 1 (middle) this measure is equal to 3, as the second and last blocks have 3 nodes. In Section 2, we provide the basic definitions around EFGs. In Section 3, we introduce the algorithms to find an MSA segmentation minimizing the height of the resulting indexable EFG: for the gapless case, we show how the left-to-right segmentation algorithm by Norri et al. exploiting *left extensions* [21] can be combined with the computation of the *minimal left extensions* by Equi et al. [18] to obtain an $O(mn)$ algorithm in the case where $|\Sigma| \in O(m)$; since left extensions cannot be used in the general case, we develop an equivalent left-to-right solution exploiting *meaningful right extensions* that is also correct in the case with gaps. In the general case, the number of these

extensions is $O(mn^2)$ and computing them takes $O(mn\alpha \log|\Sigma|)$ time, where α is the length of the longest run in the MSA where a row spells a prefix of the string spelled by another row and, unfortunately, $\alpha \in \Theta(n)$ in the worst case. Hence, we continue with a different generalization of the gapless height that we call *prefix-aware height*, equal to the maximum number of distinct strings in a block but omitting strings that are prefixes of others in such block. In the example of Figure 1, this measure is equal to 2. The number of meaningful right extensions is $O(mn)$ for this refined height, so in Section 4 we obtain an $O(mn)$ time solution on general MSAs: the segmentation minimizing the maximum prefix-aware height provides a useful lower bound on the optimal segmentation under the original height. The linear time is achieved thanks to the computation of the generalized suffix tree built from the MSA rows, its symmetrical prefix tree counterpart, and the constant-time navigation between the two offered by the suffix tree data structure of Belazzougui et al. answering weighted ancestor queries [2]. We leave it for future work to study whether the modified EFG suggested by our refined height, as seen in Figure 1, supports an adapted version of the index for fast pattern matching queries.

2 Definitions

We follow the notation of Equi et al. [10].

Strings. We denote integer intervals by $[x..y]$. Let $\Sigma = [1..\sigma]$ be an alphabet of size $|\Sigma| = \sigma$. A *string* $T[1..n]$ is a sequence of symbols from Σ , i.e. $T \in \Sigma^n$, where Σ^n denotes the set of strings of length n over Σ . In this paper, we assume that σ is always smaller or equal to the length of the strings we are working with. The *reverse* of T , denoted with T^{-1} , is the string T read from right to left. A *suffix* (*prefix*) of string $T[1..n]$ is $T[x..n]$ ($T[1..y]$) for $1 \leq x \leq n$ ($1 \leq y \leq n$). A *substring* of string $T[1..n]$ is $T[x..y]$ for $1 \leq x \leq y \leq n$. The *length* of a string T is denoted $|T|$ and the *empty string* ε is the string of length 0. In particular, substring $T[x..y]$ where $y < x$ is the empty string. For convenience, we denote with Σ^* and Σ^+ the set of finite strings and finite nonempty strings over Σ , respectively. We say that a substring $T[x..y]$ is *proper* if it is non-empty and different from T . String Q *occurs* in T if $Q = T[x..y]$; we say that x is the starting position (occurrence) of Q in T , and y is the ending position (ending occurrence). The *lexicographic order* of two strings A and B is naturally defined by the order of the alphabet: $A < B$ iff $A[1..y] = B[1..y]$ and $A[y+1] < B[y+1]$ for some $y \geq 0$. If $y+1 > \min(|A|, |B|)$, then the shorter one is regarded as smaller. However, we usually avoid this implicit comparison by adding an *end marker* $\$ \notin \Sigma$ to the strings and we consider $\$$ to be the lexicographically smallest character. The concatenation of strings A and B is denoted $A \cdot B$, or just AB .

Elastic founder graphs. MSAs can be compactly represented by elastic founder graphs, the vertex-labeled graphs that we formalize in this section.

A *multiple sequence alignment MSA* $[1..m, 1..n]$ is a matrix with m strings drawn from $\Sigma \cup \{-\}$, each of length n , as its rows. Here, $- \notin \Sigma$ is the *gap* symbol. For a string $X \in (\Sigma \cup \{-\})^*$, we denote $\text{spell}(X)$ the string resulting from removing the gap symbols from X . If an MSA does not contain gaps then we say it is *gapless*, otherwise we say that it is a *general MSA*. Given $I \subseteq [1..m]$, we denote with $\text{MSA}[I, 1..n]$ the MSA obtained by considering only rows $\text{MSA}[i, 1..n]$ with $i \in I$.

Let \mathcal{P} be a *partitioning* of $[1..n]$, that is, a sequence of subintervals $\mathcal{P} = [x_1..y_1], [x_2..y_2], \dots, [x_b..y_b]$ where $x_1 = 1$, $y_b = n$, and $x_j = y_{j-1} + 1$ for all $j > 2$. A *segmentation* S of $\text{MSA}[1..m, 1..n]$ based on partitioning \mathcal{P} is the sequence of b sets $S^k =$

$\{\text{spell}(\text{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\}$ for $1 \leq k \leq b$; in addition, we require for a (proper) segmentation that $\text{spell}(\text{MSA}[i, x_k..y_k])$ is not an empty string for any i and k . We call set S^k a *block*, while $\text{MSA}[1..m, x_k..y_k]$ or just $[x_k..y_k]$ is called a *segment*. The *length* of block S^k is $L(S^k) = y_k - x_k + 1$ and its *height* is $H(S^k) = |S^k|$. Since each block is derived from a segment $[x..y]$, we denote *segment length* and *height* with $L(\text{MSA}[1..m, x..y])$ and $H(\text{MSA}[1..m, x..y])$ or just $L([x..y])$ and $H([x..y])$, respectively.

Segmentation naturally leads to the definition of a founder graph through the block graph concept.

► **Definition 1** (Block Graph). *A block graph is a graph $G = (V, E, \ell)$ where $\ell : V \rightarrow \Sigma^+$ is a function that assigns a string label to every node and for which the following properties hold:*

1. *set V can be partitioned into a sequence of b blocks V^1, V^2, \dots, V^b , that is, $V = V^1 \cup V^2 \cup \dots \cup V^b$ and $V^i \cap V^j = \emptyset$ for all $i \neq j$;*
2. *if $(v, w) \in E$ then $v \in V^i$ and $w \in V^{i+1}$ for some $1 \leq i \leq b - 1$; and*
3. *if $v, w \in V^i$ then $|\ell(v)| = |\ell(w)|$, and if $v, w \in V^i$ and $v \neq w$ then $\ell(v) \neq \ell(w)$.*

For *gapless* MSAs, block S^k equals segment $\text{MSA}[1..m, x_k..y_k]$, and in that case the *founder graph* is a block graph induced by segmentation S [18]. The idea is to have a graph in which the nodes represent the strings in S while the edges retain the information of how such strings can be recombined to spell any sequence in the original MSA.

For general MSAs with gaps, we consider the following extension.

► **Definition 2** (Elastic block and founder graphs). *We call a block graph elastic if its third condition is relaxed in the sense that each V^i can contain non-empty variable-length strings. An elastic founder graph (EFG) is an elastic block graph $G(S) = (V, E, \ell)$ induced by a segmentation S of $\text{MSA}[1..m, 1..n]$ as follows: for each $1 \leq k \leq b$ we have $S^k = \{\text{spell}(\text{MSA}[i, x_k..y_k]) \mid 1 \leq i \leq m\} = \{\ell(v) : v \in V^k\}$. It holds that $(v, w) \in E$ if and only if there exist $k \in [1..b-1]$, $i \in [1..m]$ such that $v \in V^k$, $w \in V^{k+1}$, and $\text{spell}(\text{MSA}[i, x_k..y_{k+1}]) = \ell(v)\ell(w)$.*

For example, in the general $\text{MSA}[1..4, 1..13]$ of Figure 1, the segmentation based on partitioning $[1..4]$, $[5..9]$, $[10..14]$ induces an EFG $G(S) = (V^1 \cup V^2 \cup V^3, E, \ell)$ where the nodes in V^1 , V^2 , and V^3 have labels of variable length. As noted by Equi et al. [10], block graphs are connected to Generalized Degenerate Strings [1] and elastic founder graphs are connected to Elastic Degenerate Strings [3].

By definition, (elastic) founder and block graphs are acyclic. For convention, we interpret the direction of the edges as going from left to right. Consider a path P in $G(S)$ between any two nodes. The label $\ell(P)$ of P is the concatenation of the labels of the nodes in the path. Let Q be a query string. We say that Q *occurs* in $G(S)$ if Q is a substring of $\ell(P)$ for any path P of $G(S)$.

► **Definition 3** ([18]). *EFG $G(S)$ is repeat-free if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as a prefix of paths starting with v .*

► **Definition 4** ([18]). *EFG $G(S)$ is semi-repeat-free if each $\ell(v)$ for $v \in V$ occurs in $G(S)$ only as a prefix of paths starting with $w \in V$, where w is from the same block as v .*

For example, the EFG of Figure 1 is not repeat-free, since **AGC** occurs as a prefix of two distinct labels of nodes in the same block, but it is semi-repeat-free since all node labels $\ell(v)$ with $v \in V^k$ occur in $G(S)$ only starting from block V^k , or they do not occur at all elsewhere in the graph. These definitions also apply to general elastic block graphs and to elastic degenerate strings as their special case.

Basic tools. A *trie* or *keyword tree* [6] of a set of strings is a rooted directed tree with outgoing edges of each node labeled by distinct symbols such that there is a root-to-leaf path spelling each string in the set; the shared part of the root-to-leaf paths of two different leaves spell the common prefix of the corresponding strings. In a *compact trie*, the maximal non-branching paths of a trie become edges labeled with the concatenation of labels on the path. The *suffix tree* of $T \in \Sigma^*$ is the compact trie of all suffixes of string $T\$$. Such tree takes linear space and can be constructed in linear time so that when reading the leaves from left to right, the suffixes are listed in their lexicographic order [12]. A *generalized suffix tree* is one built on a set of m strings [15]. In this case, string T above is the concatenation of the strings after appending a unique end marker $\$i$ to each string¹, with $1 \leq i \leq m$.

Let $Q[1..m]$ be a query string. If Q occurs in T , then the *locus* or *implicit node* of Q in the suffix tree of T is (v, k) such that $Q = XY$, where X is the string spelled from the root to the parent of v and Y is the prefix of length k of the edge from the parent of v to v . The leaves of the subtree rooted at v , or *the leaves covered by v* , are then all the suffixes sharing the common prefix Q . Let aX and X be paths spelled from the root of a suffix tree to nodes v and w , respectively; then, one can store a *suffix link* from v to w . For suffix trees, a *weighted ancestor query* asks for the computation of the implicit or explicit node corresponding to substring $T[x..y]$ of the text, given x and y .

String $B[1..n]$ from a binary alphabet is called a *bitvector*. Operation $\text{rank}(B, i)$ returns the number of 1s in $B[1..i]$. Operation $\text{select}(B, j)$ returns the index i containing the j -th 1 in B . Both queries can be answered in constant time using an index constructible in linear time and requiring $o(n)$ bits of space in addition to the bitvector itself [16].

3 Construction of EFGs of minimum height

Recall that in the absence of gaps the semi-repeat-free and repeat-free notions (Definitions 3 and 4) are equivalent, and the strings in a block induced by a segment cannot have variable length. In this section, we study the construction of EFGs under the goal of minimizing the maximum block height. Indeed, after showing that semi-repeat-free EFGs are easy to index for fast pattern matching, Equi et al. [10] extended the previous results for the gapless setting showing that semi-repeat-free EFGs are equivalent to specific segmentations of the MSA: the semi-repeat-free property has to be checked only against the MSA, and not the final EFG. We recall these arguments in Section 3.1, along with the resulting recurrence to compute an optimal segmentation under three score functions: *i.* maximizing the number of blocks; *ii.* minimizing the maximum length of a block; and *iii.* minimizing the maximum height of a block.

In the gapless repeat-free setting, scores *i.* and *ii.* admit the construction of indexable founder graphs in $O(mn)$ time, thanks to previous research on founder graphs and MSA segmentations [18, 21, 4]. In Section 3.2 we combine these works to obtain an $O(mn)$ time solution for score *iii.* as well: the optimal segmentation is found mainly by computing the *meaningful left extensions*, that is, the positions $x_1 > \dots > x_k$ where the height of repeat-free segment $[x_i..y]$ increases, with $y \in [1..n]$. In the general and semi-repeat-free setting, extending a segment to the left can violate the semi-repeat-free property and the height can decrease. Thus, Equi et al. in [10] gave $O(n)$ - and $O(n \log \log n)$ -time algorithms for scores *i.* and *ii.*, respectively, exploiting the semi-repeat-free *right extensions* after a

¹ For our purposes, the suffix tree of the concatenated strings is functionally equivalent to the “trimmed” generalized suffix tree seen in Figure 3.

common $O(mn \log m)$ -time preprocessing of the MSA. We recently improved the second algorithm to $O(n)$ time and the preprocessing to $O(mn)$, reaching global linear time [22]. In Section 3.3, we develop a similar algorithm for the construction of a semi-repeat-free segmentation processing the *meaningful right extensions*. Although the number of these extensions is $O(n^2)$ in total, we manage to provide a parameterized linear-time solution providing an upper bound based on the length of the longest run where any two rows spell strings that are one prefix of the other. Instead, an alternative notion of height, the *prefix-aware height*, generates $O(mn)$ *meaningful prefix-aware right extensions*: they can be processed in the same fashion as the original height to obtain an optimal segmentation, and we will show how to compute them efficiently in Section 4.

3.1 Optimal EFGs correspond to optimal segmentations

Consider a segmentation $S = S^1, S^2, \dots, S^b$ inducing a semi-repeat-free EFG $G(S) = (V, E, \ell)$, as per Definition 2. It is easy to see that the strings occurring in $G(S)$ are a superset of the substrings of the MSA rows: for example, string CACTAGA occurs in the EFG of Figure 1 but it does not occur in any row of the original MSA. These new strings, as it was proven by Mäkinen et al. [18] and Equi et al. [10], do not affect the semi-repeat-free property. Intuitively, this is because they involve three or more vertices of $G(S)$.

► **Lemma 5** (Characterization, gapless setting [18]). *We say that a segment $[x..y]$ of a gapless MSA $[1..m, 1..n]$ is repeat-free if string $\text{MSA}[i, x..y]$ occurs in the MSA only at position x of any row. Then $G(S)$ is repeat-free if and only if all segments of S are repeat-free.*

► **Lemma 6** (Characterization [10]). *We say that segment $[x..y]$ of a general MSA $[1..m, 1..n]$ is semi-repeat-free if for any $i, i' \in [1..m]$ string $\text{spell}(\text{MSA}[i, x..y])$ occurs in gaps-removed row $\text{spell}(\text{MSA}[i', 1..n])$ only at position $g(i', x)$, where $g(i', x)$ is equal to x minus the number of gaps in $\text{MSA}[i', 1..x - 1]$. Similarly, $[x..y]$ is repeat-free if the possible occurrence of $\text{spell}(\text{MSA}[i, 1..n])$ in row i' at position $g(i', x)$ also ends at position $g(i', y)$. Then $G(S)$ is semi-repeat-free if and only if all segments of S are semi-repeat-free.*

Thanks to Lemmas 5 and 6, we can compute recursively the score $s(j)$ of an optimal segmentation of prefix $\text{MSA}[1..m, 1..j]$ under our three scoring schemes, using semi-repeat-free segments, that is, respecting the global semi-repeat-free property on the whole MSA:

$$s(j) = \bigoplus_{\substack{j' : 0 \leq j' < j \text{ s.t.} \\ \text{MSA}[1..m, j'+1..j] \text{ is} \\ \text{semi-repeat-free}}} E(s(j'), j', j) \quad (1)$$

where operator \bigoplus and function E extend the optimal partial solutions, and they depend on the desired scoring scheme. Indeed, for $s(j)$ to be equal to the optimal score of a segmentation: *i.* maximizing the number of blocks, set $\bigoplus = \max$ and $g(s(j'), j', j) = s(j') + 1$; for a correct initialization set $s(0) = 0$ and where there is no semi-repeat-free segmentation set $s(j) = -\infty$; *ii.* minimizing the maximum block length, set $\bigoplus = \min$ and $g(s(j'), j', j) = \max(s(j'), L([j' + 1..j])) = \max(s(j'), j - j')$; set $s(0) = 0$ and if there is no semi-repeat-free segmentation set $s(j) = +\infty$. *iii.* minimizing the maximum block height, set $\bigoplus = \min$ and $g(s(j'), j', j) = \max(s(j'), H([j' + 1..j]))$; set $s(0) = 0$ and if there is no semi-repeat-free segmentation set $s(j) = +\infty$.

3.2 The linear time solution for the gapless setting

For gapless MSAs, an $O(mn)$ solution for the construction of segmentations minimizing the maximum block height has been found by Norri et al. [21] for the case where the length of a block is limited by a given lower bound L , rather than with the repeat-free property. This result holds under the assumption that Σ is an integer alphabet of size $O(m)$. In this section, we combine the algorithm by Norri et al. with the computation of values $v(j)$ – that we call the *minimal left extensions* – by Mäkinen et al. [18], obtaining a linear-time solution to the construction of repeat-free founder graphs minimizing the maximum block height.

► **Observation 7** (Monotonicity of left extensions [21, 8]). *Given a gapless MSA[1..m, 1..n], for any $1 \leq x \leq y \leq n$ we say that $[x..y]$ is a left extension of suffix MSA[1..m, y + 1..n]. Then:*

- *if $[x..y]$ is repeat-free then $[x'..y]$ is repeat-free for all $x' < x$;*
- *$m \geq H([x'..y]) \geq H([x..y])$ for all $x' < x$.*

Thus, for each $j \in [1..n]$ we define value $v(j)$ as the greatest column index smaller or equal to j such that $[v(j)..j]$ is repeat-free, and we say that $v(j)$ or $[v(j)..j]$ is the minimal left extension of MSA[1..m, j + 1..n]. If there is no valid left extensions then $v(j) = -\infty$.

► **Definition 8** (Meaningful left extensions [21, 8]). *Given a gapless MSA[1..m, 1..n], for any $j \in [1..n]$ we denote with $L_j = \ell_{j,1}, \dots, \ell_{j,c_j}$ the meaningful (repeat-free) left extensions of MSA[1..m, j + 1..n], meaning the strictly decreasing sequence of all positions smaller than or equal to j such that:*

- *$\ell_{j,c_j} < \dots < \ell_{j,2} < \ell_{j,1} = v(j)$, so that L_j captures all repeat-free left extensions of MSA[1..m, j + 1..n];*
- *$H([\ell_{j,k}..j]) > H([\ell_{j,k} + 1..j])$ for $2 \leq k \leq c_j$, so that each $\ell_{j,k}$ marks a column where the height of the left extension increases; it follows from Observation 7 that $|L_j| = c_j \leq m$.*

If MSA[1..m, j + 1..n] has no repeat-free left extension, we define $L_j = ()$ and $c_j = 0$. Otherwise, for completeness we define $\ell_{j,c_j+1} = -1$.

Under score *iii*. Equation (1) can be rewritten using $L_j = \ell_{j,1}, \dots, \ell_{j,c_j}$ as follows:

$$s(j) = \min_{k \in [1..c_j]} \max \left(\min_{j' \in [\ell_{j,k+1} + 1.. \ell_{j,k}]} s(j'), H([\ell_{j,k}..j]) \right) \quad (2)$$

and $s(j) = +\infty$ if $c_j = 0$, so knowing values L_j , $H([\ell_{j,k}..j])$, and $\min_{j' \in [\ell_{j,k+1} + 1.. \ell_{j,k}]} s(j')$ for $k \in [1..c_j]$ makes it possible to compute $s(j)$ in $O(m)$ time. On one hand, given a fixed length L , Norri et al. [21] developed an algorithm to compute these values under the variant of Definition 8 considering segments of length at least L – instead of repeat-free segments – in $O(mn)$ total time. On the other hand, Mäkinen et al. [18] developed a linear-time algorithm to compute values $v(j)$ of a gapless MSA. The two solutions can be combined by finding values $v(j)$ with the latter, and by using the values as a dynamic lower bound on the minimum accepted segment length. Since the algorithm we develop in Section 3.3 for the general setting also solves this problem, using the symmetrically defined right extensions, we will not describe such modification in this paper.

► **Theorem 9.** *Given a gapless MSA[1..m, 1..n] from an integer alphabet Σ of size $O(m)$, an optimal repeat-free segmentation of MSA[1..m, 1..n] minimizing the maximum block height can be computed in time $O(mn)$.*

3.3 Revisiting the linear time solution for right extensions

For MSAs with gaps and under the semi-repeat-free notion, the monotonicity of left extensions (Observation 7) fails [11, Table 1]: fixing $j \in [1..n]$, left-extensions $\text{MSA}[1..m, x..y]$ are not always semi-repeat-free, or *valid*, from $x = v(j)$ backwards, and their height could decrease when extending a valid segment. For example, in the MSA of Figure 1, segment $[5..9]$ is semi-repeat-free but segment $[4..9]$ is not, and $H([5..9]) < H([6..9])$. In this section, we resolve the former of the two issues, developing an algorithm exploiting right extensions and computing the optimal MSA segmentation from left to right, in the same fashion as [10, Algorithms 1 and 2] and [23, Algorithm 2]. We will discuss the complexity of computing these right extensions in Section 3.4.

► **Observation 10** (Semi-repeat-free right extensions [10]). *Given general MSA[1..m, 1..n], for any $0 \leq x < y \leq n$ we say that $[x + 1..y]$ is an extension of prefix MSA[1..m, 1..x]. If segment $[x + 1..y]$ is semi-repeat-free, then segment $[x + 1..y']$ is semi-repeat-free for all $y' > y$. Thus, for each $x \in [0..n - 1]$ we define value $f(x)$ as the smallest column index greater than x such that $[x + 1..f(x)]$ is semi-repeat-free, and we say that $f(x)$ or $[x + 1..f(x)]$ is the minimal right extension of MSA[1..m, 1..x]. If there is no valid right extension, then $f(x) = \infty$.*

► **Definition 11** (Meaningful right extensions). *Given general MSA[1..m, 1..n], for any $x \in [0..n - 1]$ we denote with $R_x = r_{x,1}, \dots, r_{x,d_x}$ the meaningful (semi-repeat-free) right extensions of MSA[1..m, 1..x], meaning the strictly increasing sequence of all positions greater than x such that:*

- $f(x) = r_{x,1} < r_{x,2} < \dots < r_{x,d_x}$, so that R_x captures all semi-repeat-free right extensions of MSA[1..m, 1..x];
- $H([x + 1..r_{x,k}]) \neq H([x + 1..r_{x,k} - 1])$ for $2 \leq k \leq d_x$, so that each $r_{x,k}$ marks a column where the height of the right extensions changes.

If MSA[1..m, 1..x] has no semi-repeat-free right extension, then $R_x = ()$ and $d_x = 0$. Otherwise, for completeness we define value $r_{x,d_x+1} = n + 1$.

Since we will treat all R_0, \dots, R_{n-1} together, we complement each value $r_{x,k}$ with column x and the height of the corresponding MSA segment, obtaining triple $(x, r_{x,k}, H([x + 1..r_{x,k}]))$.

Thus, under score *iii*. Equation (1) can be rewritten as follows:

$$s(j) = \min_{\substack{x \in [0..j-1], k \in [1..d_x]: \\ r_{x,k} \leq j < r_{x,k+1}}} \max \left(s(x), H([x + 1..r_{x,k}]) \right). \quad (3)$$

Since each R_x defines non-overlapping ranges $[r_{x,k}, r_{x,k+1} - 1]$ over $[1, n]$, at most one range $[r_{x,k}..r_{x,k+1} - 1]$ per R_x with $x < j$ is involved in the computation of $s(j)$, and the corresponding score depends on which range contains j . Also, note that Equation (3) is simpler than Equation (2). Finally, the algorithm computing the score of an optimal semi-repeat-free segmentation minimizing the maximum block height is described in Algorithm 1, and it works by processing all meaningful right extensions in R_0, \dots, R_{n-1} expressed as triples (x, r, h) and sorted from smallest to largest order by second component. The main strategy is to keep at each iteration j the best scores of the semi-repeat-free segmentations of MSA[1..m, 1..j] ending with a right extension $[1, j]$, $[2, j]$, \dots , or $[j, j]$ described by ranges in R_0, R_1, \dots , or R_{j-1} . Checking each currently valid range individually would result in a quadratic-time solution, so we need to represent these ranges in some other form. Indeed, by counting these scores with an array $\mathbf{C}[1..m]$ such that $\mathbf{C}[i]$ is equal to the number of available solutions having score i , score $s(j)$ can be computed by finding the smallest i such that $\mathbf{C}[i]$ is greater than zero. Array \mathbf{C} needs to be updated only when j reaches some $r_{x,k}$; in other words,

■ **Algorithm 1** Main algorithm to find the optimal score of a semi-repeat-free segmentation minimizing the maximum block height.

Input: Meaningful right extensions $(x_1, r_1, h_1), \dots, (x_k, r_k, h_k)$ sorted from smallest to largest order by second component.

Output: Score of an optimal semi-repeat-free segmentation minimizing the maximum block height.

- 1 Initialize array $R[0..n-1]$ with values in $[0..m] \cup \{\perp\}$ and set all values to \perp ;
- 2 Initialize array $C[1..m]$ with values in $[0..m]$ and set all values to 0;
- 3 $y \leftarrow 1$;
- 4 $\text{minmaxheight}[0] \leftarrow 0$;
- 5 **for** $j \leftarrow 1$ **to** n **do**
- 6 **while** $j = r_y$ **do**
- 7 **if** $R[x_y] \neq \perp$ **then**
- 8 $C[R[x_y]] \leftarrow C[R[x_y]] - 1$; ▷ Remove last solution of R_{x_y}
- 9 $s \leftarrow \max(\text{minmaxheight}[x_y], h_y)$;
- 10 $R[x_y] \leftarrow s$; ▷ Save score corresponding to (x_y, r_y, h_y)
- 11 $C[s] \leftarrow C[s] + 1$; ▷ Add solution corresponding to (x_y, r_y, h_y)
- 12 $y \leftarrow y + 1$;
- 13 $\text{minmaxheight}[j] \leftarrow \min_{i=1}^m \{i : C[i] > 0\}$;
- 14 **return** $\text{minmaxheight}[n]$;

when $j = r$ for some (x, r, h) , the score $\max(s(x), h)$ of an optimal segmentation ending with $[x+1..j]$ must be added to C , and the old score relative to the previous range of R_x must be removed. We can keep track of the scores in an array $R[0..n-1]$ such that $R[x]$ is equal to the score associated with the currently valid extension of R_x . A possible implementation of the solution is described in Algorithm 1. To compute the actual segmentation, instead of just its score, we can use two backtracking arrays B and C_{bt} : $C_{\text{bt}}[i] = x$ where $[r_{x,k}..r_{x,k+1} - 1]$ is a currently valid range of minimum score finishing last, that is, with maximum value of $r_{x,k+1}$; values in C_{bt} can be used to compute $B[j]$, equal to x where $[x+1..j]$ is the last segment of an optimal solution for $\text{MSA}[1..m, 1..j]$. Then, B reconstructs an optimal segmentation. Algorithm 1 can be easily modified to update these arrays, if each meaningful right extension $(x, r_{x,k}, H([x+1..r_{x,k}]))$ is augmented with value $r_{x,k+1} - 1$.

► **Lemma 12.** *Given the meaningful right extensions R_0, R_1, \dots, R_{n-1} of $\text{MSA}[1..m, 1..n]$, we can compute the optimal semi-repeat-free segmentation minimizing the maximum block height in time $O(mn + R)$, with $R := \sum_{x=0}^{n-1} |R_x|$.*

Proof. The correctness follows from Equation (3) and from the arguments above. Sorting the meaningful right extensions (x, r, h) by their second component can be done in time $O(n + R)$, as the meaningful right extensions take value in $[1..n]$. Moreover, the management of arrays R and C takes constant time per meaningful right extensions, and the computation of each $s(j)$ takes $O(m)$ time, reaching the time complexity of $O(mn + R)$. ◀

For the gapless case, this is an alternative solution to that of Section 3.2, since $R \in O(mn)$ and the algorithms by Norri et al. and Equi et al. can be used to compute the meaningful right extensions.

3.4 The complexity of minimizing the maximum block height

As Lemma 12 states, we can process the meaningful right extensions of Definition 11 to compute the score of an optimal segmentation minimizing the maximum block height. Unfortunately, in the general setting with gaps, the total number of meaningful right extensions is $O(n^2)$: as it can be seen in Figure 2, if any two row suffixes starting from the same column x spell the same string but the spelling is interleaved by gaps, then the height of segment $[x..y]$ can change at any column y ; this pattern could involve any two rows in any segment of a general $\text{MSA}[1..m, 1..n]$.

		1	2	3	4	5	6	7	8		$n-2$	n	
	1	T	-	A	-	A	-	A	-	...	A	-	C
	2	T	-	-	A	-	A	-	A		-	A	C
$H([1..y])$	1	1	2	1	2	1	2	1	1	...	2	1	1

■ **Figure 2** Example of $\text{MSA}[1..2, 1..n]$ such that $|R_0| \in O(n)$.

► **Observation 13.** *Given $\text{MSA}[1..m, 1..n]$ over alphabet $\Sigma \cup \{-\}$, we have that $H([x..y]) > H([x..y+1])$ only if there exist rows $i, i' \in [1..m]$ such that $\text{MSA}[i, y+1] = -$, $\text{MSA}[i', y+1] = c$, and $\text{spell}(\text{MSA}[i, x..y]) = \text{spell}(\text{MSA}[i', x..y+1]) = S \cdot c$, with $c \in \Sigma$ and $S \in \Sigma^+$.*

The example of Figure 2 and the context described by Observation 13 seem intuitively artificial, as a high-scoring MSA would try to align the rows to avoid such a situation. Nonetheless, without further assumptions about gaps in the MSA portions reading the same strings, we are left to compute all meaningful right extensions. Indeed, let $K_{x..y}$ be the keyword tree of the set of strings $S_{x..y} := \{\text{spell}(\text{MSA}[i, x..y]) : 1 \leq i \leq m\}$; since $|S_{x..y}| = H([x..y])$, the height of $[x..y]$ is equal to the number of distinct nodes of $K_{x..y}$ corresponding to the strings in $S_{x..y}$. We can obtain a parameterized solution by noting that if $K_{x..y}$ has m leaves then no two strings in $S_{x..y}$ are one prefix of the other and $H([x..y']) = m$ for all $y' > y$.

► **Lemma 14.** *Given general $\text{MSA}[1..m, 1..n]$ over integer alphabet $\Sigma \cup \{-\}$ of size $\sigma \in O(mn)$, we denote with α the maximum length $y - x + 1$ of any segment $[x..y]$ such that $\text{spell}(\text{MSA}[i, x..y])$ is a prefix of $\text{spell}(\text{MSA}[i', x..y])$ for some $i, i' \in [1..m]$. Then, we can compute all meaningful right extensions in time $O(mn\alpha \log \sigma)$.*

Proof. For each $x \in [0..n-1]$, we can find R_x by incrementally computing trees $K_{x+1..x+1}$, $K_{x+1..x+2}$, \dots , $K_{x+1..n}$ using a dynamic keyword tree \mathcal{T} supporting the traversal from the root to the leaves and the insertion of a c -child to an arbitrary node v , with $c \in \Sigma$. A possible implementation of the procedure is described by Algorithm 3 in Appendix A. During the computation, array $V[1..m]$ keeps track of the nodes corresponding to strings $\text{spell}(\text{MSA}[i, x+1..r])$, each variable $v.\text{count}$ counts the number of rows reading the corresponding string, and a variable h counts the number of distinct nodes v such that $v.\text{count}$ is greater than zero; if h changes then the corresponding meaningful right extension of $[x+1..r]$ is (x, r, h) . For each R_x , the algorithm stops if the number of leaves of \mathcal{T} is m , that is it computes at most α keyword trees; no meaningful right extension is missed, thanks to Observation 13 and the above arguments. We can compute R_0, \dots, R_{n-1} in time $O(mn\alpha \log \sigma)$, since the traversal and insertion operations of \mathcal{T} can be implemented in time $O(\log \sigma)$,² the other operations can be supported in constant time. ◀

² Since only insertion is needed and alphabet Σ is fixed, the addition of a children to each node v can be implemented with a dynamic binary tree with height at most $\lceil \log_2 \sigma \rceil$ leaves, growing downwards as the number of children grows.

Since the total number of meaningful right extensions is $O(mn\alpha)$, Lemma 14 and Lemma 12 give the following solution to our segmentation problem.

► **Theorem 15.** *Given general MSA[1..m, 1..n] over an integer alphabet Σ of size $O(mn)$, we can compute the score of an optimal segmentation minimizing the maximum block height in time $O(mn\alpha \log \sigma)$, where α is the length of the longest MSA segment where any two rows spell strings S, S' such that S is a prefix of S' .*

In the worst case, the number of meaningful right extension is $O(mn^2)$, $\alpha \in \Theta(n)$, and the time complexity of Lemma 14 is $\Theta(mn^2 \log \sigma)$: thus, we introduce a different generalization of block height from the gapless setting to the general one.

► **Definition 16 (Prefix-aware height).** *Given MSA[1..m, 1..n], we define the prefix-aware height of a segment $[x..y]$, denoted as $\overline{H}(\text{MSA}[1..m, x..y])$ or just $\overline{H}([x..y])$, as the number of distinct strings S in $\{\text{spell}(\text{MSA}[i, x..y]) : 1 \leq i \leq m\}$ such that S is not a prefix of some other string of the set.*

Since $\overline{H}([x..y])$ is equal to $H([x..y])$ minus the number of strings spelled in $[x..y]$ that are proper prefixes of other strings of the segment, this refined height is always smaller or equal to the original height: the relative optimal segmentation provides a lower bound for the maximum height in the original setting. Moreover, the necessary condition for the decrease in height stated in Observation 13 is no longer valid, and it is easy to see that the monotonicity of prefix-aware right extensions holds (see Observation 7). Indeed, if we define the *meaningful prefix-aware right extensions* $\overline{R}_0, \dots, \overline{R}_{n-1}$ as in Definition 11, it is easy to see that $|\overline{R}_x| \leq m + 1$ for all $x \in [0..n - 1]$, so the number of these extensions is $O(mn)$ in total. Finally, given $\overline{R}_0, \dots, \overline{R}_{n-1}$ as input, Algorithm 1 correctly computes the score of an optimal segmentation under our refined height, since Equation (3) still holds. In Section 4, we will provide an algorithm based on the generalized suffix tree of the gaps-removed MSA rows computing the prefix-aware extensions in time linear in the MSA size, obtaining the following result.

► **Theorem 17.** *Given MSA[1..m, 1..n] over integer alphabet $\Sigma \cup \{-\}$ of size $\sigma \leq mn$, computing a semi-repeat-free segmentation minimizing the maximum prefix-aware block height takes $O(mn)$ time.*

4 Preprocessing the MSA for the prefix-aware height

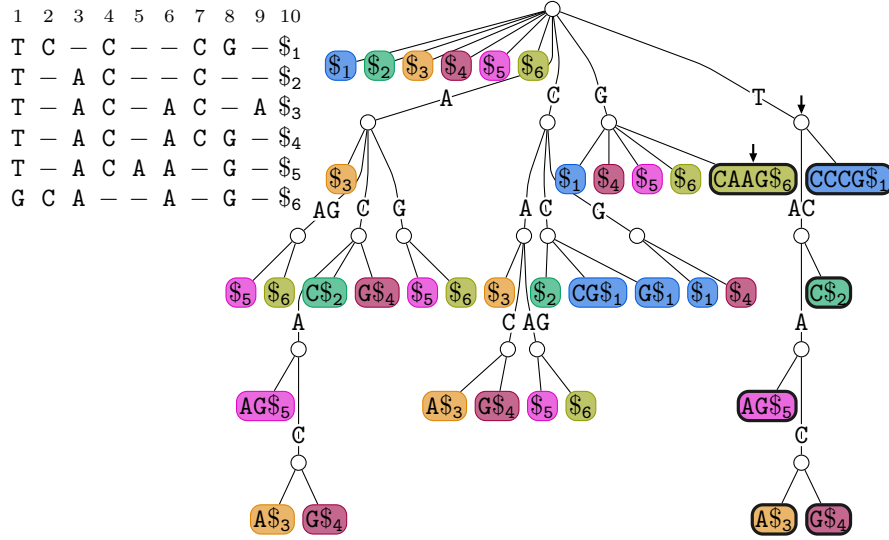
As stated in Section 3.4, the meaningful prefix-aware right extensions are $O(mn)$ in total, so the goal of this section is to compute them in time linear in the MSA size. First, in Section 4.1 we provide an overview of the $O(mn)$ time computation of the minimal right extensions $f(x)$ (Observation 10), that we recently obtained in [22]. The solution consists in solving multiple instances of the *exclusive ancestor problem* – a novel ancestor problem on trees asking for the shallowest nodes covering all and only the given set of leaves – on the following structure, built from the gaps-removed rows of the MSA.

► **Definition 18** ([10, 22]). *Given a general MSA[1..m, 1..n] from alphabet Σ , we define GST_{MSA} as the generalized suffix tree of the set of strings $\{\text{spell}(\text{MSA}[i, 1..n]) \cdot \$i : 1 \leq i \leq m\}$, with $\$, \dots, \m m new distinct terminator symbols not in Σ .*

An example of GST_{MSA} is given in Figure 3. Then, in Section 4.2, we extend these techniques to show that the forests inside GST_{MSA} identified by the exclusive ancestors can describe the meaningful prefix-aware right extensions: by computing for each node of these forests the

19:12 Elastic Founder Graphs of Minimum Height

position indicating where the first occurrence of the related string ends, and by sorting these positions, we can compute the meaningful right extension in $O(m^2n)$ global time. Finally, in Section 4.3 we describe how these positions can be computed efficiently, thanks to the *generalized prefix tree* of the gap-removed rows and the data structure for weighted ancestor queries of Belazzougui et al. [2]. This structure, after its linear-time construction, makes it possible to navigate from the suffix tree to the prefix tree in constant time, reaching global $O(mn)$ time.



■ **Figure 3** Example of an $\text{MSA}[1..6, 1..9]$ and its GST_{MSA} , where the label to each leaf has been moved inside the leaf itself. Leaves are colored according to the corresponding row. We have also highlighted, with a black outline, the leaves \mathcal{L}_0 corresponding to suffixes $\text{spell}(\text{MSA}[i, 1..n])$; their exclusive ancestors W_0 , the nodes corresponding to $G \cdot \text{CAAG}\$6$ and T , are marked with arrows.

4.1 Computing the minimal right extensions

Consider the generalized suffix tree GST_{MSA} (Definition 18) built from the gaps-removed rows $S_i := \text{spell}(\text{MSA}[i, 1..n])\i , for $i \in [1..m]$. Each suffix $S_i[x..|S_i|]$ corresponds to a unique leaf $\ell_{i,x}$ of GST_{MSA} and vice versa, for $1 \leq x \leq |S_i|$. Moreover, each substring $S_i[x..y]$ corresponds to an explicit or implicit node of GST_{MSA} in the root-to- $\ell_{i,x}$ path, and each explicit or implicit node v of GST_{MSA} corresponds to one or more such substrings, described by the leaves of $\text{GST}_{\text{MSA}}(v)$, the subtree rooted at v . Note that in GST_{MSA} we stripped away essential gap information: we will implicitly add it back by considering a certain set of leaves and of nodes, corresponding to the strings occurring at a certain MSA column x .

But first, the notion of semi-repeat-free segment can be broken down into each single row.

► **Definition 19** (Semi-repeat-free substring [22]). *Given a substring $\text{MSA}[i, x..y]$ such that $\text{spell}(\text{MSA}[i, x..y]) \in \Sigma^+$, we say that $\text{MSA}[i, x..y]$ is semi-repeat-free if, for all $1 \leq i' \leq m$, string $\text{spell}(\text{MSA}[i, x..y])$ occurs in gaps-removed row $\text{spell}(\text{MSA}[i', 1..n])$ only at position $g(i', x)$ (defined as in Lemma 6), or it does not occur at all.*

Indeed, Observation 10 holds also for single rows and we can split the computation of $f(x)$, the smallest integer making segment $[x+1, f(x)]$ semi-repeat-free: if substring $\text{MSA}[i, x..y]$ is semi-repeat-free, then $\text{MSA}[i, x..y']$ is semi-repeat-free for all $y' > y$. For each $x \in [0..n-1]$ we can define $f^i(x)$ as the smallest column index greater than x such that $\text{MSA}[i, x+1..f^i(x)]$ is a semi-repeat-free substring. Then, it is easy to see that $f(x) = \max_{i=1}^m f^i(x)$.

Let \mathcal{L}_x be the set of leaves of GST_{MSA} corresponding to suffixes $\text{spell}(\text{MSA}[i, x+1..n]) \cdot \$_i$, for $i \in [1..m]$. In [22] we proved that the *exclusive ancestors* of these leaves, defined as the shallowest of their ancestors covering only leaves in \mathcal{L}_x , correspond to the shortest semi-repeat-free strings starting from column $x+1$. For example, in the MSA of Figure 3 there are two exclusive ancestors for \mathcal{L}_0 , corresponding to strings $\text{G} \cdot \text{CAAG}\6 and T ; for \mathcal{L}_2 they are four and correspond to strings $\text{AAG} \cdot \$6$, $\text{AC} \cdot \text{A}$, $\text{AC} \cdot \text{C}\2 , and $\text{CC} \cdot \text{G}\1 . Let W_x be the set of exclusive ancestors of \mathcal{L}_x : for each leaf $\ell_{i,x+1}$ covered by some $w \in W_x$, the first character of the label from w 's parent to w in GST_{MSA} – the relative node is implicit or is w itself – corresponds to the smallest semi-repeat-free prefix of $\text{MSA}[i, x+1..n]$.

Thus, we have that $f^i(x)$ corresponds to the k -th non-gap symbol of row i , with $k = \text{rank}(\text{MSA}[i, 1..n], x) + \text{stringdepth}(\text{parent}(w)) + 1$, where $\text{rank}(\text{MSA}[i, 1..n], x)$ is the number of non-gap symbols in $\text{MSA}[i, 1..x]$ and $\text{stringdepth}(u) = |\text{string}(u)|$. For example, in the MSA of Figure 3 we have that values $f^i(0)$ for $i \in [1..6]$ are equal to 1, 1, 1, 1, 1, and 2, and values $f^i(2)$ are equal to 8, 7, 6, 6, 5, and 10; in particular, $f^1(2) = 8$ because CCG is the shortest semi-repeat-free substring of $\text{MSA}[1, 3..10]$, and the last G of CCG corresponds to column position 8 in $\text{MSA}[1, 1..10]$. Each \mathcal{L}_x can be transformed into \mathcal{L}_{x+1} by following the suffix links of rows $i \in [1..m]$ such that $\text{MSA}[i, x+1] \neq -$, and the exclusive ancestor set problem on each \mathcal{L}_x can be solved in time $O(m)$: the minimal right extensions can be computed in time $O(mn)$, provided Σ is an integer alphabet of size $\sigma \leq mn$ [22].

4.2 Computing the meaningful prefix-aware right extensions

Given GST_{MSA} and its leaves \mathcal{L}_x corresponding to the suffixes starting at column $x+1$, the forest with m leaves identified by the exclusive ancestors W_x of \mathcal{L}_x can also be used to study the meaningful prefix-aware right extensions \overline{R}_x (Definitions 11 and 16).

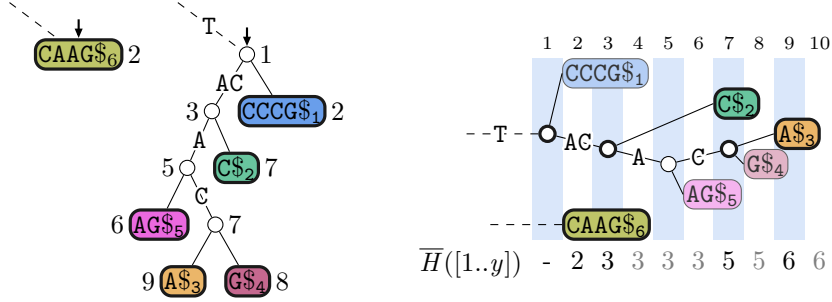
► **Definition 20** (First ending occurrence). *Given GST_{MSA} , let F_x be the set of all explicit nodes of GST_{MSA} belonging to the subtree rooted at some exclusive ancestor $w \in W_x$. Then, for each $v \in F_x$ we define value $\text{pos}(v)$ as the first ending occurrence of string $\text{string}(\text{parent}(v)) \cdot \text{char}(v)$ in the MSA, where $\text{char}(v)$ is the first character of the label from v 's parent to v . In other words, if $S = \text{string}(\text{parent}(v)) \in \Sigma^+$ and $c = \text{char}(v) \in \Sigma$, then $\text{pos}(v)$ is the minimum column index $y \in [1..n]$ such that $Sc = \text{spell}(\text{MSA}[i, x+1..y])$ for some $1 \leq i \leq m$.*

An example of set F_x and of values $\text{pos}(v)$ is shown in Figure 4. After plotting these values in a horizontal line, it is easy to notice that all increases in \overline{H} correspond to some $\text{pos}(v)$, but not the other way around: the pos values that do not affect \overline{H} , because they do not correspond to two or more rows reading strings that are not one prefix of the other, are the first-born children – with respect to the value of pos – of branching nodes.

► **Definition 21** (First-born nodes). *Given GST_{MSA} and its forest F_x corresponding to all semi-repeat-free strings starting from column $x+1$, with $0 \leq x < n$, for each internal node $v \in F_x$ we arbitrarily choose one of its children with minimum value of pos to be a first-born node of F_x . Then, let \widehat{F}_x be the subset of non-first-born nodes of F_x , obtained by removing these nodes from F_x .*

► **Lemma 22.** *Given GST_{MSA} and the set of non-first-born nodes \widehat{F}_x associated with column x , for any $y \in [f(x)..n]$ the prefix-aware height of segment $[x+1..y]$ is equal to the number of nodes in \widehat{F}_x having pos value equal or smaller than y , in symbols $\overline{H}([x+1..y]) = |\{\widehat{v} \in \widehat{F}_x : \text{pos}(\widehat{v}) \leq y\}|$.*

19:14 Elastic Founder Graphs of Minimum Height



■ **Figure 4** On the left, the forest F_0 of the example MSA of Figure 3, annotated with values $\text{pos}(v)$. On the right, the same forest plotted against the MSA columns, with only the non-first-born nodes of \widehat{F}_0 highlighted. Note that $f(0) = f^6(0) = 2$.

Proof. For any $v \in F_x$, let I_v be the set of indexes of the rows whose suffix is covered by v . From the properties of GST_{MSA} it follows that if any $v_1, v_2 \in F_x$ are not one ancestor of the other, then the corresponding strings $\text{string}(\text{parent}(v_1)) \cdot \text{char}(v_1)$ and $\text{string}(\text{parent}(v_2)) \cdot \text{char}(v_2)$ are not one prefix of the other: if $y \geq \text{pos}(v_1)$ and $y \geq \text{pos}(v_2)$, then $\overline{H}(\text{MSA}[I_{v_1} \cup I_{v_2}, x + 1..y]) = \overline{H}(\text{MSA}[I_{v_1}, x + 1..y]) + \overline{H}(\text{MSA}[I_{v_2}, x + 1..y])$. We call this key property the *independence of collateral relatives*.³ In particular, the property holds for any subset U of children of some node $v \in F_x$, provided $y \geq \max_{u \in U} \text{pos}(u)$, and it holds for the exclusive ancestors of $W_x \subseteq F_x$, because $y \geq f(x) \geq \max_{w \in W_x} \text{pos}(w)$:

$$\overline{H}(\text{MSA}[1..m, x + 1..y]) = \sum_{w \in W_x} \overline{H}(\text{MSA}[I_w, x + 1..y]). \quad (4)$$

We can now prove the modification of the thesis restricted to the rows I_v of any node $v \in F_x$. To do so, we introduce one final notation: we denote with \widehat{F}_x^v the set $(\widehat{F}_x \cap \text{GST}_{\text{MSA}}(v)) \cup \{v\}$, that also deals with the case when v is a first-born node. Then, for any $v \in F_x$ and $y \in [\max_{i \in I_v} f^i(x)..n]$ we have that

$$\overline{H}(\text{MSA}[I_v, x + 1..y]) = \begin{cases} 1 & \text{if } y < \text{pos}(v), \\ |\{\hat{v} \in \widehat{F}_x^v : \text{pos}(\hat{v}) \leq y\}| & \text{otherwise.} \end{cases} \quad (5)$$

The proof of Equation (5) proceeds by induction on the height of the subtree rooted at v .

Base case: If v is a leaf then $\widehat{F}_x^v = \{v\}$, $I_v = \{i\}$ for some $i \in [1..m]$, and $\overline{H}(\text{MSA}[\{i\}, x + 1..y]) = 1$, so Equation (5) is easily verified.

Inductive hypothesis: Equation (5) holds for all nodes v such that the subtree rooted at v has height less than or equal to $h \geq 0$.

Inductive step: Let the height of the subtree rooted at v be equal to $h + 1$, and let u_1, \dots, u_p be the $p \geq 2$ children of v , with u_1 the first-born. If $y < \text{pos}(v)$ then all occurrences of $Sc = \text{string}(\text{parent}(v)) \cdot \text{char}(v)$ in the MSA end after column y , so all strings $\text{spell}(\text{MSA}[i, x + 1..y])$ with $i \in I_v$ are prefixes of Sc and $\overline{H}(\text{MSA}[I_v, x + 1..y]) = 1$. Using the same argument, Equation (5) is also verified if $y < \text{pos}(u_1)$, so we can assume $y \geq \text{pos}(u_1) >$

³ In genealogical terms, the ancestor relationship is described as a direct line, opposed to a collateral line for relatives that are not in a direct line.

$\text{pos}(v)$. Consider the children u_k of v such that $y < \text{pos}(u_k)$, for $2 \leq k \leq p$; the strings spelled in the corresponding rows I_{u_k} are prefixes of $\text{string}(\text{parent}(u_1))$, so they are ignored in the prefix-aware height. If $U_{\leq} := \{u_k : 1 \leq k \leq p \wedge \text{pos}(u_k) \leq y\}$ then

$$\begin{aligned} \overline{H}(\text{MSA}[I_v, x + 1..y]) &= \overline{H}\left(\text{MSA}\left[\bigcup_{u \in U_{\leq}} I_u\right][x + 1..y]\right) \\ &= \sum_{u \in U_{\leq}} \overline{H}(\text{MSA}[I_u, x + 1..y]) && \text{indep. collateral relatives} \\ &= \sum_{u \in U_{\leq}} |\{\hat{u} \in \hat{F}_x^u : \text{pos}(\hat{u}) \leq y\}| && \text{inductive hypothesis} \\ &= |\{\hat{v} \in \hat{F}_x^v : \text{pos}(\hat{v}) \leq y\}|. \end{aligned}$$

Note that the last equality holds because $\text{pos}(u_1)$ of $\hat{F}_x^{u_1}$ is replaced by $\text{pos}(v)$ of \hat{F}_x^v . The thesis follows from Equations (4) and (5), because the exclusive ancestors partition the rows $[1..m]$ into $|W_x|$ sets. Also, note that $|\hat{F}_x| = m$. ◀

An example of sets F_x and \hat{F}_x can be seen in Figure 4. Unfortunately, their naive computation takes time $O(m^2)$ if done locally, because GST_{MSA} does not contain the information on the ending occurrences of MSA substrings – and it cannot be easily augmented to do so.

► **Lemma 23.** *Given a general MSA $[1..m, 1..n]$, GST_{MSA} , and the exclusive ancestors W_x , we can compute the meaningful prefix-aware right extensions \overline{R}_x in time $O(m^2)$.*

Proof. For each $v \in F_x$, we can compute $\text{pos}(v)$ by finding for each row $i \in I_v$ the ending position y_i of the occurrence of $Sc = \text{string}(\text{parent}(v)) \cdot \text{char}(v)$ in $\text{MSA}[i, 1..n]$ (Sc is a semi-repeat-free substring so there is at most one occurrence per row). In other words, position y_i corresponds to the k -th non-gap character of row i , where $k = \text{rank}(\text{MSA}[i, 1..n], x) + \text{stringdepth}(\text{parent}(v)) + 1$. Then, $\text{pos}(v) = \min_{i \in I_v} y_i$. The first-born child of v can be found by choosing one of its children with minimum pos values, and the removal of first-born nodes results in the pos values of \hat{F}_x . Given $f(x)$ and the ordered pos values, a simple algorithm like Algorithm 2 in Appendix A considers all columns containing pos values and outputs the relative prefix-aware right extension as a triple $(x, y, \overline{H}([x + 1..y]))$.

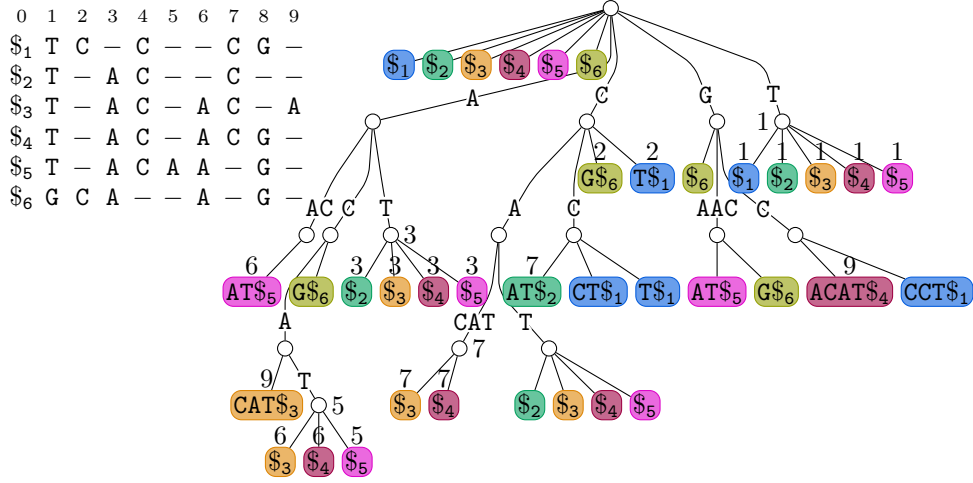
Since we can preprocess in linear time the MSA rows to answer rank and select queries in constant time, the computation of each $\text{pos}(v)$ takes $O(|I_v|)$ time. Forest F_x is composed of compacted trees with m total leaves, so it contains $O(m)$ nodes: the subtrees of F_x can be unbalanced, hence the total time is $O(m^2)$. Then, these values can be sorted in time $O(m \log m)$ and then processed in $O(m)$ time. ◀

4.3 Speedup using weighted ancestor queries

Thanks to Lemma 23, we can compute the meaningful prefix-aware right extensions in $O(m^2n)$ time, the bottleneck being the computation of values $\text{pos}(v)$ (Definition 20). The other tasks can be executed in time $O(mn)$ by generalizing the solution to a global computation: the pos values of all $\hat{F}_0, \dots, \hat{F}_{n-1}$ are $O(mn)$ in total; together they can be sorted in $O(mn)$ time since they take values in $[1..n]$, and they can be separately processed again in total linear time. GST_{MSA} does not contain the information about the ending occurrences of MSA strings, but it does contain the information on the (starting) occurrences: indeed, the sets of leaves $\mathcal{L}_0, \dots, \mathcal{L}_{n-1}$ consider each and every suffix starting from a certain MSA column. This gives us the key idea of symmetry to compute values $\text{pos}(v)$ efficiently.

19:16 Elastic Founder Graphs of Minimum Height

► **Definition 24.** Given a general MSA[1..m, 1..n] from alphabet $\Sigma \cup \{-\}$, we define GPT_{MSA} as the generalized prefix tree of the set of strings $\{\$i \cdot \text{spell}(\text{MSA}[i, 1..n]) : 1 \leq i \leq m\}$, with $\$1, \dots, \m m new distinct terminator symbols not in Σ . Alternatively, GPT_{MSA} can be constructed as the generalized suffix tree of $\{\text{spell}(\text{MSA}[i, 1..n])^{-1} \cdot \$i : 1 \leq i \leq m\}$.



■ **Figure 5** Example of the GPT_{MSA} built from the MSA of Figure 3, annotated with the pos^{-1} values relevant for the computation of R_0 (Figure 4), the meaningful right extensions starting from column 1.

- **Observation 25.** Note that in GPT_{MSA} strings are read from right to left. For each node u of GPT_{MSA} , let $\text{pos}^{-1}(u)$ be the first ending occurrence of $\text{string}(u)$ in some MSA row:
- for any leaf ℓ of GPT_{MSA} corresponding to row $i \in [1..m]$, we have that $\text{pos}^{-1}(\ell)$ is equal to the k -th non-gap character of $\text{MSA}[i, 1..n]$, with $k = |\text{string}(\ell)|$;
 - for any internal node u of GPT_{MSA} , let v_1, \dots, v_p be its children; then $\text{pos}^{-1}(u) = \min_{k=1}^p \text{pos}^{-1}(v_k)$;
 - given a node v of GST_{MSA} , let v^{-1} be the node of GPT_{MSA} corresponding to string $\text{string}(\text{parent}(v)) \cdot \text{char}(v)$ read from right to left; if this is an implicit node, then we define v^{-1} as the first explicit ancestor in GPT_{MSA} ; then $\text{pos}(v) = \text{pos}^{-1}(v^{-1})$.

For example, if v is the GST_{MSA} node of Figures 3 and 4 corresponding to string $\text{TAC} \cdot \text{A}$, v^{-1} corresponds to ACAT in the GPT_{MSA} of Figure 5 and $\text{pos}^{-1}(v^{-1}) = 5$.

► **Lemma 26.** Given a general MSA[1..m, 1..n], GST_{MSA} , and GPT_{MSA} , values $\text{pos}(v)$ for any node v of GST_{MSA} can be computed in $O(mn)$ time.

Proof. As shown in Observation 25, the tree structure of GPT_{MSA} makes it possible to compute $\text{pos}^{-1}(v)$ recursively: similar to the computation of $\mathcal{L}_0, \dots, \mathcal{L}_{n-1}$ this can be done in $O(mn)$ time. It remains to show that given $v \in \text{GST}_{\text{MSA}}$ we can find $v^{-1} \in \text{GPT}_{\text{MSA}}$ in $O(1)$ time: it is straightforward to locate one occurrence of $\text{string}(\text{parent}(v)) \cdot \text{char}(v)$ in the MSA, so we can find v^{-1} by answering the corresponding weighted ancestor query in GPT_{MSA} . Belazzougui et al. recently proved that we can preprocess suffix trees in linear time to be able to answer weighted ancestor queries in constant time [2]. ◀

This concludes the proof of Theorem 17: as we have already shown in Section 3.4, the meaningful prefix-aware right extensions are a drop-in replacement of the original meaningful extensions, so Lemma 26 implies that the optimal segmentation minimizing the maximum prefix-aware height can be computed in linear time.

References

- 1 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Degenerate string comparison and applications. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland*, volume 113 of *LIPICs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.WABI.2018.21.
- 2 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. 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 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.8.
- 3 Giulia Bernardini, Pawel Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, 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.
- 4 Bastien Cazaux, Dmitry Kosolobov, Veli Mäkinen, and Tuukka Norri. Linear time maximum segmentation problems in column stream model. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2019.
- 5 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings Bioinform.*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.
- 6 Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference, IRE-AIEE-ACM '59 (Western)*, pages 295–298, New York, NY, USA, 1959. Association for Computing Machinery. doi:10.1145/1457838.1457895.
- 7 Hannes P. Eggertsson, Snaedis Kristmundsdottir, Doruk Beyter, Hakon Jonsson, Astros Skuladottir, Marteinn T. Hardarson, Daniel F. Guðbjartsson, Kari Stefansson, Bjarni V. Halldorsson, and Pall Melsted. GraphTyper2 enables population-scale genotyping of structural variation using pangenome graphs. *Nature Communications*, 10(1):5402, November 2019. doi:10.1038/s41467-019-13341-9.
- 8 Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. 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 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICALP.2019.55.
- 9 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 Tomás Bures, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdzinski, Claus Pahl, Florian Sikora, and Prudence W. H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science - 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings*, volume 12607 of *Lecture Notes in Computer Science*, pages 608–622. Springer, 2021. doi:10.1007/978-3-030-67731-2_44.
- 10 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.

- 11 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing founder graphs. *CoRR*, abs/2102.12822, 2021. [arXiv:2102.12822](https://arxiv.org/abs/2102.12822).
- 12 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.
- 13 Erik Garrison, Jouni Sirén, Adam Novak, Glenn Hickey, Jordan Eizenga, Eric Dawson, William Jones, Shilpa Garg, Charles Markello, Michael Lin, and Benedict Paten. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36, August 2018. doi:10.1038/nbt.4227.
- 14 Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPICs*, pages 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ESA.2019.51.
- 15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 16 G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.
- 17 Daehwan Kim, Joseph Paggi, Chanhee Park, Christopher Bennett, and Steven Salzberg. Graph-based genome alignment and genotyping with hisat2 and hisat-genotype. *Nature Biotechnology*, 37:1, August 2019. doi:10.1038/s41587-019-0201-4.
- 18 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.
- 19 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
- 20 Tuukka Norri, Bastien Cazaux, Saska Dönges, Daniel Valenzuela, and Veli Mäkinen. Founder reconstruction enables scalable and seamless pangenomic analysis. *Bioinformatics*, 37(24):4611–4619, July 2021. doi:10.1093/bioinformatics/btab516.
- 21 Tuukka Norri, Bastien Cazaux, Dmitry Kosolobov, and Veli Mäkinen. Linear time minimum segmentation enables scalable founder reconstruction. *Algorithms Mol. Biol.*, 14(1):12:1–12:15, 2019.
- 22 Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. In *Proc. 33rd International Workshop on Combinatorial Algorithms (IWOCA 2022)*, 2022. To appear.
- 23 Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. *CoRR*, abs/2201.06492, 2022. [arXiv:2201.06492](https://arxiv.org/abs/2201.06492).
- 24 Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10:R98, 2009.
- 25 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.

A Pseudocode implementations of the algorithms

■ **Algorithm 2** Algorithm computing the meaningful prefix-aware right extensions, given the sorted pos values of \hat{F}_x . Set \hat{F}_x is obtained by removing the first-born nodes from F_x .

Input: Value $f(x)$, values $\text{pos}(w_1), \dots, \text{pos}(w_m)$ of nodes in \hat{F}_x , sorted from smallest to largest order.

Output: Meaningful prefix-aware right extensions \overline{R}_x .

```

1  $h \leftarrow 1$ ;
2 while  $\text{pos}(w_h) < f(x)$  do
3    $h \leftarrow h + 1$ ;
4 while  $h \leq m$  do
5   if  $h = m \vee \text{pos}(w_{h+1}) \neq \text{pos}(w_h)$  then
6      $\text{output}(x, \text{pos}(w_h), h)$ ;
7    $h \leftarrow h + 1$ ;

```

■ **Algorithm 3** Algorithm computing all meaningful right extensions R_0, \dots, R_{n-1} . To efficiently compute keyword trees $K_{x+1..r}$ for $y \in [x+1..n]$, we need a dynamic tree data structure \mathcal{T} supporting navigation and insertions in time $O(\log \sigma)$.

Input: MSA[1..m, 1..n] from an integer alphabet $\Sigma \cup \{-\}$ of size $\sigma \in O(mn)$, minimal right extensions $f(x)$ for $x \in [0..n-1]$.

Output: Meaningful prefix-aware right extensions R_0, \dots, R_{n-1} represented as triples $(x, r_{x,k}, H([x+1..r_{x,k}]))$.

```

1 for  $x \leftarrow 0$  to  $n - 1$  do
2   Initialize empty keyword tree  $\mathcal{T}$ , containing only node root;
3   Initialize array  $V[1..m]$  with values pointers to nodes of  $\mathcal{T}$  and set all values to node root;
4    $h \leftarrow 0$ ;
5    $r \leftarrow x$ ;
6   while  $\mathcal{T}.\text{leaves} < m \wedge r \leq m$  do
7      $h' \leftarrow h$ ;
8     for  $i \leftarrow 1$  to  $m$  do
9       if  $\text{MSA}[i, r] \neq -$  then
10         $V[i].\text{count} \leftarrow V[i].\text{count} - 1$ ;
11        if  $V[i].\text{count} = 0$  then
12           $h \leftarrow h - 1$ ;
13        if  $V[i]$  has an (MSA[ $i, r$ ])-child  $v$  then
14           $V[i] \leftarrow v$ ;
15        else
16          Add new (MSA[ $i, r$ ])-child  $v$  to  $V[i]$ ;
17           $V[i] \leftarrow v$ ;
18        if  $V[i].\text{count} = 0$  then
19           $h \leftarrow h + 1$ ;
20         $V[i].\text{count} \leftarrow V[i].\text{count} + 1$ ;
21   if  $r = f(x)$  then
22      $\text{output}(x, r, h)$ ;
23   else if  $r \geq f(x) \wedge h \neq h'$  then
24      $\text{output}(x, r, h)$ ;
25    $r \leftarrow r + 1$ ;

```

Longest Palindromic Substring in Sublinear Time


Panagiotis Charalampopoulos ✉ 

Reichman University, Herzliya, Israel

Solon P. Pissis ✉ 

CWI, Amsterdam, The Netherlands

Vrije Universiteit, Amsterdam, The Netherlands

Jakub Radoszewski ✉ 

Institute of Informatics, University of Warsaw, Poland

Abstract

We revisit the classic algorithmic problem of computing a longest palindromic substring. This problem is solvable by a celebrated $\mathcal{O}(n)$ -time algorithm [Manacher, *J. ACM* 1975], where n is the length of the input string. For small alphabets, $\mathcal{O}(n)$ is not necessarily optimal in the word RAM model of computation: a string of length n over alphabet $[0, \sigma)$ can be stored in $\mathcal{O}(n \log \sigma / \log n)$ space and read in $\mathcal{O}(n \log \sigma / \log n)$ time. We devise a simple $\mathcal{O}(n \log \sigma / \log n)$ -time algorithm for computing a longest palindromic substring. In particular, our algorithm works in sublinear time if $\sigma = 2^{o(\log n)}$. Our technique relies on periodicity and on the $\mathcal{O}(n \log \sigma / \log n)$ -time constructible data structure of Kempa and Kociumaka [STOC 2019] that answers longest common extension queries in $\mathcal{O}(1)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, longest palindromic substring, longest common extension

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.20

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation, grant no. 810/21.

Solon P. Pissis: Supported 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.

Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

1 Introduction

We start with some basic definitions and notation. Let $S = S[0] \cdots S[n-1]$ be a *string* of length $n = |S|$ over an alphabet Σ of σ *letters*. We consider throughout an integer alphabet $\Sigma = [0, \sigma) \subseteq [0, n)$. The *empty string* is the unique string of length 0. For any two positions i and $j \geq i$ of S , $S[i..j]$ is the *fragment* of S starting at position i and ending at position j ; it is represented in $\mathcal{O}(1)$ space by i and j . The fragment $S[i..j]$ is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$; we say that P occurs at *position* i in S . A fragment $S[i..j]$ can be equivalently written as $S[i..j+1)$, $S(i-1..j]$, or $S(i-1..j+1)$. A *prefix* of S is a fragment of the form $S[0..j]$ and a *suffix* of S is a fragment of the form $S[i..n)$. A substring of S is *proper* when it does not equal S . By ST we denote the *concatenation* of two strings S and T . We denote the reverse string of S by S^R , i.e., $S^R = S[n-1] \cdots S[0]$. A palindrome is a symmetric word that reads the same backward and forward. Formally, a string S is said to be a *palindrome* if and only if $S = S^R$.

In this work, we consider the classic algorithmic problem of computing a longest palindromic substring.



© Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 20; pp. 20:1–20:9

Leibniz International Proceedings in Informatics



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

LONGEST PALINDROMIC SUBSTRING

Input: A string S of length n over an integer alphabet $[0, \sigma)$ with $\sigma \leq n$.

Output: Positions $i, j \in [0, n)$ such that $S[i..j]$ is a longest palindromic substring of S .

LONGEST PALINDROMIC SUBSTRING can be solved in $\mathcal{O}(n)$ time by Manacher’s celebrated algorithm [34, 3], by Jeurling’s algorithm [31] or by Gusfield’s simple algorithm, which uses longest common extension queries [28]. Other settings in which the problem has been studied include the compressed setting, where the input string is given as a straight-line program [36], the streaming setting [26], the dynamic setting, where the string undergoes updates [1, 2], and a semi-dynamic setting [23]. Le Gall and Seddighin [24] have recently presented a strongly sublinear-time quantum algorithm for the problem and a quantum lower bound.

The detection of palindromes is a well-studied problem with a lot of variants [29, 30, 19, 25, 5, 22, 12, 41, 40, 39] arising out of different practical scenarios. For instance, in computational biology, palindromes are found in both prokaryotic and eukaryotic genomes and they have been linked with countless possible functions. They play an important role in the regulation of gene activity and other cell processes because these are often observed near promoters, introns, and specific untranslated regions; for more details see [38, 13, 18, 42, 43, 35].

Our Model and Result

The main contribution of our work is to improve on the existing linear-time solutions to LONGEST PALINDROMIC SUBSTRING in the word RAM model of computation when the input string is given in a packed representation. Let us now describe this model in more detail.

We assume the unit-cost word RAM model with word size $w = \Theta(\log n)$ and a standard instruction set including arithmetic operations, bitwise Boolean operations, and shifts. We count the space complexity of our algorithms in machine words used by the algorithm. The *packed representation* of a string S over an integer alphabet $[0, \sigma)$ is a list obtained by storing $\Theta(\log_\sigma n)$ letters per machine word thus representing S in $\mathcal{O}(|S|/\log_\sigma n)$ machine words. If S is given in the packed representation we simply say that S is a *packed string*.

We prove the following result.

► **Theorem 1.** *LONGEST PALINDROMIC SUBSTRING can be solved in $\mathcal{O}(n/\log_\sigma n)$ time, if the input is given in a packed representation.*

In Section 2 we provide the necessary background. In Section 3 we recall the linear-time algorithm for solving LONGEST PALINDROMIC SUBSTRING by Gusfield [28]. We provide our sublinear-time algorithm in Section 4 and conclude in Section 5.

Other Related Work

A large body of work exploits bit-level parallelism in the word RAM model to speed-up string matching algorithms; see [4, 37, 21, 9, 6, 10, 7, 14, 27, 8, 11, 15] and references therein.

2 Preliminaries

Palindromes. Let S be a string of length n . If $S[i..j]$, $0 \leq i \leq j < n$, is a palindrome, the number $\frac{i+j}{2}$ is called the *center* of $S[i..j]$ and the number $\frac{j-i+1}{2}$ is called the *radius* of $S[i..j]$. A palindromic fragment $S[i..j]$ of S is said to be a *maximal palindrome* if there is no longer palindrome in S with center $\frac{i+j}{2}$. Note that a maximal palindrome of S can be a fragment of another palindrome of S and that the longest palindrome in S must be maximal.

Periodicity. A positive integer p is called a *period* of a string S if $S[i] = S[i + p]$ for all $i \in [0, |S| - p)$. We refer to the smallest period as *the period* of the string, and denote it by $\text{per}(S)$. A string S is called *periodic* if $2 \cdot \text{per}(S) \leq |S|$. A *border* of a nonempty string S is a proper substring of S that occurs both as a prefix and as a suffix of S . A string S has a period p if and only if it has a border of length $|S| - p$.

► **Lemma 2** (Periodicity Lemma (weak version) [20]). *If a string S has periods p and q such that $p + q \leq |S|$, then $\text{gcd}(p, q)$ is also a period of S .*

Let $\mathcal{B}(S)$ denote the set of lengths of borders of S . The following characterization of long borders of a string is generally known; cf. [17]. We give a proof of the lemma for completeness.

► **Lemma 3.** *Assume that a string S of length n is periodic with smallest period p . Then $\mathcal{B}(S) \cap [p, n] = \{n - kp : k \in \mathbb{Z}^+\} \cap [p, n]$.*

Proof. (\subseteq) If $b \in \mathcal{B}(S)$, then $q = n - b$ is a period of S . As p is a period of S as well, if $b \geq p$, then by the Periodicity Lemma $\text{gcd}(p, q)$ is also a period of S . This means that p divides q , as otherwise $\text{gcd}(p, q)$ would have been a period of S smaller than p , which is impossible.

(\supseteq) For each integer $k \in [0, n/p)$, the string S has a period kp and hence a border of length $n - kp$. ◀

Longest Common Extension. An important building block of our technique is a so-called longest common extension data structure, first used by Landau and Vishkin in their textbook solution for approximate pattern matching with at most k mismatches [33]. Let us denote the lengths of the longest common prefix and the longest common suffix of two strings U and V by $\text{LCP}(U, V)$ and $\text{LCS}(U, V) = \text{LCP}(U^R, V^R)$ respectively. Given a string S , it is often useful to have a data structure that can efficiently return $\text{LCP}(S[i..n], S[j..n])$ or $\text{LCS}(S[0..i], S[0..j])$; we collectively call such queries *longest common extension (LCE) queries*. Kempa and Kociumaka presented an optimal LCE data structure for packed strings.

► **Theorem 4** ([32, Theorem 5.4]). *Given a packed representation of a string $S \in [0, \sigma)^n$, LCE queries on S can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n/\log_\sigma n)$ -time preprocessing.*

3 LCE-based Linear-Time Algorithm

We describe the linear-time algorithm given by Gusfield for LONGEST PALINDROMIC SUBSTRING [28]. Gusfield's algorithm is based on the following simple fact – its proof follows by the definition of palindromes and by the definition of $\text{LCP}(U, V)$ for two strings U, V .

► **Fact 5.** *Let S be a string of length n . $S[i..j]$ is a palindrome of odd length with center $c = \frac{i+j}{2}$ if and only if $\text{LCP}(S[c+1..n], (S[0..c-1])^R) \geq \frac{j-i}{2}$. $S[i..j]$ is a palindrome of even length with center $c = \frac{i+j}{2}$ if and only if $\text{LCP}(S[\lceil c \rceil..n], (S[0..\lfloor c \rfloor])^R) \geq \frac{j-i+1}{2}$.*

Thus, after constructing an LCE data structure for string $T = SS^R$, it suffices to perform $\mathcal{O}(n)$ LCP queries: one for each integer or half-integer possible center in $[0, n)$. By using any LCE data structure, which is constructible in $\mathcal{O}(n)$ time and answers LCP queries in $\mathcal{O}(1)$ time, such as the one by Landau and Vishkin [33], we obtain a linear-time solution to LONGEST PALINDROMIC SUBSTRING; in fact this algorithm computes all maximal palindromes.

4 Computing a Longest Palindromic Substring in Sublinear Time

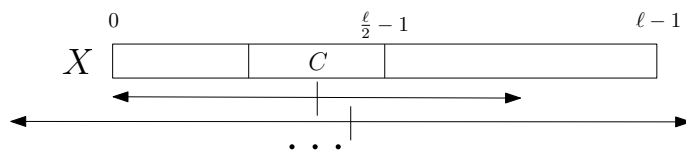
The main goal of this section is to prove Theorem 1; namely, to design an algorithm for LONGEST PALINDROMIC SUBSTRING that works in $\mathcal{O}(n/\log_\sigma n)$ time. Recall that our input is a string S of length n over alphabet $[0, \sigma)$. Let us set $\ell' = \max(1, \lfloor \frac{1}{8} \log_\sigma n \rfloor)$ and $\ell = 4\ell'$. Intuitively, ℓ' and ℓ correspond to lengths of chunks and extended chunks of S , respectively. Our algorithm proceeds with processing every chunk separately. We assume that $n \geq 8$.

Preprocessing. We compute the radii of maximal palindromes with each possible center for every distinct length- ℓ string over $[0, \sigma)$. The number of such length- ℓ strings is $\sigma^\ell = \sigma^{4\ell'} = \mathcal{O}(\sqrt{n})$ and each of them can be stored in one machine word. All the radii can be computed using Manacher's algorithm [34] in $\mathcal{O}(\ell)$ time per string, which takes $\mathcal{O}(\ell\sqrt{n}) = o(n/\log_\sigma n)$ time overall. In the end of the preprocessing step, we store, in an $\mathcal{O}(\sqrt{n})$ -sized array, for each length- ℓ string X , a constant amount of data:

- (a) a longest palindrome in X ;
- (b) a longest palindrome in X that has its center in $[\ell/2 - \ell', \ell/2 - \frac{1}{2}]$; and
- (c) the two longest prefix palindromes of X , if they exist.

Algorithm. The precomputed data allows us to compute the longest palindrome in the length- ℓ prefix and in the length- ℓ suffix of S . This will account for the longest palindrome with the center in the first and last $\ell/2$ positions of S . Let us partition $S[\ell/2..n - \ell/2]$ into chunks of length ℓ' ; if the final chunk has length smaller than ℓ' , we complete it to a length- ℓ' string by taking letters of S preceding it. Our goal is to compute, for each chunk C , the longest palindrome in the whole string S with a center in C ; let us note that this palindrome may be much longer than chunk C , as its length may even be $\Theta(n)$. We assume that a chunk $S[i..i + \ell')$ includes all centers in $[i, i + \ell' - \frac{1}{2}]$, consistently with Item b above.

For each chunk C , we consider the length- ℓ fragment X (*extended chunk*) of S such that C is the second quarter of X , i.e., C is a suffix of $X[0.. \ell/2)$. Let \mathcal{P}_X denote the set of maximal palindromes in S with centers in C that either exceed X or are prefixes of X (inspect Figure 1 for an illustration). We will show that the longest palindrome in \mathcal{P}_X can be computed in the time required to answer $\mathcal{O}(1)$ LCP queries on substrings of SS^R .



■ **Figure 1** Two of the possible palindromes from the set \mathcal{P}_X .

Using the packed representation of S , we can recover the string X packed into one machine word in constant time with word RAM operations. Using the precomputed data for X , we know the at most two longest prefix palindromes P_1, P_2 of X ; we assume that $|P_1| > |P_2|$. If any $P_i, i \in \{1, 2\}$, satisfies $|P_i| \leq \ell - 2\ell'$, we discard it, as the center of the occurrence of this palindrome as a prefix of X does not lie in C . Let \mathcal{Q}_X be the set of palindromes which are prefixes of X of length greater than $\ell - 2\ell'$. Let us note that each of P_1 and P_2 that was not discarded belongs to \mathcal{Q}_X . Each palindrome $P \in \mathcal{P}_X$ has a subpalindrome (palindromic substring) $P' \in \mathcal{Q}_X$ with the same center. If P_1 does not exist, then $\mathcal{P}_X = \emptyset$. If P_1 exists but P_2 does not, then $|\mathcal{P}_X| = 1$. In this case, we can apply Fact 5 to compute the only palindrome in \mathcal{P}_X from P_1 using one LCP query on suffixes of SS^R .

Finally, we consider the case where both P_1 and P_2 exist. Here we use the following well-known property of palindromes.

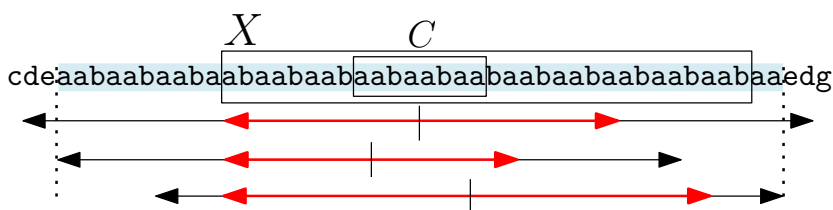
► **Lemma 6** (cf. [19, Lemma 3]). *Let U be a proper prefix of a palindrome V . Then $|V| - |U|$ is a period of V if and only if U is a palindrome. In particular, $\text{per}(V) = |V| - |U|$ if and only if U is the longest palindromic proper prefix of V .*

Let $p := |P_1| - |P_2|$. By Lemma 6, $p = \text{per}(P_1)$. We can check how far this periodicity extends on both sides by using two LCE queries. Namely, if $X = S[i..i + \ell]$, the maximal fragment with period p that contains P_1 is $S[i - a..i + b]$, where $a := \text{LCS}(S[0..i], S[0..i + p])$ and $b := p + \text{LCP}(S[i..n], S[i + p..n])$. We next provide a characterization of the lengths of the palindromes in \mathcal{Q}_X .

► **Lemma 7.** *Let P_1 be the longest prefix palindrome in \mathcal{Q}_X . Further, let $p = \text{per}(P_1)$. The set of lengths of prefix palindromes in \mathcal{Q}_X is $L := \{|P_1| - kp : k \geq 0\} \cap (\ell - 2\ell', \ell]$.*

Proof. (\subseteq) Let $Q \in \mathcal{Q}_X$. We have $p < \ell - (\ell - 2\ell') = 2\ell'$ and $|Q| > \ell - 2\ell'$, so $|Q| - p > \ell - 4\ell' = 0$. By Lemma 6, Q is a border of P_1 , so by Lemma 3, $|Q| \in L$.

(\supseteq) For each k such that $|P_1| - kp \in L$, the string P_1 has a period kp , hence a border of length $|P_1| - kp$. This border is a palindrome by Lemma 6. ◀



■ **Figure 2** Configuration in Lemmas 7 and 8 for $\ell' = 8$ and $\ell = 4\ell' = |X|$. The prefix palindromes in \mathcal{Q}_X are denoted by red arrows; the maximal palindromes in \mathcal{P}_X are denoted by black arrows. The fragment $S[i - a..i + b]$ is shaded in blue.

The periodicity of the elements of \mathcal{Q}_X enables an efficient computation of the longest palindrome in \mathcal{P}_X . For intuition, consider answering $\text{LCP}(S[[c]..n], (S[0..[c]])^R)$ (see Fact 5), for some half-integer $c \in [i - a..i + b] \setminus \mathbb{Z}$, by comparing pairs of letters: either we reach $i - a$ and $i + b - 1$ at the same time, which can happen for at most a single value of c , namely for $(i - a + i + b - 1)/2 = (2i - a + b - 1)/2$, or we reach one of the two endpoints first, in which case we get a mismatch (inspect Figure 2).

► **Lemma 8.** *Let P_1 be the longest prefix palindrome in \mathcal{Q}_X . Further let $P_1 = S[i..i + |P_1|]$, $p = \text{per}(P_1)$, $a = \text{LCS}(S[0..i], S[0..i + p])$ and $b = p + \text{LCP}(S[i..n], S[i + p..n])$. For palindromes $P \in \mathcal{P}_X$ and $Q \in \mathcal{Q}_X$ with the same center c , either $|Q| = b - a$ or $|P| = \min(|Q| + 2a, 2b - |Q|)$.*

Proof. Let us first consider the case where Q and P are even-length palindromes. Note that $[c] = i + |Q|/2$ and recall that $|Q| \geq 2p$. Let $F := S[[c]..[c] + p]$. We have

$$\lambda := \text{LCP}((S[0..[c]])^R, F^n) = |S[i - a..[c]]| = [c] - i + a = |Q|/2 + a, \text{ and}$$

$$\rho := \text{LCP}(S[[c]..n], F^n) = |S[[c]..i + b]| = i + b - [c] = b - |Q|/2.$$

20:6 Longest Palindromic Substring in Sublinear Time

Then, if $\lambda \neq \rho$, we have

$$|P| = 2 \cdot \text{LCP}((S[0..c])^R, S[c..n]) = 2 \cdot \min\{\lambda, \rho\} = \min\{|Q| + 2a, 2b - |Q|\}.$$

Else, we have $\lambda = \rho \Leftrightarrow |Q|/2 + a = b - |Q|/2 \Leftrightarrow |Q| = b - a$.

The proof for the case where Q and P are odd-length palindromes is similar, but we include it for completeness. In this case, $c = i + (|Q| - 1)/2$. Let $F := S[c..c+p]$. We have

$$\lambda := \text{LCP}((S[0..c])^R, F^n) = |S[i - a..c]| = c - i + a + 1 = |Q|/2 + 1/2 + a, \text{ and}$$

$$\rho := \text{LCP}(S[c..n], F^n) = |S[c..i+b]| = i + b - c = b - |Q|/2 + 1/2.$$

Then, if $\lambda \neq \rho$, we have

$$|P| = 2 \cdot \text{LCP}((S[0..c])^R, S[c..n]) - 1 = 2 \cdot \min\{\lambda - 1/2, \rho - 1/2\} = \min\{|Q| + 2a, 2b - |Q|\}.$$

Else, we have $\lambda = \rho \Leftrightarrow |Q| = b - a$ as before. \blacktriangleleft

We use Lemma 8 to compute the longest palindrome in \mathcal{P}_X as follows. For two palindromes $Q \in \mathcal{Q}_X$ and $P \in \mathcal{P}_X$ with the same center such that $|Q| \neq b - a$, either $|Q| < b - a \Leftrightarrow |Q| + 2a < 2b - |Q|$ and hence $|P| = |Q| + 2a$ due to Lemma 8 or $|Q| > b - a \Leftrightarrow |Q| + 2a > 2b - |Q|$ and hence $|P| = 2b - |Q|$ due to Lemma 8. Thus, it suffices to consider only three palindromes in \mathcal{Q}_X . Specifically, with the characterization of Lemma 7 we compute in $\mathcal{O}(1)$ time: the longest palindrome Q_1 in \mathcal{Q}_X of length smaller than $b - a$; the shortest palindrome Q_2 in \mathcal{Q}_X of length greater than $b - a$; and check if there is a palindrome Q_3 in \mathcal{Q}_X of length $b - a$. Finally we pick the longest of the following palindromes from \mathcal{P}_X :

- The palindrome P_I corresponding to Q_1 if Q_1 exists; the length of P_I is $|Q_1| + 2a$ due to the formula from Lemma 8.
- The palindrome P_{II} corresponding to Q_2 if Q_2 exists; the length of P_{II} is $2b - |Q_2|$ due to the formula from Lemma 8.
- The palindrome P_{III} corresponding to Q_3 if Q_3 exists; the center of P_{III} is $i + (|Q_3| - 1)/2$ and hence the length of P_{III} can be computed using one LCP query on suffixes of SS^R due to Fact 5.

Thus we have proved the following lemma.

► **Lemma 9.** *The longest palindrome in \mathcal{P}_X can be computed in the time required to answer $\mathcal{O}(1)$ LCP queries on suffixes of SS^R .*

For each chunk C (we have $\mathcal{O}(n/\ell)$ of them), we take the longer palindrome of the one computed by an application of Lemma 9 and the longest palindrome stored in Item b for the corresponding substring X . Over all chunks, using Theorem 4 to answer LCE queries in $\mathcal{O}(1)$ time, the algorithm requires time $\mathcal{O}(n/\log_\sigma n)$, and we thus obtain Theorem 1.

5 Final Remarks

We have shown an $\mathcal{O}(n/\log_\sigma n)$ -time algorithm for computing a longest palindromic substring of a string of length n over alphabet $[0, \sigma)$. Our algorithm can be easily modified to compute the number of all palindromic fragments of the string within the same time complexity.

We anticipate that our technique will be applicable in many other problems on strings, which currently admit only linear-time solutions. For instance, our approach applied to the prefix array of a string [16] can be used to compute the longest repeating prefix of a string of length n over alphabet $[0, \sigma)$, still in $\mathcal{O}(n/\log_\sigma n)$ time.

References

- 1 Amihod Amir and Itai Boneh. Dynamic palindrome detection. *CoRR*, abs/1906.09732, 2019. arXiv:1906.09732.
- 2 Amihod 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.
- 3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1):163–173, 1995. doi:10.1016/0304-3975(94)00083-U.
- 4 Ricardo A. Baeza-Yates. Improved string searching. *Software: Practice and Experience*, 19(3):257–271, 1989. doi:10.1002/spe.4380190305.
- 5 Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, and Shiho Sugimoto. Diverse palindromic factorization is NP-complete. *International Journal of Foundations of Computer Science*, 29(2):143–164, 2018. doi:10.1142/S0129054118400014.
- 6 Djamal Belazzougui. Worst case efficient single and multiple string matching in the RAM model. In *Combinatorial Algorithms – 21st International Workshop, IWOCA 2010*, volume 6460 of *Lecture Notes in Computer Science*, pages 90–102. Springer, 2010. doi:10.1007/978-3-642-19222-7_10.
- 7 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Optimal packed string matching. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011*, volume 13 of *LIPICs*, pages 423–432. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPICs.FSTTCS.2011.423.
- 8 Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Towards optimal packed string matching. *Theoretical Computer Science*, 525:111–129, 2014. doi:10.1016/j.tcs.2013.06.013.
- 9 Philip Bille. Fast searching in packed strings. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009*, volume 5577 of *Lecture Notes in Computer Science*, pages 116–126. Springer, 2009. doi:10.1007/978-3-642-02441-2_11.
- 10 Philip Bille. Fast searching in packed strings. *Journal of Discrete Algorithms*, 9(1):49–56, 2011. doi:10.1016/j.jda.2010.09.003.
- 11 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic indexing for packed strings. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 6:1–6:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.6.
- 12 Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 23:1–23:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.23.
- 13 Václav Brázda, Martin Bartas, Jiří Lýsek, Jan Coufal, and Miroslav Fojta. Global analysis of inverted repeat sequences in human gene promoters reveals their non-random distribution and association with specific biological pathways. *Genomics*, 112(4):2772–2777, 2020. doi:10.1016/j.ygeno.2020.03.014.
- 14 Dany Breslauer, Leszek Gasieniec, and Roberto Grossi. Constant-time word-size string matching. In *Combinatorial Pattern Matching – 23rd Annual Symposium, CPM 2012*, pages 83–96, 2012. doi:10.1007/978-3-642-31265-6_7.
- 15 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPICs*, pages 30:1–30:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ESA.2021.30.

- 16 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 17 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Wojciech Tyczyński, and Tomasz Waleń. The maximum number of squares in a tree. In *Combinatorial Pattern Matching – 23rd Annual Symposium, CPM 2012*, pages 27–40, 2012. doi:10.1007/978-3-642-31265-6_3.
- 18 Michaela Čutová, Jacinta Manta, Otilia Porubiaková, Patrik Kaura, Jiří Št’astný, Eva B. Jagelská, Pratik Goswami, Martin Bartas, and Václav Brázda. Divergent distributions of inverted repeats and G-quadruplex forming sequences in *Saccharomyces cerevisiae*. *Genomics*, 112(2):1897–1901, 2020. doi:10.1016/j.ygeno.2019.11.002.
- 19 Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48, 2014. doi:10.1016/j.jda.2014.08.001.
- 20 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: <http://www.jstor.org/stable/2034009>.
- 21 Kimmo Fredriksson. Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003. doi:10.1016/S0020-0190(03)00296-5.
- 22 Mitsuru Funakoshi and Takuya Mieno. Minimal unique palindromic substrings after single-character substitution. In *String Processing and Information Retrieval – 28th International Symposium, SPIRE 2021*, pages 33–46, 2021. doi:10.1007/978-3-030-86692-1_4.
- 23 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing longest palindromic substring after single-character or block-wise edits. *Theoretical Computer Science*, 859:116–133, 2021. doi:10.1016/j.tcs.2021.01.014.
- 24 François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022*, volume 215 of *LIPICs*, pages 97:1–97:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITCS.2022.97.
- 25 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter bounds and optimal algorithms for all maximal α -gapped repeats and palindromes – finding all maximal α -gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory of Computing Systems*, 62(1):162–191, 2018. doi:10.1007/s00224-017-9794-5.
- 26 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznański. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. doi:10.1007/s00453-019-00591-8.
- 27 Emanuele Giaquinta, Szymon Grabowski, and Kimmo Fredriksson. Approximate pattern matching with k -mismatches in packed text. *Information Processing Letters*, 113(19-21):693–697, 2013. doi:10.1016/j.ipl.2013.07.002.
- 28 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 29 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome pattern matching. *Theoretical Computer Science*, 483:162–170, 2013. doi:10.1016/j.tcs.2012.01.047.
- 30 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Combinatorial Pattern Matching – 25th Annual Symposium, CPM 2014*, volume 8486 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2014. doi:10.1007/978-3-319-07566-2_16.
- 31 Johan Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica*, 11(2):146–184, 1994. doi:10.1007/BF01182773.
- 32 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

- 33 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.
- 34 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.
- 35 Fernando Martínez-Alberola, Eva Barreno, Leonardo M Casano, Francisco Gasulla, Arantazu Molins, Patricia Moya, María González-Hourcade, and Eva M. Del Campo. The chloroplast genome of the lichen-symbiont microalga *Trebouxia* sp. Tr9 (Trebouxiophyceae, Chlorophyta) shows short inverted repeats with a single gene and loss of the *rps4* gene, which is encoded by the nucleus. *Journal of Phycology*, 56(1):170–184, 2020. doi:10.1111/jpy.12928.
- 36 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.
- 37 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 1998*, volume 1448 of *Lecture Notes in Computer Science*, pages 14–33. Springer, 1998. doi:10.1007/BFb0030778.
- 38 Christopher E. Pearson, Haralabos Zorbas, Gerald B. Price, and Maria Zannis-Hadjopoulos. Inverted repeats, stem-loops, and cruciforms: significance for initiation of DNA replication. *Journal of Cellular Biochemistry*, 63(1):1–22, 1996. doi:10.1002/(SICI)1097-4644(199610)63:1<1::AID-JCB1>3.0.CO;2-3.
- 39 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *String Processing and Information Retrieval – 24th International Symposium, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.
- 40 Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *European Journal of Combinatorics*, 68:249–265, 2018. doi:10.1016/j.ejc.2017.07.021.
- 41 Mikhail Rubinchik and Arseny M. Shur. Palindromic k -factorization in pure linear time. In *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020*, volume 170 of *LIPICs*, pages 81:1–81:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.MFCS.2020.81.
- 42 Xin Tao, Shaochun Yuan, Fan Chen, Xiaoman Gao, Xinli Wang, Wenjuan Yu, Song Liu, Ziwen Huang, Shangwu Chen, and Anlong Xu. Functional requirement of terminal inverted repeats for efficient ProtoRAG activity reveals the early evolution of V(D)J recombination. *National Science Review*, 7(2):403–417, 2020. doi:10.1093/nsr/nwz179.
- 43 Ran Zhou, David Macaya-Sanz, Craig H. Carlson, Jeremy Schmutz, Jerry W. Jenkins, David Kudrna, Aditi Sharma, Laura Sandor, Shengqiang Shu, Kerrie Barry, et al. A willow sex chromosome reveals convergent evolution of complex palindromic repeats. *Genome Biology*, 21(1):1–19, 2020. doi:10.1186/s13059-020-1952-4.

Permutation Pattern Matching for Doubly Partially Ordered Patterns

Laurent Bulteau ✉ 

LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-Vallée, France

Guillaume Fertin ✉ 

Nantes Université, LS2N (UMR 6004), CNRS, Nantes, France

Vincent Jugé ✉

LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-Vallée, France

Stéphane Vialette ✉ 

LIGM, CNRS, Univ Gustave Eiffel, F77454 Marne-la-Vallée, France

Abstract

We study in this paper the DOUBLY PARTIALLY ORDERED PATTERN MATCHING (or DPOP MATCHING) problem, a natural extension of the PERMUTATION PATTERN MATCHING problem. PERMUTATION PATTERN MATCHING takes as input two permutations σ and π , and asks whether there exists an occurrence of σ in π ; whereas DPOP MATCHING takes *two partial orders* P_V and P_P defined on the same set X and a permutation π , and asks whether there exist $|X|$ elements in π whose values (resp., positions) are in accordance with P_V (resp., P_P). Posets P_V and P_P aim at relaxing the conditions formerly imposed by the permutation σ , since σ yields a total order on both positions and values. Our problem being NP-hard in general (as PERMUTATION PATTERN MATCHING is), we consider restrictions on several parameters/properties of the input, e.g., bounding the size of the pattern, assuming symmetry of the posets (i.e., P_V and P_P are identical), assuming that one partial order is a total (resp., weak) order, bounding the length of the longest chain/anti-chain in the posets, or forbidding specific patterns in π . For each such restriction, we provide results which together give a(n almost) complete landscape for the algorithmic complexity of the problem.

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

Keywords and phrases Partial orders, Permutations, Pattern Matching, Algorithmic Complexity, Parameterized Complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.21

1 Preamble

Let us play the following little puzzle game. Among the selection of fifteen cities of the Czech Republic depicted in Figure 1 together with their geographic coordinates, find (if they exist) five cities, say **A**, **B**, **C**, **D** and **E**, such that:

- **A** and **C** are west of **D** and north of **B**,
- **E** is east of **B** and south of **A**,
- **D** is west of **B** and north of **A** and **C**.

It is assumed that no two cities have the same longitude or latitude. Notice that the game does not provide complete information as, for example, no information is provided about the relative positioning of **A** and **C** (and silence is tantamount to consent). We may assume that the information is minimal: requiring **C** is west of **B** is unnecessary since **C** is west of **D** and **D** is west of **B**. One solution is **A** = Praha, **B** = Brno, **C** = Plzeň, **D** = Liberec and **E** = Olomouc. Note that the solution is not unique, as **A** = Plzeň, **B** = Jindřichův Hradec, **C** = Cheb, **D** = Ústí nad Labem and **E** = Brno is another solution.



© Laurent Bulteau, Guillaume Fertin, Vincent Jugé, and Stéphane Vialette;
licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 21; pp. 21:1–21:17

Leibniz International Proceedings in Informatics



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



■ **Figure 1** A Czech Republic map showing 15 of its cities. Their names and GPS coordinates, in increasing order of their longitudes, are: Cheb ($50^{\circ}4'N$ $12^{\circ}22'E$), Plzeň ($49^{\circ}44'N$ $13^{\circ}22'E$), Ústí nad Labem ($50^{\circ}39'N$ $14^{\circ}1'E$), Praha ($50^{\circ}5'N$ $14^{\circ}25'E$), České Budějovice ($48^{\circ}58'N$ $14^{\circ}28'E$), Jindřichův Hradec ($49^{\circ}9'N$ $15^{\circ}0'E$), Liberec ($50^{\circ}46'N$ $15^{\circ}3'E$), Pardubice ($50^{\circ}2'N$ $15^{\circ}46'E$), Hradec Králové ($50^{\circ}12'N$ $15^{\circ}49'E$), Brno ($49^{\circ}11'N$ $16^{\circ}36'E$), Olomouc ($49^{\circ}35'N$ $17^{\circ}15'E$), Zlín ($49^{\circ}13'N$ $17^{\circ}40'$), Opava ($49^{\circ}56'N$ $17^{\circ}54'E$), Ostrava ($49^{\circ}50'N$ $18^{\circ}17'E$) and Karviná ($49^{\circ}51'N$ $18^{\circ}32'E$).
© Creative Commons CC0 1.0 Universal Public Domain Dedication.

We show that this puzzle game can be modeled as a permutation pattern matching problem for doubly partially ordered patterns. Let us first associate a permutation $\pi \in \mathfrak{S}(15)$ with the problem (see Figure 2). We sort the fifteen cities of the Czech Republic depicted in Figure 1 both by increasing longitude (E) and by increasing latitude (N), so that $\pi(i) = j$ if the i -th city going west to east is also the j -th city going south to north. In our example, the “Czech Republic permutation” is $\pi = 11\ 6\ 14\ 12\ 1\ 2\ 15\ 10\ 13\ 3\ 5\ 4\ 9\ 7\ 8$. For example, $\pi(2) = 6$ since Plzeň is the second city going west to east, and the sixth city going south to north. What is left is to define our pattern P : P is composed of two partially ordered sets on the variables $\{A, B, C, D, E\}$ (see Figure 3): one partially ordered set (denoted P_v for *value poset* in the sequel) describes the south-to-north constraints and another partially ordered set (denoted P_p for *position poset* in the sequel) describes the west-to-east constraints.

2 Introduction

We say that a permutation σ *occurs* in another permutation π (or that π *contains* σ) if there exists a subsequence of elements of π that has the same relative order as σ . Otherwise, we say that π *avoids* σ . For example, π contains the permutation $\sigma = 123$ (resp., $\sigma = 321$) if it has an increasing (resp., a decreasing) subsequence of size 3. Similarly, $\sigma = 4312$ occurs in $\pi = 6152347$, as shown in $\textcircled{6}1\textcircled{5}2\textcircled{3}\textcircled{4}7$, but the same $\pi = 6152347$ avoids $\sigma' = 2341$.

Deciding whether a permutation $\sigma \in \mathfrak{S}(k)$ occurs in some permutation $\pi \in \mathfrak{S}(n)$ is NP-complete [7], but is fixed-parameter tractable for the parameter k [15, 17]. Several exponential-time algorithms have been recently proposed [5, 16], improving upon [1, 10]. A vast literature is devoted to the case where both the pattern σ and the target π are restricted to a proper permutation class, e.g., 321-avoiding permutations [18, 2, 21], (213, 231)-avoiding permutations [26], (2413, 3142)-avoiding (a.k.a. *separable*) permutations [19, 25], and $(k \dots 1)$ -avoiding permutations [11]. For more background on permutation patterns and pattern avoidance, we refer to [6] and [24].

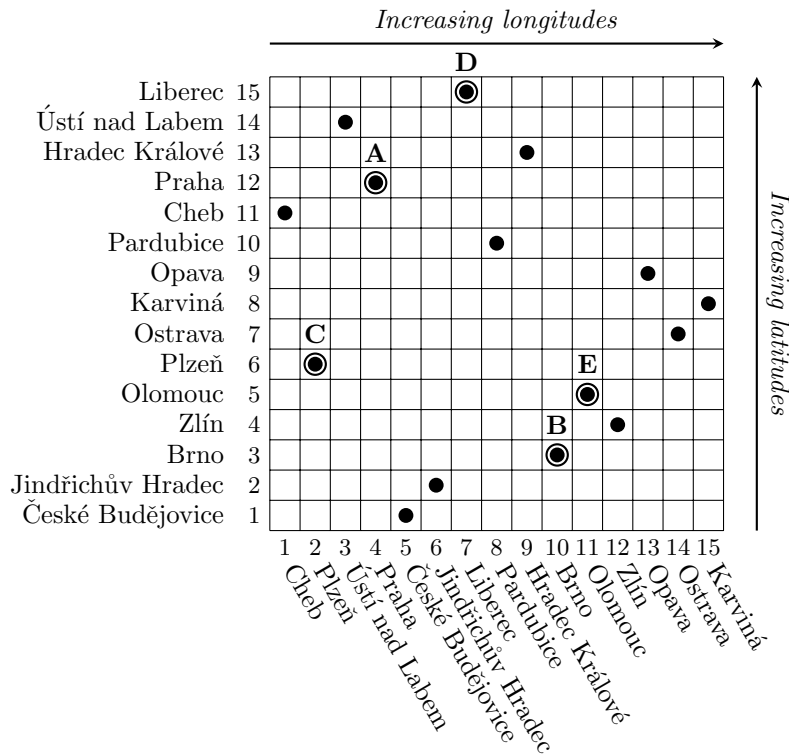


Figure 2 Permutation $\pi = 11\ 6\ 14\ 12\ 1\ 2\ 15\ 10\ 13\ 3\ 5\ 4\ 9\ 7\ 8$ corresponding to the map from Figure 1. The solution of our puzzle, depicted Figure 1, is also represented.

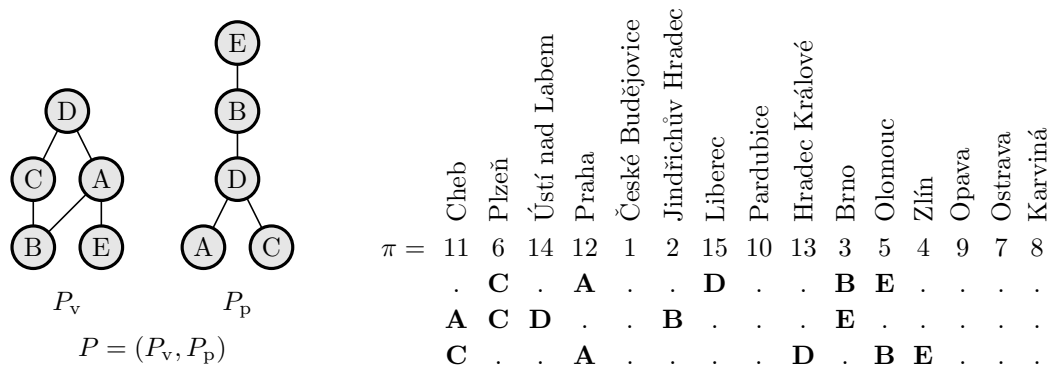


Figure 3 A dpop $P = (P_v, P_p)$ representing the pattern for our puzzle, together with three distinct occurrences. Note that partial orders are represented by Hasse diagrams, i.e., a bottom-up path in P_v (resp., P_p) implies a bottom-up (resp., left-right) relation in the occurrence of P in π .

In the last years, the notion of *pattern* has been generalized in several ways. A *vincular pattern* is a permutation in which some entries must occur consecutively [4]. *Consecutive patterns* are a special case of vincular patterns in which all entries need to be adjacent [13]. *Bivincular patterns* generalize classical patterns even further than vincular patterns by requiring that not only positions but also values of elements involved in a matching may be forced to be adjacent [8]. *Mesh patterns* (a further generalization of bivincular patterns)

impose further restrictions on the relative positions of the entries in an occurrence of a pattern [9] and *boxed mesh patterns* are special cases of mesh patterns [3]. Strongly related to our approach are *partially ordered patterns* that are vincular patterns in which the relative order of some elements is not fixed [23]. The best general reference is [24].

In this paper, we consider a new generalization of classical patterns in which both the relative order and the relative positioning of some elements are not fixed. The idea is to allow the possibility for some elements to be incomparable in value (i.e., their relative order is unknown) and to go one step further by allowing the possibility for some elements to be incomparable in position (i.e., their relative positioning in the occurrence is unknown). Since the problem is clearly NP-hard (as it contains PERMUTATION PATTERN MATCHING as a sub-problem), our goal is to identify tractable cases when restrictions apply to the pattern and/or to the permutation.

The restrictions we consider here apply to the following parameters of the problem: size of the pattern; symmetry (i.e., same partial order in positions and values); one partial order is a total (resp., weak) order; size of the longest chain (resp., anti-chain) in the partial orders (height and width); forbidden patterns in π . On the positive side, we show that the FPT algorithm for PERMUTATION PATTERN MATCHING can be generalized to our setting (with the pattern size as parameter). We further give polynomial-time algorithms when the pattern is a symmetric disjoint union of a constant number of weak orders. Finally, we also provide polynomial-time algorithms when the pattern is symmetric and the permutation belongs to some restricted classes, such as (123, 132)-avoiding permutations. We complement these positive results with NP- or W[1]-hardness proofs in most of the remaining cases.

3 Definitions

Permutations and Patterns

A permutation σ is said to be *contained* in (or is a *sub-permutation* of) another permutation π , which we denote by $\sigma \preceq \pi$, if π has a (not necessarily contiguous) subsequence whose terms are order-isomorphic to σ . We also say that π *admits an occurrence* of the *pattern* σ . If no such subsequence exists, we say that π *avoids* σ (or is σ -*avoiding*). A permutation is *separable* if it avoids both 2413 and 3142. PERMUTATION PATTERN MATCHING is the problem of deciding whether a permutation is contained into another permutation.

For any non-negative integer n , we denote by $[n]$ the set $\{1, 2, \dots, n\}$. When $n \geq 1$, we also note $\text{ip}(n) = 1\,2\,\dots\,n$ the *increasing permutation* of length n and $\text{dp}(n) = n\,(n-1)\,\dots\,1$ the *decreasing permutation* of length n . Let $\pi \in \mathfrak{S}(n)$. The *reverse* (resp., *complement*) of π is the permutation $\pi^r = \pi(n)\pi(n-1)\,\dots\,\pi(1)$ (resp., $\pi^c = (n-\pi(1)+1)(n-\pi(2)+1)\,\dots\,(n-\pi(n)+1)$). The *inverse* of π is the permutation $\pi^{-1} \in \mathfrak{S}(n)$ defined by $\pi^{-1}(j) = i$ if and only if $\pi(i) = j$. Given a permutation π of size m and a permutation σ of size n , the *skew sum* of π and σ is the permutation of size $m+n$ defined by

$$(\pi \ominus \sigma)(i) = \begin{cases} \pi(i) + n & \text{for } 1 \leq i \leq m, \\ \sigma(i - m) & \text{for } m + 1 \leq i \leq m + n, \end{cases}$$

and the *direct sum* of π and σ is the permutation of size $m+n$ defined by

$$(\pi \oplus \sigma)(i) = \begin{cases} \pi(i) & \text{for } 1 \leq i \leq m, \\ \sigma(i - m) + m & \text{for } m + 1 \leq i \leq m + n. \end{cases}$$

Orders

A relation \leq is a *partial order* on a set X if it has:

- *reflexivity*: for all $x \in X$, $x \leq x$ (i.e., every element is related to itself);
- *transitivity*: for all $x, x', x'' \in X$, if $x \leq x'$ and $x' \leq x''$, then $x \leq x''$;
- *antisymmetry*: for all $x, x' \in X$, if $x \leq x'$ and $x' \leq x$, then $x = x'$ (i.e., no two distinct elements precede each other).

If \leq has the following additional property, we say that it is a *weak order* on X :

- *transitivity of incomparability*: for all pairwise distinct $x, x', x'' \in X$, if x is incomparable with x' (i.e., neither $x \leq x'$ nor $x' \leq x$ is true) and if x' is incomparable with x'' , then x is incomparable with x'' .

Two subsets X_1, X_2 are *independent* if there is no $x_1 \in X_1, x_2 \in X_2$ such that $x_1 \leq x_2$ or $x_2 \leq x_1$. We say that a partial order is *k-weak* if there exists a partition of X into k pairwise independent sets X_1, \dots, X_k such that, for each i , the restriction of \leq to X_i is a weak order (in other words, \leq is the disjoint union of k weak partial orders).

Let $\mathcal{P} = (X, \leq)$ be a finite partially ordered set. A *chain* in \mathcal{P} is a set of pairwise comparable elements (i.e., a totally ordered subset) and an *antichain* in \mathcal{P} is a set of pairwise incomparable elements. The *partial order height* of \mathcal{P} , denoted by $\text{height}(\mathcal{P})$, is defined as the maximum cardinality of a chain in \mathcal{P} , and the *partial order width* of \mathcal{P} , denoted by $\text{width}(\mathcal{P})$, is defined as the maximum cardinality of an antichain in \mathcal{P} . By Dilworth Theorem, $\text{width}(\mathcal{P})$ is also the minimum number of chains in any partition of \mathcal{P} into chains. The *dual* of \mathcal{P} is the partial order $\mathcal{P}^\partial = (X, \leq^\partial)$ defined by letting \leq^∂ be the converse relation of \leq , i.e., $x \leq^\partial x'$ if and only if $x' \leq x$. The dual of a partial order is a partial order and the dual of the dual of a relation is the original relation. A *total order* is a partial order in which any two elements are comparable, and a set equipped with a total order is a *totally ordered set*. A *linear extension* of a partial order is a total order that is compatible with the partial order. It will be convenient to represent a linear extension of a poset $\mathcal{P} = (X, \leq)$ as the mapping $\tau_{\mathcal{P}} : X \rightarrow [|X|]$ such that $\tau_{\mathcal{P}}(i) < \tau_{\mathcal{P}}(j)$ if $i < j$ in the linear extension.

A *doubly partially ordered pattern* (dpop) P is a pair, denoted by $P = (P_v, P_p)$, of posets $P_v = (X, \leq_v)$ and $P_p = (X, \leq_p)$ defined over the same set X . We call P_v and P_p the *value poset* and the *position poset*, respectively. A dpop $P = (P_v, P_p)$ is *symmetric* if $P_v = P_p$, *dual* if $P_v = P_p^\partial$, and *semi-total* if one of P_p or P_v is a total order. We let $\text{height}(P)$ and $\text{width}(P)$ stand for $\max\{\text{height}(P_v), \text{height}(P_p)\}$ and $\max\{\text{width}(P_v), \text{width}(P_p)\}$, respectively. Finally, the *size* of P is defined as the cardinality $|X|$ and is denoted by $|P|$.

► **Definition 1** (DPOP MATCHING). *Given a permutation $\pi \in \mathfrak{S}(n)$ and a dpop $P = (P_v, P_p)$, an occurrence (or mapping) of P in π is an injective function $\varphi : X \rightarrow [n]$ such that:*

- $\pi \circ \varphi$ is \leq_v -non-decreasing, i.e., for all $x, y \in X$, if $x \leq_v y$ then $\pi(\varphi(x)) \leq \pi(\varphi(y))$, and
- φ is \leq_p -non-decreasing, i.e., for all $x, y \in X$, if $x \leq_p y$ then $\varphi(x) \leq \varphi(y)$.

The DPOP MATCHING problem consists in deciding whether P occurs in π .

First Observations

► **Observation 2.** *PERMUTATION PATTERN MATCHING is the special case of DPOP MATCHING where both \leq_v and \leq_p are total orders.*

We note that applying a vertical and/or horizontal symmetry on both pattern and permutation does not alter the existence of an occurrence.

► **Observation 3.** Let $P = (P_V, P_P)$ be a dpop and π be a permutation. The following statements are equivalent:

1. (P_V, P_P) occurs in π ;
2. $(P_V, (P_P)^\partial)$ occurs in π^r ;
3. $((P_V)^\partial, P_P)$ occurs in π^c ;
4. $((P_V)^\partial, (P_P)^\partial)$ occurs in π^{cr} ;
5. (P_P, P_V) occurs in π^{-1} .

The following reformulation will prove useful.

► **Observation 4.** Let $P = (P_V, P_P)$ be a dpop with $P_V = (X, \leq_V)$, $P_P = (X, \leq_P)$ and $k = |X|$, and let $\pi \in \mathfrak{S}(n)$ be a permutation. The following statements are equivalent:

- P occurs in π .
- There exists a linear extension $\tau_V : X \rightarrow [k]$ of P_V and a linear extension $\tau_P : X \rightarrow [k]$ of P_P such that the permutation $\sigma \in \mathfrak{S}(k)$ defined by $\sigma(i) = \tau_V(\tau_P^{-1}(i))$ for $1 \leq i \leq k$ is contained in π .

The rationale for the reformulation introduced in Observation 4 stems from the following corollary that sets the general context.

► **Corollary 5** ([17]). *DPOP MATCHING* is FPT for the parameter $|P|$.

Indeed, it is enough to guess two linear extensions $\tau_V : X \rightarrow [k]$ of P_V and $\tau_P : X \rightarrow [k]$ of P_P , and to check if the permutation $\sigma \in \mathfrak{S}(k)$ defined by $\sigma(i) = \tau_V(\tau_P^{-1}(i))$ for $1 \leq i \leq k$ is contained in π . There are $O(k!^2)$ pairs of such extensions and, for each of them, one can check whether σ occurs in π in $n 2^{O(k^2 \log k)}$ time [17].

4 Semi-Total Patterns

In this section we focus on semi-total patterns, i.e., without loss of generality, on the case where P_P is a total order (up to symmetry by Observation 3). This case still contains PERMUTATION PATTERN MATCHING as a special case, and is thus NP-hard. We focus on small-height value partial orders, i.e., on dpop with constant height(P_V), and give an XP algorithm for weak orders (Proposition 6) and paraNP-hardness in general (Proposition 7).

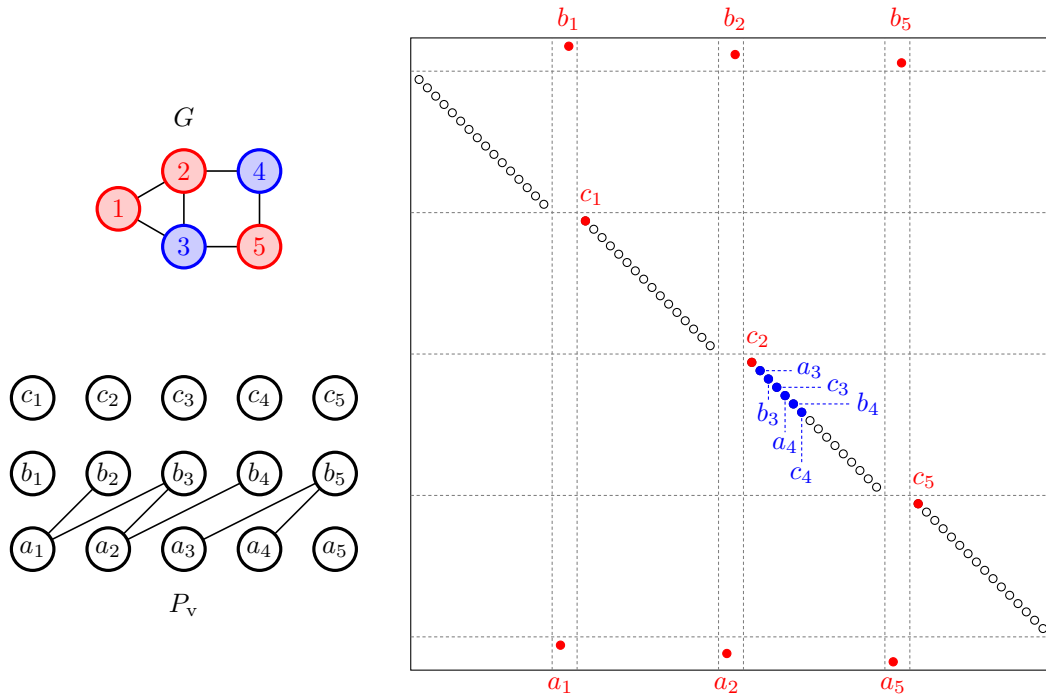
► **Proposition 6.** *DPOP MATCHING* is solvable in $O(n^{\text{height}(P_V)})$ time if P_V is a weak order and P_P is a total order.

Proof. Let $\pi \in \mathfrak{S}(n)$ be a permutation and $P = (P_V, P_P)$ be a dpop on some ground set X , where P_V is a weak order and P_P is a total order. Without loss of generality, we assume that X is the set $[k]$ and that P_P is the usual order on integers. For every $x \in X$, we abusively denote by $\text{height}(x)$ the maximum cardinality of a chain with maximum element x in P_V . Finally, set $\ell = \text{height}(P_V)$.

For any two distinct variables $x, y \in X$, we have $x <_V y$ if and only if $\text{height}(x) < \text{height}(y)$. Thus, P occurs in π if and only there exists a sequence $0 = a_0 < a_1 < a_2 < \dots < a_\ell = n$ such that w_σ is a subsequence of w_π , where $w_\pi \in [\ell]^n$ and $w_\sigma \in [\ell]^k$ are the two words defined by $w_\pi[i] = \min\{j : a_{j-1} < \pi(i) \leq a_j\}$ and $w_\sigma[i] = \text{height}(i)$.

As for the running time, there exist $\binom{n-1}{\ell-1}$ distinct sequences $(a_i)_{0 \leq i \leq \ell}$ and deciding whether w_σ occurs in w_π as a subsequence is a linear-time procedure. ◀

► **Proposition 7.** *DPOP MATCHING* is NP-complete even if $\text{height}(P_V) = 2$, P_P is a total order and π avoids 1234.



■ **Figure 4** Top left: vertex cover (in red) in a 5-vertex graph G . Bottom left: partially ordered set P_v constructed from G . Right: an occurrence of P in π obtained from our size-3 vertex cover.

Proof. We perform a reduction from VERTEX COVER, which is known to be NP-complete [22]. Let $G = (V, E)$ be a graph and let k be a positive integer. We identify V with the set $[n]$.

We construct a dpop $P = (P_v, P_p)$, where $P_v = (X, \leq_v)$ is a height-2 partial order and $P_p = (X, \leq_p)$ a total order, as follows. We set $X = \{a_1, b_1, c_1, a_2, b_2, c_2, \dots, a_n, b_n, c_n\}$, so that $|X| = 3n$. Then, we set $a_1 \leq_p b_1 \leq_p c_1 \leq_p a_2 \leq_p b_2 \leq_p c_2 \leq_p \dots \leq_p a_n \leq_p b_n \leq_p c_n$, which defines the total order \leq_p . Finally, for each edge $\{i, j\}$ in E with $i < j$, we set $a_i \leq_v b_j$; all other elements of X are pairwise incomparable by \leq_v . This defines a partial order \leq_v such that $\text{height}(\leq_v) = 2$.

Write now $N = 3n + 3$ and $m = (k + 1)N - 2$, and define a permutation $\pi \in \mathfrak{S}(m)$ as follows:

- $\pi(iN + j) = (m + 1) - (iN + j + k)$ whenever $0 \leq i \leq k$ and $1 \leq j \leq N - 2$;
- $\pi(iN - 1) = (k + 1) - i$ and $\pi(iN) = (m + 1) - i$ whenever $1 \leq i \leq k$.

It is straightforward to check that π is 1234-avoiding. It is also easy to see how the construction, illustrated in Figure 4, can be accomplished in polynomial-time.

Let us see under which conditions an injective \leq_p -non-decreasing function $\varphi: X \rightarrow [m]$ maps P into π . We say that a vertex i belongs to the j -th gadget if one of the integers $\varphi(a_i)$ or $\varphi(b_i)$ is equal to $jN - 1$ or to jN , i.e., if $\{\varphi(a_i), \varphi(b_i)\} \cap \{jN - 1, jN\} \neq \emptyset$. When two elements in the range of φ are consecutive, either they are integers $\varphi(a_i)$ and $\varphi(b_i)$ for a given i , or one of them is an integer $\varphi(c_i)$ for some i . Therefore, no two distinct vertices i and i' can belong to the same j -th gadget. Consequently, and since there are k gadgets, the set V' of vertices i that belong to some gadget is of size at most k .

Then, we define a notion of *height* as follows: for each element x of X , we set $\text{height}(x) = 0$ if N divides $\varphi(x) + 1$, $\text{height}(x) = 2$ if N divides $\varphi(x)$, and $\text{height}(x) = 1$ otherwise. By construction, for all $x, y \in X$ such that $x \leq_p y$, we have $\pi(\varphi(x)) \leq \pi(\varphi(y))$ if and only if x

is of smaller height than y . Therefore, if φ maps P into π , and for each relation $a_i \leq_v b_j$, either a_i has height 0 or b_j has height 2. In particular, either i or j must belong to V' , and therefore V' is a vertex cover of size at most k .

Conversely, provided that there exist vertices $v(1) < v(2) < \dots < v(k)$ that form a vertex cover V' , we construct an occurrence of P in π as follows. First, we abusively set $v(0) = 0$. Then, for all $i \in [k]$, we set $f(i) = jN + 3(i - v(j))$, where j is the largest integer such that $v(j) \leq i$. We set $\varphi(a_j) = f(j) - 1$, $\varphi(b_j) = f(j)$ and $\varphi(c_j) = f(j) + 1$.

By construction, we have $f(i) + 3 \leq f(i + 1)$ for all i , and therefore φ is an injective \leq_p -non-decreasing function. Moreover, for every $i \in [n]$, the elements a_i and b_i have heights 0 and 2 if $i \in V'$, and they have height 1 if $i \notin V'$. It follows that $\pi(\varphi(a_i)) \leq \pi(\varphi(b_j))$ whenever $a_i \leq_v b_j$, i.e., that φ is an occurrence of P in π . ◀

5 Symmetric Patterns

This section is devoted to studying complexity issues of pattern matching for symmetric dpop (i.e., those dpop $P = (\mathcal{P}, \mathcal{P})$, whose value and position posets coincide). We further focus on two special cases, first when \mathcal{P} has a bounded width, then when π is restricted to constrained pattern-avoiding classes of permutations.

5.1 Symmetric Pattern with Bounded Width

We first observe that the problem is polynomial for width 1 (Observation 8). We further prove $W[1]$ -hardness for the parameter k when \mathcal{P} is a disjoint union of k chains (Proposition 10). We complement this result with an XP algorithm for the slightly more general case where \mathcal{P} is a disjoint union of weak orders (Proposition 11). Note that the existence of an XP algorithm for the width parameter remains open, and we conjecture that the problem is NP-hard even for constant width.

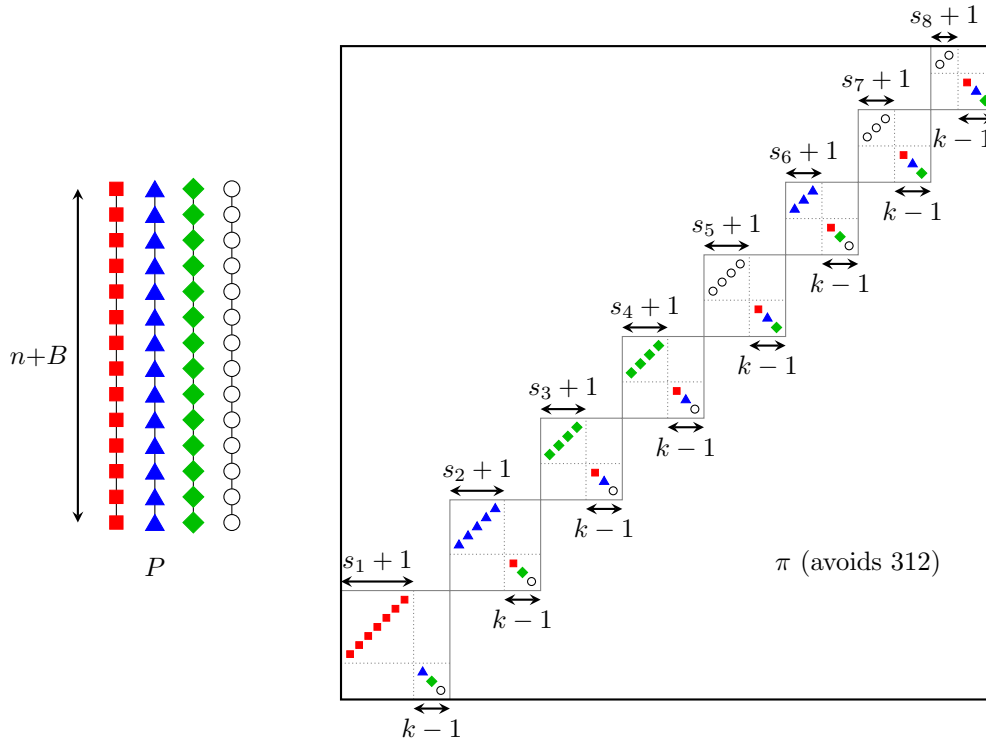
► **Observation 8.** *DPOP MATCHING is solvable in $O(n \log \log |P|)$ time for a symmetric dpop P of width 1 (i.e., a total symmetric dpop P).*

Proof. If P has width 1, then $P = (\mathcal{P}, \mathcal{P})$ for some total order $\mathcal{P} = (X, \leq)$. In particular, we can write $X = \{x_1, \dots, x_{|X|}\}$ with $x_i < x_j$ for $i < j$, and in any mapping $\phi : X \rightarrow [n]$, the elements $\pi_{\phi(x_1)}, \dots, \pi_{\phi(x_{|X|})}$ must form an increasing subsequence of π . Conversely, any size- $|X|$ increasing subsequence of π can be used as an image for ϕ , so in this setting DPOP MATCHING corresponds to the longest increasing subsequence problem, which can be solved in $O(n \log \log |X|)$ time [12]. ◀

To simplify the exposition of our next result, we introduce a new problem that may be of independent interest. Given a positive integer k and a permutation $\pi \in \mathfrak{S}(kn)$, BALANCED k -INCREASING COLORING is the problem of deciding whether there exists a balanced k -coloring of π (i.e., a partition of $[kn]$ into k subsets of size exactly n) such that each color induces an increasing subsequence of π .

► **Proposition 9.** *BALANCED k -INCREASING COLORING for 312-avoiding permutations is $W[1]$ -hard for the parameter k .*

Proof. We perform a reduction from UNARY BIN PACKING parameterized by the number of bins, which is known to be $W[1]$ -hard [20]. In this version of BIN PACKING, we are given a list of integers s_1, s_2, \dots, s_n encoded in unary, and two integers B and k . These integers are interpreted as item sizes, and the task is to decide whether the items can be partitioned



■ **Figure 5** Reduction from UNARY BIN PACKING to BALANCED k -INCREASING COLORING for the list $6, 4, 3, 3, 3, 2, 2, 1$, which admits the partition $(\{6\}, \{4, 2\}, \{3, 3\}, \{3, 2, 1\})$, and the integers $B = 6, k = 4$. Left: dpop P that consists of k chains, each of length $n + B$. Right: 312-avoiding permutation π that consists of n blocks. Each color/shape induces an increasing subsequence of π .

into k subsets, each of total size B . We show that there is a reduction from UNARY BIN PACKING, parameterized by the number of bins, to BALANCED k -INCREASING COLORING, parameterized by the number of colors.

Consider an arbitrary instance of UNARY BIN PACKING containing n items with item sizes $S = \{s_1, s_2, \dots, s_n\}$, and two integers B and k . Define $\pi \in \mathfrak{S}(kB + kn)$ by

$$\pi = \bigoplus_{i=1}^n (\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)).$$

Each pattern $\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)$ is called the i -th block of π . See Figure 5 for an illustration. It is straightforward to check that π is 312-avoiding.

We claim that the n items s_1, s_2, \dots, s_n can be partitioned into k subsets, each of total size B , if and only if there exists a k -coloring of π such that each color induces an increasing pattern of length $B + n$.

Suppose first that the n items s_1, s_2, \dots, s_n can be partitioned into k subsets, each of total size B . Write $S = S_1 \cup S_2 \cup \dots \cup S_k$ such a partition. Define a k -coloring of π as follows. Consider the i -th block $\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)$ of π , and suppose that $s_i \in S_j$. Color the whole ascending pattern $\text{ip}(s_i + 1)$ with color c_j and arbitrarily color the elements of the descending pattern $\text{dp}(k - 1)$ with the remaining $k - 1$ colors (each element of $\text{dp}(k - 1)$ is assigned to a distinct color). We claim that every color c_j induces an increasing pattern of length $B + n$ in π . First, it is clear that the above k -coloring induces increasing patterns only. As for the length of each induced increasing pattern, focus on any color c_j . We note that,

in every block $\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)$ of π , either the whole subpattern $\text{ip}(s_i + 1)$ is colored with color c_j (if $s_i \in S_j$) or exactly one element of the subpattern $\text{dp}(k - 1)$ is colored with color c_j (if $s_i \notin S_j$). It follows that the increasing pattern induced by color c_j in π has length $\sum_{s_i \in S_j} (s_i + 1) + n - |S_i| = \sum_{s_i \in S_j} s_i + |S_i| + n - |S_i| = B + n$.

For the reverse direction, suppose now that there exists a k -coloring of π such that each color induces an increasing pattern of length $B + n$. Every block $\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)$ requires at least k colors, as it contains a decreasing subpattern of length k . Therefore, the whole subpattern $\text{ip}(s_i + 1)$ is colored with the same color. For every $j \leq k$, let S_j be the set of all s_i such that, in the i -th block $\text{ip}(s_i + 1) \ominus \text{dp}(k - 1)$, the subpattern $\text{ip}(s_i + 1)$ is colored with color c_j . We have $B + n = \sum_{s_i \in S_j} (s_i + 1) + n - |S_j| = \sum_{s_i \in S_j} s_i + |S_j| + n - |S_j|$, and hence $\sum_{s_i \in S_j} s_i = B$. Therefore, the n items s_1, s_2, \dots, s_n can be packed into k bins, each of capacity B . ◀

Most of the interest in Proposition 9 stems from the following proposition.

► **Proposition 10.** *DPOP MATCHING for symmetric dpop and 312-avoiding permutations is W[1]-hard for the parameter $\text{width}(P)$.*

Proof. We perform a reduction from BALANCED k -INCREASING COLORING, which is W[1]-hard for the parameter k . Let $\pi \in \mathfrak{S}(kn)$ for some positive integers k and n . We construct a symmetric dpop $P = (\mathcal{P}, \mathcal{P})$, where $\mathcal{P} = (X, \preceq)$, as follows: $X = [k] \times [n]$ and $(i, j) \preceq (i', j')$ if and only if $i = i'$ and $j \leq j'$. We claim that P occurs in π if and only if π admits a k -coloring for which every color induces an increasing pattern of length n .

If π admits such a k -coloring into colors c_1, c_2, \dots, c_k , the function $\varphi: X \rightarrow [kn]$ that maps each pair (i, j) to the j -th smallest position with color c_i is an occurrence of P in π .

Conversely, suppose that some injective function $\varphi: X \rightarrow [kn]$ is an occurrence of P in π . For each $i \leq k$, the set $\{i\} \times [n]$ forms a chain of \preceq , and therefore it is mapped to an increasing pattern of size n . Coloring this pattern in color c_i produces the desired k -coloring. ◀

We show now that the problem where P consists of k independent chains is XP for the parameter k . In fact, we generalize this result to k -weak partial orders (i.e., if P consists of k independent weak orders).

► **Proposition 11.** *DPOP MATCHING for k -weak symmetric dpop is XP with parameter k .*

Proof. Let P be a disjoint union of k weak symmetric dpop P_1, P_2, \dots, P_k . For each dpop P_i , let \preceq_i be a linear extension of P_i , and let $P_{i,1}, P_{i,2}, \dots, P_{i,p_i}$ be the maximal antichains of P_i , ordered by \preceq_i . Finally, for each k -tuple $\mathbf{a} = (a_1, a_2, \dots, a_k)$ of integers such that $a_i \leq |P_i|$, we denote by $P_{\mathbf{a}}$ the dpop obtained from P by removing the a_i \preceq_i -least elements of each dpop P_i , and by $P_{i,\mathbf{a}}^{\min}$ the set of \preceq_i -minimal elements of $P_{\mathbf{a}}$.

Then, given a permutation $\pi \in \mathfrak{S}(n)$, a k -tuple $\mathbf{I} = (I_1, \dots, I_k)$ of intervals of $[n]$, a k -tuple \mathbf{a} and an integer ℓ , a function $\varphi: P_{\mathbf{a}} \rightarrow \{\ell, \ell + 1, \dots, n\}$ is called a *partial matching* for $(\pi, \mathbf{I}, \mathbf{a}, \ell)$ if:

- φ and $\pi \circ \varphi$ are \preceq_i -non-decreasing for each i , and
- for each i , and each element x of $P_{\mathbf{a}}$, $\pi(x) \in I_i$ if and only if $x \in P_{i,\mathbf{a}}^{\min}$.

Before going further, we denote by $\mathbf{1}_i$ the k -tuple with one element 1 (in position i) and $k - 1$ elements 0. We also denote by $<_i$ the partial order on tuples \mathbf{I} of intervals, where $\mathbf{I} <_i \mathbf{I}'$ if $I_j = I'_j$ whenever $j \neq i$ and $x < x'$ whenever $x \in I_i$ and $x' \in I'_i$.

When $a_i = |P_i|$ for all i , such a partial matching exists for all permutations π , tuples of intervals \mathbf{I} and integers ℓ . When $a_i = 0$ for all i and $\ell = 1$, and once π is fixed, such partial matchings coincide with (standard) matchings, and thus we are interested in checking whether a partial matching exists. Finally, for all tuples \mathbf{I} and \mathbf{a} and for all $\ell \leq n$, a partial matching φ for $(\pi, \mathbf{I}, \mathbf{a}, \ell)$ exists precisely when one of the following cases occur:

1. φ is a partial matching for $(\pi, \mathbf{I}, \mathbf{a}, \ell + 1)$, i.e., $\ell \notin \varphi(P_{\mathbf{a}})$;
2. there exists an integer $i \leq k$ for which the \preceq_i -least element of $P_{i, \mathbf{a}}^{\min}$, say x , is such that $\varphi(x) = \ell$ and $\pi(\ell) \in I_i$, and either
 - x is not the only element of $P_{i, \mathbf{a}}^{\min}$, and φ is a partial matching for $(\pi, \mathbf{I}, \mathbf{a} + \mathbf{1}_i, \ell + 1)$, or
 - x is the only element of $P_{i, \mathbf{a}}^{\min}$ and there exists a tuple $\mathbf{I}' >_i \mathbf{I}$ such that φ is a partial matching for $(\pi, \mathbf{I}', \mathbf{a} + \mathbf{1}_i, \ell + 1)$.

Consequently, we can compute by dynamic programming the list of triples $(\mathbf{I}, \mathbf{a}, \ell)$ such that there exists a partial matching for $(\pi, \mathbf{I}, \mathbf{a}, \ell)$: deciding whether adding a triple $(\mathbf{I}, \mathbf{a}, \ell)$ to the list simply requires to check which triples of the form $(\mathbf{I}', \mathbf{a}', \ell + 1)$ already belong to the list. Since there are less than n^{3k+1} triples, this provides us with a $\tilde{O}(n^{6k+2})$ algorithm. ◀

5.2 Symmetric Pattern and Pattern-Avoiding π

In this final section, we consider restrictions on the *shape* of π , via pattern-avoiding restrictions. Our goal here is to identify tractable cases among classes of permutations avoiding one or more size-3 patterns. We give an almost complete dichotomy of polynomial/NP-hard cases, as shown in Table 1. Hardness results are proven in Proposition 12, and also apply to height-2 dpops. Polynomial cases are proven in Proposition 13 and apply to dpops of any height.

► **Proposition 12.** *DPOP MATCHING for height-2 symmetric dpop P and permutation π is NP-hard even if π is separable (it avoids 2413 and 3142) and one of the following restrictions occurs:*

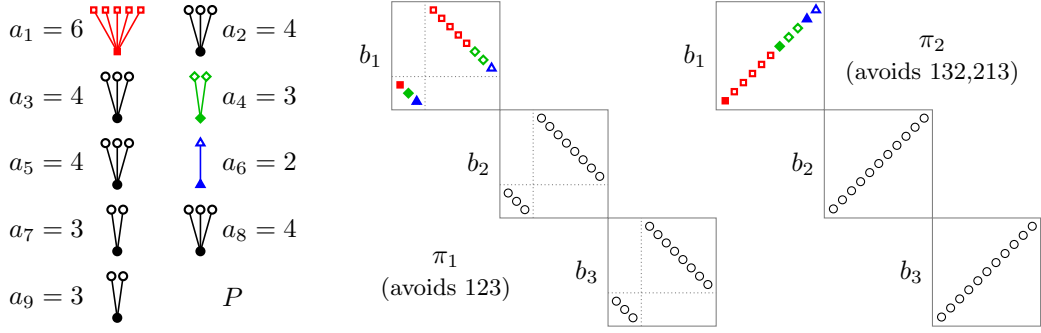
1. π is 123-avoiding;
2. π is (132, 213)-avoiding;
3. π is (132, 321)-avoiding;
4. π is (231, 312)-avoiding;
5. π is (132, 312)-avoiding;
6. π is (213, 321)-avoiding;
7. π is (213, 312)-avoiding;
8. π is (132, 231)-avoiding;
9. π is (213, 231)-avoiding.

Proof. In each of the cases presented below, we define a symmetric dpop $P = (\mathcal{P}, \mathcal{P})$ for some partially ordered set $\mathcal{P} = (X, \preceq)$. Each time, we identify P with the partial order \preceq .

■ **Table 1** Polynomial (green/light) and NP-hard (red/dark) cases for DPOP MATCHING with symmetric dpop and pattern-avoiding permutation π , for combinations of size-3 avoided patterns. For each case, see the referenced proposition and case for more details. Diagonal cases follow from any other hard case in the same row or column. For hard cases, the problem used for reduction is indicated as follows: BIC: Biclique, 3P: 3-PARTITION, BIN: UNARY BIN PACKING, BIS: BISECTION.

π avoids ... and ...	123	132	213	231	312	321
123	12.1 3P					
132	13.2					
213	13.5	12.2 BIN				
231	13.1	12.8 BIC	12.9 BIC			
312	13.4	12.5 BIC	12.7 BIC	12.4 BIC		
321	13.3	12.3 BIS	12.6 BIS	(open)		

NP-hard cases



■ **Figure 6** Reductions from 3-PARTITION and UNARY BIN PACKING to DPOP MATCHING on 123-avoiding and (132, 213)-avoiding permutations. Left: the height-2 dpop P used in both reductions. Right: the 123-avoiding and (132, 213)-avoiding permutations used in each reduction. The mapping of three subsets of P corresponding to a first bin gadget is highlighted in each figure.

Case 1: π is 123-avoiding and separable. We use a reduction from 3-PARTITION, as illustrated in Figure 6 with permutation π_1 . Let (A, B) be an instance of 3-PARTITION, where A is a list of integers a_1, a_2, \dots, a_{3n} with sum nB , all being larger than 1.

For all $p \leq n$, we define a *bin gadget* b_p as the permutation $\text{dp}(3) \oplus \text{dp}(B-3)$: we see this gadget as consisting of two parts. Our permutation π is now defined by $\pi = \bigoplus_{p=1}^n b_p$. Then, our partial order \preceq is defined on a set X of nB elements, noted $x_i, y_{i,2}, \dots, y_{i,a_i}$ for each $i \leq 3n$, so that $x_i \preceq y_{i,j}$ for all i and j .

If P has an occurrence $\varphi: X \rightarrow [nB]$ in π , this occurrence is bijective. Moreover, each element x_i is sent to the bottom-left of $y_{i,2}$, and thus it must be mapped to the left part of some gadget, say $b_{f(i)}$. Each element $y_{i,j}$ must then be mapped to the right part of the same gadget. Now, for each $p \leq n$, the set $S_p = \{i: f(i) = p\}$ is of size 3, and exactly B elements of X are mapped to the gadget b_p , which means that $\sum_{i \in S_p} a_i = B$. Moreover, $S_1 \cup \dots \cup S_n$ forms a partition of $[3n]$, hence it yields a 3-partition of A .

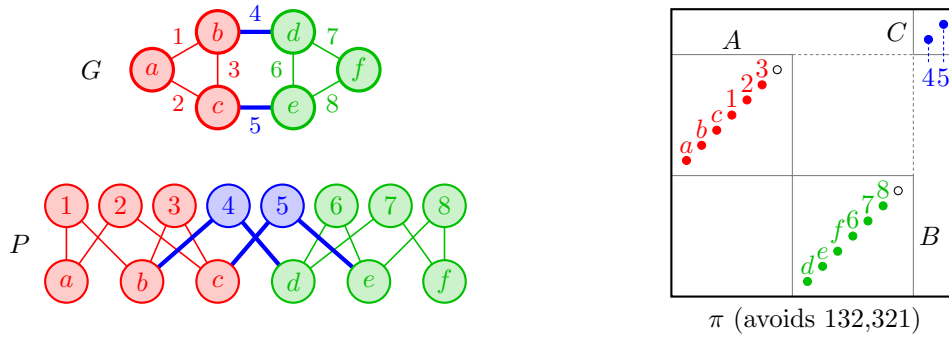
Conversely, given a partition $S_1 \cup \dots \cup S_n$ of $[3n]$ such that $|S_p| = 3$ and $\sum_{i \in S_p} a_i = B$ for each p , we build an occurrence of P in π by mapping the three elements x_i (for $i \in S_p$) to the left part of b_p , and the $B-3$ elements $y_{i,j}$ (for $i \in X_p$) to the right part of b_p .

Case 2: π is (132, 213)-avoiding. We use a reduction from UNARY BIN PACKING, as illustrated in Figure 6 with permutation π_2 . Given an instance (A, B, k) of UNARY BIN PACKING, where A is a list of integers a_1, a_2, \dots, a_n larger than 1, we use the same dpop as in Case 1: our partial order \preceq is defined on a set X of nB elements, noted $x_i, y_{i,2}, \dots, y_{i,a_i}$ for each $i \leq n$, so that $x_i \preceq y_{i,j}$ for all i and j . However, this time, our gadget b_p is the permutation $\text{ip}(B)$, and our permutation π is again defined by $\pi = \bigoplus_{p=1}^n b_p$.

If P has an occurrence $\varphi: X \rightarrow [nB]$ in π , this occurrence is bijective. Each element x_i is sent to some gadget, say $b_{f(i)}$, and the elements $y_{i,j}$ must then be mapped to the same gadget. Now, for each $p \leq k$, let $S_p = \{i: f(i) = p\}$. Exactly B elements of X are mapped to the gadget b_p , which means that $\sum_{i \in S_p} a_i = B$. This means that (S, B, k) is a positive instance of the UNARY BIN PACKING problem.

Conversely, given a partition $S_1 \cup \dots \cup S_k$ of $[n]$ such that $\sum_{i \in S_p} a_i = B$ for each i , we build an occurrence of P in π by mapping the B elements x_i and $y_{i,j}$ (for $i \in S_p$) to b_p .

Case 3: π is (132, 321)-avoiding. We use a reduction from BISECTION, as illustrated in Figure 7. Given a graph $G = (V, E)$ and an integer k , the BISECTION problem consists in deciding whether V admits a partition $V_1 \cup V_2$ such that $|V_1| = |V_2|$ and that *splits* at most k edges (i.e., at most k edges have one endpoint in V_1 and one endpoint in V_2).



■ **Figure 7** Simplified version (for $W = 1$ and $L = 7$) of the reduction from BISECTION to DPOP MATCHING on (132, 321)-avoiding permutations. Top left: a size-6 graph with a bisection $(\{a, b, c\}, \{d, e, f\})$ that splits $k = 2$ edges. Bottom left: height-2 dpop P (note that, in general, each element from the bottom line should appear W times and not just once). Right: permutation π ; elements of P are mapped to A, B or C depending on whether they are colored in red, green or blue.

Our reduction is as follows. Let $n = |V|/2$, $m = |E|$, $W = m + k + 1$, and $L = nW + m$. Our permutation is defined by $\pi = (\text{ip}(L) \ominus \text{ip}(L)) \oplus \text{ip}(k)$. These three parts of π are noted A, B and C , from left to right. Then, our partial order \preceq is defined on a set X of $2nW + m$ elements: $2nW$ elements, noted $x_{v,i}$ for each $v \in V$ and $i \leq W$, and m elements, noted y_e for each $e \in E$. This order contains the relations $x_{v,i} \preceq y_e$ for which v is an endpoint of e .

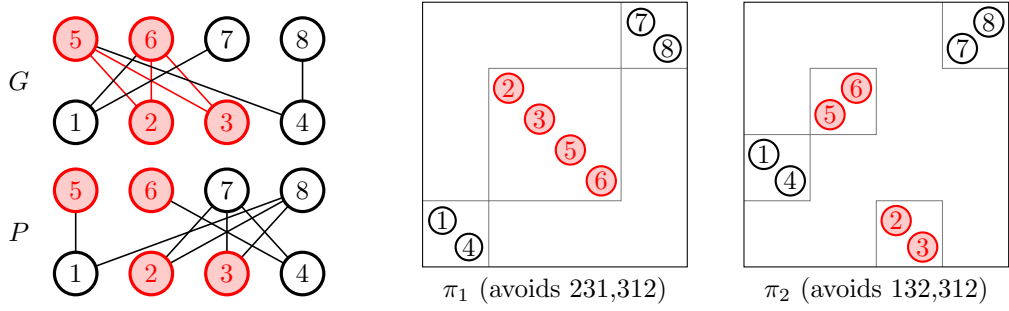
Assume that there exists a mapping of P into π . For each $v \in V$, and since C has size $k < W$, at least one of the elements $x_{v,i}$ is mapped to A , in which case we say that v has type A , or to B , in which case v has type B . Then, each vertex has at least one type, and possibly both. We partition V into three sets V_A, V_B, V_{AB} containing the vertices of type A, B and both A and B , respectively. Moreover, for each $v \in V_A$, each element $x_{v,i}$ must be mapped either to A or to C : these two parts together contain $L + k$ elements, so $|V_A| \leq (L + k)/W = n + 1 - 1/W$, and $|V_A| \leq n$. Similarly, $|V_B| \leq n$.

We build a set V_1 as the union of V_A with $n - |V_A|$ vertices of V_{AB} , and V_2 as $V \setminus V_1$, so that $|V_1| = |V_2| = n$. Moreover, for every $v \in V_1$ (resp., $v \in V_2$), some element $x_{v,i}$, say $x_{v,1}$, is mapped to A (resp., to B). Then, each edge $e = (u, v)$ that is split by (V_1, V_2) must be mapped to a point above some point of A and to the right of some point of B . This means that y_e is mapped to C , and that (V_1, V_2) splits at most k edges, i.e., is a valid bisection.

Conversely, given a bisection (V_1, V_2) splitting at most k edges, we map P into π as follows: map elements $x_{v,i}$ for $v \in V_1$ (resp., V_2) to the first nW elements of A (resp., B), map elements y_e for which e is induced by V_1 (resp., V_2) to the following elements of A (resp. B), and finally map all elements y_e such that e is split by (V_1, V_2) into C . This mapping is an occurrence of P in π .

Case 4: π is (231, 312)-avoiding. We use a reduction from BICLIQUE, as illustrated in Figure 8 with permutation π_1 . Given a bipartite graph $G = (V, E)$ and an integer k , the BICLIQUE problem consists in deciding whether V admits a complete bipartite subgraph $K_{k,k}$. If $V = A \cup B$ is a partition of V into two independent sets, adding independent vertices if needed allows us to assume that A and B have the same size n , and that no vertex in either side is fully connected to the other side.

Our permutation π is defined by $\pi = \text{dp}(n - k) \oplus \text{dp}(2k) \oplus \text{dp}(n - k)$. These three parts of π are noted b_1, b_2 and b_3 . Our partial order \preceq is the order on V such that $x \preceq y$ whenever $x \in A, y \in B$ and $\{x, y\} \notin E$.



■ **Figure 8** Reduction from BICLIQUE to DPOP MATCHING on $(231, 312)$ -avoiding and $(213, 231)$ -avoiding permutations. Left: a bipartite graph G with a $(2, 2)$ biclique and the corresponding height-2 dpop P , built from the complement of G . Right: permutations π_1 and π_2 with a mapping of the vertices 1 to 8, including the biclique vertices mapped into the central $2k$ positions.

Consider a mapping of P into π . For each element $x \in A$, there exists $y \in B$ such that $x \preceq y$, and therefore x cannot be mapped into b_3 . Symmetrically, no element $y \in B$ may be mapped into b_1 . Overall, since $|\pi| = |V| = 2n$, b_1 contains $n - k$ elements from A , b_3 contains $n - k$ elements from B , and b_2 contains a size- k subset A' of A and a size- k subset B' of B . No two elements $x \in A'$ and $y \in B'$ that are mapped into b_2 are comparable for \preceq , which means that $\{x, y\} \in E$ for each such pair, i.e., that (A', B') is a biclique.

Conversely, if G has a biclique (A', B') , we map all elements of $A \setminus A'$ into b_1 , all elements of $A' \cup B'$ into b_2 , and all elements of $B \setminus B'$ into b_3 . This mapping satisfies all relations $x \preceq y$ with $x \in A$ and $y \in B$, except for $x \in A'$ and $y \in B'$, but indeed there is no such relation since (A', B') is a biclique.

Case 5: π is $(132, 312)$ -avoiding. We also use a reduction from BICLIQUE, as illustrated in Figure 6 with permutation π_2 . Our partial order \preceq is the same as in Case 4, and our permutation π is defined by $\pi = ((\text{dp}(n - k) \oplus \text{ip}(k)) \ominus \text{dp}(k)) \oplus \text{ip}(n - k)$. These three parts of π are noted b_1 , b_2 , b_3 and b_4 .

Consider a mapping of P into π . For each element $y \in B$, there exists $x \in A$ such that $x \preceq y$, and therefore y cannot be mapped into b_1 or b_3 . Thus, and since $|B| = n$, the elements of B are mapped to b_2 or b_4 , and the elements of A are mapped to b_1 or b_3 . Hence, b_2 contains a size- k subset B' of B and b_3 contains a size- k subset A' of A . No element of b_2 is comparable to any element of b_3 , and therefore (A', B') is a biclique.

Conversely, if G has a biclique (A', B') , we map all elements of $A \setminus A'$ into b_1 , all elements of A' into b_3 , all elements of B' into b_2 and all elements of $B \setminus B'$ into b_4 . This mapping satisfies all relations $x \preceq y$ with $x \in A$ and $y \in B$, except for $x \in A'$ and $y \in B'$, but indeed there is no such relation since (A', B') is a biclique.

Cases 6–9: These cases are symmetric to Cases 3, 5, 5 and 7, respectively. Indeed, if (P, π) is an instance of DPOP MATCHING with P a height-2 symmetric dpop, $(P^\partial, \pi^{\text{cr}})$ and (P, π^{-1}) are equivalent instances of DPOP MATCHING with height-2 symmetric dpops, and

Case 6: if π avoids 132 and 321 (Case 3), π^{cr} avoids $132^{\text{cr}} = 213$ and $321^{\text{cr}} = 321$;

Case 7: if π avoids 132 and 312 (Case 5), π^{cr} avoids $132^{\text{cr}} = 213$ and $312^{\text{cr}} = 312$;

Case 8: if π avoids 132 and 312 (Case 5), π^{-1} avoids $132^{-1} = 132$ and $312^{-1} = 231$;

Case 9: if π avoids 213 and 312 (Case 7), π^{-1} avoids $213^{-1} = 213$ and $312^{-1} = 231$. ◀

► **Proposition 13.** *DPOP MATCHING is in P for symmetric dpop P if one of the following restrictions on π occurs:*

1. π is (123, 231)-avoiding;
2. π is (123, 132)-avoiding;
3. π is (123, 321)-avoiding;
4. π is (123, 312)-avoiding;
5. π is (123, 213)-avoiding.

Proof. In each of the cases presented below, we are given a permutation π and a symmetric dpop $P = (\mathcal{P}, \mathcal{P})$ for some partially ordered set $\mathcal{P} = (X, \preceq)$. Each time, we identify P with the partial order \preceq .

Case 1: π is (123, 231)-avoiding. There exist integers k, ℓ and m , with sum n , such that $\pi = \text{dp}(k) \ominus (\text{dp}(\ell) \oplus \text{dp}(m))$. These three parts of π are noted b_1, b_2 and b_3 . Then, for every pair (u, v) such that $u \prec v$, we must map u into b_2 and v into b_3 . Such values can be mapped greedily, since elements in b_2 are pairwise incompatible, as well as those in b_3 . Thus, P can be mapped into π if and only if it has height at most 2, there are at most a elements that are lower bounds, and at most b elements that are upper bounds.

Note that, if P is not symmetric, the problem becomes NP-hard, since reversing the horizontal order of $P_{\mathcal{P}}$ and π transforms π into the (132, 321)-avoiding permutation of the NP-hard Case 3 in Proposition 12.

Case 2: π is (123, 132)-avoiding. The permutation π is a skew sum $\pi = \ominus_{p=1}^k d_p$ of patterns of the form $d_p = \text{dp}(a_p) \oplus \text{dp}(1)$ for some integer $a_p \geq 0$. Then, no two elements in X can share a strict lower bound, i.e., if $u \prec v$ and $u \prec w$ then $v = w$. Thus, P is of height at most 2, and there exists a partition $S_1 \cup \dots \cup S_\ell$ of X in which each set S_i contains a distinguished element s_i , such that $x \preceq s_i$ if and only if $x \in S_i$. Up to reordering the patterns d_p and the sets S_i , which are pairwise incomparable, we assume that $a_1 \geq a_2 \geq \dots \geq a_k$ and that $|S_1| \geq |S_2| \geq \dots \geq |S_\ell|$. Let also m be the number of sets S_i with size at least 2.

Each set S_i must be mapped into a single pattern, say $d_{p(i)}$, and if $i \leq m$, i.e., if $|S_i| \geq 2$, the element s_i must be mapped to the unique top-right element of $d_{p(i)}$. Such a mapping exists if and only if $k \geq m$ and $a_i \geq |S_i| - 1$ for all $i \leq m$: we shall choose $p(i) = i$ and map greedily the elements of $S_i \setminus \{s_i\}$ to the bottom-left part of d_i . Finally, the elements of singleton sets S_i can be mapped to the remaining places in π .

Case 3: π is (123, 321)-avoiding. Erdős-Szekeres theorem [14] proves that $n \leq 4$.

Cases 4–5: These cases are symmetric to Cases 1 and 2, respectively. Indeed, if (P, π) is an instance of DPOP MATCHING with P a height-2 symmetric dpop, $(P^\partial, \pi^{\text{cr}})$ and (P, π^{-1}) are equivalent instances of DPOP MATCHING with height-2 symmetric dpops, and

Case 4: if π avoids 123 and 231 (Case 1), π^{-1} avoids $123^{-1} = 123$ and $231^{-1} = 312$;

Case 5: if π avoids 123 and 132 (Case 2), π^{cr} avoids $123^{\text{cr}} = 123$ and $132^{\text{cr}} = 213$. ◀

6 Concluding Remarks

Some open complexity questions remain among the parameters we identified for DPOP MATCHING. For semi-total dpops, the complexity is open for constant width, and for most classes of pattern-avoiding permutations (although, according to Propositions 7 and 10, the problem is NP-hard when π avoids 1234 or 312, respectively). For symmetric dpops, it would be interesting to settle the complexity status of deciding whether a dpop occurs in a (231, 321)-avoiding or (312, 321)-avoiding permutation. In particular, for these cases, we conjecture that the problem becomes polynomial when $\text{height}(P)$ is constant.

Regarding the original puzzle formulation of the problem, an interesting question is to generate instances that yield a unique solution, i.e., given a permutation π , find a dpop with a unique occurrence in π . This can be done by using a semi-total dpop (e.g., take X

with $|X| = |\pi|$, let P_P be a total order and P_V be an empty order), but one could try to minimize $|X|$ or the number of pairs of comparable elements in P (i.e., the number of clues) in order to have a unique solution.

References

- 1 Shlomo Ahal and Yuri Rabinovich. On complexity of the subpattern problem. *SIAM J. Discret. Math.*, 22(2):629–649, 2008.
- 2 Michael H. Albert, Marie-Louise Lackner, Martin Lackner, and Vincent Vatter. The complexity of pattern matching for 321-avoiding and skew-merged permutations. *Discret. Math. Theor. Comput. Sci.*, 18(2), 2016.
- 3 Sergey V. Avgunstovich, Sergey Kitaev, and Alexandr Valyuzhenich. Avoidance of boxed mesh patterns on permutations. *Discret. Appl. Math.*, 161(1-2):43–51, 2013.
- 4 Eric Babson and Einar Steingrímsson. Generalized permutation patterns and a classification of the mahonian statistics. *Séminaire Lotharingien de Combinatoire [electronic only]*, 44:B44b, 18 p.–B44b, 18 p., 2000. URL: <http://eudml.org/doc/120841>.
- 5 Benjamin Aram Berendsohn, László Kozma, and Dániel Marx. Finding and counting permutations via csps. *Algorithmica*, 83(8):2552–2577, 2021.
- 6 Miklós Bóna. *Combinatorics of Permutations, Second Edition*. Discrete mathematics and its applications. CRC Press, 2012.
- 7 Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998.
- 8 Mireille Bousquet-Mélou, Anders Claesson, Mark Dukes, and Sergey Kitaev. (2+2)-free posets, ascent sequences and pattern avoiding permutations. *J. Comb. Theory, Ser. A*, 117(7):884–909, 2010.
- 9 Petter Brändén and Anders Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *Electron. J. Comb.*, 18(2), 2011.
- 10 Marie-Louise Bruner and Martin Lackner. A fast algorithm for permutation pattern matching based on alternating runs. *Algorithmica*, 75(1):84–117, 2016.
- 11 Laurent Bulteau, Romeo Rizzi, and Stéphane Vialette. Pattern matching for k-track permutations. In Costas S. Iliopoulos, Hon Wai Leong, and Wing-Kin Sung, editors, *Combinatorial Algorithms - 29th International Workshop, IWOCA 2018, Singapore, July 16-19, 2018, Proceedings*, volume 10979 of *Lecture Notes in Computer Science*, pages 102–114. Springer, 2018.
- 12 Maxime Crochemore and Ely Porat. Fast computation of a longest increasing subsequence and application. *Inf. Comput.*, 208(9):1054–1059, 2010.
- 13 Sergi Elizalde and Marc Noy. Consecutive patterns in permutations. *Adv. Appl. Math.*, 30(1-2):110–125, 2003.
- 14 Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio mathematica*, 2:463–470, 1935.
- 15 Jacob Fox. Stanley-wilf limits are typically exponential. *CoRR*, abs/1310.8378, 2013. [arXiv:1310.8378](https://arxiv.org/abs/1310.8378).
- 16 Paweł Gawrychowski and Mateusz Rzepecki. Faster exponential algorithm for permutation pattern matching. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 279–284. SIAM, 2022.
- 17 Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 82–101. SIAM, 2014.
- 18 Sylvain Guillemot and Stéphane Vialette. Pattern matching for 321-avoiding permutations. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 1064–1073. Springer, 2009.

- 19 Louis Ibarra. Finding pattern matchings for permutations. *Inf. Process. Lett.*, 61(6):293–295, 1997.
- 20 Klaus Jansen, Stefan Kratsch, Dániel Marx, and Ildikó Schlotter. Bin packing with fixed number of bins revisited. *J. Comput. Syst. Sci.*, 79(1):39–49, 2013.
- 21 Vít Jelínek and Jan Kynčl. Hardness of permutation pattern matching. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 378–396. SIAM, 2017.
- 22 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.
- 23 Sergey Kitaev. Introduction to partially ordered patterns. *Discret. Appl. Math.*, 155(8):929–944, 2007.
- 24 Sergey Kitaev. *Patterns in Permutations and Words*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2011.
- 25 Both Emerite Neou, Romeo Rizzi, and Stéphane Vialette. Pattern matching for separable permutations. 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 260–272, 2016.
- 26 Both Emerite Neou, Romeo Rizzi, and Stéphane Vialette. Permutation pattern matching in (213, 231)-avoiding permutations. *Discret. Math. Theor. Comput. Sci.*, 18(2), 2016.



Linear-Time Computation of Shortest Covers of All Rotations of a String

Maxime Crochemore  

King's College London, UK
Université Gustave Eiffel,
Marne-la-Vallée, France

Costas S. Iliopoulos  

King's College London, UK

Jakub Radoszewski  

University of Warsaw, Poland

Wojciech Rytter  


University of Warsaw, Poland

Juliusz Straszypiński  

University of Warsaw, Poland

Tomasz Waleń  

University of Warsaw, Poland

Wiktor Zuba  

University of Warsaw, Poland
CWI, Amsterdam, The Netherlands

Abstract

We show that lengths of shortest covers of all rotations of a length- n string over an integer alphabet can be computed in $\mathcal{O}(n)$ time in the word-RAM model, thus improving an $\mathcal{O}(n \log n)$ -time algorithm from Crochemore et al. (*Theor. Comput. Sci.*, 2021). Similarly as Crochemore et al., we use a relation of covers of rotations of a string S to seeds and squares in S^3 . The crucial parameter of a string S is the number $\xi(S)$ of primitive covers of all rotations of S . We show first that the time complexity of the algorithm from Crochemore et al. can be slightly improved which results in time complexity $\Theta(\xi(S))$. However, we also show that in the worst case $\xi(S)$ is $\Omega(|S| \log |S|)$. This is the main difficulty in obtaining a linear time algorithm. We overcome it and obtain yet another application of runs in strings.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases cover, quasiperiod, cyclic rotation, seed, run

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.22

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Juliusz Straszypiński: 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

A string C is a *cover* of a string S if each position in S is inside at least one occurrence of C in S . We say that a string Y is a *rotation* of a string X if $X = AB$ and $Y = BA$ for some strings A and B ; in this case we write $Y = \text{rot}_{|A|}(X)$, where $|A|$ is the length of string A .

Let us denote by $\text{CC}[i]$ the length of the shortest cover of $\text{rot}_i(S)$, where S is an input string. We consider the following problem.

COVERS OF ALL ROTATIONS

Input: A length- n string S .

Output: The array $\text{CC}[0..n-1]$.



© Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszypiński, Tomasz Waleń, and Wiktor Zuba;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 22; pp. 22:1–22:15



Leibniz International Proceedings in Informatics

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

22:2 Linear-Time Computation of Shortest Covers of All Rotations of a String

Let Fib_n denote the n -th Fibonacci string ($Fib_0 = b$, $Fib_1 = a$, $Fib_n = Fib_{n-1}Fib_{n-2}$).

► **Example 1.** For the Fibonacci string $S = Fib_6$ we have

$$CC = [5, 5, 13, 3, 13, 5, 5, 13, 3, 8, 8, 3, 13]$$

The following table gives shortest covers for consecutive rotations $rot_0(S), \dots, rot_{12}(S)$.

i	rotation $rot_i(S)$	shortest cover	length $CC[i]$
0	abaababaabaab	abaab	5
1	baababaabaaba	baaba	5
2	aababaabaabab	aababaabaabab	13
3	ababaabaababa	aba	3
4	babaabaababaa	babaabaababaa	13
5	abaabaababaab	abaab	5
6	baabaababaaba	baaba	5
7	aabaababaabab	aabaababaabab	13
8	abaababaababa	aba	3
9	baababaababaa	baababaa	8
10	aababaababaab	aababaab	8
11	ababaababaaba	aba	3
12	babaabaababaa	babaabaababaa	13

Covers in strings are an extensively studied notion in stringology; algorithms computing covers in a string were proposed in [1, 4, 19, 18], not to mention approximate and generalized variants of covers (for a recent survey, see [10]). In [8] the authors showed an $\mathcal{O}(n \log n)$ -time (and $\mathcal{O}(n)$ space) algorithm that computes the lengths of shortest covers of all rotations of a length- n string. Later in [7] the authors developed a data structure over a length- n string that allows to answer queries about lengths of shortest covers of factors of the string. If combined with a data structure answering Weighted Ancestor Queries in the suffix tree in constant time after linear-time preprocessing, proposed in a very recent result [3] or in an off-line setting in [15], the data structure of [7] requires $\mathcal{O}(n \log n)$ -time and space preprocessing and allows to answer shortest cover queries for factors in $\mathcal{O}(\log n)$ time (amortized in the case that off-line Weighted Ancestor Queries are used). Hence, this result also yields an $\mathcal{O}(n \log n)$ -time algorithm for shortest covers of all rotations of a length- n string S if applied for all length- n factors of S^2 , despite being far more general. This suggests that perhaps shortest covers of all rotations can be computed in $o(n \log n)$ time. In comparison, shortest covers of all prefixes of a string can be computed in linear time using the on-line algorithm for computing shortest covers by Breslauer [4] (regardless of the alphabet size).

We show that this supposition is right by developing a linear-time algorithm computing shortest covers of all rotations of a given string.

Our algorithm works on a word-RAM model with word size $w = \Omega(\log n)$. We assume that the input string is over a so-called integer alphabet $[0..n^{\mathcal{O}(1)}]$, where n is the length of the string, which is a common assumption in the field (see, e.g., [11]).

Our approach. We use an approach based on runs and on packed representations of sets. We say that a string C is a seed of a string S if C is a cover of a superstring of S . In [8] it is shown that a string C is a cover of a rotation of a string S , $|C| \leq |S|$, if and only if C^2 is a factor of S^3 and C is a seed of S^3 . Each run in S^3 represents in a natural way the set of occurrences of primitively-rooted squares and we need to extract from these sets the occurrences of squares which are also seeds.

2 Preliminaries and algorithmic toolbox

We consider strings over an integer alphabet. Letters of a string S are numbered 0 through $|S| - 1$, with $S[i]$ being the i th letter. A factor of S is a string $S[i] \dots S[j]$, for any $0 \leq i \leq j < |S|$; it is denoted as $S[i..j]$ or $S[i..j+1)$. If $i > j$, we assume that $S[i..j]$ is the empty string. Throughout the paper, factors of S are represented in $\mathcal{O}(1)$ space by specifying the indices i, j of any of their occurrences $S[i..j]$. By $Occ(U, S)$ we denote the set of starting positions of occurrences of U in S . If $U = rot_i(V)$, we say that U and V are cyclically equivalent.

A string S has a period $p > 0$ if $S[i] = S[i + p]$ for all $i \in [0..|S| - p - 1]$. The smallest period of S is denoted as $\text{per}(S)$. A string S is called periodic if $2 \cdot \text{per}(S) \leq |S|$, and aperiodic otherwise. A string S is called primitive if $S = V^k$ for a positive integer k implies that $k = 1$.

2.1 Suffix tree

Let $ST(S)$ denote the suffix tree of string S . It can be constructed in linear time for a string over an integer alphabet [11].

The locus of a factor U of S is an (explicit or implicit) node of $ST(S)$ such that the path from the root to this node has string label U . An implicit node is represented by its nearest explicit descendant and its distance to the descendant. The string depth of an (explicit or implicit) node v is the length of the string label of v .

We use *Weighted Ancestor Queries* on a suffix tree. Such queries, given an explicit node v and an integer value ℓ that does not exceed the string depth of v , ask for the highest explicit ancestor u of v with string depth at least ℓ . We use the following very recent result.

► **Lemma 2** ([3]). *Let $ST(S)$ be the suffix tree of S . *Weighted Ancestor Queries* on $ST(S)$ can be answered in $\mathcal{O}(1)$ time after linear-time preprocessing.*

► **Corollary 3.** *After $\mathcal{O}(n)$ time preprocessing, the locus of any factor of a length- n string S can be computed in $\mathcal{O}(1)$ time.*

A simpler off-line version of *Weighted Ancestor Queries*, that would be sufficient for our purposes, with the same time guarantees was proposed earlier in [15]. We also use the following application of the queries.

► **Lemma 4.** *Any $\mathcal{O}(n)$ factors of a length- n string S can be ordered lexicographically in $\mathcal{O}(n)$ time.*

Proof. Assume that the explicit nodes of $ST(S)$ are numbered in pre-order. Let the locus of a factor be a pair (v, d) where v is the number of the explicit descendant and d is the distance; $d = 0$ for an explicit locus. Then it suffices to use Radix Sort to order the loci of the factors by non-decreasing first components, and by non-increasing second components in case of a tie. ◀

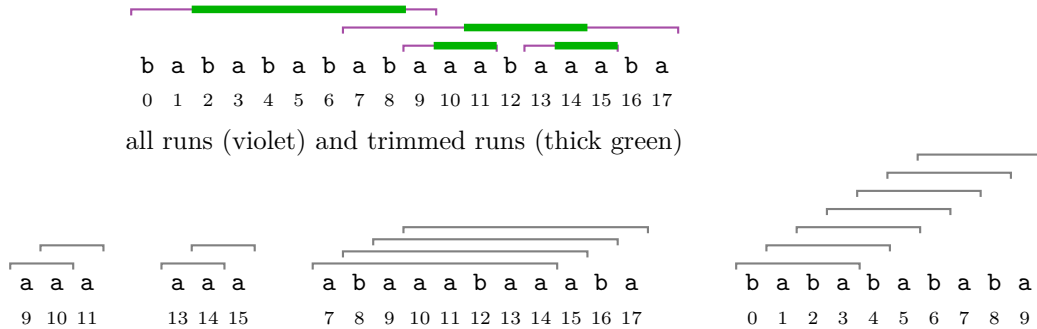
2.2 Runs, trimmed runs and p-squares

A string of the form $U^2 = UU$ is called a square. A square U^2 is called a p-square if U is primitive. We denote by $\frac{1}{2}\text{PSquares}(S)$ (p-square halves) the set of primitive factors Z of S such that the square Z^2 is also a factor of S .

The maximum number of occurrences of p-squares in a string of length n is $\Theta(n \log n)$ [9], whereas the total number of distinct square factors (hence, of distinct p-square factors) in a string is $\mathcal{O}(n)$ [13].

A *run* (also known as a *maximal repetition*) in S is a periodic fragment $R = S[i..j]$ which can be extended neither to the left nor to the right without increasing the period $p = \text{per}(R)$, i.e. if $i > 0$ then $S[i - 1] \neq S[i + p - 1]$ and if $j < |S| - 1$ then $S[j + 1] \neq S[j - p + 1]$. The exponent of a run R , denoted as $\text{exp}(R)$, is defined as $(j - i + 1)/p$. Let $\mathcal{R}(S)$ denote the set of all runs of string S . For a length- n string S it holds that $|\mathcal{R}(S)| = \mathcal{O}(n)$; moreover, the sum of exponents of runs in S is $\mathcal{O}(n)$ [17].

The center of an occurrence $S[a..a + 2\ell)$ of a square U^2 is defined as the position $a + \ell$. A square occurrence $S[a..b]$ is said to be *induced* by a run $R = S[i..j]$ if $i \leq a, b \leq j$ and $\text{per}(R) = \text{per}(U)$. Every square is induced by exactly one run [6]. A run $R = S[i..j]$ with period p induces p -squares with centers at positions in $[i + p..j - p + 1]$; the length of this interval is $|R| - 2 \cdot \text{per}(R) + 1$.



■ **Figure 1** Four runs (presented at the top) generate all p -squares (bottom). The total length of trimmed runs is the same as the number of occurrences of p -square factors of S .

For a run $R = S[i..j]$ with period p , we define a *trimmed run* as $S[i + p..j - p + 1]$. We assume that the trimmed run stores the period of the original run. We denote by $\mathcal{R}'(S)$ the set of trimmed runs of S . The following lemma was already shown in [5]; here we give a more direct proof in terms of p -squares.

► **Lemma 5.** *For any string S of length n we have that $\sum_{R \in \mathcal{R}'(S)} |R| = \mathcal{O}(n \log n)$.*

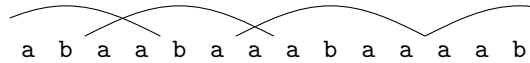
Proof. The run with period p corresponding to a trimmed run R induces $|R|$ occurrences of p -squares with half length p , whose centers are positions of R , and vice versa. Now the thesis follows directly from the fact that the number of occurrences of primitively rooted squares is $\mathcal{O}(n \log n)$. ◀

The *Lyndon root* of a run R is the lexicographically smallest rotation of its length- $\text{per}(R)$ prefix. If L is the Lyndon root of a run R , then R can be uniquely represented as (L, y, a, b) for $0 \leq a, b < |L|$ such that $R = L[|L| - a..|L| - 1]L^yL[0..b]$; we call this the *Lyndon representation* of R . One can group all runs in S by Lyndon roots and compute the Lyndon representations of all runs in $\mathcal{O}(n)$ time; see [6].

The *Lyndon type* of a primitive string U , denoted as $\text{LynType}(U)$, is the lexicographically smallest rotation of U .

2.3 Relation to seeds

A factor C of a string S is called a *seed* of S if there exists a string S' having S as a factor such that C is a cover of S' . In other words, C is a seed of S if S is covered by occurrences and left and right overhangs of C ; see Figure 2.



■ **Figure 2** A string with a seed aaba.

► **Lemma 6** ([15]). *All the seeds of a string of length n can be represented in $\mathcal{O}(n)$ space as a collection of a linear number of disjoint paths in the suffix tree of the string. This representation can be computed in $\mathcal{O}(n)$ time.*

Weighted Ancestor Queries together with a representation of all the seeds of a string from Lemma 6 can be used to show the following lemma.

► **Lemma 7** ([8, see Lemma 8]). *Let S be a string of length n . Given a family \mathcal{U} of $\mathcal{O}(n)$ factors of S , all strings in \mathcal{U} that are seeds of S can be reported in $\mathcal{O}(n)$ time.*

By $\mathbf{PrimCov}[i]$ we denote the set of lengths of covers of $rot_i(S)$ that are primitive strings. Let us observe that each of these sets is non-empty.

► **Observation 8.** $\mathbf{CC}[i] = \min \mathbf{PrimCov}[i]$.

Proof. Every string U has a cover, possibly equal to U . The shortest cover C of every string U is primitive. Otherwise, if we had $C = D^k$ for integer $k > 1$, then D would be a shorter cover of U . ◀

We further denote by $\mathbf{Seeds}(S)$ the set of factors which are seeds of S . The following lemma uses these sets for the string $X := S^3$ in order to characterize covers of all rotations of S . Denote by $\mathbf{Centers}(C^2, X)$ the centers of all occurrences of C^2 in X .

► **Lemma 9** ([8, Lemma 3]). [**Covers = Seeds+Squares**]

Let S be a string of length n , $X = S^3$, and C be a string of length up to n . Then $|C| \in \mathbf{PrimCov}[i]$ if and only if $C \in \mathbf{Seeds}(X) \cap \frac{1}{2}\mathbf{PSquares}(X)$ and $n + i \in \mathbf{Centers}(C^2, X)$.

We denote $\xi(S) = \sum_{i=0}^{n-1} |\mathbf{PrimCov}[i]|$.

3 A version of the algorithm in [8]

The algorithm from [8] computes the array \mathbf{CC} in $\mathcal{O}(n \log n)$ time and is based on the characterization of Lemma 9. For each p-square Z^2 in X such that $|Z| \leq n$, if Z is a seed of X , then for every $j \in \mathit{Occ}(Z^2, X)$, we set $\mathbf{CC}[(j + |Z|) \bmod n]$ to the minimum of its current value (starting from n) and $|Z|$; see Algorithm 1.

■ **Algorithm 1** Computing \mathbf{CC} array as in [8].

```

1  $X := S^3$ ;  $\mathbf{CC}[0..n] = (n, \dots, n)$ 
2 foreach  $Z \in \frac{1}{2}\mathbf{PSquares}(X) \cap \mathbf{Seeds}(X)$  do
3   foreach  $j \in \mathit{Occ}(Z^2, X)$  do
4      $i := (j + |Z|) \bmod n$ 
5      $\mathbf{CC}[i] := \min(\mathbf{CC}[i], |Z|)$ 

```

Lemma 9 directly implies the following observation.

► **Observation 10.** *For each execution of line 5 in Algorithm 1, we have $|Z| \in \mathbf{PrimCov}[i]$.*

In the following lemma we improve the worst case complexity analysis of Algorithm 1. The proof of the lemma generally follows the details of the algorithm from [8].

► **Lemma 11.** *The time complexity of Algorithm 1 is $\Theta(\xi(S))$.*

Proof. All distinct p-squares in X can be computed in $\mathcal{O}(n)$ time using the algorithm from [6]. Lemma 7 can be used to check which of the p-square halves are seeds of X . For each p-square Z^2 whose half satisfies this condition, we find its locus v in $ST(X)$ using Corollary 3. Up to this point, all the steps work in $\mathcal{O}(n)$ time. Finally, all $j \in Occ(Z^2, X)$ can be listed in time proportional to the number of these elements by inspecting all leaves in the subtree of v in $ST(X)$. For each j we perform the instruction in line 5; by Observation 10, it corresponds to a (distinct) element from $\mathbf{PrimCov}[i]$. Overall the time complexity is $\Theta(n + \xi(S))$; however, $\xi(S) \geq n$ by Observation 8. ◀

A proof of the following theorem is deferred until Section 6. The theorem implies that, even with the improved complexity analysis, the algorithm from [8] works in $\Omega(n \log n)$ time in the worst case.

► **Theorem 12.** *There exist infinitely many strings S such that $\xi(S) = \Omega(|S| \log |S|)$.*

► **Remark 13.** The algorithm from Section 3 can be easily modified with the aid of internal Two-Period Queries of [16, 2] (such a query computes the smallest period of a factor of the text in case that the factor is periodic) to output only aperiodic covers of rotations of a string in time proportional to their total number. Still, the construction of Theorem 12 actually provides $\Omega(|S| \log |S|)$ aperiodic covers of rotations of a string.

In our approach we heavily use runs. Hence the next version of the algorithm is based on runs and is more suitable for further improvements. We also substitute the formula with the modulo operation by a formula that closely follows Lemma 9.

■ **Algorithm 2** A version of Algorithm 1 employing runs.

```

1  $X := S^3$ ;  $\mathbf{CC}[0..n] = (n, \dots, n)$ 
2 foreach trimmed run  $X[a..b]$  in  $X$  with period  $p$  do
3   for  $i := a$  to  $b$  do
4     if  $i \in [n..2n]$  and  $X[i..i+p] \in \mathbf{Seeds}(X)$  then
5        $\mathbf{CC}[i-n] := \min(\mathbf{CC}[i-n], p)$ 

```

4 Two useful representations of p-squares: occurrences and values

Our problem reduces to finding for each position i the length of the shortest p-square centered at i , whose half is a seed of the string.

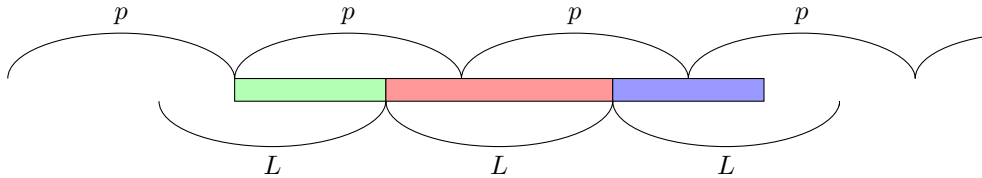
The trimmed runs, accompanied by additional useful data, can be treated as package representations of p-squares. Each occurrence of a p-square can be identified with a pair (i, p) , where i is the center and p is the period of the square.

► **Definition 14.** *An occurrence package $\gamma = (I, p)$ of occurrences of p-squares (occ-package, in short) corresponds to an interval I of consecutive centers of cyclically equivalent p-squares with period p such that $|I| \leq p$.*

With each occ-package γ we keep the following data

- $L = \mathit{LynType}(\gamma)$: the (common) Lyndon type of all the corresponding half-squares;
- $\mathit{first}(\gamma) = j_1$ and $\mathit{last}(\gamma) = j_2$, where $\mathit{rot}_{j_1}(L)$ and $\mathit{rot}_{j_2}(L)$ are the halves of the first and last p-square in γ .

An occ-package γ is called *canonical* if $first(\gamma) \leq last(\gamma)$. In this case the p-squares generated by γ are $rot_j(L)^2$ for $first(\gamma) \leq j \leq last(\gamma)$ and $L = LynType(\gamma)$. Our algorithms will only use canonical occ-packages.



■ **Figure 3** A trimmed run with period p (colored rectangles) represents the set of occurrences of p-squares with half length p that are induced by the corresponding run (top). In this example this set of occurrences is split into three (canonical) occ-packages. L is the Lyndon root of the run. A single run R generates at most $exp(R)$ occ-packages.

An *occurrence-representation* of occurrences of p-squares in X consists of a set of occ-packages for X containing all p-squares of X (together with all parameters defined above).

The lemma below follows from the fact that all runs can be computed in linear time, together with their Lyndon roots; see Figure 3.

► **Lemma 15.** *For any string X we can compute in linear time, using the runs of X , an occurrence-representation $\Gamma(X)$ of occurrences of all p-square factors of X .*

Proof. We compute the Lyndon representations of all runs in X [6]. For a run $R = X[i..j]$ with period p and Lyndon representation (L, y, a, b) , we form y occ-packages with intervals $[i + p..i + p + a)$, $[i + p + a..i + 2p + a)$, \dots , $[i + (y - 2)p + a..i + (y - 1)p + a)$, $[i + (y - 1)p + a..i + (y - 1)p + a + b]$ if $y \geq 2$, and a single package $[i + p..i + a + b]$ if $y = 1$. ◀

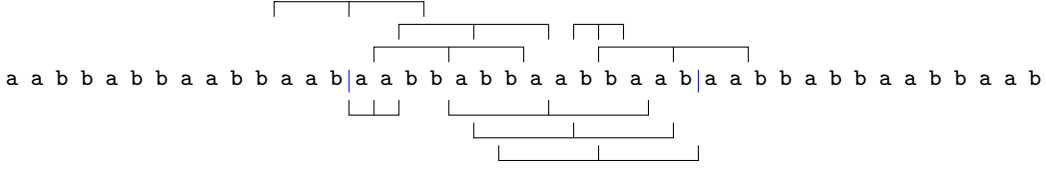
Let us note that the size of $\Gamma(X)$ is $\mathcal{O}(n)$ if $|X| = \mathcal{O}(n)$, even though the total length of intervals in occ-packages can be $\Omega(n \log n)$.

$\Gamma(X)$ is an interval representation of all occurrences of p-squares. We also need an interval-type representation of all distinct p-squares in X ; this time the sum of lengths will have linear total length.

Let a p-square U^2 be formally identified with (L, s) , where $U = rot_s(L)$ and L is the Lyndon root of U . Let us assume that all p-square factors of length up to $2n$ in X are ordered with respect to their (L, s) pairs. Let $HalfSquares[i]$ be the i -th p-square half U in this order. The length of the table $HalfSquares$ is $\mathcal{O}(n)$ [13].

► **Example 16.** For the string $S = aabbabbaabbaab$ and $X = S^3$, the table $HalfSquares$ looks as follows; see also Figure 4.

i	$HalfSquares[i]$	L	s	i	$HalfSquares[i]$	L	s
1	a	a	0	18	abba	aabb	1
2	aab	aab	0	19	bbaa	aabb	2
3	baa	aab	2	20	baab	aabb	3
4	aabaabbabbaabb	aabaabbabbaabb	0	21	abb	abb	0
...	22	bba	abb	1
17	baabaabbabbaab	aabaabbabbaabb	13	23	b	b	0



■ **Figure 4** All distinct p-squares of half length smaller than $|S|$ in $X = S^3$ for $S = \text{aabbabbaabbaab}$. For each of them, an example occurrence with the center in the middle S in X is shown.

We denote by *SeedMask* the Boolean vector such that

$$\text{SeedMask}[i] = 1 \Leftrightarrow \text{HalfSquares}[i] \in \mathbf{Seeds}(X).$$

In other words *SeedMask* is the characteristic vector of the set $\mathbf{Seeds}(X)$ as a subset of the set $\frac{1}{2}\mathbf{PSquares}(X)$.

Let us assume that an occ-package $\gamma = ([a..b], p)$ represents consecutive p-squares with halves U_1, U_2, \dots, U_k . We introduce a Boolean vector *SeedMask* $_{\gamma}$ such that *SeedMask* $_{\gamma}[i] = 1$ if and only if U_i is a seed of X .

- **Lemma 17.** (a) *The table SeedMask can be computed in $\mathcal{O}(n)$ time.*
 (b) *We can compute in $\mathcal{O}(n)$ time for all $\gamma = ([a..b], p) \in \Gamma(X)$ the values $\Phi(\gamma) := (a', b')$ such that $\text{SeedMask}_{\gamma}[a..b] = \text{SeedMask}[a'..b']$.*

Proof. (a) The main part is to compute the table *HalfSquares*. This is done as shown below in Algorithm 3.

■ **Algorithm 3** Compute *HalfSquares*.

```

1  $\mathcal{L} :=$  list of all occ-packages  $\gamma \in \Gamma(X)$  ordered by  $(\text{LynType}(\gamma), \text{first}(\gamma), -\text{last}(\gamma))$ 
2 foreach occ-package  $\gamma$  in  $\mathcal{L}$  but the first one do
3    $\gamma' :=$  previous occ-package in  $\mathcal{L}$ 
4   if  $\text{LynType}(\gamma') = \text{LynType}(\gamma)$  and  $[\text{first}(\gamma').. \text{last}(\gamma')] \supseteq [\text{first}(\gamma).. \text{last}(\gamma)]$  then
5     Remove  $\gamma$  from  $\mathcal{L}$ 
6 foreach occ-package  $\gamma$  in  $\mathcal{L}$  do
7   if there is a next occ-package  $\gamma'$  in  $\mathcal{L}$  and  $\text{LynType}(\gamma') = \text{LynType}(\gamma)$  then
8      $\text{end} := \min(\text{first}(\gamma') - 1, \text{last}(\gamma))$ 
9   else
10     $\text{end} := \text{last}(\gamma)$ 
11   Let  $\gamma = ([a..b], p)$ 
12   for  $i := a$  to  $a + \text{end} - \text{first}(\gamma)$  do
13     Append  $X[i..i+p]$  to HalfSquares

```

In the algorithm we use Lemma 15 to compute in $\mathcal{O}(n)$ time an occurrence representation $\Gamma(X)$ of occurrences of p-squares in X . We can sort all occ-packages $\gamma \in \Gamma(X)$ with respect to $(\text{LynType}(\gamma), \text{first}(\gamma), -\text{last}(\gamma))$ using Lemma 4 and then Radix Sort. Intuitively, the intervals $[\text{first}(\gamma).. \text{last}(\gamma)]$ for packages with equal Lyndon type are sorted from left to right, and intervals with equal start point are ordered by non-increasing end points. Then we scan the sorted list from left to right, removing redundant intervals $[\text{first}(\gamma).. \text{last}(\gamma)]$, that is, intervals that are contained in another such interval corresponding to the same Lyndon type. Finally, each of the non-redundant occ-packages generates some number of consecutive elements of *HalfSquares* list that were not generated by any previous occ-package in \mathcal{L} .

Let us recall that $SeedMask[i] = 1$ if and only if $HalfSquares[i]$ is a seed of X . Consequently the whole table $SeedMask$ can be computed in $\mathcal{O}(n)$ time due to Lemma 7.

(b) The function Φ can be inferred from the construction of the table $HalfSquares$. For each occ-package γ that remained on the list \mathcal{L} until the end, we set the first component of $\Phi(\gamma)$ to the position in $HalfSquares$ of the first half p-square that was introduced due to γ in line 13. For each remaining package γ , if γ' is the package that was used to remove γ from \mathcal{L} in line 4 and $\Phi(\gamma') = (a', b')$, then the first component of $\Phi(\gamma)$ is $a + first(\gamma) - first(\gamma')$. In either case the second component of $\Phi(\gamma)$ can be calculated from the first component and the length of the interval I in $\gamma = (I, p)$. ◀

Thus the bit-mask of each occ-package is a copy of a fragment of a single Boolean vector of length $\mathcal{O}(n)$; see also Figure 5.

Intuition. We need to know, in the representation $\Gamma(X)$, for each interval, a Boolean vector which says which half p-square is a seed. If we do it directly, this needs $\Omega(n \log n)$ space. However, these Boolean vectors are parts of the representation $SeedMask(X)$. Hence we can have a reference to a part of $SeedMask(X)$. The total number of references is asymptotically the same as the sum of exponents of runs, which is known to be linear. However, we need to pack $\mathcal{O}(\log n)$ -sized chunks of Boolean vectors into machine words. With further bit-level optimizations this eventually results in a linear-time algorithm. The exact implementation is given in Section 5, but first we provide an $\mathcal{O}(n \log n)$ -time implementation to illustrate the main ideas of our approach.

We introduce bitmasks $T_p[n..2n]$ for $p \in [1..n]$ such that $T_p[n+i] = 1$ if and only if $rot_i(S)$ has a cover of length at most p . We have

$$CC[i] = \min \{ p : T_p[n+i] = 1 \}.$$

For two equal-length bitmasks $F_1[a..a+\ell]$, $F_2[a'..a'+\ell]$ we define

$$\Delta(F_1, F_2) = \{ j \in [0.. \ell] : F_1[a+j] < F_2[a'+j] \}.$$

Using the bitmask tables T and $SeedMask$ we can write the next version of the algorithm. The variable New is the set of centers of new p-squares, whose halves are seeds of X (which is stored in the fragment of $SeedMask$ as a Boolean vector). In the actual implementation the bitmask T is extended to the range $[0..3n]$; this will be more convenient in the next version of the algorithm.

■ **Algorithm 4** Extraction of shortest covers of rotations.

```

1 Compute  $\Gamma(X)$ ,  $SeedMask$  and  $\Phi$  function for  $X = S^3$ 
2  $T[0..3n] := (0, \dots, 0)$ 
3 for  $p := 1$  to  $n$  do  $\triangleright$  Invariant:  $T = T_{p-1}$ 
4   foreach occ-package  $\gamma = ([a..b], p)$  in  $\Gamma(X)$  do
5      $New := \Delta(T[a..b], SeedMask[a'..b'])$ 
6     foreach  $i \in New$  do
7       if  $a+i \in [n..2n]$  then  $CC[a+i-n] := p$ 
8        $T[a+i] := 1$ 

```

Algorithm 4 still has $\mathcal{O}(n \log n)$ time complexity since the total size of all processed fragments can be $\Omega(n \log n)$. However we process only a linear number of fragments. If we implement the assignment $T[i] := 1$ in constant time, and the operation Δ in time

least significant set bit in a machine word. If `ffs` is not supported by the model, we can set the chunk length to $\frac{1}{2} \log n$ and preprocess the `ffs` values for each possible machine word in $\mathcal{O}(\sqrt{n} \log n)$ time.

We redefine the operation Δ to work on two machine words representing Boolean vectors in time proportional to the size of output plus $\mathcal{O}(1)$. To this end we apply the operation `ffs` as shown in following function (Algorithm 5).

■ **Algorithm 5** Bitmask realisation of $\Delta(F_1, F_2)$.

```

1  $F := \text{and}(\text{not}(F_1), F_2)$ 
2  $R := \emptyset$ 
3 while  $F \neq 0$  do
4    $i := \text{ffs}(F)$ 
5   Set  $i$ th bit of  $F$  to 0
6    $R := R \cup \{i\}$ 
7 return  $R$ 

```

We also use an operation $\text{Extract}(B, a, b, s)$ that, given a packed bitmask B , indices $0 \leq a \leq b < |B|$ and shift value $0 \leq s < w$, returns the fragment consisting of bits $B[a], \dots, B[b]$ also represented as a packed bitmask but shifted by s bits, i.e. the packed representation of the bitmask $0^s B[a..b] 0^t$, where $t = w - (b - a + 1 + s) \bmod w$. It can be implemented in $\mathcal{O}((b - a + 1)/\log n + 1)$ time on the word-RAM.

In the algorithm $\text{Packed}T$ and PackedSeed are packed representations, using $\mathcal{O}(n/\log n)$ machine words, of T and SeedMask .

■ **Algorithm 6** Packed bitmask realisation of Algorithm 4.

```

▷  $w = \Omega(\log n)$  is the length of machine word
1 Compute  $\Gamma(X)$ ,  $\text{SeedMask}$  and  $\Phi$  function for  $X = S^3$ 
2  $\text{Packed}T[0.. \lceil 3n/w \rceil - 1] := (0, \dots, 0)$ 
3 for  $p := 1$  to  $n$  do
4   foreach occ-package  $\gamma = ([a..b], p)$  do
5      $(a', b') := \Phi(\gamma)$ 
6      $F := \text{Extract}(\text{PackedSeed}, a', b', a \bmod w)$       ▷  $|F| = \mathcal{O}((b - a)/\log n + 1)$ 
7      $d := a \text{ div } w$ 
8     for  $j := 0$  to  $|F| - 1$  do
9        $\text{New} := \Delta(F[j], \text{Packed}T[j + d])$ 
10      foreach  $k \in \text{New}$  do
11         $i := (j + d) \cdot w + k$ 
12        if  $i \in [n..2n)$  then  $\text{CC}[i - n] := p$ 
13        Set  $k$ -th bit in  $\text{Packed}T[j + d]$  to 1

```

Thus we obtain the main result of this paper.

► **Theorem 18.** *The lengths of shortest covers of all rotations of a string can be computed in $\mathcal{O}(n)$ time.*

Proof. We apply Algorithm 6. The initial computations in line 1 are performed in $\mathcal{O}(n)$ time by Lemma 15 (computation of the occurrence-representation $\Gamma(X)$) and Lemma 17 (computation of the global SeedMask and the Φ function on occ-packages).

In [8] it was shown that the lengths of shortest covers of rotations of Fib_m can be expressed in a concise way in relation to the shortest covers of rotations of Fib_k for $k < m$. More precisely, if S_m is the prefix of $\mathbf{CC}(Fib_m)$ of length $F_{m-1} - 1$, then [8, Theorem 2(b)] shows that for $m \geq 4$,

$$\mathbf{CC}(Fib_m) = S_{m-2}, F_m, S_{m-3}, F_m, S_{m-2}, F_m, S_{m-1}, F_m.$$

The proof of the theorem, however, never used the property that the covers were shortest, but only a division into covers of length F_{m-1} and the shorter ones.

Let A_m be a table of length $F_{m-1} - 1$ of lists such that $A_m[i]$ consists of all lengths of aperiodic covers of $rot_i(Fib_m)$. Since we only care about asymptotics, we will limit the proof to the first $F_{m-1} - 1$ rotations of Fib_m . We claim that $A_m = A'_{m-2}\{F_m\}A'_{m-3}\{F_m\}A'_{m-2}$, where A'_k equals A_k with every list appended with the value F_m .

Thanks to this limitation we do not need to care about covers of lengths F_{m-1} (they are not present for those rotations), which can behave differently than F_k for $k < m - 1$ in the recurrence. The values F_k for $k < m - 1$ behave according to the described recursive formula as shown in the proof of [8, Theorem 2(b)], while values F_m occur in each list since the whole rotations are their own covers and are aperiodic due to Fact 19.

Let C_m be the number of elements in all the lists in A_m . Then the definition of A_m provides a recursive formula $C_m = 2 \cdot C_{m-2} + C_{m-3} + F_{m-1} - 1$. From here it can be checked by simple induction that $C_m \geq cmF_m$ for any $c < \frac{1}{2+\phi}$ (where ϕ is the golden ratio) and sufficiently large m :

$$\begin{aligned} 2c(m-2)F_{m-2} + c(m-3)F_{m-3} + F_{m-1} - 1 &\geq cmF_m \Leftrightarrow \\ c(mF_m - (2m-4)F_{m-2} - (m-3)F_{m-3}) &\leq F_{m-1} - 1 \Leftrightarrow \\ c(mF_{m-1} - (m-4)F_{m-2} - (m-3)F_{m-3}) &\leq F_{m-1} - 1 \Leftrightarrow \\ c(4F_{m-2} + 3F_{m-3}) &\leq F_{m-1} - 1 \Leftrightarrow \\ c(F_m + 2F_{m-1}) &\leq F_{m-1} - 1 \Leftrightarrow \\ c &\leq \frac{F_{m-1} - 1}{F_m + 2F_{m-1}} \sim \frac{1}{2 + \phi} \end{aligned}$$

This concludes the proof. ◀

Proof of Theorem 12. By Lemma 21 for the family of Fibonacci strings we have $\xi(Fib_m) = \Omega(F_m \cdot m) = \Omega(|Fib_m| \log |Fib_m|)$. ◀

7 Final Remarks

In our algorithm we extract fragments consisting of $\mathcal{O}(n \log n)$ bits in total of a packed bitmask consisting of $\mathcal{O}(n)$ bits. The fact that all these fragments can be represented in $\mathcal{O}(n)$ machine words allows us to obtain linear time complexity. Each of these bitmask fragments carries the information about which subsequent occurrences of p-squares of the same half length are seeds of the string S^3 . Based on combinatorial properties of squares and seeds, it can be the case that the total size of RLE representations of these bitmask fragments is $\mathcal{O}(n)$. This would simplify the algorithm.

References

- 1 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

- 2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The “runs” theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
- 3 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. 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 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.8.
- 4 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.
- 5 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. The number of repetitions in 2D-strings. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPICs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ESA.2020.32.
- 6 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.
- 7 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Internal quasiperiod queries. 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 60–75. Springer, 2020. doi:10.1007/978-3-030-59212-7_5.
- 8 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, 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.
- 9 Maxime Crochemore and Wojciech Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.
- 10 Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *Theoretical Computer Science*, 854:17–29, 2021. doi:10.1016/j.tcs.2020.11.033.
- 11 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.
- 12 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: <http://www.jstor.org/stable/2034009>.
- 13 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.
- 14 Costas S. Iliopoulos, Dennis W. G. Moore, and William F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1-2):281–291, 1997. doi:10.1016/S0304-3975(96)00141-7.
- 15 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.
- 16 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, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

- 17 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFPCS.1999.814634.
- 18 Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002. doi:10.1007/s00453-001-0062-2.
- 19 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.


Rectangular Tile Covers of 2D-Strings

Jakub Radoszewski ✉ 

University of Warsaw, Poland

Wojciech Rytter ✉ 

University of Warsaw, Poland

Juliusz Straszyński ✉ 

University of Warsaw, Poland

Tomasz Waleń ✉ 

University of Warsaw, Poland

Wiktor Zuba ✉ 

University of Warsaw, Poland

CWI, Amsterdam, The Netherlands

Abstract

We consider tile covers of 2D-strings which are a generalization of periodicity of 1D-strings. We say that a 2D-string A is a *tile cover* of a 2D-string S if S can be decomposed into non-overlapping 2D-strings, each of them equal to A or to A^T , where A^T is the transpose of A . We show that all tile covers of a 2D-string of size N can be computed in $\mathcal{O}(N^{1+\varepsilon})$ time for any $\varepsilon > 0$. We also show a linear-time algorithm for computing all 1D-strings being tile covers of a 2D-string.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases tile cover, periodicity, efficient algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.23

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Juliusz Straszyński: 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.

Acknowledgements We thank Panagiotis Charalampopoulos for helpful discussions.

1 Introduction

A 1D-string (or simply a string) is a finite sequence of letters. A 2D-string S is a rectangular 2D-matrix consisting of n rows being strings of equal length m . The dimensions of the 2D-string are $n \times m$ and its size is $|S| = N = n \cdot m$. We say that a 1D-string S has period p if $S[i] = S[i + p]$ for all $i = 1, \dots, |S| - p$. We say that a 2D-string S has horizontal (vertical) period p if each row (column, respectively) of S has period p .

Periodicity in 2D-strings has different properties than in 1D-strings. Let us consider an example based on the following known notions of periodicity. A *run* in a 1D-string (2D-string, respectively) S is a maximal periodic factor of S (maximal submatrix that is periodic in both dimensions). Two natural 2D-generalizations of squares in a 1D-string are known; a *quartic* is a configuration that is composed of 2×2 occurrences of an array W and a *tandem* is a configuration consisting of two occurrences of an array W that share one side. A string of length n has $\mathcal{O}(n)$ distinct square factors [8, 7, 14] and $\mathcal{O}(n)$ runs [11, 3]. However, it was recently shown that a 2D-string of size N can have $\Omega(N^{3/2})$ distinct tandems, $\Omega(N \log N)$ distinct quartics and $\Omega(N \log N)$ runs [9].



© Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba; licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 23; pp. 23:1–23:14

Leibniz International Proceedings in Informatics



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

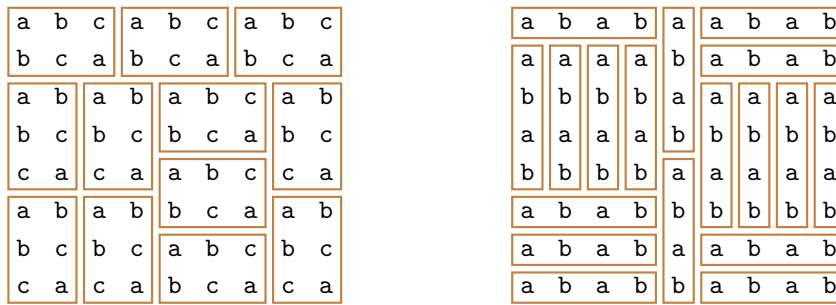
23:2 Rectangular Tile Covers of 2D-Strings

Motivated by these differences, we introduce a natural generalization of periodicity to 2D-strings which we call tile covers. A 1D-string S has a full period P if $S = P^k$ for some positive integer k . We say that a 2D-string A is a *2D-tile cover* or simply *tile cover* of an $n \times m$ 2D-string S if S can be decomposed into (non-overlapping) 2D-strings each of them equal to A or to A^T , where A^T is the transpose of A (the i -th row of A becomes in A^T the i -th column); see Figure 1. In this paper we consider the following problem.

Tile covers problem. Compute efficiently all tile covers A of a given 2D-string S .

We consider this problem in full generality as well as a special case in which A is a 1D-string (a rectangle consisting of a single row) which we call the **1D-tile covers problem**. Note that for a 2D-string of size N , there are at most N tile covers; each of them corresponds to a submatrix of S starting in its top left corner. In particular, each tile cover of S can be represented in $\mathcal{O}(1)$ space.

► **Observation 1.** *In case when S is a 1D-string, the tile cover problem is trivial: a 1D-string A is a tile cover of a 1D-string S if and only if S is a string power of A . However in case of 2D-strings the problem becomes complicated, even for 1D-tile covers.*



■ **Figure 1** Two examples of tile coverings of 2D-strings; the first one is by a 2×3 2D-tile cover and the second one is by a 1D-tile cover of length 4. Let us notice that in the case of 1D-tile covers both **abab** and its primitive root **ab** are 1D-tile covers of the 2D-string.

A known generalization of periodicity in 1D-strings is quasiperiodicity. A string C is a cover of a 1D-string S if S can be created by possibly overlapping occurrences of C ; see e.g. [2, 4]. Versions of covers of 2D-strings were studied before; the main difference between these notions and tile covers is that tile covers do not allow covering the text with overlapping occurrences of the cover. In [5] two notions of covers of 2D-strings, called 1D-covers and 2D-covers, were considered. A 2D-string C is a *2D-cover* of a 2D-string T if each position of T is inside an occurrence of C in T . Various algorithms computing 2D-covers were presented in [5, 6, 13]. A (1D) string C is a *1D-cover* of a 2D-string T if each position of T is inside an occurrence of C or C^T . A linear-time algorithm computing all 1D-covers of a string was proposed in [5].

Our Results. Let S be a 2D-string of size N . We show that:

- all 1D-tile covers of S can be computed in $\mathcal{O}(N)$ time;
- all tile covers of S can be computed in $\mathcal{O}(N^{1+\varepsilon})$ time for any $\varepsilon > 0$.

Let us recall that each tile cover can be represented in $\mathcal{O}(1)$ space as a submatrix of S .

2 1D-Tile covers of 2D-strings

In this section we show how to compute 1D-tile covers of an $n \times m$ 2D-string S of size N . Henceforth by ℓ we denote the length of the 1D-tile cover.

Let us recall some notation and properties of periodicity of 1D-strings. A string B that is both a prefix and a suffix of a string U is called a border of U . A factor F of a string U is called proper if $|F| < |U|$. By $root(U)$, called the primitive root of U , we denote the shortest string Z such that U is a power of Z . We say that U is primitive if $root(U) = U$. The string $root(U)$ is primitive. Let us also recall the solution to the following classical word equation; cf. [12].

► **Lemma 2.** *If strings X, Y satisfy $XY = YX$, then there exists a string Z such that $X = Z^x$ and $Y = Z^y$ for some positive integers x, y .*

2.1 Unary 1D-tile covers

The unary case is simpler, but not completely trivial.

► **Remark 3.** The number of distinct unary 1D-tile coverings is potentially exponential, even if $\ell = 2$. For example, there are 12,988,816 ways to tile a standard 8×8 chessboard with dominoes (to tile the 8×8 unary 2D-text by a linear unary tile of length $\ell = 2$). The numbers of distinct domino tilings of a $2 \times n$ text are Fibonacci numbers.

We define the following auxiliary $n \times m$ table D_ℓ : the first row is a prefix of $(1, 2, \dots, \ell)^\infty$, and each subsequent row results by adding 1 to the elements of the previous row, substituting each $\ell + 1$ with 1; see Figure 2.

1	2	3	4	5	6	1	2	3
2	3	4	5	6	1	2	3	4
3	4	5	6	1	2	3	4	5
4	5	6	1	2	3	4	5	6
5	6	1	2	3	4	5	6	1
6	1	2	3	4	5	6	1	2
1	2	3	4	5	6	1	2	3
2	3	4	5	6	1	2	3	4

■ **Figure 2** Illustration of the proof of Lemma 5 for $n = 8, m = 9, \ell = 6$. The two framed rectangles are balanced; however the remaining 2×3 rectangle A is not, hence a unary 8×9 matrix does not have a 1D-tile cover of length 6 (even though 6 divides $72 = 8 \cdot 9$).

We say that a submatrix of D_ℓ is *balanced* if each integer in $\{1, \dots, \ell\}$ occurs the same number of times in this submatrix.

► **Observation 4.** *For $0 < k, r < \ell$, each $k \times r$ submatrix A of D_ℓ is not balanced.*

Proof. The proof is by contradiction. Assume each of the numbers $1, \dots, \ell$ occurs in A the same number of times. The integer at position $\min(k, r)$ in the first row of A occurs in the first $\min(k, r)$ rows of A . Hence each integer in A should occur at least $\min(k, r)$ times, altogether we have at least $\min(k, r) \cdot \ell$ occurrences in A . However, it is impossible since A has only $k \cdot r$ positions and $k \cdot r < \min(k, r) \cdot \ell$ due to the inequality $k, r < \ell$. ◀

► **Lemma 5.** *A unary 2D-string of size $n \times m$ has a (unary) 1D-tile cover of length ℓ if and only if ℓ divides n or m .*

23:4 Rectangular Tile Covers of 2D-Strings

Proof. If ℓ divides m , then the text can be covered trivially with the use of horizontal occurrences; symmetrically with the use of vertical ones if ℓ divides n . Let us consider the other implication.

Each occurrence of a 1D-tile cover of length ℓ is balanced in the array D_ℓ , hence the whole 2D-string can be covered only if D_ℓ is balanced. Since every submatrix of D_ℓ of dimensions $k \times \ell$ and $\ell \times k$ is balanced, the problem reduces to the case of a matrix of dimensions $m \bmod \ell$ and $n \bmod \ell$, both smaller than ℓ , which is covered by Observation 4. ◀

► **Remark 6.** Let us note that it is not sufficient in the lemma for ℓ to divide $n \cdot m$; see Figure 2.

2.2 Combinatorics of non-unary 1D-tile covers

In this section we consider only non-unary texts S and non-unary tile covers (it is impossible for a non-unary text to have a unary tile cover or for a unary text to have a non-unary tile cover). Assume that the first row of S starts with the letter a . Consequently each 1D-tile cover starts with an occurrence of the letter a .

► **Definition 7.** For a nonempty 2D-string U we define

$$\text{ratio}(U) = \frac{\#_a(U)}{|U|}$$

where $\#_c(U)$ is the number of occurrences of the letter c in U .

We say that two 2D-strings are *similar*, written $U_1 \sim U_2$, if $\text{ratio}(U_1) = \text{ratio}(U_2)$. Obviously if U is a tile cover of S , then $U \sim S$.

Notice that since the top left position of S has to be covered, any 1D-tile cover U has to be a prefix of the first row or the first column of S , and by symmetry also the suffix of the last row or the last column of S . Henceforth we consider the case in which U is a prefix of the first row of S and a suffix of the last row of S ; the remaining cases can be handled in a symmetric way.

► **Definition 8.** In this section we say that a 1D-string U of length at most $\min(n, m)$ is a candidate if it satisfies the following conditions:

- U is a prefix of the first row of S and a suffix of the last row of S ;
- $U \sim S$;
- The first row of S belongs to $(U \cup a)^*$, that is, it can be factorized into factors equal to U and a .

► **Observation 9.** If U is a 1D-tile cover of S , then U is a candidate.

► **Observation 10.** If U is a 1D-tile cover of S , then so is $\text{root}(U)$.

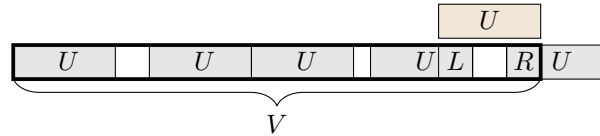
The following auxiliary lemma plays an important role in a proof of Lemma 12 that gives a key characterization of candidates.

► **Lemma 11.** Assume $U = L a^k R = R a^k L$, U is primitive and non-unary, and $L, R \neq \varepsilon$. Then $\text{ratio}(a^k R) > \text{ratio}(U)$.

Proof. We have that $k > 0$, as otherwise $U = LR = RL$, which implies that U is not primitive (see Lemma 2). The word equation from the statement of the lemma is equivalent to $a^k U = (a^k L)(a^k R) = (a^k R)(a^k L)$, hence, again by Lemma 2, strings $a^k U$ and $a^k R$ have a common primitive root, which concludes that $\text{ratio}(a^k U) = \text{ratio}(a^k R)$. However, since U is non-unary, we have that $\text{ratio}(a^k R) = \text{ratio}(a^k U) > \text{ratio}(U)$. ◀

► **Lemma 12.** *Take two candidates U, V , such that U is primitive and $|U| < |V|$. Then V is a power of U .*

Proof. Both U and V are candidates, which implies that U is a border of V . At the same time, since U is a candidate, the occurrence of V that covers the top left corner of S can be factorized into occurrences of U , letters a and a (possibly empty) string R – a proper prefix of U – at its end; see Figure 3. Let \mathcal{F} denote this factorization. We will show that $R = \varepsilon$, which will conclude that V is a power of U . Indeed, otherwise we would have $ratio(V) > ratio(U)$, contradicting the definition of a candidate.



■ **Figure 3** The series of grey boxes in the main line represents a factorization of the first row of 2D-string S into (non-unary) U 's separated by a^* 's (white spaces, possibly zero a 's). The upper brown box corresponds to the suffix of V equal to U . We focus on the first occurrence of V in the text. If additionally we assume that $ratio(U) = ratio(V)$, and that U is primitive, then V must be a power of U (that is, there are no white spaces, and the last U ends at the same point as V).

The length- $|U|$ suffix of V , by the factorization \mathcal{F} , has a form La^kR for some (possibly empty) suffix L of U and $k \geq 0$. However, this suffix is equal to U , so $U = La^kR$. At the same time, since R is a prefix of U , we also have that $U = RWL$ for some string W . From these two decompositions of U we see that $ratio(a^k) = ratio(W)$, i.e. $W = a^k$.

This will allow us to apply Lemma 11. First, if any of the strings L, R is empty and $k > 0$, by Lemma 2, we obtain that U is unary, and so is V , which concludes the proof. If any of L, R is empty and $k = 0$, then $R = \varepsilon$ (since R was chosen as a proper prefix of U) and we obtain the conclusion as shown before. Otherwise we can indeed apply Lemma 11 and obtain that $ratio(a^kR) > ratio(U)$. From the aforementioned factorization \mathcal{F} of V we obtain $ratio(V) > ratio(U)$, which contradicts the definition of a candidate. ◀

► **Corollary 13.** *Assume a non-unary text S has a 1D-tile cover. Then the shortest 1D-tile cover of S is the shortest candidate U . It is a primitive string. All other 1D-tile covers are powers of U , though not all powers of U are necessarily 1D-tile covers of S .*

The corollary suggests the following algorithm for finding the shortest 1D-tile cover:

1. Find the shortest primitive candidate U .
2. Then check if it is a 1D-tile cover.

The first step is easy because it involves only 1D-strings: first row/column and the last row/column. The second step can be done using a greedy approach.

However, testing which powers of U are 1D-tile covers requires a slightly different number-theoretic approach.

2.3 Computing non-unary candidates

All 1D-tile covers U satisfying $|U| > \min(n, m)$ can be trivially computed, hence we later assume that the length of the cover is at most $\min(n, m)$.

We use the following algorithm to compute all candidates.

Algorithm 1 ALL-CAND(S).

```

Compute  $ratio(S)$ 
 $Cand$  is initially the set of all proper borders of  $S_1\$S_2$  of length  $\leq n$ , where  $S_1$  and
 $S_2$  are respectively the first and the last row of  $S$  and  $\$$  is a special symbol
using prefix-sums computation we compute  $ratio(Z)$  for each prefix of  $S_1$ 
remove from  $Cand$  all  $U$  such that  $ratio(U) \neq ratio(S)$ 
foreach  $U \in Cand$  do
    if  $S_1 \notin (U \cup a)^*$  then remove  $U$  from  $Cand$ 
    // It is checked in  $\mathcal{O}(m)$  time per each  $U$ 
return  $Cand$ 

```

► **Corollary 14.** *ALL-CAND(S) computes in $\mathcal{O}(N)$ time all candidates.*

Proof. Borders of a string of length m can be computed in $\mathcal{O}(m)$ time [10]. We have $|Cand| \leq \min(m, n)$ and each $U \in Cand$ is checked in $\mathcal{O}(m)$ time using linear-time pattern matching [10]. ◀

2.4 Testing a single non-unary candidate

Assume $\#$ does not occur in S . In the course of the next algorithm certain entries will be marked, i.e., changed to $\#$. For a 2D-string U we define two operations:

- $Match(i, j, U)$: return true if and only if there is a full occurrence of U starting in position (i, j) in S ,
- $Mark(i, j, U)$: changes each symbol in S in the occurrence of U starting in (i, j) to $\#$.

► **Remark 15.** The following algorithm GREEDY is also well defined for a 2D-string U , but it does not work correctly for all tile cover candidates which are not 1D-strings. We reuse it later in Section 3.2.1.

Algorithm 2 GREEDY(S, U).

```

Output: True if a non-unary 2D-string  $U$  is a 1D-tile cover of  $S$ 
for  $i := 1$  to  $n$  do
    for  $j := 1$  to  $m$  do
        if  $S[i, j] \neq \#$  then
            if  $Match(i, j, U)$  then
                ▷ choosing occurrence of  $U$ 
                 $Mark(i, j, U)$ 
            else if  $Match(i, j, U^T)$  then
                ▷ choosing occurrence of  $U^T$ 
                 $Mark(i, j, U^T)$ 
            else
                return false
        ▷ All positions are now marked
    return true

```

► **Theorem 16.** *Assume U is a 1D-string. Then GREEDY(S, U) checks if U is a 1D-tile cover of S in $\mathcal{O}(N)$ time. Covering of S with U is unique (occurrences of U forming the tile cover can be chosen only in a single, unique way).*

Proof.

Correctness. When we are processing the i -th row then the preceding rows are already completely marked. Hence each non-marked position in this row should be marked now by an occurrence of U or U^T starting in this row. If $\text{Match}(i, j, U)$, then U must be used instead of U^T .

Indeed, U is non-unary. Let k be its first position containing letter different from a . $\text{Match}(i, j, U)$ determines an existence of an uncovered letter different from a at position $(i, j + k - 1)$. This position cannot be covered with an occurrence of U beginning further (k is the first such position), nor with a U^T , which determines that U must be used at position (i, j) .

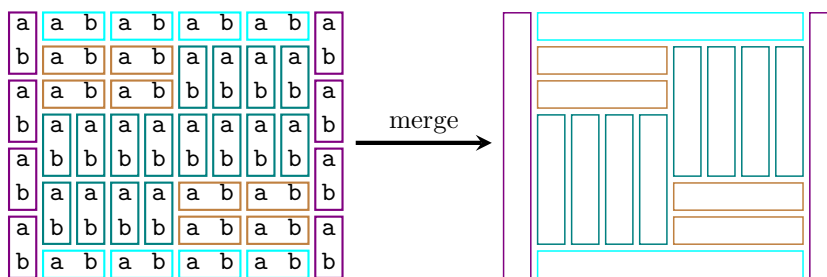
Complexity. Each operation Match can be done in $\mathcal{O}(|U|)$ time. It is amortized by marking $|U|$ positions which were not marked previously. Consequently, the total time is $\mathcal{O}(N)$. ◀

2.5 Finding all 1D-tile covers in non-unary texts

Denote by $\text{gcd}(M)$ the greatest common divisor of integers in a set M . We use the following algorithm (see also Figure 4).

■ **Algorithm 3** ALL-TILES(S).

Output: All lengths of 1D-tile covers
 $U :=$ shortest element of $\text{Cand}(S)$
 if $\text{GREEDY}(S, U) = \text{false}$ then return \emptyset
 now $\text{GREEDY}(S, U)$ can partition S into horizontal occurrences of U and vertical occurrences of U^T
 we merge consecutive horizontal occurrences and consecutive vertical occurrences in possibly larger disjoint horizontal and vertical strips
 $M :=$ set of lengths of obtained strips
 $d := \text{gcd}(M)$
 return all powers of U whose lengths divide d



■ **Figure 4** Merging of occurrences of the smallest 1D-tile cover in the covering to compute the larger ones. Here rectangles of length 4 and 8 appear, their greatest common divisor is 4, hence 1D-tile covers have lengths 2 and 4 (multiples of 2 and divisors of 4).

► **Theorem 17.** All 1D-tile covers of a 2D-string of size N can be computed in $\mathcal{O}(N)$ time.

Proof.

Correctness. Let U be the shortest 1D-tile cover. By Corollary 13 only the powers of U can be 1D-tile covers. The merged rectangles partition the 2D-string into 1D-parts. If the length of a given power of U divides the length of each rectangle, then each part of the division is trivially covered, hence the power is a 1D-tile cover. On the other hand if a given power of U is a 1D-tile cover, then by applying Algorithm 3 with it as the candidate we would obtain a different set of rectangles. This however would contradict the uniqueness of covering of S with U given by Theorem 16.

Complexity. We find the shortest 1D-tile cover with the use of Algorithms 1 and 2 both running in $\mathcal{O}(N)$ time. Algorithm 2 as a byproduct returns the set of $N/|U|$ rectangles, which can be merged in $\mathcal{O}(N/|U|)$ time. The greatest common divisor of their lengths is computed in exactly the same time. ◀

3 2D-Tile covers in 2D-strings

In this section we consider tile covers of shape $d \times \ell$, where $d \leq \ell$ (we find the other ones as tile covers of S^T).

3.1 Unary 2D-tile covers

► **Lemma 18.** *Unary $d \times \ell$ 2D-string is a tile cover of a unary $n \times m$ 2D-string if either case applies:*

- (a) d is a divisor of one dimension, and ℓ is a divisor of the other dimension of S ,
- (b) both d and ℓ divide the same dimension of S and the length of the other dimension is of the form $a \cdot d + b \cdot \ell$, for integers $a, b \geq 0$.

Proof. From Lemma 5 we know that both d and ℓ have to divide one of the dimensions n or m (unary tiling with $d \times \ell$ rectangle easily divides into a one with $1 \times d$ or $1 \times \ell$ rectangles). If each dimension of U divides a different dimension of S (case (a)), then we can tile cover S with U trivially with only occurrences of U or occurrences of U^T . For the other case we assume, that both d and ℓ divide m .

If $n = a \cdot d + b \cdot \ell$ for integers $a, b \geq 0$ we can divide S into two parts of size $(a \cdot d) \times m$ and $(b \cdot \ell) \times m$, and cover the first one with only U 's and the second one with only U^T 's.

On the other hand if U is a tile cover of S , then the tiling divides the first column into segments of lengths d or ℓ , hence n must be of a form $a \cdot d + b \cdot \ell$ for some integers $a, b \geq 0$. ◀

3.2 Non-unary 2D-tile covers – testing a candidate

We make use of 2-dimensional properties of 2D-tiles related to symmetry and horizontal periodicity. We assume later in this section that all considered 2D-tiles are of shape $d \times \ell$, where $d \leq \ell$.

► **Definition 19.** *A square matrix A is called symmetric if $A = A^T$. A matrix is called horizontally periodic (H-periodic, in short) if its i -th column equals its $(i + d)$ -th column, for $i \leq \ell - d$.*

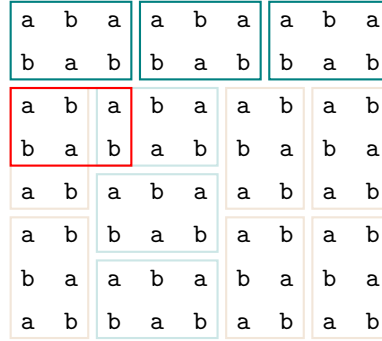
Denote by $\text{Pref}(U)/\text{Suf}(U)$ the square matrix consisting of the first/last d columns of U .

► **Definition 20.** *Define*

$$\gamma(U) \equiv \text{Pref}(U), \text{Suf}(U) \text{ are symmetric and } U \text{ is H-periodic.}$$

3.2.1 Case: not $\gamma(U)$

Observe that the algorithm $\text{GREEDY}(S, U)$ can be applied also to 2D-tiles. The main deterministic choices of this algorithm are whether to use U or U^T . In this algorithm the priority is given to U . It works correctly for 1D-tiles, unfortunately such simple solution is incorrect for 2D-tile covers U , see Figure 5.



■ **Figure 5** The algorithm $\text{GREEDY}(S, U)$ from Section 2 does not work in this case: we start in the top-left corner and take as U the prefix of S of shape 2×3 (2 rows, 3 columns) and fill the first two rows this way. However when we do the same thing when we start covering the third row we cannot continue later and GREEDY returns *false*. It is incorrect because U covers S in a different non-greedy way.

► **Lemma 21.** *If $\text{Pref}(U)$ is not symmetric, or U is not H-periodic then $\text{GREEDY}(S, U)$ works correctly.*

Proof. The only way, the algorithm GREEDY can give a bad answer is the case, where both $\text{Match}(i, j, U)$ and $\text{Match}(i, j, U^T)$ return true in a given position (such that all of the previous positions are covered), and choose to use $\text{Mark}(i, j, U)$ even though, the right occurrence to choose is U^T .

In the case, where $\text{Pref}(U)$ is not symmetric U and U^T cannot both match in any position ($U[1..d, 1..d] \neq U^T[1..d, 1..d]$).

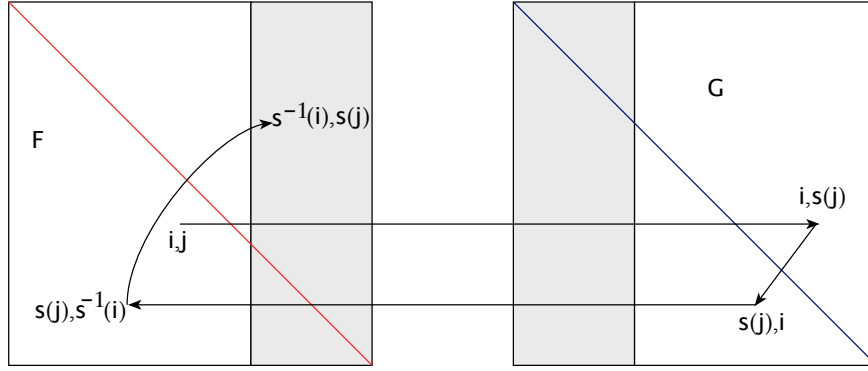
If U is not H-periodic, then if $\text{Match}(i, j, U)$ return true, then one of the $d \times d$ submatrices of S with its top-left corner at position (i, k) , where $j < k < j + l, k \bmod d = j \bmod d$ is different from $\text{Pref}(U)$.

If $\text{Mark}(i, j, U^T)$ was used, then position (i, k) cannot be covered neither with $\text{Mark}(i, p, U)$ for $j < p \leq k, p \bmod d = j \bmod d$ nor with $\text{Mark}(i, k, U^T)$, and only such occurrences are available to use. ◀

Denote by \hat{U} the matrix resulting from U by reversing each row, and then reversing each column.

► **Lemma 22.** *If $\text{Suf}(U)$ is not symmetric then $\text{GREEDY}(\hat{S}, \hat{U})$ returns true iff U is a tile cover of S since it is never possible to match both U and U^T in the same position.*

The last two lemmas give simple linear time tile test for the case when $\text{Pref}(U)$ or $\text{Suf}(U)$ is not symmetric or U is not H-periodic.



■ **Figure 6** Migration of an element in the matrix F via the matrix G .

3.2.2 Reduction to case $\gcd(d, \ell) = 1$

It may be the case, that both S and U are in fact composed of smaller, symmetric $z \times z$ matrices. If all of those submatrices are equal, then the tile covering of S may not be unique even though it is not unary. In this section we show, that it is only possible for $z = \gcd(d, \ell)$, and that the problem can be reduced to the case, where $\gcd(d, \ell) = 1$. This also simplifies the algorithm in the next section.

Assume operations \oplus, \ominus are addition and subtraction modulo k , respectively.

► **Fact 23.** Assume $\gcd(r, k) = 1$. If $Z \subseteq \{0, 1, 2, \dots, k-1\}$ is non-empty and has a property, that if $x \in Z$ implies $x \oplus r \in Z$, then $Z = \{0, 1, 2, \dots, k-1\}$.

In the lemma below we count rows and columns starting from zero.

► **Lemma 24.** Assume $0 < r < k$ and $\gcd(k, r) = 1$. Let matrices A, B be of shapes $k \times r, k \times (k-r)$, respectively, and $F = A \cdot B, G = B \cdot A$ (where \cdot denotes horizontal concatenation of matrices).

If F, G are symmetric, then all elements $F[i \oplus x, j \ominus x]$ are equal, for given i, j and all $x \in \{0, 1, 2, \dots, k-1\}$.

Proof. Let $s(x) = x \ominus r$. Then $F[i, j] = G[i, s(j)]$; see Figure 6.

Due to symmetry of G we have $G[i, s(j)] = G[s(j), i]$. Then $G[s(j), i] = F[s(j), s^{-1}(i)]$, and using symmetry of F we have

$$F[s(j), s^{-1}(i)] = F[s^{-1}(i), s(j)] = F[i \oplus r, j \ominus r]$$

Consequently $F[i, j] = F[i \oplus r, j \ominus r]$. The thesis follows now from Fact 23, by iterating the last equality. ◀

Denote $z = \gcd(d, \ell)$ and $k = d/z$.

► **Lemma 25.** Assume $\gamma(U)$. Let us decompose U into disjoint submatrices $z \times z$. Then each of these submatrices is symmetric.

Proof. Let us treat each of these $z \times z$ submatrices as single elements. Then we obtain the $k \times (\ell/z)$ matrix U' . We can apply Lemma 24 to $F = Pref(U')$, $G = Suf(U')$. Then Lemma 24 implies that each of our $z \times z$ sub-matrices equals a $z \times z$ sub-matrix on the main diagonal of $Pref(U)$ (or $Suf(U)$), consequently it is symmetric due to symmetry of diagonal $z \times z$ submatrices (see Figure 7). ◀

a	b	d	e	g	h	a	b	d	e
b	c	e	f	h	i	b	c	e	f
d	e	g	h	a	b	d	e	g	h
e	f	h	i	b	c	e	f	h	i
g	h	a	b	d	e	g	h	a	b
h	i	b	c	e	f	h	i	b	c

■ **Figure 7** The figure illustrate the most general example of U of size 6×10 such that $\gamma(U)$. Notice that it is composed of symmetric submatrices 2×2 ($z = 2 = \gcd(6, 10)$) of only $k = 3 = 6/2$ types.

► **Corollary 26.** *If $\gamma(U)$ and $\gcd(d, \ell) > 1$ then we can reduce in linear time our problem to the case of $d' \times \ell'$ candidate U' , where $\gamma(U')$ and $\gcd(d', \ell') = 1$.*

Proof. We decompose U and S into disjoint $z \times z$ submatrices. By Lemma 25 we know that the submatrices composing U are symmetric. If a decomposition of S contains a non-symmetric matrix, then U cannot be a tile cover of S . Otherwise we replace each $z \times z$ submatrix with 1×1 submatrix, with symbol identifying the corresponding submatrix. Then the resulting 2D-texts U', S' prove the thesis. ◀

3.2.3 Case: $\gamma(U)$ and $\gcd(d, \ell) = 1$

The following fact can be shown using Fact 23 and similar arguments as in the proof of Lemma 24.

► **Fact 27.** *Assume $\gamma(U)$ and $\gcd(d, \ell) = 1$. Then if two distinct columns of $\text{Pref}(U)$ are equal, then U is unary.*

Denote by $\text{first}(U)$ the first column of U , by $\text{first}(\text{Suf}(U))$ the first column of $\text{Suf}(U)$ and by $\text{Col}(i, j, S)$ the fragment of size d of the j -th column of S starting in row i . We use the following algorithm.

■ **Algorithm 4** GREEDY $'(S, U)$.

Output: True if U is a 2D-tile cover

▷ Assume U is non-unary

for $i := 1$ **to** n **do**

for $j := 1$ **to** m **do**

if $S[i, j] \neq \#$ **then**

if $\text{Match}(i, j, U)$ **and** $\text{First}(\text{Suf}(U)) \neq \text{Col}(i, j + \ell, S)$ **then**

 ▷ choose occurrence of U

 Mark(i, j, U)

else if $\text{Match}(i, j, U^T)$ **then**

 ▷ choose occurrence of U^T

 Mark(i, j, U^T)

else

return false

 ▷ All positions are now marked

return true

23:12 Rectangular Tile Covers of 2D-Strings

► **Example 28.** Figure 8 shows how our algorithm is working. The figure illustrates the case when $First(Suf(U)) = Col(i, j + \ell, S)$ for $i = j = 1$, (the first column of $Suf(U)$ starts in the 4-th column of S). Hence instead of using $Mark(1, 1, U)$ we execute $Mark(1, 1, U^T)$.

The next (i, j) with $S[i, j] \neq \#$ is $(1, 3)$. For $j = 3$ we have $j + \ell = 6$ and $First(Suf(U)) \neq Col(1, 6, S)$, hence we perform here $Mark(1, 3, U)$.

After using $Mark(5, 1, U^T)$ the next (i, j) with $S[i, j] \neq \#$ is $(3, 3)$. Here $S[i, j + 3] = \#$, hence we perform $Mark(3, 3, U)$.

a	b	a	b	a	b
b	a	b	a	b	a
a	b	a	b	a	b
a	b	b	a	b	a
b	a	a	b	a	b
a	b	b	a	b	a

■ **Figure 8** U is here the 2×3 prefix of S , $d = 2$, $\ell = 3$. The algorithm GREEDY' starts with U^T (due to additional comparison of two columns), while a naive greedy would start with U (and later fail).

► **Theorem 29.** *We can test if a given 2D-tile candidate is a 2D-tile cover in $\mathcal{O}(N)$ time.*

Proof. If not $\gamma(U)$, then we use Algorithm 2. This case was already covered by Lemmas 21 and 22. Otherwise we use Algorithm 4. Assume, that $\gamma(U)$ and $\gcd(d, \ell) = 1$.

If $Match(i, j, U)$, then $First(Suf(U)) \neq Col(i, j + \ell, S)$ represents a break in H-periodicity (possibly due to an occurrence of $\#$ or the end of the text). In this case if the algorithm decides to use U^T instead of U at the next not covered positions $(i, j + d)$, $(i, j + 2d)$, \dots , $(i, j + kd)$ for $\ell - d < kd < \ell$, the use of U will not be possible, and hence it will have to use U^T .

However then, when trying to cover position $(i, j + kd)$ for $\ell - d < kd < \ell$ it will be also unable to use U^T due to the same break of the period. If however $First(Suf(U)) = Col(i, j + \ell, S)$, then by Fact 27 $Col(i, j + \ell, S) \neq First(U)$, hence after using U at position (i, j) we will not be able to cover position $(i, j + \ell)$, and we know that $j + \ell \leq m$ and the position is not covered yet. ◀

3.3 Computing all non-unary 2D-tile covers

Let $D(n)$ denote the number of natural divisors of a natural number n .

► **Fact 30** ([1]). $D(n) = o(n^\varepsilon)$ for every constant $\varepsilon > 0$.

► **Corollary 31.** *There are only $\mathcal{O}(n^\varepsilon)$ possible shapes of 2D-tile covers of S , for $m \leq n$ and any $\varepsilon > 0$.*

► **Theorem 32.** *We can compute all 2D-tile covers in time $\mathcal{O}(N^{1+\varepsilon})$.*

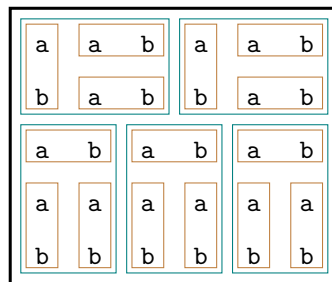
Proof. For a given candidate U we can in $\mathcal{O}(|U|)$ time check if property $\gamma(U)$ holds, and then use Algorithm 2 or Algorithm 4, each working in $\mathcal{O}(N)$ time. Due to Corollary 31 there are only $\mathcal{O}(N^\varepsilon)$ candidates for a tile cover, which we can check in $\mathcal{O}(N^{1+\varepsilon})$ total time. ◀

► **Observation 33.** *Just like in case of 1D-tile covers (see Theorem 16) covering of a non-unary 2D-string S with its 2D-tile cover U is unique.*

4 Final remarks

We showed that 2D-tile cover problem can be solved in $\mathcal{O}(N^{1+\epsilon})$ time. Our $\mathcal{O}(N)$ time algorithm for 1D-tiles was based on Lemma 12, which says that all 1D-tiles are powers of the smallest primitive one. However it does not work for 2D-tiles, see Figure 9.

We say that $d \times \ell$ 2D-tile cover is primitive iff it is not a horizontal or vertical power of a smaller 2D-tile.



■ **Figure 9** S has three primitive 2D-tile covers of shapes 2×1 , 2×3 and 5×6 (S itself). None of them is a horizontal or vertical power of another one (but a smaller one is a 2D-cover of a larger one).

We pose the following conjectures.

► **Conjecture 34.** *If a 2D-string has two distinct primitive 2D-tile covers, then one of them is a 2D-tile cover of the other one.*

► **Conjecture 35.** *There is a linear time algorithm for computing all 2D-tile covers.*

References

- 1 Tom M. Apostol. *Introduction to analytic number theory*. Undergraduate Texts in Mathematics, New York-Heidelberg: Springer-Verlag, 1976.
- 2 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.
- 3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The “runs” theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
- 4 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.
- 5 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Computing covers of 2D-strings. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, volume 191 of *LIPICs*, pages 12:1–12:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CPM.2021.12.
- 6 Maxime Crochemore, Costas S. Iliopoulos, and Maureen Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20(4):353–373, 1998. doi:10.1007/PL00009200.
- 7 Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.
- 8 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

23:14 Rectangular Tile Covers of 2D-Strings

- 9 Pawel Gawrychowski, Samah Ghazawi, and Gad M. Landau. Lower bounds for the number of repetitions in 2D strings. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval – 28th International Symposium, SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 179–192. Springer, 2021. doi:10.1007/978-3-030-86692-1_15.
- 10 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 11 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.
- 12 M. Lothaire. *Combinatorics on words, Second Edition*. Cambridge Mathematical Library. Cambridge University Press, 1997.
- 13 Alexandru Popa and Andrei Tanasescu. An output-sensitive algorithm for the minimization of 2-dimensional string covers. In *Theory and Applications of Models of Computation – 15th Annual Conference, TAMC 2019*, volume 11436 of *Lecture Notes in Computer Science*, pages 536–549. Springer, 2019. doi:10.1007/978-3-030-14812-6_33.
- 14 Adrien Thierry. A proof that a word of length n has less than $1.5n$ distinct squares, 2020. doi:10.48550/ARXIV.2001.02996.



Reordering a Tree According to an Order on Its Leaves

Laurent Bulteau  

LIGM, Université Gustave Eiffel & CNRS, Champs-sur-Marne, France

Philippe Gambette¹  

LIGM, Université Gustave Eiffel & CNRS, Champs-sur-Marne, France

Olga Seminck  

Lattice (Langues, Textes, Traitements informatiques, Cognition),
CNRS & ENS/PSL & Université Sorbonne nouvelle, France

Abstract

In this article, we study two problems consisting in reordering a tree to fit with an order on its leaves provided as input, which were earlier introduced in the context of phylogenetic tree comparison for bioinformatics, OTCM and OTDE. The first problem consists in finding an order which minimizes the number of inversions with an input order on the leaves, while the second one consists in removing the minimum number of leaves from the tree to make it consistent with the input order on the remaining leaves. We show that both problems are NP-complete when the maximum degree is not bounded, as well as a problem on tree alignment, answering two questions opened in 2010 by Henning Fernau, Michael Kaufmann and Mathias Poths. We provide a polynomial-time algorithm for OTDE in the case where the maximum degree is bounded by a constant and an FPT algorithm in a parameter lower than the number of leaves to delete. Our results have practical interest not only for bioinformatics but also for digital humanities to evaluate, for example, the consistency of the dendrogram obtained from a hierarchical clustering algorithm with a chronological ordering of its leaves. We explore the possibilities of practical use of our results both on trees obtained by clustering the literary works of French authors and on simulated data, using implementations of our algorithms in Python.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases tree, clustering, order, permutation, inversions, FPT algorithm, NP-hardness, tree drawing, OTCM, OTDE, TTDE

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.24

Related Version *Previous Version*: <https://hal.archives-ouvertes.fr/hal-03413413v1>

Supplementary Material *Software (Source Code)*: https://github.com/oseminck/tree_order_evaluation; archived at `swh:1:dir:84e3aae3efa08e4a2d296abdda6c34da0c6fca6b`

Funding *Philippe Gambette*: “Investissements d’avenir” program, reference ANR-16-IDEX-0003 (I-Site Future, programme “Cité des dames, créatrices dans la cité”).

Olga Seminck: “Investissements d’avenir” program, reference ANR19-P3IA-0001 (PRAIRIE 3IA Institute).

1 Introduction

The problem of optimizing the consistency between a tree and a given order on its leaves was first introduced in bioinformatics in the context of visualization of multiple phylogenetic trees in order to highlight common patterns in their subtree structure [6], under the name “one-layer STOP (stratified tree ordering problem)”. The authors provided an $O(n^2)$ time

¹ corresponding author



algorithm to minimize, by exchanging the left and right children of internal nodes, the number of inversions between the left-to-right order of the leaves of a binary tree and an input order on its leaves. The problem was renamed OTCM (ONE-TREE CROSSING MINIMIZATION) in [9], where an $O(n \log^2 n)$ time algorithm is provided, as well as a reduction to 3-HITTING SET of a variant of the problem where the goal is to minimize the number of leaves to delete from the tree in order to be able to perfectly match the input order on the remaining leaves, called OTDE (ONE-TREE DRAWING BY DELETING EDGES). An $O(n \log^2 n / \log \log n)$ time algorithm is later provided for OTCM by [1], improved independently in 2010 by [10] and [22] to obtain an $O(n \log n)$ time complexity. About OTDE, the authors of [10] note that “the efficient dynamic-programming algorithm derived for the related problem OTCM [...] cannot be transferred to this problem. However, we have no proof for NP-hardness for OTDE nor TTDE, either”. TTDE (TWO-TREE DRAWING BY DELETING EDGES) is a variant of OTDE where two leaf-labeled trees are provided as input and the goal is to delete the minimum number of leaves such that the remaining leaves of both trees can be ordered with the same order. We give below an answer to both sentences, providing a dynamic-programming algorithm solving OTDE for trees with fixed maximum degree as well as an NP-hardness proof in the general case for OTDE and for TTDE.

Although this problem was initially introduced in the context of comparing tree embeddings, one tree having its embedding (that is the left-to-right order of all children) fixed, we can note that only the order on the leaves of the tree with fixed embedding is useful to define both problems OTCM and OTDE. Both problems therefore consist not really in comparing trees but rather in reordering the internal nodes of one tree in order to optimize its consistency with an order on its leaves provided as input. A popular problem consisting in finding an optimal order on the leaves of a tree is “seriation”, often used for visualization purposes [7], where the optimized criterion is computed on data used to build the tree. For example, a classical criterion, called “optimal leaf ordering”, is to maximize the similarity between consecutive elements in the optimal order [2, 3, 4]. Another possibility is to minimize a distance criterion, the “bilateral symmetric distance”, computed on pairs of elements in consecutive clusters [5]. Seriation algorithms have been implemented for example in the R-packages `seriation` [12] and `dendsort` [19].

With the OTCM and OTDE problems, our goal is not to reorder a tree using only the original data from which it has been built, but using external data about some expected order on its leaves. In the context where the leaves of the tree can be ordered chronologically, for example, this would help providing an answer to the question: how much is this tree consistent with the chronological order? This issue is relevant for several fields of digital humanities, when objects associated with a publication date are classified with a hierarchical clustering algorithm, for example literature analysis [14], political discourse analysis [15] or language evolution [17], as noticed in [11]. In these articles, the comments about the chronological signal which can be observed in the tree obtained from the clustering algorithm are often unclear or imprecise. For example, in [17], the author observes about Figure 15 on page 17 that “the cluster tree gives a visual representation consistent with what is independently known of the chronological structure of the corpus”. However, the structure of the tree does not perfectly reflect the chronology². The algorithms solving the OTCM and OTDE problems can also prevent researchers from claiming having obtained perfect chronological trees with clustering, whereas there are still small inconsistencies that are not easy to spot

² For example `1380Gawain.txt` cannot be ordered between `1375AllitMorteArthur.txt` and `1400YorksPlays.txt`.

with the naked eye. For example, although “Chez Jacques Chirac, l’examen des parentés [dans ses discours de vœux] ne suppose aucune rupture, la chronologie étant parfaitement représentée”³ is claimed about Figure 2.4 in [15], the 1999 speech cannot be ordered between 1998 and 2000.

In this article, we first give useful definitions in Section 1.1. We answer two open problems from [10], proving that OTDE and TTDE are NP-complete, as well as OTCM, in Section 2. We then provide a dynamic programming algorithm solving OTDE in polynomial time for trees with fixed maximum degree in Section 3. This algorithm also works in the more general case where the order on the leaves is not strict. We then provide an FPT algorithm for the OTDE problem parameterized by the deletion-degree of the solution, which is lower than the number of leaves to delete, in Section 4. We also give an example of a tree and an order built to have a distinct solution for the OTCM and OTDE problems in Section 5. Finally, we illustrate the relevance of this problem, and of our implementations of algorithms solving them, for applications in digital humanities, with experiments on trees built from literary works, as well as simulated trees, in Section 6.

1.1 Definitions

Given a set X of elements, we define an X -tree T as a rooted tree whose leaves are bijectively labeled by the elements of X . The set of leaves of T is denoted by $L(T)$ and the set of leaves below some vertex v of T is denoted by $L(T, v)$ (or simply $L(v)$ if T is clear from the context). A set of vertices of T is *independent* if no vertex of T is an ancestor of another vertex of T .

We say that σ is a strict order on X if it is a bijection from X to $[1..n]$ and that it is a weak order on X if it is a surjection from X to $[1..m]$, where $|X| \geq m$. Given any (strict or weak) order σ , we denote by $a \leq_\sigma b$ the fact that $\sigma(a) \leq \sigma(b)$ and by $a <_\sigma b$ the fact that $\sigma(a) < \sigma(b)$. Considering the elements x_1, \dots, x_n of X such that for each $i \in [1..n - 1]$, $\sigma(x_i) \leq \sigma(x_{i+1})$, we denote by $(x_1 x_2 \dots x_n)$ the (weak or strict) order σ .

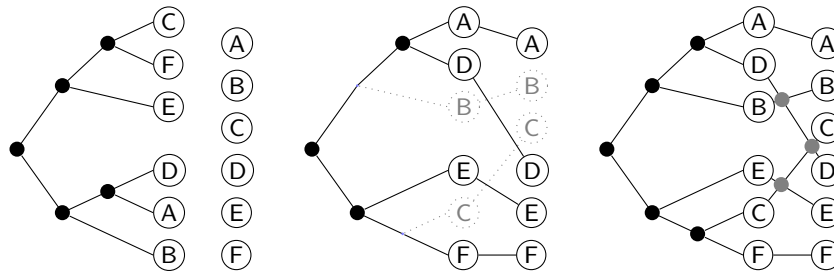
Given an X -tree T and a (weak or strict) order σ on X , we say that an independent pair $\{u, v\}$ of vertices of T is a *conflict wrt. σ* if there exist leaves $a, c \in L(u)$ and $b \in L(v)$ such that $a <_\sigma b <_\sigma c$. Conversely, if $\{u, v\}$ is not a conflict, then either $a \leq_\sigma b$ for all $a \in L(u), b \in L(v)$, or $b \leq_\sigma a$; we then write $u \preceq_\sigma v$ or $v \preceq_\sigma u$, respectively. We say that σ is *suitable* on T if T has no conflict with respect to σ .

Given two (strict or weak) orders σ_1 and σ_2 on X and two elements $a \neq b$ of X , we say that $\{a, b\}$ is an *inversion* for σ_1 and σ_2 if $a \leq_{\sigma_1} b$ and $b <_{\sigma_2} a$, or $b \leq_{\sigma_1} a$ and $a <_{\sigma_2} b$.

Given an X -tree T , a subset X' of X and an order σ on X , we denote by $\sigma[X']$ the order σ restricted to X' , and by $T[X']$ the tree T restricted to X' , that is the X' -tree obtained from T by removing leaves labeled by $X \setminus X'$ and contracting any arc to a non-labeled leaf, any arc from an out-degree-1 vertex. We define the *deletion-degree* of X' as the maximum degree of the tree induced by the deleted leaves, i.e., $T[X \setminus X']$. Intuitively, the deletion-degree measures how deletions in different branches converge on a few nodes or if they merge progressively. Note that by definition, the deletion-degree of X' is upper-bounded both by the maximum degree of T and by the size of $X \setminus X'$.

We now define the two main problems addressed in this paper (see Figure 1 for an illustration). As explained in the introduction, we differ from previous definitions which considered two trees, one with a fixed order on the leaves, as input, as only the leaf order of the second tree is useful to define the problem and not the tree itself.

³ “For Jacques Chirac, the examination of the genealogy [of his new year addresses] shows no discontinuity, the chronology being perfectly represented”



■ **Figure 1** Example for the OTDE and OTCM problem. Left: a tree T on leaves $\{A, \dots, F\}$, the reference permutation is $\sigma = (A, B, C, D, E, F)$ (more precisely, $\sigma(A) = 1, \dots, \sigma(F) = 6$). Middle: a solution for OTDE with cost 2. The subtree $T[X']$ for $X' = \{A, D, E, F\}$ is ordered to show the absence of conflicts with $\sigma[X']$. Right: a solution for OTCM with cost 3. The order $\sigma' = (A, D, B, E, C, F)$ is suitable for T and yields three inversions with σ .

We therefore define the OTCM (ONE-TREE CROSSING MINIMIZATION) problem as follows:

- **Input:** An X -tree T , an order σ on X and an integer k .
- **Output:** Yes if there exists an order σ' on X suitable on T such that the number of inversions for σ' and σ is at most k , no otherwise.

We also define the OTDE (ONE-TREE DRAWING BY DELETING EDGES) problem as follows:

- **Input:** An X -tree T , an order σ on X and an integer k .
- **Output:** Yes if there exists a subset X' of X of size at least $|X| - k$ such that $\sigma[X']$ is suitable on $T[X']$, no otherwise.

We finally define the TTDE (TWO-TREE DRAWING BY DELETING EDGES) problem in the following way:

- **Input:** Two X -trees T_1 and T_2 and an integer k .
- **Output:** Yes if there exists a subset X' of X of size at least $|X| - k$ and an order σ' on X' that is suitable on $T_1[X']$ and on $T_2[X']$, no otherwise.

2 NP-hardness

2.1 OTDE and TTDE are NP-complete for trees with unbounded degree

► **Theorem 1.** *The OTDE problem is NP-complete for strict orders and therefore for weak orders.*

Proof. First note that OTDE is in NP, since, given an X -tree T , an order σ and a set L of leaves to remove, we can check in linear time, by a recursive search of the tree, saving on each node the minimum and the maximum leaf in $\sigma[X - L]$ appearing below, whether $\sigma[X - L]$ is suitable on $T[X - L]$. Regarding NP-hardness, we now give a reduction from INDEPENDENT SET, which is NP-hard on cubic graphs [16], to OTDE when the input trees have unbounded degree.

We consider an instance of the INDEPENDENT SET problem, that is a cubic graph $G = (V = \{v_1, \dots, v_n\}, E)$ such that $|E| = m = 3n/2$ and an integer k . For each vertex v_i , we write e_i^1, e_i^2 and e_i^3 for the three edges incident with v_i (ordered arbitrarily).

We now define an instance of the OTDE problem. The set of leaf labels consists of *vertex labels* denoted v_i and v'_i for each $i \in [1..n]$, one *edge label* for each edge (also denoted e_i^j for the j th edge incident on vertex v_i), and a set of n^2 *separating labels* $B_i = \{b_i^1, b_i^2, \dots, b_i^{n^2}\}$ for each $i \in [1..n-1]$.

First, we define the strict order $\sigma(G) = (v_1 e_1^1 e_1^2 e_1^3 v'_1 b_1^1 b_1^2 \dots b_1^{n^2} v_2 e_2^1 e_2^2 e_2^3 v'_2 b_2^1 b_2^2 \dots b_2^{n^2} v_n e_n^1 e_n^2 e_n^3 v'_n)$. Then, let T_{v_i} be the tree with leaves v_i and v'_i attached below the root, T_e be the tree with leaves $e_i^{j'}$ and $e_j^{i'}$ attached below the root for each edge $e = \{v_i, v_j\}$ of G (with $i', j' \in [1..3]$), and T_{B_i} be the tree with leaves $b_i^1, \dots, b_i^{n^2}$ attached below the root for each $i \in [1..n-1]$. We finally define $T(G)$ as the tree such that $T_{v_1}, T_{v_2}, \dots, T_{v_n}, T_{e_1}, T_{e_2}, \dots, T_{e_m}, T_{B_1}, T_{B_2}, \dots$ and $T_{B_{n-1}}$ are attached below the root.

We claim that G has an independent set of size at least $k \Leftrightarrow$ the instance $(T(G), \sigma(G))$ of the OTDE problem has a solution with a set L of at most $m + n - k$ leaves to remove.

\Rightarrow : Suppose that there exists a size- k independent set $S = \{s_1, \dots, s_k\}$ of G . We then remove the following leaves (also contracting along the way the edge from their parent to the root of $T(G)$) in order to get a new tree T' :

- for each edge $e = \{v_i, v_j\} = e_i^{i'} = e_j^{j'}$ with $i < j$, we remove $e_i^{i'}$ and call $T_{e_j^{j'}} = T_e$ if $v_i \in S$ or if neither v_i nor v_j belong to S ; and we remove $e_j^{j'}$ and call $T_{e_i^{i'}} = T_e$ if $v_j \in S$ (as S is an independent set we cannot have both v_i and v_j in S);
- for each vertex v_i not in S we remove v'_i .

By ordering the children of the root of $T(G)$ such as in Figure 2(1), that is by putting, for each v_i with $i \in [1..n]$, T_{v_i} , then $T_{e_i^1}, T_{e_i^2}$ and $T_{e_i^3}$ for each of the $e_i^{i'}$ which were not removed and then T_{B_i} (except for $i = n$), the order $\sigma(G)$ restricted to the remaining $m + n + k + n^2(n-1)$ leaves is suitable on T' .

\Leftarrow : Suppose that there exists a set L of at most $m + n - k$ leaves such that $\sigma(G)[X - L]$ is suitable on $T(G)[X - L]$. For each parent p_{B_i} of the leaves of B_i and any other vertex v of T such that $\{p_{B_i}, v\}$ is a conflict wrt. $\sigma(G)$, we can delete this conflict either by deleting no leaf of B_i or all leaves of B_i . As each B_i has size $n^2 > m + n - k$, its leaves cannot belong to the set L of leaves to be deleted.

We now consider the trees T_{e_i} for each $i \in [1..m]$: by construction of $\sigma(G)$, as both leaves of each such tree are separated by some $B_{i'}$, therefore by $n^2 > m + n - k$ leaves, one of these two leaves has to be removed, so it has to belong to L . We call L' the set of such leaves of L , therefore there exists a set $L - L'$ of at most $n - k$ other leaves to delete. So there exists a subset S_L of $[1..n]$ of size at least k such that for any element $i \in S_L$, neither v_i , nor v'_i , nor any of the leaves e_i^j for $j \in \{1, 2, 3\}$ belong to $L - L'$. Note that for such $i \in S_L$, all vertices v_i and v'_i are not in L and all e_i^j are in L' . We claim that the vertices of G corresponding to S_L are an independent set of G . Suppose for contradiction that it is not the case, then there exists an edge $e = e_i^{i'} = e_j^{j'}$ between two vertices v_i and v_j of G . By construction of L' , exactly one of the leaves labeled by $e_i^{i'}$ and $e_j^{j'}$ is in L' so the second one is in $L - L'$: contradiction. \blacktriangleleft

► **Corollary 2.** *The TTDE problem is NP-complete.*

Proof. TTDE is clearly in NP. We prove hardness by reduction from OTDE (see Figure 2(2) for an illustration). Consider an instance (T, σ) of OTDE with σ a strict order on n labels X . Introduce a set Y of n new labels. Build T_1 as a caterpillar with $n + 1$ internal nodes forming a path r_1, \dots, r_{n+1} (with root r_1) and $2n$ leaves where each r_i with $i \leq n$ has one leaf attached with label $\sigma^{-1}(i) \in X$ (in the same order), and r_{n+1} has n leaves attached labelled with Y . Build T_2 as a tree, where the root has two children y, t , where y has n children which are leaves labelled with Y , and t is the root of a subtree equal to T .

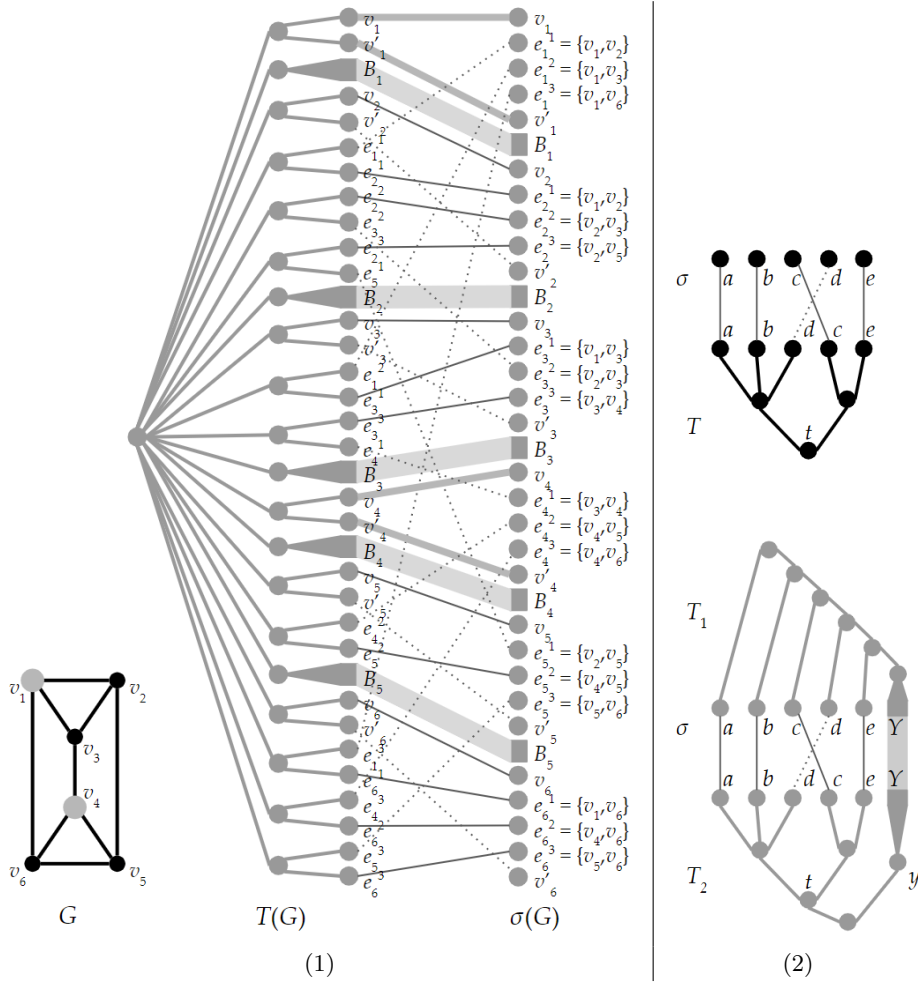


Figure 2 Illustration of the reductions of INDEPENDENT SET to OTDE and of OTDE to TTDE. (1, left) A graph G with independent set $S = \{v_1, v_4\}$ of size 2. (1, right) The corresponding tree $T(G)$ as well as the order $\sigma(G)$. By removing all leaves connected with dotted lines to the corresponding element in $\sigma(G)$, the resulting subtree of $T(G)$ is suitable for the order (since the remaining arcs are non-crossing). (2) Reduction from an OTDE instance (T, σ) (top) to a TTDE instance (T_1, T_2) (bottom). A large set of leaves labelled Y can be seen as a fixed-point, around which T_1 must be ordered according to σ , and T_2 according to the input tree T .

We now show our main claim: given $0 \leq k < n$, $\text{OTDE}(T, \sigma)$ admits a solution with at most k deletions \Leftrightarrow $\text{TTDE}(T_1, T_2)$ admits a solution with at most k deletions.

\Rightarrow Let X' be a size- $(n - k)$ subset of X such that $\sigma[X']$ is suitable on $T[X]$. Then let γ be any order on Y : the concatenation $\sigma[X']\gamma$ is suitable both on $T_1[X' \cup Y]$ and $T_2[X' \cup Y]$, so it is a valid solution for $\text{TTDE}(T_1, T_2)$ of size $2n - k$, i.e., with k deletions.

\Leftarrow Let X', Y' be subsets of X, Y , respectively, and σ' be an order on $X' \cup Y'$ such that σ' is suitable on both $T_1[X' \cup Y']$ and $T_2[X' \cup Y']$, and such that $|X' \cup Y'| \geq 2n - k > n$ (in particular, Y' contains at least one element denoted y , and $|X'| \geq n - k$). From T_2 , it follows that σ' is the concatenation (in any order) of an order σ_x of X' suitable for $T[X']$ and an order σ_y of Y' . Assume first that σ_x appears before σ_y . Then consider each internal node r_i of the caterpillar T_1 with $i \leq n$ and a child c labelled with an element X' . Then this child must be ordered before all leaves below r_{i+1} since the corresponding subtree contains

all leaves labelled with Y . Thus, the nodes in X' are ordered according to $\sigma[X']$, hence $\sigma_x = \sigma[X']$, and $T[X']$ is suitable with $\sigma[X']$. For the other case, where σ_y is ordered before σ_x , then for each r_i with a child in X' , this child must be after the subtree with root r_{i+1} (containing Y), and the nodes in X' are ordered according to the reverse of $\sigma[X']$ (i.e., $\sigma_x = \overline{\sigma[X']}$). Thus, the reverse of $\sigma[X']$ is suitable for $T[X']$, and $\sigma[X']$ as well (this is obtained by reversing the permutation of all children of internal nodes of T). In both cases, X' is a solution for $\text{OTDE}(T, \sigma)$ with $|X'| \geq n - k$. ◀

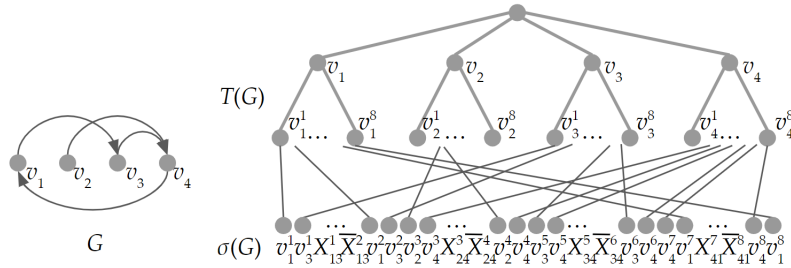
2.2 OTCM is NP-complete for trees with unbounded degree

► **Theorem 3.** *The OTCM problem is NP-complete for strict orders and therefore for weak orders.*

Proof. First note that OTCM is in NP, since, given an X -tree T with its leaves ordered according to an order σ' on X suitable on T , an order σ and a set L of leaves, the number of inversions between σ' and σ can be counted in $O(|L|^2)$. Regarding NP-hardness, we now give a reduction from FEEDBACK ARC SET, which is NP-hard [13], to OTCM.

We consider an instance of the FEEDBACK ARC SET problem, that is a directed graph $G = (V = \{v_1, \dots, v_n\}, A)$ such that $|A| = m$ and an integer f .

We now define an instance of the OTCM problem, illustrated in Figure 3. The set X of leaf labels is $\{v_i^j \mid i \in [1..n], j \in [1..2m]\}$. We define the order $\sigma(G)$ in the following way. For each arc (v_i, v_j) of G , whose rank in the lexicographic order is k , we add to $\sigma(G)$ a k^{th} supplementary ordered sequence (which we will later call a “block” corresponding to this arc) $v_i^{2k-1} v_j^{2k-1} X_{i,j}^{2k-1} \overline{X}_{i,j}^{2k} v_i^{2k} v_j^{2k}$, where $X_{i,j}^{k'}$ is the ordered sequence of $v_{i'}^{k'}$ where i' ranges from 1 to n , excluding i and j , and $\overline{X}_{i,j}^{k'}$ is the reverse of $X_{i,j}^{k'}$ (i.e., the ordered sequence of $v_{i'}^{k'}$ where i' ranges from n down to 1, excluding i and j). The tree $T(G)$ is made of a root with n children v_1 to v_n , each v_i having $2m$ children, the leaves labeled by $v_i^{k'}$ for $k' \in [1..2m]$.



■ **Figure 3** Illustration of the reduction of FEEDBACK ARC SET to OTCM: a graph G with feedback arc set $S = \{(v_4, v_1)\}$ of size 1 and the corresponding tree $T(G)$ as well as the order $\sigma(G)$.

Given an ordering σ' suitable for T , and an inversion $(v_i^k, v_{i'}^{k'})$ forming an inversion between $\sigma(G)$ and σ' , we say that this pair is *short-ranged* if $k = k'$, and *long-ranged* otherwise. Furthermore, we say that σ' is *vertex-consistent* if, for every i and $k < k'$, we have $\sigma'(v_i^k) < \sigma(v_i^{k'})$. Finally, given σ' , we write σ'' for the permutation of the $[1..n]$ corresponding to the children of the root.

We first claim that for any σ' suitable for T , there are at least $2 \binom{n}{2} \binom{2m}{2}$ long-range inversions between σ' and $\sigma(G)$, and this bound is reached if σ' is vertex-consistent. Indeed, pick any pair $(v_i^k, v_{i'}^{k'})$ with $i \neq i'$ and $k \neq k'$. Then $v_i^k <_{\sigma(G)} v_{i'}^{k'}$ iff $k < k'$ (since they are in blocks k and k' of $\sigma(G)$), respectively, and $v_i^k <_{\sigma'} v_{i'}^{k'}$ iff $\sigma''(i) < \sigma''(i')$ (since they are in $L(T, v_i)$ and $L(T, v_{i'})$, respectively). Overall, among $4 \binom{n}{2} \binom{2m}{2}$ such pairs of elements, there

are $2 \binom{n}{2} \binom{2m}{2}$ pairs creating an inversion (which is long-range by definition). For the case $i = i'$, note that pairs $(v_i^k, v_i^{k'})$ do not create any inversion iff σ' is vertex-consistent, which completes the proof of the claim.

Towards counting the number of short-ranged inversions, we say that an arc (v_i, v_j) of G is *satisfied* by σ'' if $\sigma''(i) < \sigma''(j)$. Let $i, j \in [1..n]$ and $k \in [1..m]$, and consider the two pairs (v_i^{2k-1}, v_j^{2k-1}) and (v_i^{2k}, v_j^{2k}) . Then these two pairs are, by construction of T , in the same order in σ' (as defined by σ''). If the k^{th} arc of G is (v_i, v_j) , then these two pairs are also in the same order in σ , i.e., together they account for either 0 or 2 (short-ranged) inversions. More precisely they yield 0 short-ranged inversions if (v_i, v_j) is satisfied by σ'' , and 2 inversions otherwise. If the k^{th} arc of G is any other arc, then exactly one of (v_i^{2k-1}, v_j^{2k-1}) , (v_i^{2k}, v_j^{2k}) forms a short-ranged inversion. Overall a pair $\{i, j\}$ such that one of $(v_i, v_j), (v_j, v_i)$ is a satisfied arc yields $m - 1$ short-ranged inversions, a pair $\{i, j\}$ such that one of $(v_i, v_j), (v_j, v_i)$ is an unsatisfied arc yields $m + 1$ short-range inversions, and any other pair $\{i, j\}$ with $i \neq j$ yields m short-ranged inversions. Overall, if there are f unsatisfied arcs, σ' yields $\binom{n}{2}m - m + 2f$ inversions.

We can now complete the proof with our main claim: G has a feedback arc set of size at most $f \Leftrightarrow$ the OTCM problem has a solution with at most $2 \binom{n}{2} \binom{2m}{2} + \binom{n}{2}m - m + 2f$ inversions.

\Rightarrow : If G has a feedback arc set F of size f , as $G[A - F]$ is acyclic, we consider an order σ'' over n such that for all arcs (v_i, v_j) in $A - F$, $\sigma''(i) < \sigma''(j)$ (i.e., σ'' is the topological order of the vertices in $G[A - F]$). We now order the children v_i of the root of $T(G)$ according to this order σ'' and call σ' the induced order on the leaves of $T(G)$ (also sorting all leaves v_i^j below each v_i by increasing values of j). Note that σ' is vertex-consistent, and that an arc (v_i, v_j) is satisfied by σ'' iff $(v_i, v_j) \notin F$. Thus, σ' yields $2 \binom{n}{2} \binom{2m}{2} + \binom{n}{2}m - m + 2f$ inversions.

\Leftarrow : Consider an order σ' suitable for T with at most $2 \binom{n}{2} \binom{2m}{2} + \binom{n}{2}m - m + 2f$ inversions. Let σ'' be the corresponding order on the leaves of the root, and let F be the set of arcs unsatisfied by σ'' . Since σ' has at least $2 \binom{n}{2} \binom{2m}{2}$ long-range inversions, it has at most $\binom{n}{2}m - m + 2f$ short-range inversions, and $|F| \leq f$. Finally, since all arcs in $A - F$ are satisfied by σ'' , $G[A - F]$ is acyclic and F is a feedback arc set. \blacktriangleleft

3 A polynomial-time algorithm for fixed-degree trees

We start by presenting a dynamic programming algorithm for fixed-degree trees, which is easy to implement and leads to an algorithm in $O(n^4)$ time for binary trees. The FPT algorithm presented in the next section has a better complexity but is more complex and reuses the dynamic programming machinery presented in this section, which explains why we start with this simpler algorithm.

► Theorem 4. *The OTDE problem can be solved in time $O(d!n^{d+2})$ for trees with fixed maximum degree d and for strict or weak orders.*

Proof. Given a vertex v of a rooted tree T , a (strict or weak) order $\sigma : L(T) \rightarrow [1..m]$ and two integers $l \leq r \in [1..m]$. We denote by $\mathcal{X}(v, l, r)$ a subset of $L(T, v)$ of maximum size such that $\sigma[\mathcal{X}(v, l, r)]$ is suitable with $T[\mathcal{X}(v, l, r)]$ and $\forall \ell \in \mathcal{X}(v, l, r), \sigma(\ell) \in [l, r]$. Note that $\mathcal{X}(v, l, r)$ also depends on T and σ but we simplify the notation by not mentioning them as they can clearly be identified from the context.

Denoting by c_1, \dots, c_k the children of v in T , we claim that the following formula allows to recursively compute $\mathcal{X}(v, l, r)$ in polynomial time:

- $|\mathcal{X}(v, l, r)| = \max_{\substack{\text{permutation } \pi \text{ of } [1..k] \\ x_1=l \leq x_2 \leq \dots \leq x_k \leq x_{k+1}=r}} \sum_{i=1}^k |\mathcal{X}(c_{\pi(i)}, x_i, x_{i+1})|$ if v is an internal node of T ;
- for any leaf ℓ of T , $|\mathcal{X}(\ell, l, r)| = 1$ if $\sigma(\ell) \in [l, r]$, 0 otherwise.

Correctness. We prove by induction on the size of $L(v)$ that $\mathcal{X}(v, l, r)$ is indeed a subset of $L(T, v)$ of maximum size such that $\sigma[\mathcal{X}(v, l, r)]$ is suitable with $T[\mathcal{X}(v, l, r)]$ and $\forall \ell \in \mathcal{X}(v, l, r), \sigma(\ell) \in [l, r]$.

This is obvious for any leaf, so let us consider a vertex v of T with a set $\{c_1, \dots, c_k\}$ of children. Suppose for contradiction that there exists a set of integers $l \leq r$ and a subset X' of $L(v)$ of size strictly greater than $\mathcal{X}(v, l, r)$ such that $\sigma[X']$ is suitable with $T[X']$ and $\forall \ell \in X', \sigma(\ell) \in [l, r]$. We then denote by X'_1, \dots and X'_k the sets of leaves $L(c_1) \cap X', \dots$ and $L(c_k) \cap X'$, respectively. Without loss of generality we consider that the children c_i of v are labeled such that $\max_{\ell \in X'_i} \{\sigma(\ell)\} \leq \min_{\ell \in X'_{i+1}} \{\sigma(\ell)\}$. For all $i \in [2..k]$, we define $m_i = \min_{\ell \in X'_i} \{\sigma(\ell)\}$, $m_1 = l$ and $m_{k+1} = r$. Using the induction hypothesis we know that for each $i \in [1..k]$, $|X'_i| \leq |\mathcal{X}(v, \min_{\ell \in X'_i} \{\sigma(\ell)\}, \max_{\ell \in X'_i} \{\sigma(\ell)\})|$, so $|X'_i| \leq |\mathcal{X}(v, m_i, m_{i+1})|$ because $[\min_{\ell \in X'_i} \{\sigma(\ell)\}, \max_{\ell \in X'_i} \{\sigma(\ell)\}] \subseteq [m_i, m_{i+1}]$. Therefore, $|X'| = \sum_{i=1}^k |X'_i| \leq \sum_{i=1}^k |\mathcal{X}(v, m_i, m_{i+1})|$ so by definition of $\sigma[\mathcal{X}(v, l, r)]$, $|X'| \leq \sigma[\mathcal{X}(v, l, r)]$: contradiction!

We therefore obtain a correct solution of $OTDE(T, \sigma)$ by computing $\mathcal{X}(\text{root}(T), 0, m)$.

Running-time. For each v , we compute the table of the $O(n^2)$ values of $\mathcal{X}(v, l, r)$ for all intervals $[l, r]$. Each of these values can be computed by generating the $k!$ permutations of children of v to consider any possible order among the children and splitting the interval $[l, r]$ into any possible configurations of d consecutive intervals with integer bounds partitioning $[l, r]$, which can be done in time $O(n^{d-1})$. So the computation of each $\mathcal{X}(v, l, r)$ is done in time $O(d!n^{d-1})$, therefore the total computation of all $\mathcal{X}(v, l, r)$ is done in time $O(n \times n^2 \times d!n^{d-1})$, that is in $O(d!n^{d+2})$. ◀

4 An FPT algorithm for the *deletion-degree* parameter for OTDE

We recall that with a reduction of OTDE to 3-HITTING SET [10], using the best algorithm known so far to solve this problem⁴, we can obtain an algorithm to solve OTDE $O^*(2.08^k)$ [23], where k is the number of leaves to delete and the O^* notation ignores the polynomial factor. In this section we obtain an FPT algorithm in time $O(n^4 d \partial 2^\partial)$, where d is the maximum degree of the tree and ∂ is the deletion-degree of the solution.

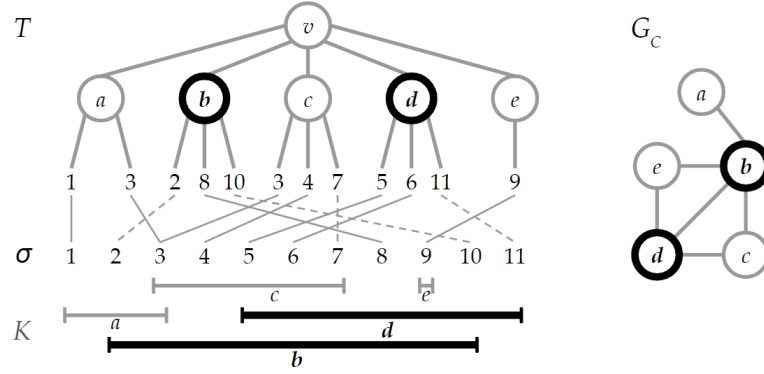
► **Theorem 5.** *The OTDE problem parameterized by the deletion-degree ∂ of the solution is FPT and can be solved in time $O(n^4 d \partial 2^\partial)$ for strict or weak orders.*

We adapt the dynamic programming algorithm from Theorem 4, using a vertex cover subroutine to have a good estimation of the permutation of the children of each node.

We first introduce some definitions (see Figure 4 for an illustration of these definitions and the algorithm in general). Given any vertex v of T , let C_v be the (independent) set of children of v , and let G_v be the *conflict graph* with vertex set C_v and with one edge per conflict. Let K be a vertex cover of G_v . Then the vertices of $C_v \setminus K$ have a *canonical order* $(w_1, \dots, w_{k'})$, with $k' = |C_v \setminus K|$ and $w_i \preceq_\sigma w_j$ for all $i \leq j$ (ties may happen when two children contain a single leaf each which are equal, such ties are broken arbitrarily). We say that $P \subseteq C_v$ is a *prefix of C_v wrt. K* if $P \setminus K$ is a prefix of this order (i.e., for some $i \leq k'$, $P \setminus K = \{w_1, \dots, w_i\}$). In other words, ignoring all subtrees below vertices of K , all leaves below vertices of a prefix P are necessarily ordered before leaves below vertices outside of P .

⁴ <http://fpt.wikidot.com/fpt-races>

24:10 Reordering a Tree According to an Order on Its Leaves



■ **Figure 4** An instance (T, σ) of OTDE (top-left), with a vertex v having children set $C_v = \{a, b, c, d, e\}$. The conflict graph of C_v (right) has a size-2 vertex cover $K = \{b, d\}$. Based on the span of each vertex (bottom-right), the dynamic programming algorithm tests permutations of C_v such that (a, c, e) appear in this order, interleaved in any possible way with b and d . In particular, the final solution corresponds to the permutation $(a c d b e)$ of C_v . Note that since σ may be a weak order (two leaves are labelled 3 in the example), the conflict graph does not correspond exactly to the intersection graph of the span intervals, e.g. vertices a and c are *not* in conflict, even though their spans overlap.

► **Lemma 6.** *If X' is a solution of OTDE with deletion-degree ∂ , then for any vertex v of T , the conflict graph G_v admits a vertex cover of size at most ∂ .*

Proof. Given a subset X' of X , we say that a node v of T has a deletion if some $L(v) \not\subseteq X'$, i.e., if v has a leaf in $X \setminus X'$. Let $\{u, v\}$ be any conflict (edge) of the conflict graph G_v , then at least one of u, v has a deletion for X' (indeed, the conflict involves three leaves a, b, c , of which at least one must be deleted). Thus, the vertices with a deletion in G_v form a vertex cover of this graph. The lemma follows from the fact that at most ∂ vertices have a deletion in each conflict graph. ◀

The first step of our algorithm consists in computing, for each node v of the graph, the set C of children of v , its conflict graph G_v , and a minimum vertex cover K_v of G_C . Since each K_v has size at most ∂ (by Lemma 6), K_v can be computed in time $O(1.3^\partial + \partial n)$ [5], and overall this first step takes $O(1.3^\partial n + \partial n^2)$.

We proceed with the dynamic programming part of our algorithm. To this end, we generalize the table \mathcal{X} to sets of nodes (instead of only v) as follows: $\mathcal{X}(P, l, r)$ corresponds to the largest set X of leaves in $\bigcup_{u \in P} L(u)$ such that σ_X is suitable for $T[X]$. Note that for a node v with children set C , $\mathcal{X}(v, l, r) = \mathcal{X}(\{v\}, l, r) = \mathcal{X}(C, l, r)$.

We first compute $\mathcal{X}(\{v\}, l, r)$ for each leaf v : clearly $\mathcal{X}(\{v\}, l, r) = \{u\}$ if $l \leq \sigma(v) \leq r$, and $\mathcal{X}(\{v\}, l, r) = \emptyset$ otherwise. For each internal vertex v (visiting the tree bottom-up), we obtain $\mathcal{X}(\{v\}, l, r)$ by first computing $\mathcal{X}(P, l, r)$ for each prefix P of C_v by increasing order of size, using the following formulas:

$$\begin{aligned}
 |\mathcal{X}(P, l, r)| &= \emptyset \text{ if } P = \emptyset \\
 &= \max_{\substack{x \in [l, r], u \in P \\ P \setminus \{u\} \text{ prefix of } C_v}} |\mathcal{X}(P \setminus \{u\}, l, x)| + |\mathcal{X}(\{u\}, x, r)| \\
 |\mathcal{X}(\{v\}, l, r)| &= |\mathcal{X}(C_v, l, r)|.
 \end{aligned}$$

Each vertex v has at most $d2^\partial$ prefixes, so the dynamic programming table \mathcal{X} has at most $n^3 d^2 2^\partial$ cells to fill. For each prefix P , there exist at most $\partial + 1$ vertices $u \in P$ such that $P \setminus \{u\}$ is a prefix (u can be any vertex in $P \cap K_v$, or the maximum vertex for \preceq_σ in $P \setminus K_v$). Overall, the *max* is taken over $O(n\partial)$ elements, and \mathcal{X} can be filled in time $O(n^4 d \partial 2^\partial)$.

Before proving the correctness of the above formula, we need a final definition: given a set of leaves $X' \subseteq X$ and a vertex v of T , we write $\text{span}_{X'}(v)$ for the smallest interval containing $\sigma(u)$ for each leaf $u \in L(u) \cap X'$ (note that $\text{span}_{X'}(v)$ may be empty, if all its leaves are deleted in X').

► **Lemma 7.** *Let X' be a solution of $\text{OTDE}(T, \sigma)$, $v \in T$ and $1 \leq l \leq r \leq m$ such that $\text{span}_{X'}(v) \subseteq [l, r]$. Then there exists a permutation $(c_1 \dots c_k)$ of the children of v and integers $x_0 = l \leq x_1 \leq \dots \leq x_k = r$ such that, for each $i \leq k$,*

(a) $\text{span}_{X'}(c_i) \subseteq [x_{i-1}, x_i]$, and

(b) $P_i = \{c_1, \dots, c_i\}$ is a prefix of the children of v wrt. σ .

Proof. Recall that we write C_v and K_v , respectively, for the set of children of v and the vertex cover in the conflict graph induced by these children. For each element c of C_v with a non-empty span, let $x(c) = \max(\text{span}(c))$. For each element w_i of $C_v \setminus K_v$ with an empty span (taking i for the rank according to the canonical order), let $x(w_i) = x(w_{i-1})$ (and $x(w_1) = l$ for $i = 1$). For the remaining vertices (in K_v with an empty span), set $x(c) = l$. Finally, order vertices c_1, \dots, c_k by increasing values of $x(c_i)$ (breaking ties according to the canonical order when applicable, or arbitrarily otherwise), and set $x_i = x(c_i)$.

Condition (a) follows from the fact that X' is a solution for $\text{OTDE}(T, \sigma)$, so that the span covered by the leaves under siblings do not overlap. For condition (b) we refer to the definition of prefix: each $P_i \setminus K_c$ is indeed a prefix in the canonical ordering of $C_v \setminus K_v$. ◀

The dynamic programming formula follows from the above remark: one can build the solution by incrementing prefixes one vertex at a time (rather than trying all possible permutations of children, as in Theorem 4).

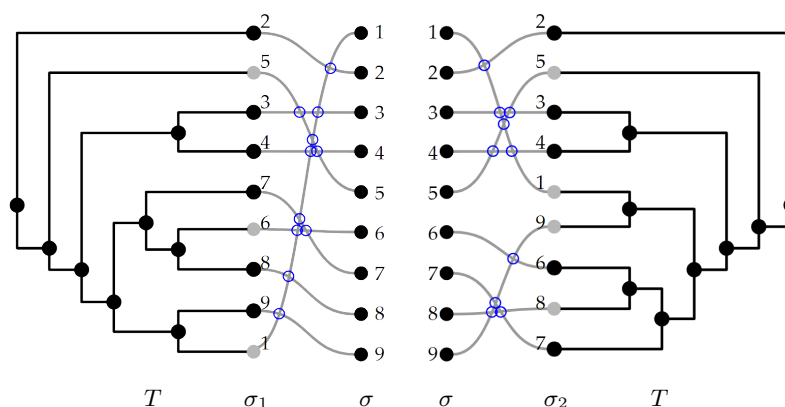
5 Optimizing OTCM and OTDE are two different things

In order to ensure that finding the smallest k such that OTCM or OTDE outputs a positive answer actually consists in optimizing different criteria, we provide in Figure 5 an example of X -tree and an order of its leaves where the order reaching the best k for a positive answer of the OTCM problem does not provide the optimal value for the number of leaves to delete in a positive answer of OTDE and where the best k for a positive answer of the OTDE problem does not provide an optimal value for the number of inversions for a positive answer of the OTCM problem.

We checked the optimality for both criteria by implementing the “naive” dynamic programming $O(n^2)$ algorithm described in Section 2.1 of [10] to solve the OTCM problem and the $O(n^4)$ algorithm described in Section 3 to solve the OTDE problem on binary trees. Both implementations are available in Python, under the GPLv3 licence, at https://github.com/oseminck/tree_order_evaluation, as well as the file `inputCounterExample1b.txt` containing the Newick encoding for the tree of Figure 5.

6 Experiments and discussion

In this section, we investigate the potential for use of OTCM and OTDE in applications where the tree of elements is obtained from a clustering algorithm taking as input distances between those elements, and where we want to test whether this clustering reflects some



■ **Figure 5** Two planar embeddings of a rooted tree T : the one on the left is optimal for the OTDE problem (deleting the 3 gray leaves makes the order σ suitable on T restricted to the remaining leaves, but the order σ_1 suitable on T has 11 inversions, shown with empty circles, with σ); the other one is optimal for the OTCM problem with the order σ_2 suitable on T having 10 inversions with σ but not for the OTDE problem (4 leaves, for example the 4 gray ones, need to be deleted to make the order σ suitable on T restricted to the remaining leaves).

intrinsic order on the elements, for example the chronological order. We both test the running time of OTCM and OTDE on real data, and the performance of OTDE on simulated data to detect possibly misplaced leaves in the order.

The first experiment deals with text data: the CIDRE corpus [20] that contains the works of 11 French 19th century fiction writers dated by year (every file contains a book that is annotated with its year of writing). We apply hierarchical clustering on the different corpora using the `AgglomerativeClustering` class from the package `sklearn` [18]. Distance matrices on which the clustering is based are obtained by using the relative frequencies of the 500 most frequent tokens⁵ in each corpus. Distance matrices were generated using the R package `stylo` [8], with the `canberra` distance metric. We obtain the results given in Table 1, which provides the running time in milliseconds of the algorithms we implemented to solve OTCM and OTDE. They show that both algorithms on binary trees are quick enough to handle typical instances of the OTCM and the OTDE problems relevant for digital humanities, a few milliseconds for the first one and a few seconds for the second one, for instances of about 50 elements in the tree and in the order.

Investigating precisely whether the numbers of inversions or deleted leaves shown in Table 1 are sufficiently small to reflect consistency with a chronological signal is beyond the scope of this paper. However, we also provide p_{OTCM} and p_{OTDE} , the percentage of cases when the best order on the leaves of the tree has the same number of inversions, or less than the chronological order, among 10000 randomly generated orders for OTCM and 100 randomly generated orders for OTDE, respectively⁶. These numbers illustrate that in all cases, it is unlikely that the observed optimal numbers of inversions or deleted leaves are due

⁵ A token is (a part of) a word form or a punctuation marker. The last sentence would yield the following tokens: [“A”, “token”, “is”, “(”, “a”, “part”, “of”, “)”, “a”, “word”, “form”, “or”, “a”, “punctuation”, “marker”, “.”] Deliberately, we do not use the term “word”, because the word can be seen as a linguistic unit of form and meaning, and henceforward “punctuation marker” would be one word and the period in the end of the sentence would not be one.

⁶ We chose to generate less random orders for OTDE in our simulations, as our algorithm is slower to solve this problem than OTCM.

■ **Table 1** Results of our implementations for problems OTCM and OTDE on binary trees generated from corpora of French novels of the 19th century. Time durations are given in milliseconds.

tree	# leaves	OTCM time	# inversions	p_{OTCM}	OTDE time	# deleted leaves	p_{OTDE}
Ségur	22	1	40	0.24	200	9	1
Féval	23	2	47	0.38	268	8	0
Aimard	24	1	35	0	401	8	0
Lesueur	31	1	48	0	676	13	0
Zévaco	29	1	42	0	727	11	0
Zola	35	2	60	0	1203	9	0
Gréville	36	2	105	0	2211	18	1
Ponson	42	3	167	2.23	3447	18	0
Balzac	59	4	248	0	8292	34	0
Verne	58	3	183	0	13446	27	0
Sand	62	4	283	0	17557	39	1

to chance, as we get equal or smaller values of inversions or deleted leaves on less than 3% of random orders (for Ponson du Terrail the number of inversions is 167 or less for 2.23% of random orders; for one of the 10 000 simulated random orders, it reached as little as 124 inversions). These preliminary results obtained thanks to reasonably small running times open new perspectives in investigating further the practical use of these algorithms, and comparing their results with other methods to search for signals of chronological evolution in textual data [21].

Our second experiment involves simulated data, to check whether, in the case the tree is built to be consistent with the input order, our algorithm finding the minimum of leaves in the tree to remove inconsistencies with the order is able to detect errors that we intentionally add to the order. We produced 100 instances of the OTDE problem, for each chosen value of n , the number of leaves, and $e < n$, the number of errors, in the following manner:

1. we randomly pick n distinct integers from the interval $[0, 999]$, which will be our set X of leaves;
2. we build a distance matrix in which the distance between two elements from X is simply the absolute difference between both; we add some noise to this matrix by adding or subtracting in each cell a random quantity equal to at most 10% of the cell value, obtaining a noisy matrix, from which we build an X-tree T using the `AgglomerativeClustering` class from the package `sklearn`;
3. we randomly pick a set L_e of e leaves in X and replace their value by another integer, randomly chosen from the interval $[0, 999]$, distinct from other leaf labels; σ is the set of leaves ordered by increasing value taking into account these new values;
4. by solving the OTDE problem on T and σ , we compute the minimum set L of leaves to remove to make $\sigma[X - L]$ suitable on $T[X - L]$, and check whether $L = L_e$.

This experiment simulates the situation where we would have dating errors on the elements we clustered in a tree. Note that like in the case of dating errors, the error in our simulation may not change the overall order on the leaves. Table 2 provides, for each chosen values of n and e , the proportion of simulated instances of OTDE where $L = L_e$, that is when our algorithm removed exactly the e leaves whose label had been randomly modified. We can observe that this happens in a majority of cases only when the number of modified leaves is small compared with the total number of leaves (up to 2 for 20 leaves, up to 4 for 50 leaves). Solving OTDE still allows to identify $e - 1$ among the e modified leaves in a majority of cases in all our experiments.

■ **Table 2** Results of the attempts to perfectly detect the set L_e of randomly relabeled leaves in simulated trees (when $L = L_e$); the situation when $|L - L_e| = 1$ corresponds to finding only $e - 1$ leaves among the e randomly relabeled leaves).

$n = \# \text{ leaves}$	$e = \# \text{ errors}$	proportion of cases when $L = L_e$	when $ L - L_e = 1$
20	1	0.79	1
20	2	0.62	0.96
20	3	0.39	0.88
20	4	0.33	0.77
20	5	0.27	0.67
50	1	0.93	1
50	2	0.83	0.99
50	3	0.70	0.98
50	4	0.59	0.91
50	5	0.56	0.90

7 Conclusion and perspectives

In this article, we addressed two problems initially introduced with motivations from bioinformatics, OTCM and OTDE. We stated them in a more simple framework with a tree and an order as input, instead of two trees as was the case when they were introduced, opening perspectives for new practical uses in digital humanities and proving that they are not equivalent. We proved that both problems, as well as a problem on two trees, TTDE, are NP-complete in the general case. We gave a polynomial-time algorithm for OTDE on trees with fixed maximum degree and an FPT algorithm in a parameter possibly smaller than the size of the solution for arbitrary trees.

We also investigated their potential for practical use, checking that the algorithms we implemented with open source code in Python to solve them are well suited for applications in digital humanities in terms of running time. We also observed on simulated data that it is possible to identify a small number of leaves for which there would be an ordering error if the tree is built from distance data derived from an order on its leaves. Future research includes the search for FPT algorithms, with relevant parameters, for OTCM and TTDE.

References

- 1 Mukul S Bansal, Wen-Chieh Chang, Oliver Eulenstein, and David Fernández-Baca. Generalized binary tanglegrams: Algorithms and applications. In *International Conference on Bioinformatics and Computational Biology*, pages 114–125. Springer, 2009. doi:10.1007/978-3-642-00727-9_13.
- 2 Ziv Bar-Joseph, Erik D. Demaine, David K. Gifford, Nathan Srebro, Angèle M. Hamel, and Tommi S. Jaakkola. K-ary clustering with optimal leaf ordering for gene expression data. *Bioinformatics*, 19(9):1070–1078, 2003. doi:10.1093/bioinformatics/btg030.
- 3 Ziv Bar-Joseph, David K Gifford, and Tommi S Jaakkola. Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics*, 17(suppl 1):S22–S29, 2001. doi:10.1093/bioinformatics/17.suppl_1.S22.
- 4 Ulrik Brandes. Optimal leaf ordering of complete binary trees. *Journal of Discrete Algorithms*, 5(3):546–552, 2007. doi:10.1016/j.jda.2006.09.003.
- 5 Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40):3736–3756, 2010. doi:10.1016/j.tcs.2010.06.026.

- 6 Tim Dwyer and Falk Schreiber. Optimal leaf ordering for two and a half dimensional phylogenetic tree visualisation. In *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*, volume 35, pages 109–115, 2004. doi:10.5555/1082101.1082114.
- 7 Denise Earle and Catherine B. Hurley. Advances in dendrogram seriation for application to visualization. *Journal of Computational and Graphical Statistics*, 24(1):1–25, 2015. doi:10.1080/10618600.2013.874295.
- 8 Maciej Eder, Jan Rybicki, and Mike Kestemont. Stylometry with R: a package for computational text analysis. *R Journal*, 8(1):107–121, 2016. URL: <https://journal.r-project.org/archive/2016/RJ-2016-007/index.html>.
- 9 Henning Fernau, Michael Kaufmann, and Mathias Poths. Comparing trees via crossing minimization. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 457–469. Springer, 2005. doi:10.1007/11590156_37.
- 10 Henning Fernau, Michael Kaufmann, and Mathias Poths. Comparing trees via crossing minimization. *Journal of Computer and System Sciences*, 76(7):593–608, 2010. doi:10.1016/j.jcss.2009.10.014.
- 11 Philippe Gambette, Olga Seminck, Dominique Legallois, and Thierry Poibeau. Evaluating hierarchical clustering methods for corpora with chronological order. In *EADH2021: Interdisciplinary Perspectives on Data. Second International Conference of the European Association for Digital Humanities*, Krasnoyarsk, Russia, September 2021. EADH. URL: <https://hal.archives-ouvertes.fr/hal-03341803>.
- 12 Michael Hahsler, Kurt Hornik, and Christian Buchta. Getting things in order: an introduction to the R package seriation. *Journal of Statistical Software*, 25(3):1–34, 2008. doi:10.18637/jss.v025.i03.
- 13 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.
- 14 Cyril Labbé and Dominique Labbé. Existe-t-il un genre épistolaire? Hugo, Flaubert et Maupassant. In *Nouvelles Journées de l'ERLA*, pages 53–85. L'Harmattan, 2013.
- 15 Jean-Marc Leblanc. *Analyses lexicométriques des vœux présidentiels*. ISTE Group, 2016.
- 16 Bojan Mohar. Face covers and the genus problem for apex graphs. *Journal of Combinatorial Theory, Series B*, 82(1):102–117, 2001. doi:10.1006/jctb.2000.2026.
- 17 Hermann Moisl. How to visualize high-dimensional data: a roadmap. *Journal of Data Mining & Digital Humanities*, 2020. doi:10.46298/jdmh.5594.
- 18 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- 19 Ryo Sakai, Raf Winand, Toni Verbeiren, Andrew Vande Moere, and Jan Aerts. dendsort: modular leaf ordering methods for dendrogram representations in R. *F1000Research*, 3, 2014. doi:10.12688/f1000research.4784.1.
- 20 Olga Seminck, Philippe Gambette, Dominique Legallois, and Thierry Poibeau. The corpus for idiolectal research (CIDRE). *Journal of Open Humanities Data*, 7:15, 2021. doi:10.5334/johd.42.
- 21 Olga Seminck, Philippe Gambette, Dominique Legallois, and Thierry Poibeau. The evolution of the idiolect over the lifetime: A quantitative and qualitative study on French 19th century literature, 2022. Under review.
- 22 Balaji Venkatachalam, Jim Apple, Katherine St John, and Dan Gusfield. Untangling tanglegrams: comparing trees by their drawings. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(4):588–597, 2010. doi:10.1109/TCBB.2010.57.
- 23 Magnus Wahlström. *Algorithms, measures and upper bounds for satisfiability and related problems*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, 2007. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-8714>.

A Theoretical and Experimental Analysis of BWT Variants for String Collections

Daive Cenzato  

Department of Computer Science, University of Verona, Italy

Zsuzsanna Lipták  

Department of Computer Science, University of Verona, Italy

Abstract

The extended Burrows-Wheeler-Transform (eBWT), introduced by Mantaci et al. [Theor. Comput. Sci., 2007], is a generalization of the Burrows-Wheeler-Transform (BWT) to multisets of strings. While the original BWT is based on the lexicographic order, the eBWT uses the omega-order, which differs from the lexicographic order in important ways. A number of tools are available that compute the BWT of string collections; however, the data structures they generate in most cases differ from the one originally defined, as well as from each other. In this paper, we review the differences between these BWT variants, both from a theoretical and from a practical point of view, comparing them on several real-life datasets with different characteristics. We find that the differences can be extensive, depending on the dataset characteristics, and are largest on collections of many highly similar short sequences. The widely-used parameter r , the number of runs of the BWT, also shows notable variation between the different BWT variants; on our datasets, it varied by a multiplicative factor of up to 4.2.

2012 ACM Subject Classification Theory of computation → Data compression; Applied computing → Bioinformatics

Keywords and phrases Burrows-Wheeler-Transform, extended BWT, string collections, repetitiveness measures, r , compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.25

Related Version *Full Version*: <https://arxiv.org/abs/2202.13235>

Supplementary Material *Software (Source Code and Data)*: <https://github.com/davidecenzato/BWT-variants-for-string-collections>

Acknowledgements We would like to thank Massimiliano Rossi who supplied us with some cleaned and filtered datasets.

1 Introduction

The Burrows-Wheeler-Transform [9] (BWT) is a fundamental string transformation which is at the heart of many modern compressed data structures for text processing, in particular in bioinformatics [34, 36, 33]. With the increasing availability of low-cost high-throughput sequencing technologies, the focus has moved from single strings to large string collections, such as the 1000 Genomes project [53], 10,000 Genomes Project [44], the 100,000 Human Genome Project [55], the 1001 Arabidopsis Project [54], and the 3,000 Rice Genomes Project (3K RGP) [52]. This has led to a widespread use of compressed data structures for string collections.

Concurrently, ever increasing text sizes have been driving a trend towards ever smaller data structures. The size of BWT-based data structures is typically measured in the number of runs (maximal substrings consisting of the same letter) of the BWT, commonly denoted r . This parameter r has become fundamental as a measure of storage space required by such



© Davide Cenzato and Zsuzsanna Lipták;

licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 25; pp. 25:1–25:18

Leibniz International Proceedings in Informatics



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

■ **Table 1** The different BWT variants on the multiset $\mathcal{M} = \{\text{ATATG, TGA, ACG, ATCA, GGA}\}$. For detailed explanations, see Section 3.

variant	result on example	tools
eBWT	CGGGATGTACGTTAAAAA	pfpebwt [6]
dolEBWT	GGAAACGG\$\$\$TTACTGT\$AAA\$	G2BWT [14], pfpebwt [6], msbwt [28]
mdolBWT	GAGAAGCG\$\$\$TTATCTG\$AAA\$	BCR [3], ropebwt2 [35], nvSetBWT [48], Merge-BWT [50], eGSA [38], eGAP [16], bwt-lcp-parallel [5], gsufsort [37]
concBWT	\$AAGAGGC\$#\$TTACTGT\$AAA\$	BigBWT [8], tools for single-string BWT
colexBWT	AAAGCGG\$\$\$TTACTGT\$AAA\$	ropebwt2 [35]

data structures. Moreover, much recent research effort has concentrated on the construction of data structures which can not only store but query, process, and mine strings in space and time proportional to r [22, 2, 47, 12].

The parameter r is also being increasingly seen as a measure of repetitiveness of the string, with several recent works theoretically exploring its suitability as such a measure, as well as its relationship to other such measures [43, 24, 1].

Several tools exist that compute variants of the BWT for string collections, among these BCR [3], ropebwt2 [35], nvSetBWT [48], msbwt [28], Merge-BWT [50], eGSA [38], BigBWT [8], bwt-lcp-parallel [5], eGAP [16], gsufsort [37], G2BWT [14], and pfpebwt [6]. It should be noted though that, when the input is a collection of strings, it is not completely straightforward how to compute the BWT – since the BWT was originally designed for individual strings. In fact, there exists more than one way to compute a Burrows-Wheeler-type transform for a collection of strings, and it turns out that different tools not only use different algorithms, but they output different data structures. As a first example, in Table 1, we give the BWT variants as computed by 12 tools on a toy example of 5 DNA-strings.

The classical way of computing text indexes of string collections is to concatenate the strings, adding a different end-of-string-symbol at the end of each string, and then computing the index for the concatenated string. This is the method traditionally used for generating classical data structures such as suffix trees and suffix arrays for more than one string, and results in the so-called *generalized suffix tree* resp. *generalized suffix array* (see e.g. [27, 45]). The drawback of this method is an increase in the size of the alphabet, from σ , often a small constant in applications, to $\sigma + k$, where k is the number of elements in the collection, typically in the thousands or even tens or hundreds of thousands. One way to avoid this is to use only conceptually different end-of-string-symbols, i.e. to have only one dollar-sign and apply string input order to break ties. This is the method used e.g. by ropebwt2 [35] and by BCR [3]. Another method to avoid increasing the alphabet is to separate the input strings using the same end-of-string-symbol; in this case, a different end-of-string-symbol has to be added to the end of the concatenated string, to ensure correctness, as e.g. in BigBWT [8]. An equivalent solution is to concatenate the input strings without removing the end-of-line or end-of-file characters, since these act as separators; or to concatenate them without separators and use a bitvector to mark the end of each string. Many studies nowadays use string collections in experiments (e.g. [49, 2, 32]); often the input strings are turned into one single sequence using one of the methods described above, and then the single-string BWT is computed; it is, however, not always stated explicitly which was the method used to obtain one sequence. Underlying this is the implicit assumption that all methods are equivalent.

In 2007, Mantaci et al. [40] introduced the *extended Burrows-Wheeler-Transform* (eBWT), which generalizes the BWT to a multiset of strings. The eBWT, like the BWT, is reversible; moreover, it is independent of the order in which the strings in the collection are presented. This is not true of any of the other methods mentioned above. Note that the eBWT differs from the BWT in several ways, most importantly in the order relation for sorting conjugates: while the BWT uses lexicographic order, the eBWT uses the so-called omega-order. (For precise definitions, see Section 2.)

The only tool up to date that computes the eBWT according to the original definition is `pfpebwt` [6]; all other tools append an end-of-string character to the input strings, explicitly or implicitly, and as a consequence, the resulting data structures differ from the one defined in [40]. Moreover, the output in most cases depends on the input order of the sequences (except for [14], [28], and, using a specific option, [35]). As a further complication, the exact nature of this dependence differs from one data structure to another.

The result is that the BWT variants computed by different tools on the same dataset, or by the same tool on the same dataset but given in a different order, may vary considerably. This variability extends to the parameter r , the number of runs of the BWT. This is all the more important given the fact that r (and the related parameter n/r , the average length of a run) is increasingly being used as a parameter characterizing the dataset itself, namely as a measure of its repetitiveness (see e.g. [12, 2, 7]).

1.1 Our contribution

To the best of our knowledge, this is the first systematic treatment of the different BWT variants in use for collections of strings. Our contributions are:

1. We define five distinct BWT variants which are computed by 12 current tools specifically designed for string collections and formally describe the differences between these, identifying specific intervals to which differences are restricted.
2. We show the influence of the input order on the output, in dependence of the BWT variant.
3. We describe the consequences on the number r of runs of the BWT and give an upper bound on the amount by which the colexicographic order (sometimes referred to as “reverse lexicographic order”) can differ from the optimal order of Bentley et al. [4].
4. We complement our theoretical analysis with extensive experiments, comparing the five BWT variants on eight real-life datasets with different characteristics.

1.2 Related work

This paper deals with tools for string collections, so we did not include any tool that computes the BWT of a single string, such as `libdivsufsort` [42], `sais-lite-lcp` [20], `libsais` [26], `bwtdisk` [17]. Even though, in many cases, these are the tools used for collections of strings, the data structure they compute depends on the method used for turning the string collection into a single string, as explained above. Nor did we include other BWT variants for single strings such as the bijective BWT [23, 30], since, again, these were not designed for string collections.

The Big-xBWT [21] is a tool for compressing and indexing read collections, using the xBWT of Ferragina et al. [18, 19]. In addition to the string collection, it requires a reference sequence as input, in contrast to the other tools. Moreover, the output is not comparable either, since its length can vary – as opposed to all other BWT variants we review, the xBWT

is not a permutation of the input characters but can be shorter, due to the fact that it first maps the input to a tree and then applies the xBWT to it, a BWT-like index for labeled trees, rather than for strings. Likewise, the tool [46] for reference-free xBWT is not included in this review: even though it does not require a reference sequence, it, too, computes the xBWT, which is a data structure that does not fall within the category we focus on. Further, we did not include SPRING [11], a reference-free compressor for FASTQ and FASTA files: even though it employs a BWT-based compressor (BSC) during computation, it does not output the BWT.

There has been considerable interest recently in the parameter r , the number of runs of the BWT: it was put in relation with other measures of repetitiveness in [29], while both [10] and [4] studied the question which permutation of the input strings of the collection results in the lowest value for r . Since the method for concatenating the input strings used in [10] (using the same separator symbol but without an additional end-of-string character) differs from all BWT variants that have been implemented by some tool, we do not include it in this study. The result by Bentley et al. [4], on the other hand, is more general, and we will employ it as a benchmark in our experimental comparisons (see Section 5).

1.3 Overview

We give the necessary definitions in Section 2; note that we assume familiarity of the reader with the Burrows-Wheeler-Transform. In Section 3, we present the BWT variants and analyse their differences. In Section 4 we discuss the effects on the repetitiveness measure r , while our experimental results are presented in Section 5. We draw some conclusions from our study in Section 6. Due to space restrictions, most proofs have been omitted and can be found in the full version, along with the full tables with detailed results on all eight datasets.

2 Preliminaries

Let Σ be a finite ordered alphabet of size σ . We use the notation $T = T[1..n]$ for a string T of length n over Σ , $T[i]$ for the i th character, and $T[i..j]$ for the substring $T[i] \cdots T[j]$ of T , where $i \leq j$; $|T|$ denotes the length of T , and ε the empty string. For a string T over Σ and an integer $m > 0$, T^m denotes the m -fold concatenation of T . A string T is called *primitive* if $T = U^m$ implies $T = U$ and $m = 1$. Every string T can be written uniquely as $T = U^m$, where U is primitive. We refer to U as *root*(T) and to m as *exp*(T), i.e., $T = \text{root}(T)^{\text{exp}(T)}$. A *run* in string T is a maximal substring consisting of the same character; we denote by $\text{runs}(T)$ the number of runs of T . Often, an end-of-string character (usually denoted $\$$) is appended to the end of T ; this character is not element of Σ and is assumed to be smaller than all characters from Σ . Note that appending a $\$$ makes any string primitive.

For two strings S, T , the (*unit-cost*) *edit distance* $\text{dist}_{\text{edit}}(S, T)$ is defined as the minimum number of operations necessary to transform S into T , where an operation can be deletion or insertion of a character, or substitution of a character by another. The *Hamming distance* $\text{dist}_{\text{H}}(S, T)$, defined only if $|S| = |T|$, is the number of positions i such that $S[i] \neq T[i]$.

The *lexicographic order* on Σ^* is defined by $S <_{\text{lex}} T$ if S is a proper prefix of T , or if there exists an index j s.t. $S[j] < T[j]$ and for all $i < j$, $S[i] = T[i]$. The *colexicographic order*, or *colex-order* (referred to as *reverse lexicographic order* in [35, 13]) is defined by $S <_{\text{colex}} T$ if $S^{\text{rev}} <_{\text{lex}} T^{\text{rev}}$, where $X^{\text{rev}} = X[n]X[n-1] \cdots X[1]$ denotes the reverse of the string $X = X[1..n]$. String S is a *conjugate* of string T if $S = T[i..n]T[1..i-1]$ for some $i \in \{1, \dots, n\}$ (also called the *i th rotation* of T).

Given a string $T = T[1..n]$ over Σ , the *Burrows-Wheeler-Transform* [9], $\text{BWT}(T)$, is a permutation of the characters of T , given by concatenating the last characters of the lexicographically sorted conjugates of T . The number of runs of the BWT of string T is denoted $r(T)$, i.e. $r(T) = \text{runs}(\text{BWT}(T))$. To make the BWT uniquely reversible, one can add an index to it marking the lexicographic rank of the conjugate in input. For example, $\text{BWT}(\text{banana}) = \text{nbbaaa}$, hence $r(\text{banana}) = 3$, and the index 4 specifies that the input was the 4th conjugate in lexicographic order. Alternatively, one adds a $\$$ to the end of T , which makes the input unique: $\text{BWT}(\text{banana}\$) = \text{annb}\$aa$. Note that BWT with and without end-of-string symbol can be quite different.

Next we define the *omega-order* [40] on Σ^* : $S \prec_\omega T$ if $\text{root}(S) = \text{root}(T)$ and $\text{exp}(S) < \text{exp}(T)$, or if $S^\omega <_{\text{lex}} T^\omega$ (implying $\text{root}(S) \neq \text{root}(T)$), where T^ω denotes the infinite string obtained by concatenating T infinitely many times. The omega-order relation coincides with the lexicographic order if neither of the two strings is a proper prefix of the other. The two orders can differ otherwise, e.g. $\text{GT} <_{\text{lex}} \text{GTC}$ but $\text{GTC} \prec_\omega \text{GT}$.

Given a multiset of strings $\mathcal{M} = \{T_1, \dots, T_k\}$, the *extended Burrows-Wheeler-Transform*, $\text{eBWT}(\mathcal{M})$ [40], is a permutation of the characters of the strings in \mathcal{M} , given by concatenating the last characters of the conjugates of each T_i , for $i = 1, \dots, k$, listed in omega-order. For example, the omega-sorted conjugates of $\mathcal{M} = \{\text{GTC}, \text{GT}\}$ are: $\text{CGT}, \text{GTC}, \text{GT}, \text{TCG}, \text{TG}$, hence, $\text{eBWT}(\mathcal{M}) = \text{TCTGG}$. Again, adding the indices of the input conjugates, in this case 2, 3, makes the eBWT uniquely reversible.

3 BWT variants for string collections

We identified five distinct transforms, which we list below, that were computed by the programs listed above. Let $\mathcal{M} = \{T_1, \dots, T_k\}$ be a multiset of strings, with total length $N_{\mathcal{M}} = \sum_{i=1}^k |T_i|$. Since several of the data structures depend on the order in which the strings are listed, we implicitly regard \mathcal{M} as a list $[T_1, \dots, T_k]$, and write (\mathcal{M}, π) explicitly for a specific permutation π in which the strings are presented.

1. $\text{eBWT}(\mathcal{M})$: the extended BWT of \mathcal{M} of Mantaci et al. [40]
2. $\text{dolEBWT}(\mathcal{M}) = \text{eBWT}(\{T_i\$ \mid T_i \in \mathcal{M}\})$ (“dollar-eBWT”)
3. $\text{mdolBWT}(\mathcal{M}) = \text{BWT}(T_1\$_1 T_2\$_2 \cdots T_k\$_k)$, where dollars are assumed to be smaller than characters from Σ and $\$_1 < \$_2 < \dots < \$_k$ (“multidollar BWT”)
4. $\text{concBWT}(\mathcal{M}) = \text{BWT}(T_1\$T_2\$ \cdots T_k\#\#)$, where $\# < \$$ (“concatenated BWT”)
5. $\text{colexBWT}(\mathcal{M}) = \text{mdolBWT}(\mathcal{M}, \gamma)$, where γ is the permutation corresponding to the colexicographic (‘reverse lexicographic’) order of the strings in \mathcal{M} .

Because all BWT variants except the eBWT use additional end-of-string symbols as string separators, we refer to these four by the collective term *separator-based BWT variants*. In Table 2 we show the five data structures on our running example of 5 DNA-strings, and give first properties. For ease of exposition and comparison, we replaced all separator-symbols by the same dollar-sign $\$$ for all string separator symbols, even where, conceptually or concretely, different dollar-signs are assumed to terminate the individual strings, as is the case for mdolBWT. Moreover, the concBWT contains one additional character, the final end-of-string symbol, here denoted by $\#$, which is smaller than all other characters; thus, the additional rotation starting with $\#$ is the smallest and results in an additional dollar in the first position of the transform. For ease of comparison, we remove this first symbol from concBWT and replace the $\#$ by $\$$.

■ **Table 2** Overview of properties of the five BWT variants considered in this paper. The colors in the example BWTs correspond to interesting intervals in separator-based variants, see Section 3.2.

BWT variant	example	order of shared suffixes	independent of input order?
<i>non-sep.-based</i> eBWT(\mathcal{M})	CGGGATGTACGTTAAAAA	omega-order of strings	yes
<i>separator-based</i> dolEBWT(\mathcal{M})	GGAAACGG\$\$\$\$TTACTGT\$AAA\$	lexicographic order of strings	yes
mdolBWT(\mathcal{M})	GAGAAACGG\$\$\$\$TTATCTG\$AAA\$	input order of strings	no
concBWT(\mathcal{M})	AAGAGGGC\$\$\$\$TTACTGT\$AAA\$	lexicographic order of subsequent strings in input	no
colexBWT(\mathcal{M})	AAAGCGC\$\$\$\$TTACTGT\$AAA\$	colexicographic order	yes

It is important to point out that the programs listed in Table 1 do not necessarily use the definitions given here; however, in each case, the resulting transform is the one claimed, up to renaming or removing separator characters, see Section 3.1 and 3.2.

3.1 The effect of adding separator symbols

The first obvious difference between the eBWT and the separator-based variants is their length: eBWT(\mathcal{M}) has length $N_{\mathcal{M}}$, while all other variants have length $N_{\mathcal{M}} + k$, since they contain an additional character (the separator) for each input string.

In all four separator-based transforms, the k -length prefix consists of a permutation of the last characters of the input strings. This is because the rotations starting with the dollars are the first k lexicographically; in the eBWT, these k characters occur interspersed with the rest of the transform; namely, in the positions corresponding to the omega-ranks of the input strings T_i (see Table 2).

The next point is that adding a \$ to the end of the strings introduces a distinction, not present in the eBWT, between suffixes and other substrings: since the separators are smaller than all other characters, occurrences of a substring as suffix will be listed en bloc before all other occurrences of the same substring. On the other hand, in the eBWT, these occurrences will be listed interspersed with the other occurrences of the same substring.

► **Example 1.** Let $\mathcal{M} = \{\text{AACGAC}, \text{TCAC}\}$ and $U = \text{AC}$. U occurs both as a suffix and as an internal factor; the characters preceding it are A (internal substring) and C, G (suffix), and we have eBWT(\mathcal{M}) = CGACATAACC, dolEBWT(\mathcal{M}) = CC\$GCAAATAC\$.

Finally, it should be noted that adding end-of-string symbols to the input strings changes the definition of the order applied. As observed above, the omega-order coincides with the lexicographic order on all pairs of strings S, T where neither is a proper prefix of the other; but with end-of-strings characters, no input string can be a proper prefix of another. Thus, on rotations of the T_i 's, the omega-order equals the lexicographic order. As an example, consider the multiset $\mathcal{M} = \{\text{GTC}$, \text{GT}$\}$ from Section 2: we have the following omega-order among the rotations: \$GT, \$GTC, C\$GT, GT\$, GTC\$, T\$G, TC\$G, which coincides with the lexicographic order. Similarly, adding *different* dollars \$₁, \$₂, ..., \$_k and applying the omega-order results again in the lexicographic order between the rotations, with different dollar symbols considered as distinct characters. Indeed, if we append a different dollar-sign to each input string, then the omega-order, the lexicographic order, and the order of the suffixes of the concatenated string (i.e. our mdolBWT) are all equivalent.

Regarding the differences among the four separator-based BWT variants, we will show that all differences occur in certain well-defined intervals of the BWT, and that the differences themselves depend only on a specific permutation of $\{1, \dots, k\}$, given by the combination of the input order, the lexicographic order of the input strings, and the BWT variant applied. In Tables 3 and 4 we give the full BWT matrices for all five BWT variants, along with the optimal one minimizing the number of runs, see Section 4.

■ **Table 3** From left to right we show the mdolBWT, the dolEBWT, and the concBWT of the string collection $\mathcal{M} = \{\text{ATATG}, \text{TGA}, \text{ACG}, \text{ATCA}, \text{GGA}\}$.

index	mdol	rotation	index	dolE	rotation	index	conc	rotation
(1,6)	G	\$ ₁ ATATG	(3,4)	G	\$ACG	23	A	\$\$ATATG\$TGA\$ACG\$ATCA\$GGA
(2,4)	A	\$ ₂ TGA	(1,6)	G	\$ATATG	10	A	\$ACG\$ATCA\$GGA\$#ATATG\$TGA
(3,4)	G	\$ ₃ ACG	(4,5)	A	\$ATCA	14	G	\$ATCA\$GGA\$#ATATG\$TGA\$ACG
(4,5)	A	\$ ₄ ATCA	(5,4)	A	\$GGA	19	A	\$GGA\$#ATATG\$TGA\$ACG\$ATCA
(5,4)	A	\$ ₅ GGA	(2,4)	A	\$TGA	6	G	\$TGA\$ACG\$ATCA\$GGA\$#ATATG
(2,3)	G	A\$ ₂ TG	(4,4)	C	A\$ATC	22	G	A\$#ATATG\$TGA\$ACG\$ATCA\$GG
(4,4)	C	A\$ ₄ ATC	(5,3)	G	A\$GG	9	G	A\$ACG\$ATCA\$GGA\$#ATATG\$TGA
(5,3)	G	A\$ ₅ GG	(2,3)	G	A\$TG	18	C	A\$GGA\$#ATATG\$TGA\$ACG\$ATC
(3,1)	\$ ₃	ACG\$ ₃	(3,1)	\$	ACG\$	11	\$	ACG\$ATCA\$GGA\$#ATATG\$TGA\$
(1,1)	\$ ₁	ATATG\$ ₁	(1,1)	\$	ATATG\$	1	\$	ATATG\$TGA\$ACG\$ATCA\$GGA\$#
(4,1)	\$ ₄	ATCA\$ ₄	(4,1)	\$	ATCA\$	15	\$	ATCA\$GGA\$#ATATG\$TGA\$ACG\$
(1,3)	T	ATG\$ ₁ AT	(1,3)	T	ATG\$AT	3	T	ATG\$TGA\$ACG\$ATCA\$GGA\$#AT
(4,3)	T	CA\$ ₄ AT	(4,3)	T	CA\$AT	17	T	CA\$GGA\$#ATATG\$TGA\$ACG\$AT
(3,2)	A	CG\$ ₃ A	(3,2)	A	CG\$A	12	A	CG\$ATCA\$GGA\$#ATATG\$TGA\$A
(1,5)	T	G\$ ₁ ATAT	(3,3)	C	G\$AC	13	C	G\$ATCA\$GGA\$#ATATG\$TGA\$AC
(3,3)	C	G\$ ₃ AC	(1,5)	T	G\$ATAT	5	T	G\$TGA\$ACG\$ATCA\$GGA\$#ATAT
(2,2)	T	GA\$ ₂ T	(5,2)	G	GA\$G	21	G	GA\$#ATATG\$TGA\$ACG\$ATCA\$GG
(5,2)	G	GA\$ ₅ G	(2,2)	T	GA\$T	8	T	GA\$ACG\$ATCA\$GGA\$#ATATG\$T
(5,1)	\$ ₅	GG\$ ₅	(5,1)	\$	GG\$A	20	\$	GG\$#ATATG\$TGA\$ACG\$ATCA\$
(1,2)	A	TATG\$ ₁ A	(1,2)	A	TATG\$A	2	A	TATG\$TGA\$ACG\$ATCA\$GGA\$#A
(4,2)	A	TCA\$ ₄ A	(4,2)	A	TCA\$A	16	A	TCA\$GGA\$#ATATG\$TGA\$ACG\$A
(1,4)	A	TG\$ ₁ ATA	(1,4)	A	TG\$ATA	4	A	TG\$TGA\$ACG\$ATCA\$GGA\$#ATA
(2,1)	\$ ₂	TGA\$ ₂	(2,1)	\$	TGA\$	7	\$	TGA\$ACG\$ATCA\$GGA\$#ATATG\$

3.2 Interesting intervals

Let us call a string U a *shared suffix* w.r.t. multiset \mathcal{M} if it is the suffix of at least two strings in \mathcal{M} . Let b be the lexicographic rank of the smallest rotation beginning with $U\$$ and e the lexicographic rank of the largest rotation beginning with $U\$$, among all rotations of strings $T\$$, where $T \in \mathcal{M}$. (One can think of $[b, e]$ as the suffix-array interval of $U\$$.) We call $[b, e]$ an *interesting interval* if there exist $i \neq j$ s.t. U is a suffix of both T_i and T_j , and the preceding characters in T_i and T_j are different, i.e., the two occurrences of U as suffix of T_i and T_j constitute a left-maximal repeat. (Interesting intervals correspond to internal nodes in the suffix tree of the reverse string, within the subtree of $\$$.) Clearly, $[1, k]$ is an interesting interval unless all strings end with the same character. Note that interesting intervals differ both from the *SAP-intervals* of [13] and from the *tuples* of [4] (called *maximal row ranges* in [41]): the former are the intervals corresponding to *all* shared suffixes U , even if not left-maximal, while the latter include also suffixes U that are not shared. The next lemma follows from the fact that no two substrings ending in $\$$ can be one prefix of the other.

25:8 An Analysis of BWT Variants of String Collections

■ **Table 4** From left to right we show the eBWT, the colexBWT, and the optimal BWT of the string collection $\mathcal{M} = \{\text{ATATG, TGA, ACG, ATCA, GGA}\}$, see Section 4.

index	eBWT	rotation	index	colexBWT	rotation	index	optimum	rotation
(4,4)	C	AATC	(1,5)	A	\$ ₁ ATCA	(1,4)	A	\$ ₁ TGA
(3,1)	G	ACG	(2,4)	A	\$ ₂ GGA	(2,4)	A	\$ ₂ GGA
(5,3)	G	AGG	(3,4)	A	\$ ₃ TGA	(3,5)	A	\$ ₃ ATCA
(1,1)	G	ATATG	(4,4)	G	\$ ₄ ACG	(4,4)	G	\$ ₄ ACG
(4,1)	A	ATCA	(5,6)	G	\$ ₅ ATATG	(5,6)	G	\$ ₅ ATATG
(1,3)	T	ATGAT	(1,4)	C	A\$ ₁ ATC	(1,3)	G	A\$ ₁ TG
(2,3)	G	ATG	(2,3)	G	A\$ ₂ GG	(2,3)	G	A\$ ₂ GG
(4,3)	T	CAAT	(3,3)	G	A\$ ₃ TG	(3,4)	C	A\$ ₃ ATC
(3,2)	A	CGA	(4,1)	\$	ACG\$ ₄	(4,1)	\$	ACG\$ ₄
(3,3)	C	GAC	(5,1)	\$	ATATG\$ ₅	(5,1)	\$	ATATG\$ ₅
(5,2)	G	GAG	(1,1)	\$	ATCA\$ ₁	(3,1)	\$	ATCA\$ ₃
(1,5)	T	GATAT	(5,3)	T	ATG\$ ₅ AT	(5,3)	T	ATG\$ ₅ AT
(2,2)	T	GAT	(1,3)	T	CA\$ ₁ AT	(3,3)	T	CA\$ ₃ AT
(5,1)	A	GGA	(4,2)	A	CG\$ ₄ A	(4,2)	A	CG\$ ₄ A
(1,2)	A	TATGA	(4,3)	C	G\$ ₄ AC	(4,3)	C	G\$ ₄ AC
(4,2)	A	TCAA	(5,5)	T	G\$ ₅ ATAT	(5,5)	T	G\$ ₅ ATAT
(1,4)	A	TGATA	(2,2)	G	GA\$ ₂ G	(1,2)	T	GA\$ ₁ T
(2,1)	A	TGA	(3,2)	T	GA\$ ₃ T	(2,2)	G	GA\$ ₂ G
			(2,1)	\$	GG\$ ₂	(2,1)	\$	GG\$ ₂
			(5,2)	A	TATG\$ ₅ A	(5,2)	A	TATG\$ ₅ A
			(1,2)	A	TCA\$ ₁ A	(3,2)	A	TCA\$ ₃ A
			(5,4)	A	TG\$ ₅ ATA	(5,4)	A	TG\$ ₅ ATA
			(3,1)	\$	TGA\$ ₃	(1,1)	\$	TGA\$ ₁

► **Lemma 2.** Any two distinct interesting intervals are disjoint.

We can now narrow down the differences between any two separator-based BWTs of the same multiset to interesting intervals. This implies that the dollar-symbols appear in the same positions in all separator-based variants except for one very specific case. Moreover, we get an upper bound on the Hamming distance between two separator-based BWTs:

► **Proposition 3.** Let L_1 and L_2 be two separator-based BWTs of the same multiset \mathcal{M} .

1. If $L_1[i] \neq L_2[i]$ then $i \in [b, e]$ for some interesting interval $[b, e]$.
2. Let \mathcal{I}_1 resp. \mathcal{I}_2 be the positions of the dollars in L_1 resp. L_2 . If $\mathcal{I}_1 \neq \mathcal{I}_2$ then there exist $i \neq j$ such that T_i is a proper suffix of T_j .
3. $\text{dist}_H(L_1, L_2) \leq \sum_{[b,e] \text{ interesting interval}} (e - b + 1)$.

Proof. 1. Let $L_1[i] = x$ and $L_2[i] = y$. Since all separator-based BWT variants use the lexicographical order of the rotations, this means that there exists a substring U which is preceded by x in one string T_j and by y in another $T_{j'}$, the first occurrence has rank i in one BWT and the other has rank i in the other BWT variant. This implies that the two occurrences are followed by two dollars, and either the two dollars are different, or they are the same dollar, and the subsequent substrings are different. Therefore, U defines an interesting interval. Parts 2. and 3. follow from 1. ◀

Proposition 3 implies that the variation of the different transforms can be explained based solely on what rule is used to break ties for shared suffixes. We will see next how the different BWT variants determine this tie-breaking rule.

3.3 Permutations induced by separator-based BWT variants

Let us now restrict ourselves to \mathcal{M} being a set, i.e., no string occurs more than once. (This is just for convenience since now the input order uniquely defines a permutation w.r.t. lexicographic order; the results of this section apply equally to multisets \mathcal{M} .) As we showed in the previous subsection, the only differences between the different separator-based BWT variants are given by the order in which shared suffixes are listed. It is also clear that the same order applies in each interesting interval, as well as to the k -length prefix of the transform, whether or not it is an interesting interval.

Since the strings are all distinct, they each have a unique lexicographic rank within the set \mathcal{M} . Thus the input order can be seen as a permutation ρ of the lexicographic ranks¹; if the strings are input in lexicographic order, then $\rho = id$. For our toy example $\mathcal{M} = [\text{ATATG}, \text{TGA}, \text{ACG}, \text{ATCA}, \text{GGA}]$, we have $\rho = 25134$.

Let us now define as *output permutation* π the permutation of the last characters of the input strings, as found in the k -length prefix of the BWT variant in question. We will denote the output permutations of the dolEBWT, mdolBWT, concBWT, and colexBWT by π_{de} , π_{md} , π_{conc} , and π_{colex} , respectively. Again, we give these permutations w.r.t. the lexicographic ranks of the strings. In our running example, we have $\pi_{de} = 12345$, $\pi_{md} = 25134$, $\pi_{conc} = 45132$, and $\pi_{colex} = 34512$.

It is easy to see that the permutation π_{md} is equal to ρ , since the dollar-symbols are ordered according to ρ . For the dolEBWT, the rank of $\$T_i$ equals the lexicographic rank of T_i among all input strings, i.e., $\pi_{de} = id$. Further, $\pi_{colex} = \gamma$ by definition, where γ denotes the colexicographic order of the input strings. The situation is more complex in the case of concBWT. Since the $\#$ is the smallest character, the last string of the input will be the first, while for the others, the lexicographic rank of the following string decides the order. In our running example, $\pi_{conc} = 45132$. We next formalize this.

Let Φ_ρ be the *linking permutation* [31] of ρ , defined by $\Phi_\rho(i) = \rho(\rho^{-1}(i) + 1)$, for $i \neq \rho(k)$, and $\Phi_\rho(\rho(k)) = \rho(1)$, the permutation that maps each element to the element in the next position and the last element to the first. Let us also define, for $j \in \{1, \dots, k\}$ and $i \neq j$, $f_j(i)$ by $f_j(i) = i$ if $i < j$ and $i - 1$ otherwise. The next lemma gives the precise relationship between ρ and π_{conc} . It says², essentially, that π_{conc} is the BWT of ρ .

► **Lemma 4.** *Let ρ be the permutation of the input order w.r.t. the lexicographic order, i.e. the i th input string has lexicographic rank $\rho(i)$. Then $\pi_{conc} = \pi_{conc}(\rho)$ is given by:*

$$\pi_{conc}(1) = \rho(k), \quad \text{and for } i \neq \rho(k) : \pi_{conc}^{-1}(i) = f_{\rho(1)}(\Phi_\rho(i)) + 1. \quad (1)$$

► **Example 5.** The mapping $\rho \mapsto \pi_{conc}$ for $k = 3$ is as follows: $123 \mapsto 312$, $132 \mapsto 231$, $312 \mapsto 231$, $213 \mapsto 321$, $231 \mapsto 132$, and $321 \mapsto 123$. Note that no ρ maps to 213.

As can be seen already for $k = 3$, not all permutations π are reached by this mapping. We will call a permutation π *feasible* if there exists an input order ρ such that $\pi_{conc}(\rho) = \pi$. For $k = 4$, there are 18 feasible permutations (out of 24), for $k = 5$, 82 (out of 120). In

¹ For those used to thinking about suffix arrays, ρ can be seen as the inverse suffix array of the input if the strings are thought of as meta-characters.

² We thank Massimiliano Rossi for this observation.

25:10 An Analysis of BWT Variants of String Collections

Table 5, we give the percentage of feasible permutations π , for k up to 11. The lexicographic order is always feasible, namely with $\rho = k, k-1, \dots, 2, 1$; however, the colex order is not always feasible, as the following example shows.

► **Example 6.** Let $\mathcal{M} = \{\text{GAA}, \text{ACA}, \text{TGA}\}$, thus $\gamma = 213$, but as we have seen, no permutation of the strings in \mathcal{M} will yield this order for concBWT. In addition, the $\text{colexBWT}(\mathcal{M}) = \text{AAAACGG\$AT\$\$}$ has 7 runs, while all feasible ones have at least 8: $\text{AAAGACG\$AT\$\$}$, $\text{AAACGAG\$AT\$\$}$, $\text{AAAAGCG\$AT\$\$}$, $\text{AAAGCAG\$AT\$\$}$, $\text{AAACAGG\$AT\$\$}$.

■ **Table 5** Percentage of feasible permutations w.r.t. concBWT.

no. of seq's k	3	4	5	6	7	8	9	10	11
	83.33%	75.0%	68.33%	63.89%	60.12%	57.29%	54.8%	52.81%	51.0%

An important consequence is that the permutations induced by mdolBWT and concBWT are always different: $\pi_{md} \neq \pi_{conc}$ holds always, since $\pi_{conc}(1) = \rho(k)$. This means that, in whatever order the strings are given w.r.t. lexicographic order, on most string sets the resulting transforms mdolBWT and concBWT will differ.

4 Effects on the parameter r

What is the effect of the different permutations π of the strings in \mathcal{M} , induced by these BWT variants, on the number of runs of the BWT? As the following example shows, the number of runs can differ significantly between different variants.

► **Example 7.** Let $\mathcal{M} = \{\text{AAAA}, \text{AGCA}, \text{GCAA}, \text{GTCA}, \text{CAAA}, \text{CGCA}, \text{TCAA}, \text{TTCA}\}$. Then $\text{mdolBWT}(\mathcal{M}) = \text{AAAAAAAAACACACACACACAC\$\$GTGTGT\$\$AC\$\$GT\$\$}$ has 28 runs, while $\text{colexBWT}(\mathcal{M}) = \text{AAAAAAAAAAAAACCCCAACCAC\$\$GGTTGT\$\$AC\$\$GT\$\$}$ has 18 runs.

► **Lemma 8.** Let $[b, e]$ be an interesting interval, and (n_1, \dots, n_σ) the Parikh vector of $L[b..e]$, i.e. n_i is the number of occurrences of the i th character. Let \mathbf{a} be such that $n_{\mathbf{a}} = \max_i n_i$, and $N_{\mathbf{a}} = (e - b + 1) - n_{\mathbf{a}}$, the sum of the other character multiplicities. Then the maximum number of runs in interval $[b, e]$ is $e - b + 1$ if $n_{\mathbf{a}} - 1 \leq N_{\mathbf{a}}$, and $2N_{\mathbf{a}} + 1$ otherwise.

We will use this lemma to measure the variability of a dataset:

► **Definition 9.** Let \mathcal{M} be a multiset. For an interesting interval $[b, e]$, let $\text{var}([b, e])$ be the upper bound on the number of runs in $[b, e]$ from Lemma 8. Then the variability of \mathcal{M} is

$$\text{var}(\mathcal{M}) = \frac{\sum_{[b,e] \text{ interesting interval}} \text{var}([b, e])}{\sum_{[b,e] \text{ interesting interval}} (e - b + 1)}.$$

Which of the BWT variants produces the fewest runs? As we have shown, this depends on the input order with most BWT variants, and the only possible variation is within interesting intervals. The colexBWT has been shown experimentally to yield a low number of runs of the BWT [35, 13]. Even though it does not always minimize r (one can easily create small examples where other permutations yield a lower number of runs), we can bound its distance from the optimum.

► **Proposition 10.** Let L be the colexBWT of multiset \mathcal{M} , and let r_{OPT} denote the minimum number of runs of any separator-based BWT of \mathcal{M} . Then $\text{runs}(L) \leq r_{OPT} + 2 \cdot c_{\mathcal{M}}$, where $c_{\mathcal{M}}$ is the number of interesting intervals.

Bentley, Gibney, and Thankachan recently gave a linear-time algorithm for computing the order of the dollars which minimizes the number of runs [4], i.e. the optimal order for `mdolBWT`. The idea is, in effect, to start from the `colex`-order and then adjust, where possible, the order of the runs within interesting intervals in order to minimize character changes at the borders, i.e. such that the first and the last run of each interesting interval is identical to the run preceding and following that interesting interval. This is equivalent to sorting groups of sequences sharing the same left-maximal suffix. This sorting can be done on each interesting interval independently without affecting the other interesting intervals. In Table 4, we show the result on our toy example, where it reduces the number of runs by 2 w.r.t. `colex` order. We implemented an algorithm that computes the number of optimal runs according to the method of [4] and applied it to our datasets. In the next section, we compare the number of runs of each of the five BWT variants to the optimum.

5 Experimental results

We computed the five BWT variants for eight different genomic datasets, with different characteristics. Four of the datasets contain short reads: SARS-CoV-2 short [51], Simons Diversity reads [39], 16S rRNA short [57], Influenza A reads [56], and four contain long sequences: SARS-CoV-2 long [25], 16S rRNA long [15], *Candida auris* reads [58], one of which, SARS-CoV-2 genomes, whole viral genomes [6]. The main features of the datasets, including the number of sequences, sequence length, and the mean runlength of the optimal BWT are reported in Table 6. Details of the experiment setup are included in the full version.

On each of the datasets, we computed the pairwise Hamming distance between separator-based BWTs. To compare them to the `eBWT`, we computed the pairwise edit distance on a small subset of the sequences (for obvious computational reasons), computing also the Hamming distance on the small set, for comparison. We generated some statistics on each of the data sets: the number of interesting intervals, the fraction of positions within interesting intervals (total length of interesting intervals divided by total length of the dataset), and the dataset’s variability (Def. 9). To study the variation of the r -parameter, we implemented the algorithm by Bentley et al. [4] for the optimal input order and computed r_{OPT} for each data set, comparing it to the number of runs of all five BWT variants. In Table 8 and 9, we include a compact version of these results for the two datasets with the highest and the lowest variation between the BWT variants, the SARS-CoV-2 short sequences and the SARS-CoV-2 genomes, respectively. The full experimental results for all eight datasets are contained in the full version.

In Table 7 we give a brief summary of the results, reporting, for each dataset, the fraction of positions in interesting intervals, the dataset’s variability, the average pairwise Hamming distance between separator-based BWT variants, and the maximum and minimum value, among the five BWT variants, of the average runlength of the BWT.

The experiments showed a high variation in the number of runs in particular on datasets of short sequences. The highest difference was between `colexBWT` and `concBWT`, by a multiplicative factor of over 4.2, on the SARS-CoV-2 short dataset. In Figure 1 we plot the average runlength n/r for the four short sequence datasets, and the percentage increase of the number of runs w.r.t. r_{OPT} . The variation is less pronounced on the one dataset which is less repetitive, namely Simons Diversity reads. Recall that the `mdolBWT` and `concBWT` vary depending on the input permutation. On most long sequence datasets, on the other hand, the differences were quite small (see full version). To better understand how far the `colexBWT` is from the optimum, we plot in Figure 2 the number of runs of `colexBWT`

25:12 An Analysis of BWT Variants of String Collections

w.r.t. to r_{OPT} , on all eight datasets. The strongest increase is on short sequences, where the variation among all BWT variants is high, as well; on the long sequence datasets, with the exception of SARS-CoV-2 long sequences, the `colexBWT` is very close to the optimum; however, note that on those datasets, all BWTs are close to the optimum.

The average number of runs and the average pairwise Hamming distance strongly depend on the length of the sequences in the input collection. If the collection has a lot of short sequences which are very similar, then the differences between the BWTs both w.r.t. the number of runs, and as measured by the Hamming distance, can be large. This is because there are a lot of maximal shared suffixes and so many positions are in interesting intervals. To better understand this relationship, we plotted, in Figure 3, the average Hamming distance against the two parameters variability and fraction of positions in interesting intervals. We see that the two datasets with highest average Hamming distance, SARS-CoV-2 short dataset and the Simons Diversity reads, have at least one of the two values very close to 1, while for those datasets where both values are very low, the BWT variants do not differ very much.

■ **Table 6** Table summarizing the main parameters of the eight datasets. From left to right we report the dataset name, the number of sequences, the total length, the average, minimum and maximum sequence length and the optimum average runlength (n/r), according to [4].

dataset	no. seq	total length	avg	min	max	n/r (opt)
SARS-CoV-2 short	500,000	25,000,000	50	50	50	35.125
Simons Diversity reads	500,000	50,000,000	100	100	100	8.133
16S rRNA short	500,000	75,929,833	152	69	301	44.873
Influenza A reads	500,000	115,692,842	231	60	251	50.275
SARS-CoV-2 long	50,000	53,726,351	1,075	265	3,355	74.498
16S rRNA long	16,741	25,142,323	1,502	1,430	1,549	47.140
Candida auris reads	50,000	124,150,880	2,483	214	8,791	1.732
SARS-CoV-2 genomes	2,000	59,610,692	29,805	22,871	29,920	523.240

■ **Table 7** Table summarizing the results on the eight datasets. From left to right we report dataset names followed by the ratio of positions in interesting intervals, the variability of the dataset (see Def. 9), the average normalized Hamming distance between any two separator-based BWT variants. In the last two columns we report the maximum and minimum average runlength (n/r) taken over all five BWT variants.

dataset	ratio pos.s in intr.int.s	vari- ability	avg. Hamming d. betw. $\$$ -sep. BWTs	max n/r (avg. runlength)	min n/r (avg. runlength)
SARS-CoV-2 short	0.792	0.210	0.11754	31.524	7.494
Simons Diversity reads	0.107	0.976	0.07195	7.873	5.299
16S rRNA short	0.741	0.058	0.02982	44.253	18.836
Influenza A reads	0.103	0.363	0.02609	49.172	23.100
SARS-CoV-2 long	0.175	0.037	0.00464	73.204	57.568
16S rRNA long	0.047	0.104	0.00289	46.879	45.015
Candida auris reads	0.007	0.497	0.00246	1.732	1.726
SARS-CoV-2 genomes	0.001	0.148	0.00012	521.610	499.549

■ **Table 8** Results for the SARS-CoV-2 short dataset. Top left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. Top right: summary of the dataset properties. Bottom left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Bottom right: number of runs and average runlength (n/r) taken over all BWT variants.

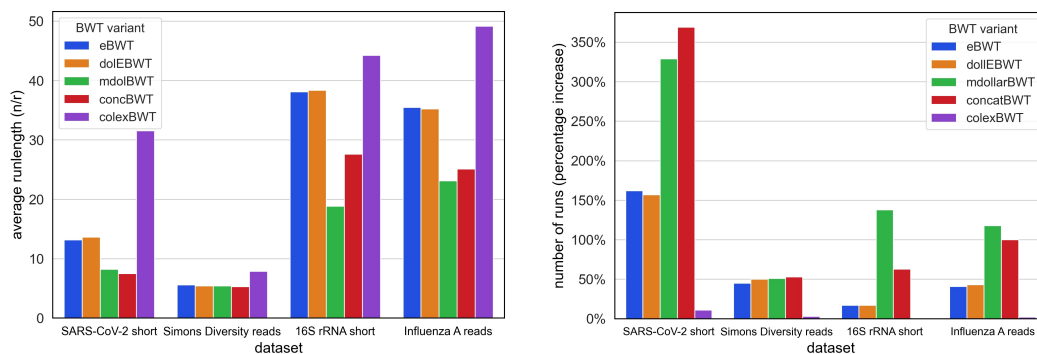
SARS-CoV-2 short (500,000 short sequences)

<i>norm. Hamming d.</i>	<i>Hamming distance on the big dataset</i>			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	3,014,183	2,926,602	2,912,860
mdolBWT	0.11820	0	3,013,908	3,102,887
concBWT	0.11477	0.11819	0	3,013,634
colexBWT	0.11423	0.12168	0.11818	0

<i>dataset properties</i>	
no. sequences	500,000
average length	50
total length	25,000,000
no. of interesting intervals	116,598
total length intr.int.s	20,187,840
fraction pos.s in intr.int.s	0.792
variability	0.210

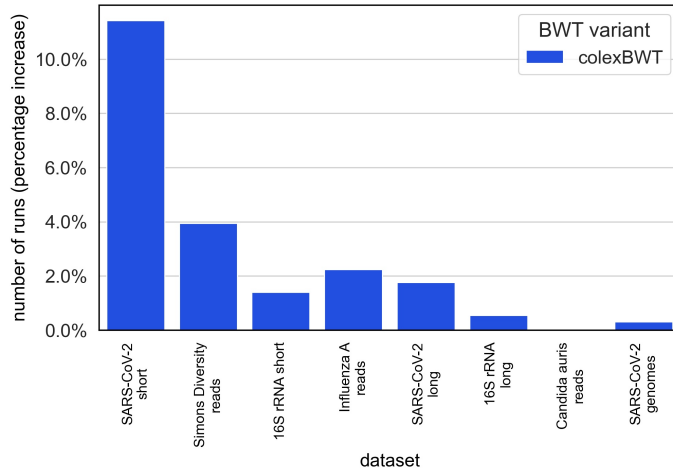
<i>norm. edit d.</i>	<i>edit distance on a subset of 5,000 sequences</i>				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	28,702	43,903	43,828	46,936
dolEBWT	0.11256	0	17,000	16,921	20,104
mdolBWT	0.17217	0.06667	0	16,130	20,812
concBWT	0.17187	0.06636	0.06325	0	20,830
colexBWT	0.18406	0.07884	0.08162	0.08169	0

<i>no. runs big dataset</i>		
	<i>r</i>	<i>n/r</i>
eBWT	1,902,148	13.143
dolEBWT	1,868,581	13.647
mdolBWT	3,113,818	8.189
concBWT	3,402,513	7.494
colexBWT	808,906	31.524
optimum	725,979	35.125

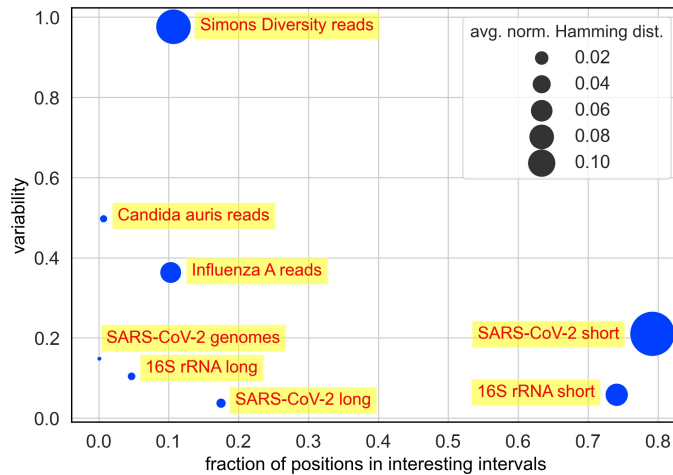


■ **Figure 1** Results regarding r on short sequence datasets, of all BWT variants. Left: average runlength (n/r). Right: number of runs (percentage increase with respect to optimal BWT).

25:14 An Analysis of BWT Variants of String Collections



■ **Figure 2** Number of runs of the colexBWT with respect to optimal BWT (percentage increase) on all eight datasets.



■ **Figure 3** Average normalized Hamming distance variations with respect to variability and fraction of positions in interesting intervals on all datasets.

■ **Table 9** Results for the SARS-CoV-2 genomes dataset. Top left: absolute and normalized pairwise Hamming distance between separator-based BWT variants. Top right: summary of the dataset properties. Bottom left: absolute and normalized pairwise edit distance between all BWT variants on a subset of the input collection. Bottom right: number of runs and average runlength (n/r) taken over all BWT variants.

SARS-CoV-2 genomes (2,000 long sequences)

<i>Hamming d.</i> <i>norm. Hamming d.</i>	<i>Hamming distance on the big dataset</i>			
	dolEBWT	mdolBWT	concBWT	colexBWT
dolEBWT	0	7,958	7,900	7,263
mdolBWT	0.00013	0	7,958	7,957
concBWT	0.00013	0.00013	0	7,990
colexBWT	0.00012	0.00013	0.00013	0

<i>dataset properties</i>	
no. sequences	2,000
total length	59,612,692
average length	29,085
no. interesting intervals	1863
total length intr.int.s	80,486
fraction pos.s in intr.int.s	0.001
variability	0.148

<i>edit d.</i> <i>norm. edit d.</i>	<i>edit distance on a subset of 50 sequences</i>				
	eBWT	dolEBWT	mdolBWT	concBWT	colexBWT
eBWT	0	786	795	801	791
dolEBWT	0.00053	0	98	107	86
mdolBWT	0.00053	0.00007	0	105	112
concBWT	0.00054	0.00007	0.00007	0	114
colexBWT	0.00053	0.00006	0.00008	0.00008	0

<i>no. runs big dataset</i>		
	<i>r</i>	<i>n/r</i>
eBWT	117,628	506.773
dolEBWT	117,410	507.731
mdolBWT	118,870	501.495
concBWT	119,334	499.549
colexBWT	114,287	521.605
optimum	113,930	523.240

6 Conclusion

We presented the first study of the different variants of the Burrows-Wheeler-Transform for string collections. We found that the data structures computed by different tools differ not insignificantly, as measured by the pairwise Hamming distance: up to 12% between different BWT variants on the same dataset in our experiments. We showed that most BWT variants in use are input order dependent, so the same tool can produce different variants if the input set is permuted. These differences extend also to the number of runs r , a parameter that is central in the analysis of BWT-based data structures, and which is increasingly being used as a measure of the repetitiveness of the dataset itself.

With string collections replacing individual sequences as the prime object of research and analysis, and thus becoming the standard input for text indexing algorithms, we believe that it is all the more important for users and researchers to be aware that not all methods are equivalent, and to understand the precise nature of the BWT variant produced by a particular tool. We suggest further to standardize the definition of the parameter r for string collections, using either the colexicographic order or the optimal order of Bentley et al. [4].

References

- 1 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *CoRR*, abs/2107.08615, 2021. [arXiv:2107.08615](https://arxiv.org/abs/2107.08615).
- 2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r -index. *Theor. Comput. Sci.*, 812:96–108, 2020. [doi:10.1016/j.tcs.2019.08.005](https://doi.org/10.1016/j.tcs.2019.08.005).

- 3 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013. doi:10.1016/j.tcs.2012.02.002.
- 4 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *Proc. of 28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *LIPIcs*, pages 15:1–15:13, 2020. doi:10.4230/LIPIcs.ESA.2020.15.
- 5 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Multithread multistring Burrows-Wheeler Transform and Longest Common Prefix array. *J. Comput. Biol.*, 26(9):948–961, 2019. doi:10.1089/cmb.2018.0230.
- 6 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. of 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *LNCS*, pages 129–142, 2021. doi:10.1007/978-3-030-86692-1_11.
- 7 Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub, Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. of 23rd Symposium on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 60–72. SIAM, 2021. doi:10.1137/1.9781611976472.5.
- 8 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019. doi:10.1186/s13015-019-0148-5.
- 9 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 10 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick automaton: Applications to run-length encoding. In *Proc. of 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, volume 128 of *LIPIcs*, pages 24:1–24:20, 2019. doi:10.4230/LIPIcs.CPM.2019.24.
- 11 Shubham Chandak, Kedar Tatwawadi, Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. SPRING: a next-generation compressor for FASTQ data. *Bioinform.*, 35(15):2674–2676, 2019. doi:10.1093/bioinformatics/bty1015.
- 12 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r -index. In *Proc. of 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *LIPIcs*, pages 13:1–13:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.13.
- 13 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012. doi:10.1093/bioinformatics/bts173.
- 14 Diego Díaz-Domínguez and Gonzalo Navarro. Efficient construction of the extended BWT from grammar-compressed DNA sequencing reads. *CoRR*, abs/2102.03961, 2021. arXiv:2102.03961.
- 15 Robert C Edgar. Updating the 97% identity threshold for 16S ribosomal RNA OTUs. *Bioinf.*, 34(14):2371–2375, 2018. doi:10.1093/bioinformatics/bty113.
- 16 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6:1–6:15, 2019. doi:10.1186/s13015-019-0140-0.
- 17 Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012. doi:10.1007/s00453-011-9535-0.
- 18 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. of 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pages 184–193, 2005. doi:10.1109/SFCS.2005.69.

- 19 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009. doi:10.1145/1613676.1613680.
- 20 Johannes Fischer and Florian Kurpicz. sais-lite-lcp. <https://github.com/kurpicz/sais-lite-lcp>. Accessed: 2022-02-05.
- 21 Travis Gagie, Garance Gourdel, and Giovanni Manzini. Compressing and indexing aligned readsets. In *Proc. of 21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*, volume 201 of *LIPICs*, pages 13:1–13:21, 2021. doi:10.4230/LIPICs.WABI.2021.13.
- 22 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. of 39th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, pages 1459–1477, 2018. doi:10.1137/1.9781611975031.96.
- 23 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012. arXiv:1201.3077.
- 24 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 *Proc. of 47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2021)*, volume 12607 of *LNCS*, pages 249–262, 2021. doi:10.1007/978-3-030-67731-2_18.
- 25 Allison J. Greaney et al. A SARS-CoV-2 variant elicits an antibody response with a shifted immunodominance hierarchy. *PLOS Pathogens*, 18:1–27, February 2022. doi:10.1101/2021.10.12.464114.
- 26 Ilya Grebnov. libsais. <https://github.com/IlyaGrebnov/libsais>. Accessed: 2022-02-05.
- 27 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 28 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinform.*, 30(24):3524–3531, 2014. doi:10.1093/bioinformatics/btu584.
- 29 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *Proc. of 61st IEEE Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1002–1013, 2020. doi:10.1109/FOCS46700.2020.00097.
- 30 Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara. In-place Bijective Burrows-Wheeler Transforms. In *Proc. of 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *LIPICs*, pages 21:1–21:15, 2020. doi:10.4230/LIPICs.CPM.2020.21.
- 31 Gregory Kucherov, Lilla Tóthmérés, and Stéphane Vialette. On the combinatorics of suffix arrays. *Inf Process Lett*, 113(22-24):915–920, 2013. doi:10.1016/j.ipl.2013.09.009.
- 32 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. In *Proc. of 23rd Annual Conference in Computational Molecular Biology (RECOMB 2019)*, volume 11467 of *LNCS*, pages 158–173, 2019. doi:10.1089/cmb.2019.0309.
- 33 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012. doi:10.1038/nmeth.1923.
- 34 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25, 2009. doi:10.1186/gb-2009-10-3-r25.
- 35 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinform.*, 30(22):3274–3275, 2014. doi:10.1093/bioinformatics/btu541.
- 36 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.
- 37 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms Mol. Biol.*, 15(1):18, 2020. doi:10.1186/s13015-020-00177-y.

- 38 Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina Dutra de Aguiar Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26:1–26:16, 2017. doi:10.1186/s13015-017-0117-9.
- 39 Swapan Mallick et al. The Simons Genome Diversity Project: 300 genomes from 142 diverse populations. *Nature*, 538(7624):201–206, 2016. doi:10.1038/nature18964.
- 40 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.
- 41 Giovanni Manzini. XBWT tricks. In *Proc. of 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, volume 9954 of *LNCS*, pages 80–92, 2016. doi:10.1007/978-3-319-46049-9_8.
- 42 Yuta Mori. libdivsufsort. <https://github.com/y-256/libdivsufsort>. Accessed: 2022-02-05.
- 43 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.
- 44 Genome 10K Community of Scientists. A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J Hered.*, 100:659–674, 2009. doi:10.1093/jhered/esp086.
- 45 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
- 46 Enno Ohlebusch, Stefan Stauß, and Uwe Baier. Trickier XBWT tricks. In *Proc. of 25th International Symposium in String Processing and Information Retrieval (SPIRE 2018)*, volume 11147 of *LNCS*, pages 325–333, 2018. doi:10.1007/978-3-030-00479-8_26.
- 47 Marco Oliva, Massimiliano Rossi, Jouni Sirén, Giovanni Manzini, Tamer Kahveci, Travis Gagie, and Christina Boucher. Efficiently merging r-indexes. In *Proc. of 31st Data Compression Conference (DCC 2021)*, pages 203–212, 2021. doi:10.1109/DCC50243.2021.00028.
- 48 Jacopo Pantaleoni. BWT of large string sets. *CoRR*, abs/1410.0562, 2014. arXiv:1410.0562.
- 49 Simon J. Puglisi and Bella Zhukova. Document retrieval hacks. In *Proc. of 19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *LIPICs*, pages 12:1–12:12, 2021. doi:10.4230/LIPICs.SEA.2021.12.
- 50 Jouni Sirén. Burrows-Wheeler Transform for terabases. In *Proc. of 26th Data Compression Conference (DCC 2016)*, pages 211–220, 2016. doi:10.1109/DCC.2016.17.
- 51 Tyler N. Starr et al. Deep mutational scanning of SARS-CoV-2 receptor binding domain reveals constraints on folding and ACE2 binding. *Cell*, 182(5):1295–1310.e20, 2020. doi:10.1016/j.cell.2020.08.012.
- 52 C. Sun et al. RPAN: rice pan-genome browser for 3000 rice genomes. *Nucleic Acids Res.*, 45(2):597–605, 2017. doi:10.1093/nar/gkw958.
- 53 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015. doi:10.1038/nature15393.
- 54 The 1001 Genomes Consortium. Epigenomic Diversity in a Global Collection of Arabidopsis thaliana Accessions. *Cell*, 166(2):492–505, 2016. doi:10.1016/j.cell.2016.06.044.
- 55 C. Turnbull et al. The 100,000 genomes project: bringing whole genome sequencing to the NHS. *Br Med J*, 361, 2018. doi:10.1136/bmj.k1687.
- 56 Silvie Van den Hoecke, Judith Verhelst, Marnik Vuylsteke, and Xavier Saelens. Analysis of the genetic diversity of influenza A viruses using next-generation DNA sequencing. *BMC Genomics*, 16(1):79, 2015. doi:10.1186/s12864-015-1284-z.
- 57 Raf Winand et al. Targeting the 16s rRNA gene for bacterial identification in complex mixed samples: Comparative evaluation of second (Illumina) and third (Oxford nanopore technologies) generation sequencing technologies. *Int. J. of Mol. Sci.*, 21(1):298, 2019. doi:10.3390/ijms21010298.
- 58 Michael H. Woodworth et al. Sentinel case of *Candida auris* in the Western United States Following Prolonged Occult Colonization in a Returned Traveler from India. *Microb Drug Resist*, 25(5):677–680, 2019. doi:10.1089/mdr.2018.0408.

RePair Grammars Are the Smallest Grammars for Fibonacci Words

Takuya Mieno^{1,2}  

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

Shunsuke Inenaga  

Department of Informatics, Kyushu University, Fukuoka, Japan

PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Takashi Horiyama  

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

Abstract

Grammar-based compression is a loss-less data compression scheme that represents a given string w by a context-free grammar that generates only w . While computing the smallest grammar which generates a given string w is NP-hard in general, a number of polynomial-time grammar-based compressors which work well in practice have been proposed. *RePair*, proposed by Larsson and Moffat in 1999, is a grammar-based compressor which recursively replaces all possible occurrences of a most frequently occurring bigrams in the string. Since there can be multiple choices of the most frequent bigrams to replace, different implementations of RePair can result in different grammars. In this paper, we show that the smallest grammars generating the Fibonacci words F_k can be completely characterized by RePair, where F_k denotes the k -th Fibonacci word. Namely, all grammars for F_k generated by any implementation of RePair are the smallest grammars for F_k , and no other grammars can be the smallest for F_k . To the best of our knowledge, Fibonacci words are the first non-trivial infinite family of strings for which RePair is optimal.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases grammar based compression, Fibonacci words, RePair, smallest grammar problem

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.26

Related Version *Full Version*: <https://arxiv.org/abs/2202.08447>

Funding *Takuya Mieno*: JSPS KAKENHI Grant Numbers 20H05964 and JP20J11983.

Shunsuke Inenaga: JST PRESTO Grant Number JPMJPR1922.

Takashi Horiyama: JSPS KAKENHI Grant Number 20H05964.

1 Introduction

A context-free grammar in the Chomsky normal form that produces only a single string w is called a *straight-line program (SLP)* for w . Highly repetitive strings that contain many long repeats can be compactly represented by SLPs since occurrences of equal substrings can be replaced by a common non-terminal symbol. *Grammar-based compression* is a loss-less data compression scheme that represents a string w by an SLP for w . We are aware of more powerful compression schemes such as run-length SLPs [24, 37, 6], composition systems [19], collage systems [26], NU-systems [36], the Lempel-Ziv 77 family [42, 39, 12, 13], and bidirectional schemes [39]. Nevertheless, since SLPs exhibit simpler structures than those, a number of efficient algorithms that can work directly on SLPs have been proposed,

¹ Corresponding author

² Current affiliation: University of Electro-Communications, Japan (tmieno@uec.ac.jp)



including pattern matching [25, 24], convolutions [40], random access [8], detection of repeats and palindromes [22], Lyndon factorizations [23], longest common extension queries [21], longest common substrings [34], finger searches [5], and balancing the grammar [17]. More examples of algorithms directly working on SLPs can be found in references therein and the survey [31]. Since these algorithms do not decompress the SLPs, they can be more efficient than solutions on uncompressed strings.

Since the complexities of the algorithms mentioned above depend on the size of the SLP, it is important to compute a small grammar for a given string. *The smallest grammar problem* is to find a grammar that derives a given string w , where the total length of the right-hand sides of the productions is the smallest possible. The smallest grammar problem is known to be NP-hard in general [39, 10]. Namely, there is no polynomial-time algorithm that finds the smallest grammar for *arbitrary* strings, unless $P = NP$. Notably, the NP-hardness holds even when the alphabet size is bounded by some constant at least 17 [9], on the other hand, it is open whether the NP-hardness holds for strings over a smaller constant alphabet, particularly on binary alphabets.

We consider a slightly restricted version of the smallest grammar problem where the considered grammars are SLPs, i.e., only those in the Chomsky normal form. We follow a widely accepted definition for the *size* of an SLP, which is the number of productions in it. Thus, in the rest of our paper, grammars mean SLPs unless otherwise stated, and our smallest grammar problem seeks the smallest SLP, which generates the input string with the fewest productions³. There are some trivial examples of strings whose smallest grammar sizes can be easily determined, e.g., a unary string $(\mathbf{a})^{2^i}$ of length power of two⁴, and non-compressible strings in which all the symbols are distinct. It is interesting to identify classes of strings whose smallest grammars can be determined in polynomial-time since it may lead to more and deeper insights to the smallest grammar problem. To the best of our knowledge, however, no previous work shows non-trivial strings whose smallest grammar sizes are computable in polynomial-time.

In this paper, we study the smallest grammars of the *Fibonacci words* $\{F_1, F_2, \dots, F_n, \dots\}$ defined recursively as follows: $F_1 = \mathbf{b}$, $F_2 = \mathbf{a}$, and $F_i = F_{i-1}F_{i-2}$ for $i \geq 3$. We show that the smallest grammars of the Fibonacci words can be completely characterized by the famous *RePair* [30] algorithm, which is the best known practical grammar compressor that recursively replaces the most frequently occurring bigram with a new non-terminal symbol in linear total time. We first prove that the size of the smallest grammar of the n -th Fibonacci word F_n is n . We then prove that applying *any implementation* of RePair to F_n always provides a smallest grammar of F_n , and conversely, only such grammars can be the smallest for Fibonacci words. This was partially observed earlier in the experiments by Furuya et al. [15], where five different implementations of RePair produced grammars of the same size for the fib41 string from the Repetitive Corpus of the Pizza&Chili Corpus (<http://pizzachili.dcc.uchile.cl/repcorpus.html>). However, to our knowledge, this paper is the first that gives theoretical evidence.

³ There is an alternative definition of the size of a grammar, that is, the total sum of the lengths of the right side of its rules. This definition is usually used for non-SLP grammars.

⁴ Grammars for unary words are closely related to *addition chains* [28], and the smallest (not necessarily SLP) grammar for $(\mathbf{a})^k$ is non-trivial for general k that is not a power of two. Also, in such a case, RePair does not provide the smallest grammar for $(\mathbf{a})^k$ [20].

Related Work

Although the smallest grammar problem is NP-hard, there exist polynomial-time approximations to the problem: Rytter's AVL-grammar [38] produces an SLP of size $O(s^* \log(N/s^*))$, where s^* denote the size of the smallest SLP for the input string and N is the length of the input string. The α -balanced grammar of Charikar et al. [10] produces a (non-SLP) grammar of size $O(g^* \log(N/g^*))$, where g^* denotes the size of the smallest (non-SLP) grammar. Upper bounds and lower bounds for the approximation ratios of other practical grammar compressors including LZ78 [43], BISECTION [27], RePair [30], SEQUENTIAL [41], LONGEST MATCH [27], and GREEDY [1], are also known [10, 2]. Charikar et al. [10] showed that the approximation ratio of RePair to the smallest (non-SLP) grammar is at most $O((N/\log N)^{2/3})$ and is at least $\Omega(\sqrt{\log N})$. The lower bound was later improved by Bannai et al. [2] to $\Omega(\log N/\log \log N)$. Furthermore, it is known that RePair has a lower bound on the approximation ratio $\log_2(3)$ to the smallest (non-SLP) grammar for *unary* strings [20]. On the other hand, RePair is known to achieve the best compression ratio on many real-world datasets and enjoy applications in web graph compression [11] and XML compression [32]. Some variants of RePair have also been proposed [33, 7, 18, 16, 15, 29].

2 Preliminaries

2.1 Strings

Let Σ be an alphabet. An element in Σ is called a symbol. An element in Σ^* is called a string. The length of string w is denoted by $|w|$. The empty string ε is the string of length 0. For each i with $1 \leq i \leq |w|$, $w[i]$ denotes the i -th symbol of w . For each i and j with $1 \leq i \leq j \leq |w|$, $w[i..j]$ denotes the *substring* of w which begins at position i and ends at position j . For convenience, let $w[i..j] = \varepsilon$ if $i > j$. When $i = 1$ (resp. $j = |w|$), $w[i..j]$ is called a *prefix* (resp. a *suffix*) of w . For non-empty strings w and b with $|b| < |w|$, b is called a *border* of w if b is both a prefix and a suffix of w . If there are no borders of w , then w is said to be *borderless*. For any non-empty string w , we call $w[|w|]w[1..|w|-1]$ the *right-rotation* of w . For a string w , σ_w denotes the number of distinct symbols appearing in w . For a non-empty string w , we denote by w^R the *reversed string* of w , namely $w^R = w[|w|] \cdots w[1]$.

2.2 Fibonacci Words and Related Words

For a binary alphabet $\{a, b\}$, *Fibonacci words* $F_i^{(a,b)}$ (starting with a for $i > 1$) are defined as follows: $F_1^{(a,b)} = b$, $F_2^{(a,b)} = a$, and $F_i^{(a,b)} = F_{i-1}^{(a,b)} F_{i-2}^{(a,b)}$ for $i \geq 3$. We call $F_i^{(a,b)}$ the i -th Fibonacci word (starting with a for $i > 1$). By the above definition of Fibonacci words, $|F_i^{(a,b)}| = f_i$ holds for each i , where f_i denotes the i -th Fibonacci number defined as follows: $f_1 = 1$, $f_2 = 1$, $f_i = f_{i-1} + f_{i-2}$ for $i \geq 3$. There is an alternative definition (e.g. [3]) of Fibonacci words using the *string morphism* $\phi^{(a,b)}$: The i -th Fibonacci word $F_i^{(a,b)}$ (starting with a for $i > 1$) is $(\phi^{(a,b)})^{i-1}(b)$, where $\phi^{(a,b)}$ is a morphism over $\{a, b\}$ such that $\phi^{(a,b)}(a) = ab$ and $\phi^{(a,b)}(b) = a$. We strictly distinguish the morphism $\phi^{(b,a)}$ from $\phi^{(a,b)}$ over the same binary alphabet $\{a, b\}$, namely, $\phi^{(b,a)}$ generates the Fibonacci words $F_i^{(b,a)}$ where a and b are flipped in $F_i^{(a,b)}$. We will omit the superscript (a, b) if it is clear from contexts or it is not essential for the discussion.

Next, we define other words, which will be utilized to analyze the smallest grammar of Fibonacci words. Let $\pi^{(a,b)}$ be the morphism over $\{a, b\}$ such that $\pi^{(a,b)}(a) = ab$ and $\pi^{(a,b)}(b) = abb$. Further, let $\theta^{(a,b)}$ be the morphism over $\{a, b\}$ such that $\theta^{(a,b)}(a) = aab$ and

■ **Table 1** Lists of $F_i^{(a,b)}$ for $i = 1, \dots, 10$, and $P_i^{(a,b)}$ and $Q_i^{(a,b)}$ for $i = 1, \dots, 5$.

i	$F_i^{(a,b)}$	length
1	b	1
2	a	1
3	ab	2
4	aba	3
5	abaab	5
6	abaababa	8
7	abaababaabaab	13
8	abaababaabaababaababa	21
9	abaababaabaababaabaabaabaabaab	34
10	abaababaabaababaabaabaabaabaabaabaabaabaabaabaabaab	55

i	$P_i^{(a,b)}$	length
1	a	1
2	ab	2
3	ababb	5
4	ababbababbabb	13
5	ababbababbabbababbababbababbabb	34

i	$Q_i^{(a,b)}$	length
1	a	1
2	aab	3
3	aabaabab	8
4	aabaababaabaababaabab	21
5	aabaababaabaababaabaabaabaabaabaabaabaabaabaabaab	55

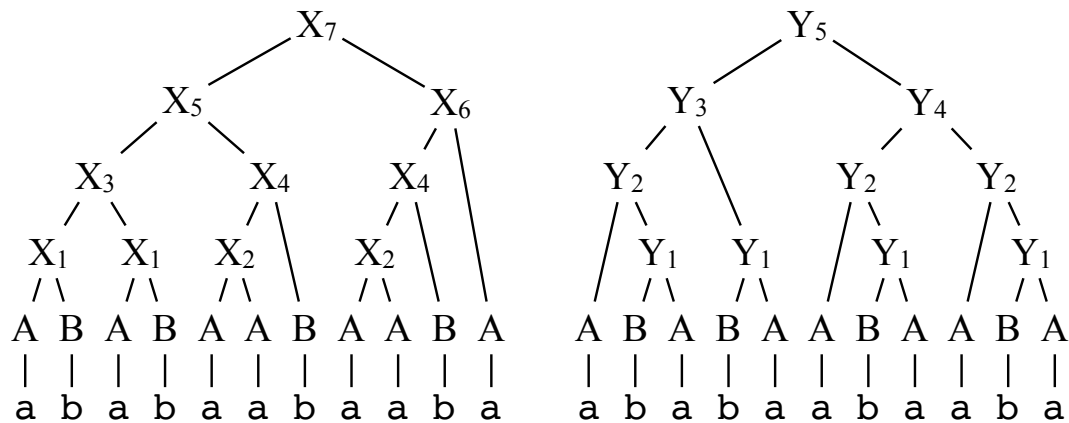
$\theta^{(a,b)}(b) = ab$. For each positive integer i , we define $P_i^{(a,b)}$ and $Q_i^{(a,b)}$ over $\{a, b\}$ by $P_i^{(a,b)} = (\pi^{(a,b)})^{i-1}(a)$ and $Q_i^{(a,b)} = (\theta^{(a,b)})^{i-1}(a)$, respectively. We treat their superscripts as for that of Fibonacci words. We will later show that $|P_i| = |F_{2i-1}| = f_{2i-1}$ and $|Q_i| = |F_{2i}| = f_{2i}$ for any $i \geq 1$. We show examples for these three words in Table 1. We remark that strings P_i and Q_i can be obtained at some point while RePair is being applied to the Fibonacci words. We will prove this in Section 4.

For a symbol X and a string y , let $\xi_{X \rightarrow y}$ be the morphism such that $\xi_{X \rightarrow y}(X) = y$ and $\xi_{X \rightarrow y}(c) = c$ for any symbol $c \neq X$. Namely, when applied to a string w , $\xi_{X \rightarrow y}(w)$ replaces all occurrences of X in w with y but any other symbols than X remain unchanged. For any morphism λ and any sequence $S = (s_1, \dots, s_m)$ of strings, let $\lambda(S) = (\lambda(s_1), \dots, \lambda(s_m))$.

2.3 Grammar Compression and RePair

A context-free grammar in the Chomsky normal form that produces a single string w is called a *straight-line program* (SLP in short) for w . Namely, any production in a grammar is of form either $X_i \rightarrow \alpha$ or $X_i \rightarrow X_j X_k$, where α is a terminal symbol and X_i, X_j , and X_k are non-terminal symbols such that $i > j$ and $i > k$, that is, there are no *cycles* in the productions. In what follows, we refer to an SLP that produces w simply as a *grammar of w* . Let $\mathcal{T}(G)$ denote the derivation tree of a grammar G , where each internal node in $\mathcal{T}(G)$ is labeled by the corresponding non-terminal symbol of G . As in [38], we conceptually identify terminal symbols with their parents so that $\mathcal{T}(G)$ is a full binary tree (i.e. every internal node has exactly two children). Let G_1 and G_2 be grammars both deriving the same string w , and

$w = \text{ababaabaaba}$



■ **Figure 1** Illustration for the derivation trees of two distinct grammars of string $w = \text{ababaabaaba}$. The size of the grammar on the left is 9 since there are nine productions; $\{A \rightarrow a, B \rightarrow b, X_1 \rightarrow AB, X_2 \rightarrow AA, X_3 \rightarrow X_1X_1, X_4 \rightarrow X_2B, X_5 \rightarrow X_3X_4, X_6 \rightarrow X_4A, X_7 \rightarrow X_5X_6, \}$. On the other hand, the size of the grammar on the right is 7. Note that the right one is a RePair grammar of w . In the rest of the paper, we sometimes identify the terminal symbols (leaves) with their parents so the derivation trees are (conceptually) full binary trees.

let Π_1 and Π_2 be the sets of non-terminal symbols of G_1 and G_2 , respectively. We say that G_1 and G_2 are *equivalent* if there exists a renaming bijection $f : \Pi_1 \rightarrow \Pi_2$ that transforms $\mathcal{T}(G_1)$ to $\mathcal{T}(G_2)$. We say that G_1 and G_2 are *distinct* if they are not equivalent. For example, two grammars $\{A \rightarrow a, B \rightarrow b, C \rightarrow AB, D \rightarrow CA\}$ and $\{X \rightarrow a, Y \rightarrow b, Z \rightarrow XY, W \rightarrow ZX\}$ are equivalent grammars both deriving string aba .

Equivalent grammars form an equivalence class of grammars, and we pick an arbitrary one as the representative of each equivalence class. A *set* S of grammars that derive the same string w is a set which consists of (some) representative grammars, which means that any two grammars in S are mutually distinct. See Figure 1 for examples of distinct grammars for the same string.

The *size* of a grammar G , denoted by $|G|$, is the number of productions in G . We denote by $g^*(w)$ the size of the smallest grammar of string w . Further, we denote by $\text{Opt}(w)$ the set of all the smallest grammars of string w . While computing $g^*(w)$ for a given string w is NP-hard in general [10], a number of practical algorithms which run in polynomial-time and construct small grammars of w have been proposed.

In this paper, we focus on *RePair* [30], which is the best known grammar-based compressor that produce small grammars in practice. We briefly describe the RePair algorithm, which consists of the three stages:

1. Initial stage: All terminal symbols in the input string are replaced with non-terminal symbols. This creates unary productions.
2. Replacement stage: The algorithm picks an *arbitrary* bigram which has the most non-overlapping occurrences in the string, and then replaces all possible occurrences of the bigram with a new non-terminal symbol. The algorithm repeats the same process recursively for the string obtained after the replacement of the bigrams, until no bigrams have two or more non-overlapping occurrences in the string. It is clear that the productions created in the replacement stage are all binary.

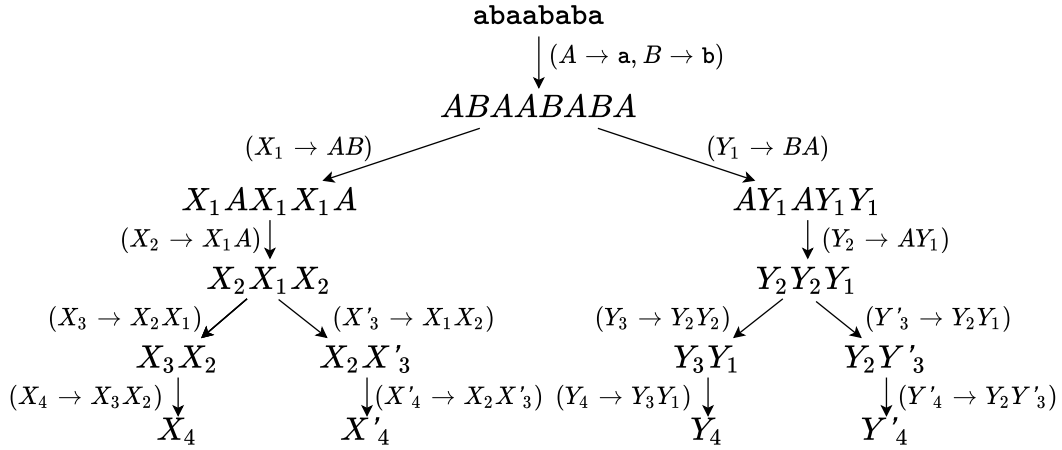


Figure 2 Illustration for the changes of strings and productions to be added when RePair is applied to string $w = \text{abaababa}$. At the second level, the most frequent bigrams in string $ABAABABA$ are AB and BA . If AB is chosen and replaced with non-terminal symbol X_1 , the string changes to $X_1AX_1X_1A$ and production $X_1 \rightarrow AB$ is added. Otherwise (if BA is chosen and replaced with non-terminal symbol Y_1), the string changes to $AY_1AY_1Y_1$ and production $Y_1 \rightarrow BA$ is added. In this example, the size of $\text{RePair}(w)$ is four.

3. Final stage: Trivial binary productions are created from the sequence of non-terminal symbols which are obtained after the last replacement. This ensures that the resulting grammar is in the Chomsky normal form. We remark that when distinct bigrams have the most non-overlapping occurrences in the string, then the choice of the bigrams to replace depends on each implementation of RePair.

A grammar of w obtained by some implementation of RePair is called a *RePair grammar* of w . We denote by $\text{RePair}(w)$ the set of all possible RePair grammars of w . We show an example of RePair grammars in Figure 2.

2.4 LZ-factorization

A sequence $S = (s_1, \dots, s_m)$ of non-empty strings is called a *factorization* of string w if $w = s_1 \cdots s_m$. Each s_i ($1 \leq i \leq m$) is called a *phrase* of S . The *size* of the factorization S , denoted $|S|$, is the number m of phrases in S .

For a factorization $S = (s_1, \dots, s_m)$ of a string w , we say that the i -th phrase s_i is *greedy* if either s_i is a fresh symbol that occurs for the first time in $s_1 \cdots s_i$, or s_i is the longest prefix of $s_i \cdots s_m$ which occurs in $s_1 \cdots s_{i-1}$. A factorization of a string w is called the *LZ-factorization* of w if all the phrases are greedy. Note that this definition of the LZ-factorization is equivalent to the one in [38]. The LZ-factorization of string w is denoted by $LZ(w)$, and the size of $LZ(w)$ is denoted by $z(w)$. We sometimes represent a factorization (s_1, s_2, \dots, s_m) of w by $s_1|s_2|\dots|s_m$, where each $|$ denotes the *boundary* of the phrases. For example, The LZ-factorization of $w = \text{ababaabaaba}$ is a|b|ab|a|aba|aba .

3 Basic Properties of Fibonacci and Related Words

In this section, we show some properties of the aforementioned words. We fix the alphabet $\Sigma = \{a, b\}$ in this section. First, for the summation of Fibonacci sequences, the next equations hold:

► **Fact 1.** $\sum_{k=1}^i f_{2k-1} = f_{2i}$ and $\sum_{k=1}^i f_{2k} = f_{2i+2} - 1$.

By the definitions of F_n , P_n , and Q_n , we have the following observation:

► **Lemma 2.** *For each $k \geq 2$, the most frequent bigrams of F_{2k} are **ab** and **ba**, and the most frequent bigram of F_{2k-1} is **ab**. Also, for each $i \geq 2$ and each $j \geq 3$, the most frequent bigram of P_i and Q_j is **ab**.*

Proof. From the fact that bigram **bb** and trigram **aaa** do not occur in any Fibonacci word (e.g., see [3]), we can see that any occurrence of **aa** is followed by **b** in Fibonacci words. Thus, **aa** cannot occur more frequently than **ab** in any Fibonacci word. Also, since the third and subsequent Fibonacci words start with **ab**, bigram **ab** occurs more frequently than **aa**. Additionally, since all the Fibonacci words F_{2k} of even order end with **b** and all the Fibonacci words F_{2k-1} of odd order end with **a**, the statements for the Fibonacci words hold.

Similarly, as for string P_i , it follows from the definition of morphism π that **aa** does not occur in P_i . Also, **bb** always succeeds **a**, and thus, **bb** cannot occur more frequently than **ab**. Furthermore, by the definition of morphism π , string P_i starts with **aba** and ends with **b** for $i \geq 3$. Thus, the most frequent bigram of P_i is **ab** for $i \geq 3$ (note that P_2 is trivial).

Finally, as for string Q_j , it follows from the definition of morphism θ that **bb** does not occur in Q_j . Also, **aa** always precedes **b**, and thus, **aa** cannot occur more frequently than **ab**. Furthermore, by the definition of morphism θ , string Q_j ends with **ab** for $j \geq 3$. Thus, the most frequent bigram of Q_j is **ab** for $j \geq 3$. ◀

A factorization $C = (c_1, \dots, c_m)$ of a string w is called the *C-factorization* of w if either c_i is a fresh symbol or c_i is the longest prefix of $c_i \cdots c_m$ which occurs twice in $c_1 \cdots c_i$. We can obtain the full characterization of the LZ-factorization of F_n immediately from the C-factorization of F_n , as follows:

► **Lemma 3.** *The LZ-factorization of F_n is $(a, b, a, F_4^R, \dots, F_{n-2}^R, s)$, where $s = ab$ if n is odd, and $s = ba$ otherwise.*

Proof. It is shown in [4] that the C-factorization of the infinite Fibonacci word \mathbf{F} is $(a, b, a, F_4^R, F_5^R, \dots)$. Also, for each $i \geq 4$, the (only) reference source of each factor F_i^R is the substring of F_n of length f_i ending at just before the factor, i.e., the source does not overlap with the factor. From these facts, it can be seen that the LZ-factorization of \mathbf{F} is the same as the C-factorization of \mathbf{F} . Then, the last phrase of the C-factorization of a finite Fibonacci word is of length two since $f_n = \sum_{i=1}^{n-2} f_i + 1 = (1 + 1 + 2 + \sum_{i=4}^{n-2} f_i) + 1 = 3 + \sum_{i=4}^{n-2} f_i + 2$. Also, since the Fibonacci words of odd order (resp. even order) end with **ab** (resp. **ba**), the last phrase is **ab** (resp. **ba**). ◀

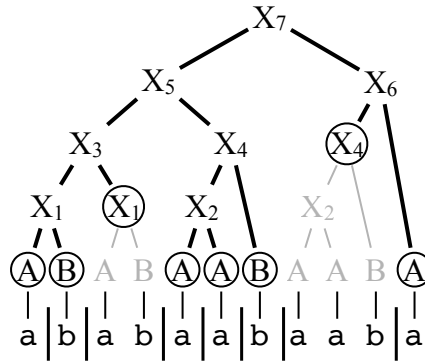
The next lemma states that P_i and Q_i are the right-rotations of Fibonacci words.

► **Lemma 4.** *For each $i \geq 1$, $P_i^{(a,b)}$ is the right-rotation of $F_{2i-1}^{(b,a)}$, and $Q_i^{(a,b)}$ is the right-rotation of $F_{2i}^{(a,b)}$.*

Proof. The next claim can be proven by induction:

▷ **Claim 5.** For any non-empty string $x \in \{a, b\}$, $(\phi^{(b,a)})^2(x)b = b\pi^{(a,b)}(x)$ and $(\phi^{(a,b)})^2(x)ab = ab\theta^{(a,b)}(x)$ hold.

We prove the lemma by using Claim 5. Assume that the lemma holds for i . Since the last symbol of $F_{2i-1}^{(b,a)}$ is **a**, we can write $F_{2i-1}^{(b,a)} = xa$ with some string x . From the induction hypothesis, $P_i^{(a,b)} = ax$ holds. Then, $P_{i+1}^{(a,b)} = \pi^{(a,b)}(ax) = ab\pi^{(a,b)}(x)$. Also, $F_{2i+1}^{(b,a)} =$



■ **Figure 4** Illustration for $\mathcal{PT}(G)$ of grammar G for string $w = ababaabaaba$. The circled nodes are leaves of $\mathcal{PT}(G)$. For this grammar G of w , $gfact(G) = a|b|ab|a|a|b|aab|a$. Since $|G| = 9$, $|gfact(G)| = 8$, and $\sigma_w = 2$, we can see that Lemma 7 holds for this example.

Let g_1 and g_2 be the numbers of productions of Type 1 and Type 2, respectively. By the definition of $\mathcal{PT}(G)$, the labels of all non-leaf nodes are distinct, and they correspond to the productions of Type 1. Thus, the number m of non-leaf nodes is at most g_1 . Also, $\sigma_w \leq g_2$ always holds. Hence, $m + \sigma_w \leq g_1 + g_2 = |G|$. On the other hand, $m = |gfact(G)| - 1$ holds since $\mathcal{PT}(G)$ is a full binary tree and the number of leaves of $\mathcal{PT}(G)$ is $|gfact(G)|$. Therefore, $|gfact(G)| - 1 + \sigma_w \leq |G|$ holds. ◀

We obtain the following tighter lower bound for the size of the smallest grammar(s):

► **Theorem 8.** *For any string w , $z(w) - 1 + \sigma_w \leq g^*(w)$ holds.*

Proof. It was shown in [38] that $z(w) \leq |gfact(G)|$ for any grammar G of w . Thus, combining it with Lemma 7, we obtain the theorem. ◀

By regarding the recursive definition of F_n as a grammar, we can construct a size- n grammar of F_n . Also, $g^*(F_n)$ is at least n by Theorem 8 since $z(F_n) = n - 1$ and $\sigma_{F_n} = 2$. Thus, we obtain the following corollary:

► **Corollary 9.** *The smallest grammar size of F_n is n .*

4.2 RePair Grammars are Smallest for Fibonacci Words

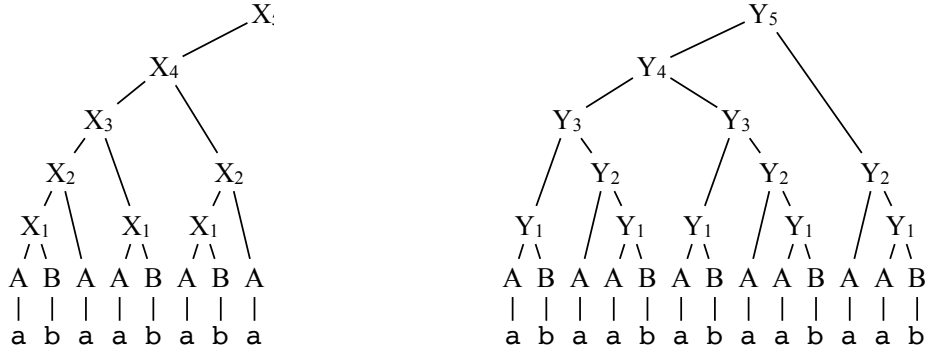
By considering the inverse of morphism ϕ , we have the next observation:

► **Observation 10.** *By replacing all occurrences of ab in $F_i^{(a,b)}$ with X , we obtain $F_{i-1}^{(X,a)}$.*

The next lemma shows how F , P , and Q can be obtained from one of the others:

► **Lemma 11.** *$\xi_{b \rightarrow ba}(P_i) = F_{2i}$, $\xi_{b \rightarrow ab}(P_i) = Q_i$, and $\xi_{a \rightarrow ab}(Q_i) = P_{i+1}$ hold.*

Proof. Let $\psi_1 = \xi_{b \rightarrow ba}$, $\psi_2 = \xi_{b \rightarrow ab}$, and $\psi_3 = \xi_{a \rightarrow ab}$. First, we consider compositions of these morphisms. Since $\phi^2(\psi_1(a)) = \phi^2(a) = \phi(ab) = aba$, $\phi^2(\psi_1(b)) = \phi^2(ba) = \phi(aab) = ababa$, $\psi_1(\pi(a)) = \psi_1(ab) = aba$, and $\psi_1(\pi(b)) = \psi_1(abb) = ababa$, we have $\phi^2 \circ \psi_1 = \psi_1 \circ \pi$. Also, since $\psi_2(\pi(a)) = \psi_2(ab) = aab$, $\psi_2(\pi(b)) = \psi_2(abb) = aabab$, $\theta(\psi_2(a)) = \theta(a) = aab$, and $\theta(\psi_2(b)) = \theta(ab) = aabab$, we have $\psi_2 \circ \pi = \theta \circ \psi_2$. Also, since $\psi_3(\theta(a)) = \psi_3(aab) = ababb$, $\psi_3(\theta(b)) = \psi_3(ab) = aab$, $\pi(\psi_3(a)) = \pi(ab) = ababb$, and $\pi(\psi_3(b)) = \pi(b) = abb$, we have $\psi_3 \circ \theta = \pi \circ \psi_3$.



■ **Figure 5** Two RePair grammars of the 7-th Fibonacci word $abaababaabaab$ over $\{a, b\}$.

When $i = 1$, the lemma clearly holds. We assume that the lemma holds for $i - 1$ with $i \geq 2$. Then, $\psi_1(P_i) = \psi_1(\pi(P_{i-1})) = \phi^2(\psi_1(P_{i-1})) = \phi^2(F_{2i-2}) = F_{2i}$, $\psi_2(P_i) = \psi_2(\pi(P_{i-1})) = \theta(\psi_2(P_{i-1})) = \theta(Q_{i-1}) = Q_i$, and $\psi_3(Q_i) = \psi_3(\theta(Q_{i-1})) = \pi(\psi_3(Q_{i-1})) = \pi(P_i) = P_{i+1}$. ◀

By considering the inverses of the three morphisms in Lemma 11, we have the next corollary:

► **Corollary 12.** *Let X denote a fresh non-terminal symbol. By replacing all occurrences of ba in $F_{2i}^{(a,b)}$ with X , we obtain $P_i^{(a,X)}$. By replacing all occurrences of ab in $Q_i^{(a,b)}$ with X , we obtain $P_i^{(a,X)}$. By replacing all occurrences of ab in $P_{i+1}^{(a,b)}$ with X , we obtain $Q_i^{(X,b)}$.*

We show examples of two RePair grammars of $F_7^{(a,b)}$ in Figure 5.

We are ready to clarify the shape of all the RePair grammars of Fibonacci words.

► **Lemma 13.** *The size of every RePair grammar of F_n is n , i.e., $\text{RePair}(F_n) \subseteq \text{Opt}(F_n)$. Also, $|\text{RePair}(F_n)| = 2\lfloor n/2 \rfloor - 2$.*

Proof. By Lemma 2, Observation 10 and Corollary 12, each string, that appears while (an implementation of) the RePair algorithm is running, is one of F , P , and Q over some binary alphabet. The change of the strings can be represented by a directed graph (V, E) such that $V = \{F_i \mid 4 \leq i \leq n\} \cup \{P_i \mid 3 \leq i \leq \lfloor n/2 \rfloor\} \cup \{Q_i \mid 2 \leq i \leq \lfloor n/2 \rfloor - 1\}$ and $E = \{(F_i, F_{i-1}) \mid 5 \leq i \leq n\} \cup \{(F_{2k}, P_k) \mid 3 \leq k \leq \lfloor n/2 \rfloor\} \cup \{(P_i, Q_{i-1}) \mid 3 \leq i \leq \lfloor n/2 \rfloor\}$. See Figure 6 for an illustration of the graph. Each edge represents a replacement of all occurrences of a most frequent bigram, and thus each path from source (F_n) to sinks (F_4 and Q_2) corresponds to a RePair grammar of F_n . The size of a RePair grammar is the number of edges in its corresponding path plus *four*, since the size of a minimal⁵ grammar of length-3-binary string, such as F_4 and Q_2 , is four. Since the number of edges in any source-to-sinks paths is $n - 4$, the size of each RePair grammar of the n -th Fibonacci word is n . Also, the number of the RePair grammars is *twice* the number of distinct source-to-sinks paths since there are exactly two possible minimal grammars of any length-3 string.

Next, let us count the number of source-to-sinks paths in the graph. There is only one path from F_n to F_4 , and there are $\lfloor n/2 \rfloor - 2$ edges from F_{2k} on the upper part to P_k on the lower part for all k with $3 \leq k \leq \lfloor n/2 \rfloor$. Thus, the number of distinct source-to-sinks paths is $\lfloor n/2 \rfloor - 1$. Therefore, the number of distinct RePair grammars is $2\lfloor n/2 \rfloor - 2$. ◀

⁵ This means that there are no redundant productions.

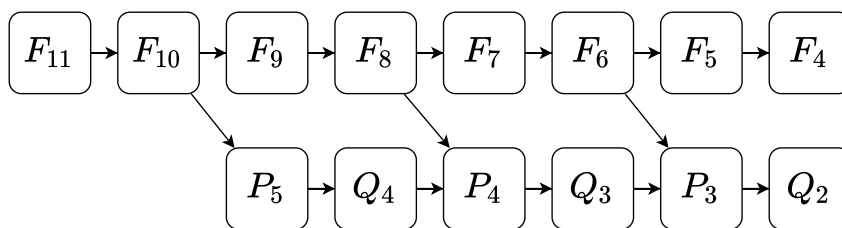


Figure 6 An example of the graph for $n = 11$ described in Lemma 13.

5 Optimality of RePair for Fibonacci Words

In this section, we prove our main theorem:

► **Theorem 14.** $\text{Opt}(F_n) = \text{RePair}(F_n)$.

The derivation tree of any grammar (i.e., SLP) G is a full binary tree. Thus, there exists a *bottom-up* algorithm which constructs the grammar G by replacing bigrams with a non-terminal symbol one by one. Thus, it suffices to consider all such algorithms in order to show the optimality of RePair for F_n . We show that any bigram-replacement that does not satisfy the condition of RePair always produces a larger grammar than the RePair grammars. For F_n , P_n , and Q_n , there are 16 strategies that do not satisfy the condition of RePair:

Bigram to replace (all/not all of them)	aa		ab		ba		bb	
	all	not all	all	not all	all	not all	all	not all
F_{2k}	2	3	RePair	1	RePair	4	-	-
F_{2k+1}			RePair		5	6	-	-
P_n	-	-	RePair	7	8	9	10	11
Q_n	15	16	RePair	12	13	14	-	-

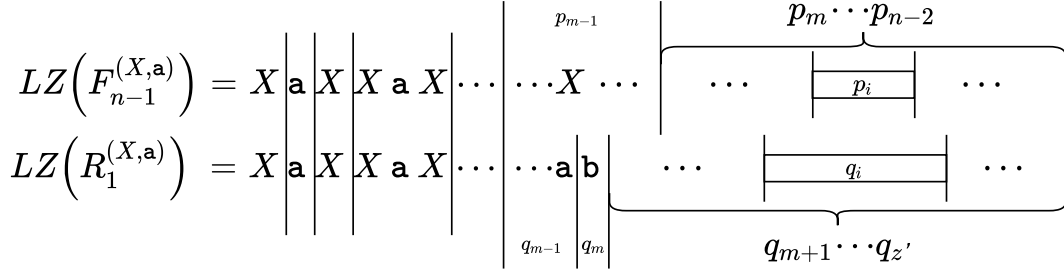
The case numbers (1–16) are written inside their corresponding cells in the table. Each hyphen shows the case where the bigram does not occur in the string, which therefore does not need to be considered.

In order to show the non-optimality of each of the above strategies, we utilize the sizes of LZ-factorizations which are lower bounds of the sizes of grammars. Let R be the string obtained by replacing occurrences of a bigram in F_n with a non-terminal symbol X by one of the above 16 strategies. We will show that $z(R) \geq n - 1$ holds for each case. Then, by Theorem 8, the size of the corresponding grammar of F_n becomes at least $(z(R) + |\{X\}| - 1) + \sigma_{F_n} \geq (n - 1) + 2 = n + 1$, i.e., that is not the smallest by Corollary 9.

To compare the LZ-factorizations between two strings transformed from the same string F_n , we treat the boundaries as if they are on F_n .

5.1 Non-optimality of Strategies for F_n

We first define a *semi-greedy* factorization $SG(w)$ of string w which will be used in the proof for the first three cases. Let $SG(w)$ be the factorization of w obtained by shifting each boundary of $LZ(w)$ except the ones whose left phase is of length 1 to the left by one. For example, $SG(F_7) = \mathbf{a|b|a|ab|abaab|aab}$ since $LZ(F_7) = \mathbf{a|b|a|aba|baaba|ab}$. Clearly, $|SG(F_n)| = |LZ(F_n)| = n - 1$. By the definition of $SG(F_n)$ and properties of $LZ(F_n)$ (cf. [4, 14]), the following claim holds:



■ **Figure 7** Illustration for contradiction of $LZ(F_{n-1}^{(X,a)}) = (p_1, \dots, p_{n-2})$ and $LZ(R_1^{(X,a,b)}) = (q_1, \dots, q_{z'})$ for Case (1). Note that the scale of this figure is based on the length of $F_n \in \{a, b\}^*$, not the lengths of phrases.

- ▷ **Claim 15.** Let $SG(F_n) = (p_1, \dots, p_{n-1})$ for $n \geq 5$. The following statements hold:
- The first four phrases are $(p_1, p_2, p_3, p_4) = (a, b, a, ab)$.
 - For each i with $5 \leq i \leq n-2$, p_i is the right-rotation of F_i^R and it is a greedy phrase.
 - The last phrase is $p_{n-1} = aba$ if n is even, and $p_{n-1} = aab$ otherwise.
 - Each boundary of $SG(F_n)$, except the first and third ones, divides an occurrence of ba .

Case (1): Replacing some but not all the occurrences of ab in F_n

Recall that $F_{n-1}^{(X,a)}$ is obtained by replacing all the occurrences of ab in $F_n^{(a,b)}$ with X . Let $R_1^{(X,a,b)}$ be any string obtained by replacing some but *not all* the occurrences of ab in $F_n^{(a,b)}$ with X . Let $LZ(F_{n-1}^{(X,a)}) = (p_1, \dots, p_{n-2})$ and $LZ(R_1^{(X,a,b)}) = (q_1, \dots, q_{z'})$ where $z' = |LZ(R_1^{(X,a,b)})|$. See Figure 7 for illustration. The first mismatch of boundaries between two factorizations is the position of the first occurrence of b in $R_1^{(X,a,b)}$. Since the b is a fresh symbol, it is a length-1 phrase. Suppose that this length-1 phrase is the m -th phrase ($m \geq 2$) in $LZ(R_1^{(X,a,b)})$. Then, $\xi_{X \rightarrow ab}(p_{m-1} \cdots p_{n-2}) = \xi_{X \rightarrow ab}(q_{m-1} \cdots q_{z'})$ holds. The next corollary follows from Claim 15:

► **Corollary 16.** *The factorization $\xi_{X \rightarrow ab}(LZ(F_{n-1}^{(X,a)}))$ of $F_n^{(a,b)}$ is the same as $SG(F_n^{(a,b)})$ except the first phrase. In other words, for each i with $2 \leq i \leq n-2$, $\xi_{X \rightarrow ab}(p_i)$ is the $(i+1)$ -th phrase of $SG(F_n^{(a,b)})$.*

From the greediness of $\xi_{X \rightarrow ab}(p_{m-1})$ in $SG(F_n^{(a,b)})$, p_{m-1} is not shorter than q_{m-1} . Thus, $\xi_{X \rightarrow ab}(p_m \cdots p_{n-2})$ is not longer than $\xi_{X \rightarrow ab}(q_{m+1} \cdots q_{z'})$. For the sake of contradiction, we assume that $z' < n-1$. Then, $z' - (m+1) + 1 < (n-2) - m + 1$ holds, and hence, there must exist a phrase q_i of $LZ(R_1^{(X,a,b)})$ and a phrase p_j of $LZ(F_{n-1}^{(X,a)})$ such that $\xi_{X \rightarrow ab}(q_i)$ contains $\xi_{X \rightarrow ab}(p_j)$ and their ending positions in $F_n^{(a,b)}$ are different. This contradicts the greediness of the phrase $\xi_{X \rightarrow ab}(p_j)$ of $SG(F_n^{(a,b)})$ on $F_n^{(a,b)}$. Therefore, $|LZ(R_1^{(X,a,b)})| = z' \geq n-1$.

Basically, most of the remaining cases can be proven by similar argumentations, however, we will write down the details because there are a few differences.

Case (2): Replacing all the occurrences of aa in F_n

Let $R_2^{(X,a,b)}$ be the string obtained by replacing all the occurrences of aa in $F_n^{(a,b)}$ with X . The next corollary holds from Claim 15 (see also Figure 8 for a concrete example):

► **Corollary 17.** *The factorization $\xi_{X \rightarrow aa}(LZ(R_2^{(X,a,b)}))$ of $F_n^{(a,b)}$ is the same as $SG(F_n^{(a,b)})$ except the first four phrases. In other words, for each i with $5 \leq i \leq n-1$, $\xi_{X \rightarrow aa}(p_i)$ is the i -th phrase of $SG(F_n^{(a,b)})$, where p_i is the i -th phrase of $LZ(R_2^{(X,a,b)})$.*

Thus, $|LZ(R_2^{(X,a,b)})| = |SG(F_n^{(a,b)})| = n-1$.

SG a b
 LZ x b



$$\begin{aligned}
 LZ(F_8) &= a \underline{b} \underline{a} \underline{a} \underline{b} \underline{a} \underline{b} \underline{a} \underline{a} \underline{b} \underline{a} \underline{a} \underline{b} \underline{a} \underline{b} \underline{a} \underline{a} \underline{b} \underline{a} \underline{b} \underline{a} \\
 LZ(P_4) &= a \underline{x} \underline{a} \underline{x} \underline{x} \underline{a} \underline{x} \underline{a} \underline{x} \underline{x} \underline{a} \underline{x} \underline{x}
 \end{aligned}$$

■ **Figure 9** Two factorizations $LZ(F_8)$ and $LZ(P_4)$.

Case (3): Replacing some but not all the occurrences of aa in F_n

Let $R_3^{(X,a,b)}$ be any string obtained by replacing some but not all the occurrences of aa in $F_n^{(a,b)}$ with X . Let $LZ(R_2^{(X,a,b)}) = (p_1, \dots, p_{n-1})$ and $LZ(R_3^{(X,a,b)}) = (q_1, \dots, q_{z'})$ where $z' = |LZ(R_3^{(X,a,b)})|$. We omit the superscripts in the following. The first mismatch of boundaries between $LZ(R_2)$ and $LZ(R_3)$ is the position of the first occurrence of aa in R_3 . Since this is the first occurrence of aa , there has to be a boundary between the two a 's. Suppose that the phrase that starts with the second a is the m -th phrase ($m \geq 4$) in $LZ(R_3)$. By Corollary 17, p_{m-1} is not shorter than q_{m-1} . Thus, $\xi_{X \rightarrow aa}(q_m \cdots q_{z'})$ is longer than $\xi_{X \rightarrow aa}(p_m \cdots p_{n-1})$. For the sake of contradiction, we assume that $z' < n - 1$. Then, $z' - m + 1 < (n - 1) - m + 1$ holds, and hence, there exist phrases q_i of $LZ(R_3)$ and p_j of $LZ(R_2)$ such that $\xi_{X \rightarrow aa}(q_i)$ contains $\xi_{X \rightarrow aa}(p_j)$ and their ending positions in F_n are different. This contradicts the greediness of the phrase $\xi_{X \rightarrow aa}(p_j)$ of $SG(F_n)$. Therefore, $|LZ(R_3)| = z' \geq n - 1$.

Case (4): Replacing some but not all the occurrences of ba in F_{2k}

Recall that $P_k^{(a,X)}$ is obtained by replacing all the occurrences of ba in $F_{2k}^{(a,b)}$ with X . Let $R_4^{(X,a,b)}$ be any string obtained by replacing some but not all the occurrences of ba in $F_{2k}^{(a,b)}$ with X . Let $LZ(P_k^{(a,X)}) = (p_1, \dots, p_{2k-2})$ and $LZ(R_4^{(X,a,b)}) = (q_1, \dots, q_{z'})$ where $z' = |LZ(R_4^{(X,a,b)})|$. Since the only boundary in $LZ(F_{2k}^{(a,b)})$ that divides an occurrence of ba is the second one, the next holds for the LZ-factorization of $P_k^{(a,X)}$ (see also Figure 9 for a concrete example):

► **Corollary 18.** *The factorization $\xi_{X \rightarrow ba}(LZ(P_k^{(a,X)}))$ of $F_n^{(a,b)}$ is the same as $LZ(F_{2k}^{(a,b)})$ except the first three phrases. In other words, for each i with $4 \leq i \leq 2k - 2$, $\xi_{X \rightarrow ba}(p_i)$ is the $(i + 1)$ -th phrase of $LZ(F_{2k}^{(a,b)})$.*

We omit the superscripts in the following. The first mismatch of boundaries between two factorizations is the position of the first occurrence of b in R_4 . Since the b is a fresh symbol, it is a length-1 phrase. Let the length-1 phrase be the m -th phrase in $LZ(R_4)$. Then, $\xi_{X \rightarrow ba}(q_m \cdots q_{z'})$ is longer than $\xi_{X \rightarrow ba}(p_m \cdots p_{2k-2})$ by Corollary 18. For the sake of contradiction, we assume that $z' \leq 2k - 2$. Then $z' - m + 1 \leq (2k - 2) - m + 1$, and hence, there exist phrases q_i of $LZ(R_4)$ and p_j of $LZ(P_k)$ such that $\xi_{X \rightarrow ba}(q_i)$ contains $\xi_{X \rightarrow ba}(p_j)$ and their ending positions in F_{2k} are different. This contradicts that the greediness of phrase $\xi_{X \rightarrow ba}(p_j)$ of $LZ(F_{2k})$, Therefore, $|LZ(R_4)| = z' > 2k - 2$.

Case (5): Replacing all the occurrences of ba in F_{2k+1}

Let $R_5^{(X,a,b)}$ be the string obtained by replacing all the occurrences of ba in $F_{2k+1}^{(a,b)}$ with X . Since $F_{2k+1}^{(a,b)}$ ends with b , the last symbol of $R_5^{(X,a,b)}$ is b and it is unique in $R_5^{(X,a,b)}$. We omit the superscripts in the following. Since $F_{2k+1} = F_{2k}F_{2k-2}F_{2k-3}$, P_kP_{k-1} is a prefix of R_5 . By Lemma 6, the first $2k - 2$ phrases of $LZ(R_5)$ is the same as that of $LZ(P_{k+1})$. Also, the $(2k - 1)$ -th phrase ends at before b and the $2k$ -th phrase is b . Thus, $|LZ(R_5)| = 2k$.

Case (6): Replacing some but not all the occurrences of ba in F_{2k+1}

Let $R_6^{(X,a,b)}$ be any string obtained by replacing some but not all the occurrences of ba in $F_{2k+1}^{(a,b)}$ with X . Let $LZ(R_5^{(X,a,b)}) = (p_1, \dots, p_{2k})$ and $LZ(R_6^{(X,a,b)}) = (q_1, \dots, q_{z'})$ where $z' = |LZ(R_6^{(X,a,b)})|$. We omit the superscripts in the following. The first mismatch of boundaries between two factorizations is the position of the first occurrence of b in R_6 . Since the b is a fresh symbol, it is a length-1 phrase. Let the length-1 phrase be the m -th phrase in $LZ(R_6)$. Then, $\xi_{X \rightarrow ba}(p_m \dots p_{2k})$ is not longer than $\xi_{X \rightarrow ba}(q_m \dots q_{z'})$ by the greediness of $\xi_{X \rightarrow ba}(p_{m-1})$. For the sake of contradiction, we assume that $z' < 2k$. Then $z' - m + 1 < 2k - m + 1$, and hence, there exist phrases q_i of $LZ(R_6)$ and p_j of $LZ(R_5)$ such that $\xi_{X \rightarrow ba}(q_i)$ contains $\xi_{X \rightarrow ba}(p_j)$ and their ending positions in F_{2k+1} are different. This contradicts the greediness of phrase $\xi_{X \rightarrow ba}(p_j)$ of $LZ(F_{2k+1})$. Therefore, $|LZ(R_6)| = z' \geq 2k$.

The proofs for the remaining ten cases can be found in a full version of this paper [35]. We remark that the remaining ten cases can also be proven by similar argumentations.

6 Conclusions

In this paper, we analyzed the smallest grammars of Fibonacci words and completely characterized them by the RePair grammar-compressor. Namely, the set of all smallest grammars that produce only the n -th Fibonacci word F_n equals the set of all grammars obtained by applying (different implementations of) the RePair algorithm to F_n . Further, we showed that the size of the smallest grammars of F_n is n and that the number of such grammars is $2\lfloor n/2 \rfloor - 2$.

To show the smallest grammar size of F_n , we revisited the result on the lower bound of the sizes of grammars shown by Rytter [38]. Here, we gave a slightly tighter lower bound of the grammar size $z(w) - 1 + \sigma_w$ for *any* string w . Independent of the above results on Fibonacci words, this result on a lower bound is interesting since the result will help show the *exact* values of the smallest grammar size of other strings.

It is left as our future work to investigate whether it is possible to characterize the smallest grammars of other binary words, such as Thue-Morse words and Period-doubling words, by similar methods to Fibonacci words.

References

- 1 Alberto Apostolico and Stefano Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Data Compression Conference, DCC 2000, Snowbird, Utah, USA, March 28-30, 2000*, pages 143–152. IEEE Computer Society, 2000. doi:10.1109/DCC.2000.838154.
- 2 Hideo Bannai, Momoko Hirayama, Danny HucKe, Shunsuke Inenaga, Artur Jez, Markus Lohrey, and Carl Philipp Reh. The smallest grammar problem revisited. *IEEE Trans. Inf. Theory*, 67(1):317–328, 2021. doi:10.1109/TIT.2020.3038147.

- 3 Jean Berstel. *Fibonacci Words – A Survey*, pages 13–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986. doi:10.1007/978-3-642-95486-3_2.
- 4 Jean Berstel and Alessandra Savelli. Crochemore factorization of sturmian and other infinite words. In Rastislav Kralovic and Pawel Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 157–166. Springer, 2006. doi:10.1007/11821069_14.
- 5 Philip Bille, Anders Roy Christiansen, Patrick Hage Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *Theory Comput. Syst.*, 62(8):1715–1735, 2018. doi:10.1007/s00224-017-9839-9.
- 6 Philip Bille, Travis Gagie, Inge Li Gørtz, and Nicola Prezza. A separation between rslps and LZ77. *J. Discrete Algorithms*, 50:36–39, 2018. doi:10.1016/j.jda.2018.09.002.
- 7 Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-efficient Re-Pair compression. 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 171–180. IEEE, 2017. doi:10.1109/DCC.2017.24.
- 8 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. doi:10.1137/130936889.
- 9 Katrin Casel, Henning Fernau, Serge Gaspers, Benjamin Gras, and Markus L. Schmid. On the complexity of the smallest grammar problem over fixed alphabets. *Theory Comput. Syst.*, 65(2):344–409, 2021. doi:10.1007/s00224-020-10013-w.
- 10 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.
- 11 Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Trans. Web*, 4(4):16:1–16:31, 2010. doi:10.1145/1841909.1841913.
- 12 Maxime Crochemore. Linear searching for a square in a word. *Bulletin of the European Association of Theoretical Computer Science*, 24:66–72, 1984.
- 13 Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998. doi:10.1007/PL00009202.
- 14 Gabriele Fici. Factorizations of the fibonacci infinite word. *J. Integer Seq.*, 18(9):15.9.3, 2015. URL: <https://cs.uwaterloo.ca/journals/JIS/VOL18/Fici/fici5.html>.
- 15 Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. Practical grammar compression based on maximal repeats. *Algorithms*, 13(4):103, 2020. doi:10.3390/a13040103.
- 16 Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with rsync. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval – 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2019. doi:10.1007/978-3-030-32686-9_3.
- 17 Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. *J. ACM*, 68(4):27:1–27:40, 2021. doi:10.1145/3457389.
- 18 Michal Ganczorz and Artur Jez. Improvements on Re-Pair grammar compressor. 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 181–190. IEEE, 2017. doi:10.1109/DCC.2017.52.
- 19 Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In Rolf G. Karlsson and Andrzej Lingas, editors, *Algorithm Theory – SWAT ’96, 5th Scandinavian Workshop on Algorithm Theory, Reykjavík, Iceland, July 3-5, 1996, Proceedings*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 1996. doi:10.1007/3-540-61422-2_148.

- 20 Danny Hucke and Carl Philipp Reh. Approximation ratios of RePair, LongestMatch and Greedy on unary strings. *Algorithms*, 14(2):65, 2021. doi:10.3390/a14020065.
- 21 Tomohiro I. Longest common extensions with recompression. 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 18:1–18:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.18.
- 22 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015. doi:10.1016/j.ic.2014.09.009.
- 23 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/j.tcs.2016.03.005.
- 24 Artur Jez. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027.
- 25 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.
- 26 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. doi:10.1016/S0304-3975(02)00426-7.
- 27 John C. Kieffer, En-Hui Yang, Gregory J. Nelson, and Pamela C. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(4):1227–1245, 2000. doi:10.1109/18.850665.
- 28 Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. URL: <https://www.worldcat.org/oclc/312898417>.
- 29 Dominik Köppl, Tomohiro I, Isamu Furuya, Yoshimasa Takabatake, Kensuke Sakai, and Keisuke Goto. Re-PAIR in small space. *Algorithms*, 14(1):5, 2021. doi:10.3390/a14010005.
- 30 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference, DCC 1999, Snowbird, Utah, USA, March 29-31, 1999*, pages 296–305. IEEE Computer Society, 1999. doi:10.1109/DCC.1999.755679.
- 31 Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complex. Cryptol.*, 4(2):241–299, 2012. doi:10.1515/gcc-2012-0016.
- 32 Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePAIR. *Inf. Syst.*, 38(8):1150–1167, 2013. doi:10.1016/j.is.2013.06.006.
- 33 Takuya Masaki and Takuya Kida. Online grammar transformation based on Re-PAIR algorithm. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 – April 1, 2016*, pages 349–358. IEEE, 2016. doi:10.1109/DCC.2016.69.
- 34 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.
- 35 Takuya Mieno, Shunsuke Inenaga, and Takashi Horiyama. Repair grammars are the smallest grammars for Fibonacci words. *CoRR*, abs/2202.08447, 2022. URL: <https://arxiv.org/abs/2202.08447>.
- 36 Gonzalo Navarro and Cristian Urbina. On stricter reachable repetitiveness measures. 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 193–206. Springer, 2021. doi:10.1007/978-3-030-86692-1_16.

- 37 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 – Kraków, Poland*, volume 58 of *LIPICs*, pages 72:1–72:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.MFCS.2016.72.
- 38 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.
- 39 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.
- 40 Toshiya Tanaka, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing convolution on grammar-compressed text. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2013 Data Compression Conference, DCC 2013, Snowbird, UT, USA, March 20-22, 2013*, pages 451–460. IEEE, 2013. doi:10.1109/DCC.2013.53.
- 41 En-Hui Yang and John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE Trans. Inf. Theory*, 46(3):755–777, 2000. doi:10.1109/18.841161.
- 42 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.
- 43 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

Minimal Absent Words on Run-Length Encoded Strings

Tooru Akagi ✉

Department of Informatics, Kyushu University, Fukuoka, Japan

Kouta Okabe ✉

Department of Information Science and Technology, Kyushu University, Fukuoka, Japan

Takuya Mieno¹ ✉ 

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

Yuto Nakashima ✉ 

Department of Informatics, Kyushu University, Fukuoka, Japan

Shunsuke Inenaga² ✉ 

Department of Informatics, Kyushu University, Fukuoka, Japan

PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan

Abstract

A string w is called a *minimal absent word* for another string T if w does not occur (as a substring) in T and all proper substrings of w occur in T . State-of-the-art data structures for reporting the set $\text{MAW}(T)$ of MAWs from a given string T of length n require $O(n)$ space, can be built in $O(n)$ time, and can report all MAWs in $O(|\text{MAW}(T)|)$ time upon a query. This paper initiates the problem of computing MAWs from a compressed representation of a string. In particular, we focus on the most basic compressed representation of a string, *run-length encoding (RLE)*, which represents each maximal run of the same characters a by a^p where p is the length of the run. Let m be the RLE-size of string T . After categorizing the MAWs into five disjoint sets $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4, \mathcal{M}_5$ using RLE, we present matching upper and lower bounds for the number of MAWs in \mathcal{M}_i for $i = 1, 2, 4, 5$ in terms of RLE-size m , except for \mathcal{M}_3 whose size is unbounded by m . We then present a compact $O(m)$ -space data structure that can report all MAWs in optimal $O(|\text{MAW}(T)|)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, combinatorics on words, minimal absent words, run-length encoding

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.27

Funding *Takuya Mieno*: JSPS KAKENHI Grant Number JP20J11983

Yuto Nakashima: JSPS KAKENHI Grant Number JP18K18002, JP21K17705

Shunsuke Inenaga: JST PRESTO Grant Number JPMJPR1922

Acknowledgements We thank the anonymous referees for their comments.

1 Introduction

An *absent word* (a.k.a. *a forbidden word*) for a string T is a non-empty string that is *not* a substring of T . An absent word X for T is said to be a *minimal absent word (MAW)* for T if all proper substrings of X occur in T . MAWs are combinatorial string objects, and their interesting mathematical properties have extensively been studied in the literature

¹ Current affiliation: University of Electro-Communications, Japan (tmieno@uec.ac.jp)

² Corresponding author



(see [5, 14, 16, 13, 23, 1] and references therein). MAWs also enjoy several applications including phylogeny [8], data compression [12, 15, 3], musical information retrieval [11], and bioinformatics [2, 9, 24, 21].

Thus, given a string T of length n over an alphabet of size σ , computing the set $\text{MAW}(T)$ of all MAWs for T is an interesting and important problem: Crochemore et al. [14] presented the first efficient data structure of $O(n)$ space which outputs all MAWs in $\text{MAW}(T)$ in $O(\sigma n)$ time and $O(n)$ working space. Since the number $|\text{MAW}(T)|$ of MAWs for T can be as large as $O(\sigma n)$ and there exist strings S for which $|\text{MAW}(S)| \in \Omega(\sigma|S|)$ [14], Crochemore et al.'s algorithm [14] runs in optimal time in the worst case. Later, Fujishige et al. [19] presented an improved data structure of $O(n)$ space, which can report all MAWs in $O(n + |\text{MAW}(T)|)$ time and $O(n)$ working space. Fujishige et al.'s algorithm [19] can easily be modified so it uses $O(|\text{MAW}(T)|)$ time for reporting all MAWs, by explicitly storing all MAWs when $|\text{MAW}(T)| \in O(n)$. The key tool used in these two algorithms is an $O(n)$ -size automaton called the *DAWG* [7], which accepts all substrings of T . The DAWG for string T can be built in $O(n \log \sigma)$ time for general ordered alphabets [7], or in $O(n)$ time for integer alphabets of size polynomial in n [19]. There also exist other efficient algorithms for computing MAWs with other string data structures such as suffix arrays and Burrows-Wheeler transforms [6, 4]. MAWs in other settings have also been studied in the literature, including length specified versions [10], the sliding window versions [13, 23, 1], circular string versions [18], and labeled tree versions [17].

In this paper, we initiate the study of computing MAWs for *compressed* strings. As the first step of this line of research, we consider strings which are compactly represented by *run-length encoding* (*RLE*). Let m be the size of the RLE of an input string T . We first categorize the elements of $\text{MAW}(T)$ into five disjoint subsets $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$, and \mathcal{M}_5 , by considering how the MAWs can be related to the boundaries of maximal character runs in T (Section 2). In Section 3 and Section 4, we present matching upper bounds and lower bounds for their sizes $|\mathcal{M}_i|$ ($i = 1, 2, 4, 5$) in terms of the RLE size m or the number σ'_T of distinct characters occurring in T . Notice that $\sigma'_T \leq m$ always holds. The exception is \mathcal{M}_3 , which can contain $\Omega(n)$ MAWs regardless of the RLE size m . Still, in Section 5 we propose our RLE-compressed $O(m)$ -space data structure that can enumerate all MAWs for T in output-sensitive $O(|\text{MAW}(T)|)$ time. Since $m \leq n$ always holds, our result is an improvement over Crochemore et al.'s and Fujishige et al.'s results both of which require $O(n)$ space to store representations of all MAWs. Charalampopoulos et al. [10] showed how one can use *extended bispecial factors* of T to represent all MAWs for T in $O(n)$ space, and to output all MAWs in optimal $O(|\text{MAW}(T)|)$ time upon a query. While the way how we characterize the MAWs may be seen as the RLE version of their method based on the extended bispecial factors, our $O(m)$ -space data structure cannot be obtained by a straightforward extension from [10], since there exists a family of strings over a constant-size alphabet for which the RLE-size is $m \in O(1)$ but $|\text{MAW}(T)| \in \Omega(n)$. We note that, by the use of *truncated RLE suffix arrays* [25], our $O(m)$ -space data structure can be built in $O(m \log m)$ time with $O(m)$ working space (the details of the construction will be presented in the full version of this paper).

2 Preliminaries

2.1 Strings

Let Σ be an ordered alphabet. An element of Σ is called a character. An element of Σ^* is called a string. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0. If $T = xyz$, then x , y , and z are called a *prefix*, *substring*, and *suffix* of T ,

respectively. They are called a *proper prefix*, *proper substring*, and *proper suffix* of T if $x \neq T$, $y \neq T$, and $z \neq T$, respectively. For any $1 \leq i \leq |T|$, the i -th character of T is denoted by $T[i]$. For any $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T starting at i and ending at j . For any $i \leq |T|$ and $1 \leq j$, let $T[..i] = T[1..i]$ and $T[j..] = T[j..|T|]$. We say that a string w occurs in a string T if w is a substring of T . Note that by definition, the empty string ε is a substring of any string T and hence ε always occurs in T .

Let $\#_T w$ denote the number of occurrences of a string w in a string T . We will abbreviate it to $\#w$ when no confusion occurs.

2.2 Run length encoding (RLE) and bridges

The *run-length encoding* $\text{rle}(T)$ of string T is a compact representation of T such that each maximal run of the same characters in T is represented by a pair of the character and the length of the maximal run. More formally, $\text{rle}(T) = a_1^{p_1} \cdots a_m^{p_m}$ encodes each substring $T[i..i+p-1]$ by a^p if $T[j] = a \in \Sigma$ for every $i \leq j \leq i+p-1$, $T[i-1] \neq T[i]$, and $T[i+p-1] \neq T[i+p]$. Each a^p in $\text{rle}(T)$ is called a (character) *run*, and p is called the exponent of this run. The j -th maximal run in $\text{rle}(T)$ is denoted by r_j , namely $\text{rle}(T) = r_1 \cdots r_m$. The *size* of $\text{rle}(T)$, denoted $R(T)$, is the number of maximal character runs in $\text{rle}(T)$. E.g., for a string $T = \text{aacccccccbbabbbb}$ of length 18, $\text{rle}(T) = \text{a}^2\text{c}^7\text{b}^2\text{a}^1\text{b}^4$ and $R(T) = 5$.

Our model of computation is a standard word RAM with machine word size $\Omega(\log |T|)$, and the space requirements of our data structures will be measured by the number of words (not bits). Thus, $\text{rle}(T)$ of size m can be stored in $O(m)$ space.

2.3 Bridges

A string $w \in \Sigma^*$ of length $|w| \geq 2$ is said to be a *bridge* if $w[1] \neq w[2]$ and $w[|w|-1] \neq w[|w|]$. In other words, both of the first run and the last run in $\text{rle}(w)$ are of length 1. A substring of T that is a bridge is called a *bridge substring* of T . Let B_ℓ denote the set of bridge substrings w of T with $R(w) = \ell$. Further let $\mathcal{B} = \bigcup_\ell B_\ell$ be the set of all bridge substrings of T . For example, for the same string $T = \text{aacccccccbbabbbb}$ as the above one, the substring $\text{ac}^7\text{b}^2\text{a}$ of T is a bridge, and $B_4 = \{\text{ac}^7\text{b}^2\text{a}, \text{cb}^2\text{a}^1\text{b}\}$. For a string w with $R(w) \geq 3$, we can obtain a bridge substring of w by removing the first and the last runs of w and then *shrinking* the runs at both ends so that their exponents are 1. We denote by $\text{shk}(w)$ such shrunk bridge. For convenience, let $\text{shk}(w) = \varepsilon$ if $R(w) \leq 2$. Also, for every $k \geq 2$, we denote $\text{shk}^k(w) = \text{shk}(\text{shk}^{k-1}(w))$. For example, consider the same T as the above again, $\text{shk}(T) = \text{acccccccbbab}$, $\text{shk}^2(w) = \text{cbba}$, $\text{shk}^3(w) = \text{b}$, and $\text{shk}^k(w) = \varepsilon$ for any $k \geq 4$.

2.4 Minimal absent words (MAWs)

A string $w \in \Sigma^*$ is called an *absent word* for a string T if w does not occur in T , namely if $\#w = 0$. An absent word w for T is called a *minimal absent word* or *MAW* for T if all proper substrings of w occur in T . We denote by $\text{MAW}(T)$ the set of all MAWs for T . An alternative definition of MAWs is such that a string aub of length at least two with $a, b \in \Sigma$ and $u \in \Sigma^*$ is a MAW of T if $\#(aub) = 0$, $\#(au) \geq 1$ and $\#(ub) \geq 1$. For a MAW of length 1 (namely a character not occurring in T), we use a convention that $u = \varepsilon$ and a and b are united into a single character.

The MAWs in $\text{MAW}(T)$ are partitioned into the following five disjoint subsets \mathcal{M}_i ($1 \leq i \leq 5$) based on their RLE sizes $R(aub)$:

27:4 Minimal Absent Words on Run-Length Encoded Strings

- $\mathcal{M}_1 = \{aub \in \text{MAW}(T) \mid R(aub) = 1\}$;
- $\mathcal{M}_2 = \{aub \in \text{MAW}(T) \mid R(aub) = 2, u = \varepsilon\}$;
- $\mathcal{M}_3 = \{aub \in \text{MAW}(T) \mid R(aub) = 3, a \neq u[1] \text{ and } b \neq u[|u|]\}$;
- $\mathcal{M}_4 = \{aub \in \text{MAW}(T) \mid R(aub) \geq 4, a \neq u[1] \text{ and } b \neq u[|u|]\}$;
- $\mathcal{M}_5 = \{aub \in \text{MAW}(T) \mid R(aub) \geq 2, a = u[1] \text{ or } b = u[|u|]\}$.

For $1 \leq i \leq 5$, a MAW aub in \mathcal{M}_i is called of *type i* .

In the rest of this paper, we will consider an arbitrarily fixed string T of length n . For convenience, we assume that $n \geq 3$ and that there are special terminal symbols $T[1] = T[n] = \$ \notin \Sigma$ not occurring inside T . Since $\$ \notin \Sigma$, we do not consider any MAW containing $\$$ for T in our arguments to follow (recall that a MAW must be an element of Σ^*). In addition, since $\$$ does not occur elsewhere in T , $\text{MAW}(T) = \text{MAW}(T[2..n-1])$ holds.

► **Example 1.** Consider $T = \$b^2ac^3ba^2\$ = \$bbaccbbaa\$$. All MAWs in $\text{MAW}(T)$ are divided into the following five types: $\mathcal{M}_1 = \{aaa, bbb, cccc\}$; $\mathcal{M}_2 = \{ca, bc\}$; $\mathcal{M}_3 = \{acb, accb\}$; $\mathcal{M}_4 = \{cbac\}$; $\mathcal{M}_5 = \{bbaa\}$.

Let Σ' denote the set of characters occurring in T except for $\$$. Let $\sigma' = |\Sigma'|$ be the number of distinct characters occurring in $T[2..n-1]$.

3 Upper bounds on the number of MAWs for RLE strings

In this section, we present upper bounds for the number of MAWs in a string T that is represented by its RLE $\text{rle}(T)$ of size $R(T) = m$.

3.1 Upper bounds for the number of MAWs of type 1, 2, 3, 5

We first consider the number of MAWs except for those of type 4.

► **Lemma 2.** $|\mathcal{M}_1| = \sigma$.

Proof. By the definition of \mathcal{M}_1 , any MAW in \mathcal{M}_1 is of the form a^k . For any character $\alpha \in \Sigma'$ that occurs in T , let $aub = \alpha^{p+1}$ such that α^p is the *longest* maximal run of α in T . Clearly $\alpha^p = au = ub$ occurs in T and α^{p+1} does not occur in T . Since $R(aub) = R(\alpha^{p+1}) = 1$, $\alpha^{p+1} \in \mathcal{M}_1$ and it is the unique MAW of type 1 consisting of α 's. For any character $\beta \in \Sigma \setminus \Sigma'$ that does not occur in T , clearly β is a MAW of T and $\beta \in \mathcal{M}_1$ since $R(\beta) = 1$. In total, we obtain $|\mathcal{M}_1| = \sigma$. ◀

Note that this upper bound for $|\mathcal{M}_1|$ is tight for any string T and alphabet Σ of size σ .

► **Lemma 3.** $|\mathcal{M}_2| \in O((\sigma')^2)$.

Proof. Any MAW in \mathcal{M}_2 is of the form ab with $a, b \in \Sigma$ and $a \neq b$. By the definition of MAWs, ab can be a MAW for T only if both a and b occur in T , which implies that $a, b \in \Sigma'$. The number of such combinations of a and b is $\sigma'(\sigma' - 1)$. ◀

Since $\sigma' \leq m$ always holds, we have that $|\mathcal{M}_2| \in O(m^2)$. Later we will show that this upper bound for $|\mathcal{M}_2|$ is asymptotically tight.

► **Lemma 4.** $|\mathcal{M}_3|$ is unbounded by m .

Proof. Consider a string $T = ac^{n-2}b$, where $a \neq c$ and $c \neq b$. Then $ac^k b$ for each $1 \leq k \leq n-3$ is a MAW of T and $R(ac^k b) = 3$. Since they are the only type 3 MAWs of T , we have that $|\mathcal{M}_3| = n - 3$. Clearly, the original length n of T cannot be bounded by $m = R(T) = 3$. ◀

Although the number of MAWs of type 3 is unbounded by m , later we will present an $O(m)$ -space data structure that can enumerate all elements in \mathcal{M}_3 in output-sensitive time.

► **Lemma 5.** $|\mathcal{M}_5| \in O(m)$.

Proof. Any MAW $aub \in \mathcal{M}_5$ can be represented by $a^{i+1}vb$ or avb^{i+1} with maximal integer $i \geq 1$, where $a^i v = u$ in the former and $vb^i = u$ in the latter. Let us consider the case of $a^{i+1}vb$ as the case of avb^{i+1} is symmetric. Then $ca^i vb$ with some character $c \neq a$ must occur in T . Let k be the beginning position of an occurrence of $ca^i vb$ in T . Then, $T[k+1..k+i] = a^i$ is a maximal run of a .

Now consider any distinct MAW $a^{i+1}v'b' \in \mathcal{M}_5 \setminus \{a^{i+1}vb\}$ with $v'b' \neq vb$. Again, $c'a^i v'b'$ with some character $c' \neq a$ must occur in T . Suppose on the contrary that $c'a^i v'b'$ has an occurrence beginning at the same position k as $ca^i vb$. This implies that $c' = c$, and both $a^i vb$ and $a^i v'b'$ are prefixes of $T[k+1..|T|]$.

- If $|a^i vb| < |a^i v'b'|$, then $a^i v'$ contains $a^i vb$ as a substring. Since $a^{i+1}v'$ occurs in T , $a^{i+1}vb$ must also occur in T . Hence $a^{i+1}vb$ is not a MAW for T , a contradiction.
- If $|a^i vb| > |a^i v'b'|$, then $a^i v$ contains $a^i v'b'$ as a substring. Thus $a^{i+1}vb$ is an absent word for T but it is not minimal. Hence $a^{i+1}vb$ is not a MAW for T , a contradiction.
- If $|a^i vb| = |a^i v'b'|$, then this contradicts that $a^i vb \neq a^i v'b'$.

Hence, at most two element of \mathcal{M}_5 can be associated with a position k in T such that $T[k] \neq T[k+1]$. The number of such positions does not exceed $2m$. ◀

3.2 Upper bound for the number of MAWs of type 4

In the rest of this section, we show an upper bound of the number of MAWs of type 4. Namely, we prove the following lemma.

► **Lemma 6.** $|\mathcal{M}_4| \in O(m^2)$.

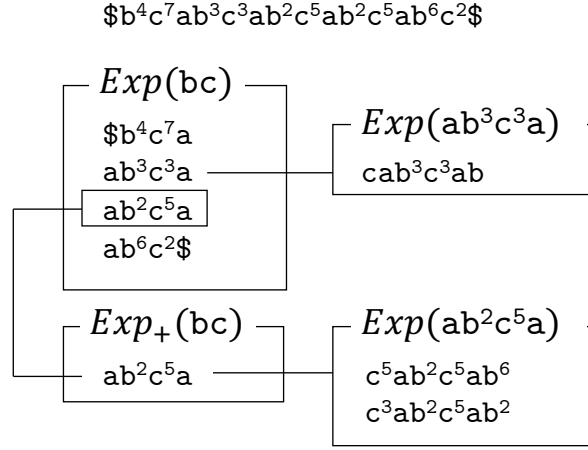
Firstly, we explain a way to characterize MAWs of type 4. For any string $w \in \Sigma^*$ and integer $t > 0$, let $\text{Exp}^t(w)$ be the set of bridges such that $\text{Exp}^t(w) = \{w' \in \mathcal{B} \mid \text{shk}^t(w') = w\}$. Namely, $\text{Exp}^t(w)$ is the *inverse image* of $\text{shk}^t(w') = w$ for bridge substrings w' of T . We use $\text{Exp}(w)$ to denote $\text{Exp}^1(w)$. Figure 1 gives an example for $\text{Exp}^t(w)$ ($\text{Exp}_+^t(w)$ in the figure will be defined later). Any MAW z in \mathcal{M}_4 is of the form $a\alpha^i u \beta^j b$ with $a, b, \alpha, \beta \in \Sigma, u \in \Sigma^*$, and positive integers i, j where a, α^i, β^j, b are the first, the second, the second last, and the last run of z , respectively. By the definition of MAWs, both the suffix $\alpha^i u \beta^j b$ and the prefix $a\alpha^i u \beta^j$ of z occur in T . From this fact, we can obtain the following observations.

► **Observation 7.** Each MAW $z \in \mathcal{M}_4$ corresponds to a pair of distinct bridges $(w_1, w_2) \in \text{Exp}(\text{shk}(z)) \times \text{Exp}(\text{shk}(z))$. Formally, for each MAW $z = a\alpha^i u \beta^j b \in \mathcal{M}_4$, there exist characters $a_1, b_1 \in \Sigma \cup \{\$\}$ and integers $i_1 \geq i, j_1 \geq j$ such that $w_1 = a_1 \alpha^{i_1} u \beta^{j_1} b, w_2 = a \alpha^i u \beta^j b_1 \in \text{Exp}(\text{shk}(z))$ and $w_1 \neq w_2$ (since these two occur in T but z does not occur in T).

This observation gives a main idea of our characterization which is stated in the following lemma.

► **Lemma 8.** For any bridge w , $|\{z \mid \text{shk}(z) = w, z \in \mathcal{M}_4\}| \leq |\text{Exp}(w)|(|\text{Exp}(w)| - 1)$.

Proof. Let $\mathcal{M}_4(w) = \{z \mid \text{shk}(z) = w, z \in \mathcal{M}_4\}$. By Observation 7, each $z \in \mathcal{M}_4(w)$ corresponds to a pair $(w_1, w_2) \in \text{Exp}(\text{shk}(z)) \times \text{Exp}(\text{shk}(z))$ where $w_1 \neq w_2$. Let $z_1 = a_1 \alpha^{i_1} u \beta^{j_1} b_1, z_2 = a_2 \alpha^{i_2} u \beta^{j_2} b_2$ be distinct MAWs in $\mathcal{M}_4(w)$ where $\text{shk}(z_1) = \text{shk}(z_2) = w$. Assume towards a contradiction that z_1 and z_2 correspond to $(a' \alpha^{i'} u \beta^{j'} b, a \alpha^i u \beta^j b') \in$



■ **Figure 1** The bridge $w_1 = ab^2c^5a \in \text{Exp}(bc)$ is an element of $\text{Exp}_+(bc)$ since $|\text{Exp}(w_1)| \geq 2$. On the other hand, the bridge $w_2 = ab^3c^3a \in \text{Exp}(bc)$ is not an element of $\text{Exp}_+(bc)$ since $|\text{Exp}(w_2)| < 2$.

$\text{Exp}(w) \times \text{Exp}(w)$. This implies that, by Observation 7, $i = i_1 = i_2, j = j_1 = j_2, a = a_1 = a_2, b = b_1 = b_2$. Thus $z_1 = z_2$ holds, a contradiction. Hence, for any distinct MAWs $z_1, z_2 \in \mathcal{M}_4(w)$, z_1 and z_2 correspond to distinct elements of $\text{Exp}(\text{shk}(z)) \times \text{Exp}(\text{shk}(z))$. Since the number of elements (w_1, w_2) in $\text{Exp}(\text{shk}(z)) \times \text{Exp}(\text{shk}(z))$ such that $w_1 \neq w_2$ is $|\text{Exp}(w)|(|\text{Exp}(w)| - 1)$, this lemma holds. ◀

Since each MAW z corresponds to an element $(w_1, w_2) \in \text{Exp}(\text{shk}(z)) \times \text{Exp}(\text{shk}(z))$ such that $w_1 \neq w_2$, it is enough for the bound to sum up all $|\text{Exp}(w)|^2$ such that $|\text{Exp}(w)| \geq 2$ holds. Let \mathcal{W} be the set of bridges w such that $|\text{Exp}(w)| \geq 2$ or $w \in B_2 \cup B_3$. Let $\mathcal{X} = \sum_{w \in \mathcal{W}} |\text{Exp}(w)|$. For considering such $\text{Exp}(w)$, we also define a subset $\text{Exp}_+^t(w)$ of $\text{Exp}^t(w)$ as follows: For any string (bridge) w and integer $t > 0$,

$$\text{Exp}_+^t(w) = \{w' \mid w' \in \text{Exp}^t(w), |\text{Exp}(w')| \geq 2\}.$$

We also use $\text{Exp}_+(w)$ to denote $\text{Exp}_+^1(w)$. Figure 2 shows an illustration for $\text{Exp}^i(w), \text{Exp}_+^i(w), \mathcal{W}$, and \mathcal{X} . We give the following lemma that explains relations between $\text{Exp}^i(w), \text{Exp}_+^i(w)$, and \mathcal{X} .

► **Lemma 9.**

$$\mathcal{X} = \sum_{w \in B_2 \cup B_3} \left(|\text{Exp}(w)| + \sum_{i=1}^{\lfloor m/2 \rfloor - 1} \sum_{z \in \text{Exp}_+^i(w)} |\text{Exp}(z)| \right).$$

Proof. Let z_{even} be a bridge where $R(z_{\text{even}}) = 2i + 2$ for some $i \geq 1$. Notice that $\text{shk}(z_{\text{even}}) = c_1c_2 \in B_2$ for some distinct characters c_1, c_2 . By the definition of $\text{Exp}_+^i(\cdot)$, if $|\text{Exp}(z_{\text{even}})| \geq 2$, then $z_{\text{even}} \in \text{Exp}_+^i(c_1c_2)$. Let z_{odd} be a bridge where $R(z_{\text{odd}}) = 2i + 3$ for some $i \geq 1$. Notice that $\text{shk}(z_{\text{odd}}) = c_1c_2^kc_3 \in B_3$ for some characters c_1, c_2, c_3 and an integer $k \geq 1$. By the definition of $\text{Exp}_+^i(\cdot)$, if $|\text{Exp}(z_{\text{odd}})| \geq 2$, then $z_{\text{odd}} \in \text{Exp}_+^i(c_1c_2^kc_3)$. Therefore the statement holds. ◀

This implies that $|\mathcal{M}_4| \leq \sum_{w \in \mathcal{W}} |\text{Exp}(w)|^2 \leq \mathcal{X}^2$. Thus, if $\mathcal{X} \in O(m)$, $|\mathcal{M}_4| \in O(m^2)$.

We can also observe that $\sum_{i=1}^{\lfloor m/2 \rfloor - 1} \sum_{z \in \text{Exp}_+^i(w)} |\text{Exp}(z)|$ is the sum of the number of children of black nodes (which have more than a single child) in the tree for w . The number of leaves of the tree is an upper bound for the sum. It is also clear that $|\text{Exp}(w)|$ can be

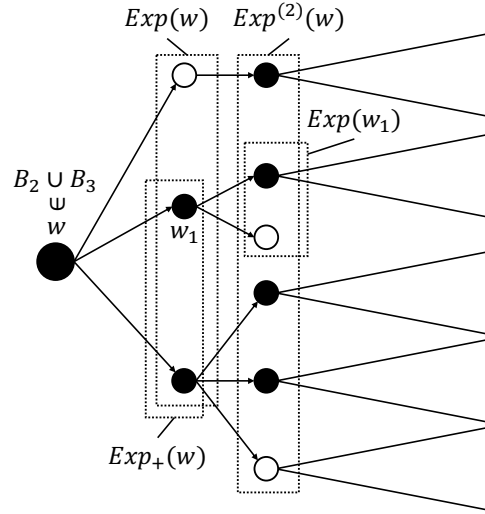


Figure 2 This tree shows an illustration for $\text{Exp}^i(w)$, $\text{Exp}_+^i(w)$, \mathcal{W} , and \mathcal{X} . The root node represents a bridge $w \in B_2 \cup B_3$. The set of children of the root corresponds to $\text{Exp}(w)$, namely, each child x represents a bridge such that $\text{shk}(x) = w$. Each black node represents a bridge x such that $|\text{Exp}(x)| \geq 2$ (i.e., each black node has at least two children) or the root. Let $W(w)$ be the set of nodes consisting of all the black nodes in the tree rooted at a bridge $w \in B_2 \cup B_3$. Then \mathcal{W} is the union of $W(w)$ for all $w \in B_2 \cup B_3$, and \mathcal{X} is the total number of children of black nodes in \mathcal{W} .

bounded by the number of leaves of the tree (In Appendix we give a more mathematical description for the above discussion as Observation 23 and Proposition 24). Consequently, we obtain $|\mathcal{X}| \in O(m)$ as in Lemma 10.

► **Lemma 10.** $|\mathcal{X}| \in O(m)$.

Proof. By Lemma 9 and the above discussion, we have

$$\begin{aligned} \mathcal{X} &= \sum_{w \in B_2 \cup B_3} \left(|\text{Exp}(w)| + \sum_{i=1}^{\lfloor m/2 \rfloor - 1} \sum_{z \in \text{Exp}_+^i(w)} |\text{Exp}(z)| \right) \\ &\leq \sum_{w \in B_2 \cup B_3} 2\#w \\ &\leq 2((m-1) + (m-2)) \in O(m). \end{aligned} \quad \blacktriangleleft$$

We are ready to prove Lemma 6:

Proof of Lemma 6. $|\mathcal{M}_4| \leq \sum_{w \in \mathcal{W}} |\text{Exp}(w)|^2 \leq |\mathcal{X}|^2 \leq (2(2m-3))^2 \in O(m^2)$. ◀

4 Lower bounds on the number of MAWs for RLE strings

In the previous section, we showed a tight bound $|\mathcal{M}_1| = \sigma$, and showed that $|\mathcal{M}_3|$ is unbounded by the RLE size m . In this section, we give tight lower bounds for the sizes of \mathcal{M}_2 , \mathcal{M}_3 , and \mathcal{M}_5 which asymptotically match the upper bounds given in the previous section. Throughout this section, we omit the terminal \$ at either end of T , since our lower bound instances do not need them.

► **Lemma 11.** *There exists a string T such that $|\mathcal{M}_2| = \sigma'(\sigma' - 2) + 1$.*

Proof. Let $T = 123 \cdots \sigma'$, where all characters in T are mutually distinct. Any bigram occurring in T is of the form $i(i+1)$ with $1 \leq i < \sigma'$. Thus, for each $1 \leq i < \sigma'$, bigram $i \cdot j$ with any $j \in \{1, \dots, i-1, i+2, \dots, \sigma'\}$ is a type-2 MAW for T , and bigram $\sigma' \cdot j$ is a type-2 MAW for T . Namely, the set \mathcal{M}_2 of type-2 MAWs for T is:

$$\mathcal{M}_2 = \left\{ \begin{array}{l} 13, \dots, 1\sigma', \\ 21, 24, \dots, 2\sigma', \\ 31, 32, 35, \dots, 3\sigma', \\ \dots, \\ (\sigma' - 1)1, \dots, (\sigma' - 1)(\sigma' - 2), \\ \sigma'1, \dots, \sigma'(\sigma' - 1) \end{array} \right\}.$$

Thus we have $|\mathcal{M}_2| = \sigma'(\sigma' - 2) + 1$ for this string T . \blacktriangleleft

Since $\sigma' = m$ for the string T of Lemma 11, we obtain a tight lower bound $|\mathcal{M}_2| \in \Omega(m^2)$ in terms of m . The string $T = 123 \cdots \sigma'$ can easily be generalized so that $m < n$, where $n = |T|$. For instance, consider $T' = 1^{p_1}2^{p_2}3^{p_3} \cdots \sigma'^{p_{\sigma'}}$ with $p_i > 1$ for each i . The set of type-2 MAWs for T' is equal to that for T .

► **Lemma 12.** *There exists a string T with $R(T) = m$ such that $|\mathcal{M}_4| \in \Omega(m^2)$.*

Proof. Consider string $T = \text{abc}^p \cdot \text{ab}^2\text{c}^{p-1} \cdot \text{ab}^3\text{c}^{p-2} \cdot \text{ab}^4\text{c}^{p-3} \cdots \text{ab}^{p-1}\text{c}^2 \cdot \text{ab}^p\text{c} \cdot \text{a}$, where a , b , and c are mutually distinct characters. Then the set of type-4 MAWs for T is a superset of the following set:

$$\left\{ \begin{array}{l} \text{abca}, \text{abc}^2\text{a}, \dots, \text{abc}^{p-1}\text{a}, \\ \text{ab}^2\text{ca}, \text{ab}^2\text{c}^2\text{a}, \dots, \text{ab}^2\text{c}^{p-2}\text{a}, \\ \text{ab}^3\text{ca}, \text{ab}^3\text{c}^2\text{a}, \dots, \text{ab}^3\text{c}^{p-3}\text{a}, \\ \dots, \\ \text{ab}^{p-2}\text{ca}, \text{ab}^{p-2}\text{c}^2\text{a}, \\ \text{ab}^{p-1}\text{ca} \end{array} \right\}.$$

Since $m = 3p + 1$, we have $|\mathcal{M}_4| > p(p-1)/2 \in \Omega(p^2) = \Omega(m^2)$. \blacktriangleleft

► **Lemma 13.** *There exists a string T with $R(T) = m$ such that $|\mathcal{M}_5| \in \Omega(m)$.*

Proof. Consider string $T = \text{abc} \cdot \text{ab}^2\text{c}^2 \cdot \text{ab}^3\text{c}^3 \cdots \text{ab}^p\text{c}^p \cdot \text{a}$, where a , b , and c are mutually distinct characters. Then the set of type-5 MAWs for T is a superset of the set

$$\{\text{b}^{i+1}\text{c}^i\text{a} \mid 1 \leq i \leq p-1\}.$$

Since $m = 3p + 1$, $|\mathcal{M}_5| > p - 1 \in \Omega(p) = \Omega(m)$. \blacktriangleleft

5 Efficient representations of MAWs for RLE strings

Consider a string T that contains σ' distinct characters. In this section, we present compact data structures that can output every MAW for T upon query, using a total of $O(m)$ space, where $m = R(T)$ is the size of $\text{rle}(T)$. We will prove the following theorem:

► **Theorem 14.** *There exists a data structure \mathcal{D} of size $O(m)$ which can output all MAWs for string T in $O(|\text{MAW}(T)|)$ time, where m is the RLE-size of T .*

In our representation of MAWs that follows, we store $\text{rle}(T)$ explicitly with $O(m)$ space. The following is a general lemma that we can use when we output a MAW from our data structures.

► **Lemma 15.** *For each MAW $w \in \text{MAW}(T)$, $\text{rle}(w)$ of size $R(w)$ can be retrieved in $O(R(w))$ time from a tuple (a, i, s, t, b, j) and $\text{rle}(T)$, where $a, b \in \Sigma$, $0 \leq i, j \leq |T|$, and $0 \leq s, t \leq m$.*

Proof. When $R(w) = 1$ (i.e. $w \in \mathcal{M}_1$), then since w is of the form a^i with $i \geq 1$, we can simply represent it by $(a, i, 0, 0, 0, 0)$.

When $R(w) \geq 2$, then let $w = aub$. When $aub \in \mathcal{M}_2$, then $w = ab$ and thus it can be simply represented by $(a, 1, 0, 0, b, 1)$. When $aub \in \mathcal{M}_3 \cup \mathcal{M}_4$, then $a \neq u[1]$ and $b \neq u[|u|]$. Hence it can be represented by $(a, 1, s, t, b, 1)$ where $r_s \cdots r_t = \text{rle}(u)$. When $aub \in \mathcal{M}_5$, then $a = u[1]$ or $u[|u|] = b$. Let i, j be the maximal integers such that $a^i u^j b = aub$. We can represent it by (a, i, s, t, b, j) with $r_s \cdots r_t = \text{rle}(u')$. ◀

For ease of discussion, in what follows, we will identify each MAW w with its corresponding tuple (a, i, s, t, b, j) which takes $O(1)$ space.

5.1 Representation for \mathcal{M}_1

We have shown that $|\mathcal{M}_1| = \sigma$ (Lemma 2), however, σ can be larger than σ' and m . However, a simple representation for \mathcal{M}_1 exists, as follows:

► **Lemma 16.** *There exists a data structure D_1 of $O(\sigma') \subseteq O(m)$ space that can output each MAW in \mathcal{M}_1 in $O(1)$ time.*

Proof. For ease of explanation, assume that the string T is over the integer alphabet $\Sigma = \{1, \dots, \sigma\}$ and let $\Sigma' = \{c_1, \dots, c_{\sigma'}\} \subseteq \{1, \dots, \sigma\}$. Let $M = \langle c_1^{p_1}, \dots, c_{\sigma'}^{p_{\sigma'}} \rangle$ be the list of type-1 MAWs in \mathcal{M}_1 that are runs of characters in Σ' , sorted in the lexicographical order of the characters, i.e. $1 \leq c_1 < \dots < c_{\sigma'} \leq \sigma$. We store M explicitly in $O(\sigma')$ space. When we output each MAW in \mathcal{M}_1 , we test the numbers (i.e. characters) in $\Sigma = \{1, \dots, \sigma\}$ incrementally, and scan M in parallel: For each $c = 1, \dots, \sigma$ in increasing order, if $c^p \in M$ with some $p > 1$ then we output c^p , and otherwise we output c . ◀

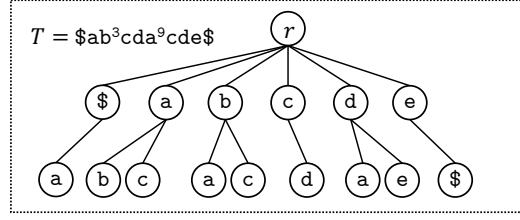
5.2 Representation for \mathcal{M}_2

Recall that $|\mathcal{M}_2| \in O(\sigma'^2) \subseteq O(m^2)$ and this bound is tight in the worst case. Therefore we cannot store all elements of \mathcal{M}_2 explicitly, as our goal is an $O(m)$ -space representation of MAWs. Nevertheless, the following lemma holds:

► **Lemma 17.** *There exists a data structure D_2 of $O(m)$ space that can output each MAW in \mathcal{M}_2 in $O(1)$ amortized time.*

Proof. If $|\mathcal{M}_2| \in O(m)$, then we explicitly store all elements of \mathcal{M}_2 .

If $|\mathcal{M}_2| \in \Omega(m)$, then let D_2 be the trie that represents all bigrams that occur in T . See Figure 3 for a concrete example of D_2 . Note that for any pair $a, b \in \Sigma'$ of *distinct* characters both occurring in T , ab is either in D_2 or in \mathcal{M}_2 . Since the number of such pairs a, b is $\sigma'(\sigma' - 1)$, we have that $\sigma'^2 = \Theta(|D_2| + |\mathcal{M}_2|)$, where $|D_2|$ denotes the size of the trie D_2 . Since $|D_2| < m$, we have $\sigma'^2 = O(|\mathcal{M}_2| + m)$. Suppose that the character labels of the out-going edges of each node in D_2 are lexicographically sorted. When we output each element in \mathcal{M}_2 , we test every bigram ab such that $a \neq b$ and $a, b \in \Sigma'$ in the lexicographical order, and traverse D_2 in parallel in a depth-first manner. We output ab if it is not in the trie D_2 . This takes $O(\sigma'^2 + |D_2|) \subseteq O(|\mathcal{M}_2| + m) = O(|\mathcal{M}_2|)$ time, since $|\mathcal{M}_2| \in \Omega(m)$. ◀



■ **Figure 3** The trie D_2 for string $T = \$ab^3cda^9cde\$$. A bigram ab with $a \neq b$, $a, b \in \Sigma'$ is in \mathcal{M}_2 iff ab is not in this trie D_2 . For instance, ae and db are MAWs of T .

5.3 Representation for \mathcal{M}_3

Recall that the number of MAWs of type 3 in \mathcal{M}_3 is unbounded by the RLE size m (Lemma 4). Nevertheless, we show that there exists a compact $O(m)$ -space data structure that can report each MAW in \mathcal{M}_3 in $O(1)$ time.

Notice that, by definition, a MAW acb of type 3 is a bridge and therefore, it is of the form $ac^k b$ with $c \in \Sigma'_T \setminus \{a, b\}$ and $k \geq 1$.

We begin with some observations. For a triple (a, c, b) of characters with $a \neq c$ and $b \neq c$, let us consider the ordered set $\mathcal{BS}_{acb}(T)$ of bridge substrings of T which are of the form $ac^\ell b$ ($\ell \geq 1$), where the elements in $\mathcal{BS}_{acb}(T)$ are sorted in increasing order of ℓ . Let $\ell_{\max} = \max\{\ell \mid ac^\ell b \in \mathcal{BS}_{acb}(T)\}$. Then, for any $1 \leq k < \ell_{\max}$, $ac^k b \in \mathcal{M}_3$ iff $ac^k b \notin \mathcal{BS}_{acb}(T)$. For instance, consider string $T = ac^3bac^9bac^5bc^4e$ for which $\mathcal{BS}_{acb}(T) = \{ac^3b, ac^5b, ac^9b\}$. Then, $\{ac^1b, ac^2b, ac^4b, ac^6b, ac^7b, ac^8b\}$ is the subset of type-3 MAWs of T of the form $ac^k b$. We remark that the above strategy that is based on bridge substrings of the string is not enough to enumerate all elements of \mathcal{M}_3 , since e.g. ac^3e and bc^2b are also type-3 MAWs in this running example. This leads us to define the notion of *combined bridges*: A bridge $ac^\ell b$ is a combined bridge of T if (1) $ac^\ell b$ is not a bridge substring of T , (2) $ac^i b'$ and $a'c^j b$ are bridge substrings of T with $b' \neq b$ and $a' \neq a$, and (3) $\ell = \min\{i, j\}$. Let $\mathcal{CB}_c(T)$ denote the set of combined bridges of T with middle character c .

► **Observation 18.** A bridge $ac^k b$ is in \mathcal{M}_3 iff $ac^k b \notin \mathcal{BS}_{acb}(T)$ and either (i) $ac^{k'} b \in \mathcal{BS}_{acb}(T)$ with $k' > k$ or (ii) $ac^{k'} b \in \mathcal{CB}_c(T)$ with $k' \geq k$.

The type-3 MAWs ac^3e and bc^2b in the running example belong to Case (ii), since ac^3e is in $\mathcal{CB}_c(T)$ and bc^3b is in $\mathcal{CB}_c(T)$, respectively.

Observation 18 leads us to the following idea: For each character $c \in \Sigma'_T$, let $\mathcal{BS}_c(T) = \bigcup_{a,b \in \Sigma'} \mathcal{BS}_{acb}(T)$ be the ordered set of bridge substrings z of T with $R(z) = 3$ whose middle characters are all c . We suppose that the elements of $\mathcal{BS}_c(T)$ are sorted in increasing order of the exponents ℓ of the middle character c . See Figure 4 for a concrete example for $\mathcal{BS}_c(T)$.

Given $\mathcal{BS}_c(T)$, we can enumerate all type-3 MAWs in \mathcal{M}_2 by incrementally constructing a trie T_c of bigrams. Initially, T_c is a trie only with the root. The algorithm has two stages: **First Stage:** The first stage deals with Case (i) of Observation 18. We perform a linear scan over $\mathcal{BS}_c(T)$. When we encounter a bridge substring $ac^\ell b$ from $\mathcal{BS}_c(T)$, we traverse the trie T_c with the corresponding bigram ab .

1. If ab is not in the current trie, then $ac^k b$ for all $1 \leq k < \ell$ are MAWs in \mathcal{M}_3 . After reporting all these MAWs, we create a node v representing ab and store ℓ .
2. If ab is already in the current trie, then the value $\hat{\ell}$ stored in the node v which represents ab is less than ℓ . Then, $ac^k b$ for all $\hat{\ell} < k < \ell$ are MAWs in \mathcal{M}_3 . After reporting all these MAWs, we update the value in v with ℓ .

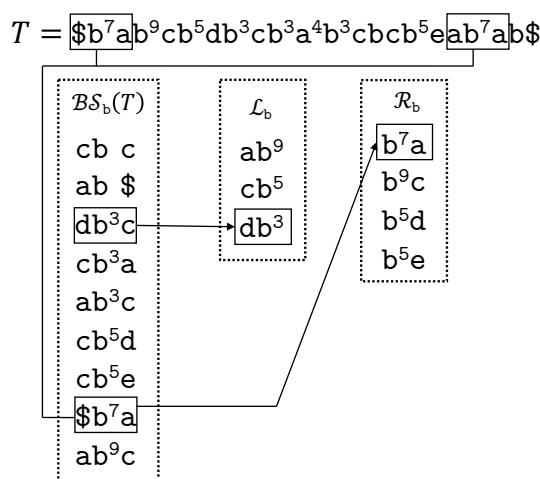


Figure 4 \mathcal{BS}_b , \mathcal{L}_b , and \mathcal{R}_b for string $T = ab^7ab^9cb^5db^3cb^3a^4b^3cbcb^5eab^7abc$ and character b .

The final trie \mathcal{T}_c after the first stage will be unchanged in the following second stage.

Second Stage: The second stage deals with Case (ii) of Observation 18. For each character $a \in \Sigma'_T \setminus \{c\}$, we store the left component ac^i of a bridge substring such that i is the largest exponent of the bridge substrings beginning with ac . Let \mathcal{L}_c be the set of ac^i 's for all characters $a \in \Sigma'_T \setminus \{c\}$. Similarly, let \mathcal{R}_c be the set of the right components $c^j b$ for all characters $b \in \Sigma'_T \setminus \{c\}$, where j is the largest exponent of the bridge substrings ending with cb . See Figure 4 for a concrete example for \mathcal{L}_c and \mathcal{R}_c .

For each pair of $ac^i \in \mathcal{L}_c$ and $c^j b \in \mathcal{R}_c$, let $ac^\ell b$ be the combined bridge with $\ell = \min\{i, j\}$.

1. If ab is not in the trie \mathcal{T}_c , then $ac^k b$ for all $1 \leq k \leq \ell$ are MAWs in \mathcal{M}_3 .
2. If ab is in the trie \mathcal{T}_c , then let $\hat{\ell}$ be the value stored in the node that represents ab .
 - a. If $\hat{\ell} < \ell$, then $ac^k b$ for all $\hat{\ell} < k \leq \ell$ are MAWs in \mathcal{M}_3 .
 - b. If $\hat{\ell} \geq \ell$, then we do nothing.

We have the following lemma:

► **Lemma 19.** *There exists a data structure D_3 of $O(m)$ space that can output each MAW in \mathcal{M}_3 in amortized $O(1)$ time.*

Proof. Analogously to the case of \mathcal{M}_2 , if $|\mathcal{M}_2| \in O(m)$, then we can explicitly store all type-3 MAWs in $O(m)$ space.

In what follows, we consider the case where $|\mathcal{M}_2| \in \Omega(m)$. For each character $c \in \Sigma'_T$, we perform the above algorithm on $\mathcal{BS}_c(T)$. The correctness of the algorithm follows from Observation 18. Since $\sum_{c \in \Sigma'_T} |\mathcal{BS}_c(T)| \in O(m)$, the total space requirement of the data structure for all characters in Σ'_T is $O(m)$. Let us consider the time complexity. The first stage takes $O(m + f) \subseteq O(|\mathcal{M}_3|)$ time, where f is the number of MAWs reported in the first stage for all characters in Σ'_T . The second stage takes $O(|\mathcal{L}_c| \cdot |\mathcal{R}_c|)$ time for each $c \in \Sigma'_T$. For each combined bridge $ac^\ell b$ created from \mathcal{L}_c and \mathcal{R}_c , when it falls into Case 1 or Case 2-a, then at least one MAW is reported. When it falls into Case 2-b, then no MAW is reported. However, in Case 2-b, there has to be a MAW $ac^k b$ that was reported in the first stage. Since we test at most one combined bridge for each pair of characters a, b , a MAW $ac^k b$ reported in the first stage is charged at most once. Therefore, the second stage takes a total of $O(\sum_{c \in \Sigma'_T} |\mathcal{L}_c| \cdot |\mathcal{R}_c|) \subseteq O(|\mathcal{M}_3|)$ time. ◀

5.4 Representation for \mathcal{M}_4

Recall that $|\mathcal{M}_4| \in O(m^2)$ and this bound is tight in the worst case. Therefore we cannot store all elements of \mathcal{M}_4 explicitly, as our goal is an $O(m)$ -space representation of MAWs. Nevertheless, the following lemma holds:

► **Lemma 20.** *There exists a data structure D_4 of $O(m)$ space that can output each MAW in \mathcal{M}_4 in $O(1)$ amortized time.*

Our data structure D_4 is based on the discussion in Section 3.2. We consider the following bipartite graph $G_w = (V_L \cup V_R, E)$ for any bridge $w \in \mathcal{W}$. We can identify each bridge $a\alpha^i u \beta^j b \in \text{Exp}(w)$ by representing the bridge as a 4-tuple (a, i, j, b) . Let F_w be the set of 4-tuples which represents all elements in $\text{Exp}(w)$. Two disjoint sets V_L, V_R of vertices and set E of edges are defined as follows:

$$\begin{aligned} V_L &= \{(a, i) \mid \exists(a, i, j, b) \in F_w\}, \\ V_R &= \{(j, b) \mid \exists(a, i, j, b) \in F_w\}, \\ E &= \{((a, i), (j, b)) \mid \exists(a, i, j, b) \in F_w\}. \end{aligned}$$

V_L (resp. V_R) represents the set of the left (resp. right) parts of bridges in \mathcal{W} . For each edge in E represents a bridge in \mathcal{W} . This implies that $|E| = |\text{Exp}(w)|$. Assume that all vertices in V_L (resp. V_R) are sorted in non-decreasing order w.r.t. the value i (resp. j) which represents the exponent of corresponding run. For any $k \in [1, |V_L|]$ and $k' \in [1, |V_R|]$, $v_L(k) = (c_L(k), e_L(k))$ denotes the k -th vertex in V_L , and $v_R(k') = (c_R(k'), e_R(k'))$ denotes the k' -th vertex in V_R . For any vertex $v_L(k) \in V_L$ and $v_R(k') \in V_R$, we also define

$$\begin{aligned} E_{max}^{LR}(k) &= \max\{e_R(i) \mid \exists(v_L(k), v_R(i)) \in E\}, \\ E_{max}^{RL}(k') &= \max\{e_L(i) \mid \exists(v_R(i), v_R(k')) \in E\}. \end{aligned}$$

Figure 5 gives an illustration for this graph. Due to Observation 7, each MAW z of type 4 corresponds to an element of $\text{Exp}(w) \times \text{Exp}(w)$ where $z^{(1)} = w$. By this idea, we detect each MAW as a pair of vertices in $V_L \times V_R$ which is not an edge in E . The following lemma explains all MAWs which can be represented by the graph.

► **Lemma 21.** *For any vertices $v_L(k) \in V_L$ and $v_R(k') \in V_R$ of $G_{\alpha u \beta}$, the string $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ is a MAW iff the following three conditions hold (see also Figure 6 for an illustration):*

- $(v_L(k), v_R(k')) \notin E$,
- $E_{max}^{LR}(k) \geq e_R(k')$, and
- $E_{max}^{RL}(k') \geq e_L(k)$.

Proof. If $(v_L(k), v_R(k')) \notin E$, $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ is an absent word. $E_{max}^{LR}(k) \geq e_R(k')$ and $E_{max}^{RL}(k') \geq e_L(k)$ implies that $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}$ and $\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ occur in the string. Thus $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ is a MAW.

On the other hand, if $(v_L(k), v_R(k')) \in E$, $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ occurs in the text. $E_{max}^{LR}(k) < e_R(k')$ implies that $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}$ does not occur in the string. $E_{max}^{RL}(k') < e_L(k)$ implies that $\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ does not occur in the string. Thus all three conditions hold if $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$ is a MAW. ◀

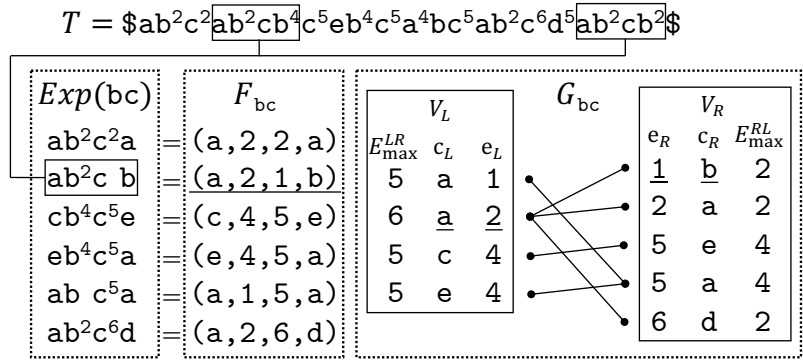


Figure 5 This figure shows G_{bc} for $T = \$ab^2c^2ab^2cb^4c^5eb^4c^5a^4bc^5ab^2c^6d^5ab^2cb^2\$$. For a bridge bc , $Exp(bc)$ has 6 bridges. F_{bc} contains 6 tuples which represents all bridges in $Exp(bc)$. For instance, a bridge $ab^2cb = (a, 2, 1, b)$ where the first character is a , the exponent of the second run is 2, the exponent of the second last run is 1, and the last character is b . V_L is the set of pairs by the left-half of elements in F_{bc} . In this example, V_L has 4 vertices $\{(a, 1), (a, 2), (c, 4), (e, 4)\}$ which are sorted in non-decreasing order of the second key (representing its exponent). V_R is the symmetric set for the right parts. Each bridge corresponds to an edge. For example, the second bridge ab^2cb in the figure corresponds to the edge from the second vertex $(a, 2)$ in V_L to the first vertex $(1, b)$ in V_R . Since the number of bridges in $Exp(bc)(F_{bc})$ is 6, the graph has 6 edges.

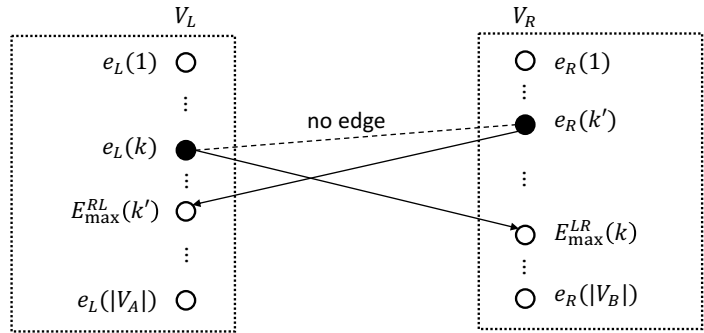


Figure 6 This is an illustration for Lemma 21. For the k -th vertex $v_L(k) \in V_L$ and k' -th vertex $v_R(k') \in V_R$, this graph satisfies the three conditions of the lemma.

Proof of Lemma 20. Let x be the number of outputs. If $x < m$, we can just store all the MAWs themselves. Assume that $x \in \Omega(m)$.

For all bridge $w = \alpha u \beta \in \mathcal{W}$, G_w represents all MAWs which correspond to elements in $Exp(w) \times Exp(w)$. Our data structure D_4 consists of G_w for any $w \in \mathcal{W}$. It is clear that G_w can be stored in $O(|Exp(w)|)$ space. This implies that the size of D_4 is linear in \mathcal{X} , namely, D_4 can be stored in $O(m)$ space (Lemma 10).

We can output all MAWs which are represented by G_w based on Lemma 21 (see Algorithm 1). For the k -th vertex $v_L(k)$, C represents all vertices $v_R(k')$ in V_B such that $(v_L(k), v_R(k')) \notin E$ and $E_{max}^{RL}(k') \geq e_L(k)$ (the first and third condition in Lemma 21). For each vertex in C , if $E_{max}^{LR}(k) \geq e_R(k')$ (the second condition in Lemma 21), the algorithm outputs a MAW $c_L(k)\alpha^{e_L(k)}u\beta^{e_R(k')}c_R(k')$. Then the running time of our algorithm is $O(x + \sum_{w \in \mathcal{W}} |G_w|) \subseteq O(x + m) = O(x)$, since $x \in \Omega(m)$. ◀

■ **Algorithm 1** Compute all MAWs in \mathcal{M}_4 .

Input: bipartite graph $G_{\alpha\beta} = (V_L, V_R, E)$

Output: all MAWs in \mathcal{M}_4 that are associated by $\alpha u \beta$, $a \alpha^{k_1} u \beta^{k_2} b$ for $a, b \in \Sigma$, $k_1, k_2 \in \mathbb{N}$

```

1:  $C_R \leftarrow V_R$ 
2: for each  $v_L(k) \in V_L$  do
3:    $C = \{v_R(k') \in C_R \mid e_R(k') \leq E_{max}^{LR}(k)\} \setminus \{v \mid (v_L(k), v) \in E\}$ 
4:   for each  $v_R(k') \in C$  do
5:     if  $E_{max}^{RL}(k') \geq e_L(v_L(k))$  then
6:       output  $c_L(k) \alpha^{e_L(k)} u \beta^{e_R(k')} c_R(k')$ 
7:     else
8:        $C_R \leftarrow C_R \setminus \{v_R(k')\}$ 
9:     end if
10:  end for
11: end for

```

5.5 Representation for \mathcal{M}_5

► **Lemma 22.** *There exists a data structure of size $O(m)$ that outputs each element of \mathcal{M}_5 in $O(1)$ time.*

Proof. By Lemma 5, $|\mathcal{M}_5| \in O(m)$. Recall that an element of \mathcal{M}_5 can be as long as $O(n)$. However, using Lemma 15 we can represent and store all elements in \mathcal{M}_5 in a total of $O(m)$ space. It is trivial that each stored element can be output in $O(1)$ time. ◀

6 Conclusions and open questions

Minimal absent words (MAWs) are combinatorial string objects that can be used in applications such as data compression (anti-dictionaries) and bioinformatics. In this paper, we considered MAWs for a string T that is described by its run-length encoding (RLE) $\text{rle}(T)$ of size m . We first analyzed the number of MAWs for a string T in terms of its RLE size m , by dividing the set $\text{MAW}(T)$ of all MAWs for T into five disjoint types. Albeit the number of MAWs of some types is superlinear in m , we devised a compact $O(m)$ -space representation for $\text{MAW}(T)$ that can output all MAWs in output-sensitive $O(|\text{MAW}(T)|)$ time.

We would like to remark that our $O(m)$ -space representation can be built in $O(m \log m)$ time with $O(m)$ space, with the help of the *truncated RLE suffix array (trLESA)* data structure [25]. A suffix s of T is called a trLE suffix of T if $s = ar_i \cdots r_m$ where the first a is the last character in the previous run r_{i-1} . trLESA(T) for $\text{rle}(T) = r_1 \cdots r_m$ is an integer array of length m such that $\text{trLESA}(T)[i] = k$ iff $ar_i \cdots r_m$ is the k -th lexicographically smallest trLE suffix for T . trLESA occupies $O(m)$ space, and can be built in $O(m \log m)$ time with $O(m)$ working space [25]. The details for our trLESA-based construction algorithm for our $O(m)$ -space MAW representation will appear in the full version of this paper.

An interesting open question is whether there exist other compressed representations of MAWs, based on e.g. grammar-based compression [20], Lempel-Ziv 77 [26], and run-length Burrows-Wheeler transform [22].

References

- 1 Tooru Akagi, Yuki Kuhara, Takuya Mieno, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Combinatorics of minimal absent words for a sliding window. *CoRR*, abs/2105.08496, 2021. [arXiv:2105.08496](#).
- 2 Yannis Almirantis, Panagiotis Charalampopoulos, Jia Gao, Costas S. Iliopoulos, Manal Mohamed, Solon P. Pissis, and Dimitris Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5, 2017.
- 3 Lorraine A. K. Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, and Solon P. Pissis. Constructing antidictionaries of long texts in output-sensitive space. *Theory Comput. Syst.*, 65(5):777–797, 2021.
- 4 Carl Barton, Alice Heliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15(1):388, 2014.
- 5 Marie Pierre Béal, Filippo Mignosi, and Antonio Restivo. Minimal forbidden words and symbolic dynamics. In *STACS 1996*, pages 555–566, 1996.
- 6 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In *ESA 2013*, pages 133–144, 2013.
- 7 Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985.
- 8 Supaporn Chairungsee and Maxime Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450:109–116, 2012.
- 9 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262:57–68, 2018.
- 10 Panagiotis Charalampopoulos, Maxime Crochemore, and Solon P Pissis. On extended special factors of a word. In *SPIRE 2018*, pages 131–138. Springer, 2018.
- 11 Tim Crawford, Golnaz Badkobeh, and David Lewis. Searching page-images of early music scanned with OMR: A scalable solution using minimal absent words. In *ISMIR 2018*, pages 233–239, 2018.
- 12 M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictionaries. *Proc. IEEE*, 88(11):1756–1768, 2000.
- 13 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Information and Computation*, 270:104461, 2020.
- 14 Maxime Crochemore, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.
- 15 Maxime Crochemore and Gonzalo Navarro. Improved antidictionary based compression. In *12th International Conference of the Chilean Computer Science Society, 2002. Proceedings.*, pages 7–13. IEEE, 2002.
- 16 Gabriele Fici. *Minimal forbidden words and applications*. PhD thesis, Università di Palermo and Université Paris-Est Marne-la-Vallée, 2006.
- 17 Gabriele Fici and Pawel Gawrychowski. Minimal absent words in rooted and unrooted trees. In *SPIRE 2019*, pages 152–161, 2019.
- 18 Gabriele Fici, Antonio Restivo, and Laura Rizzo. Minimal forbidden factors of circular words. *Theor. Comput. Sci.*, 792:144–153, 2019.
- 19 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In *MFCS 2016*, volume 58, pages 38:1–38:14, 2016.

27:16 Minimal Absent Words on Run-Length Encoded Strings

- 20 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. doi:10.1109/18.841160.
- 21 Grigorios Koulouras and Martin C Frith. Significant non-existence of sequences in genomes and proteomes. *Nucleic acids research*, 49(6):3139–3155, 2021.
- 22 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
- 23 Takuya Mieno, Yuki Kuhara, Tooru Akagi, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *SOFSEM 2020*, volume 12011 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2020.
- 24 Diogo Pratas and Jorge M Silva. Persistent minimal sequences of SARS-CoV-2. *Bioinformatics*, 36(21):5129–5132, 2020.
- 25 Yuya Tamakoshi, Keisuke Goto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. An opportunistic text indexing structure based on run length encoding. In *CIAC 2015*, volume 9079 of *Lecture Notes in Computer Science*, pages 390–402. Springer, 2015.
- 26 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23(3):337–349, 1977.

A Appendix

We give a supplemental proposition that can be useful for analyzing the upper bound on the number of MAWs of type 4.

We begin with the following observation:

► **Observation 23.** For any bridge substring $w \in \Sigma^*$ of T ,

$$|\text{Exp}(w)| = \#w - \sum_{z \in \text{Exp}(w)} (\#z - 1) \leq \#w + |\text{Exp}_+(w)| - \sum_{z \in \text{Exp}_+(w)} \#z.$$

Note that $\sum_{z \in \text{Exp}_+(w)} (\#z - 1) \leq \sum_{z \in \text{Exp}(w)} (\#z - 1)$ since $\#z - 1 = 0$ when $z \in \text{Exp}(w) \setminus \text{Exp}_+(w)$. Below we present Proposition 24 which gives an upper bound for \mathcal{X} .

► **Proposition 24.** For any bridge w and $t \geq 1$ such that $|\text{Exp}(w)| \geq 2$,

$$|\text{Exp}(w)| + \sum_{i=1}^t \sum_{z \in \text{Exp}_+^i(w)} |\text{Exp}(z)| \leq \#w + \sum_{i=1}^t |\text{Exp}_+^i(w)|. \quad (1)$$

Proof. We prove this lemma by induction on t . By Observation 23 and $|\text{Exp}(w)| \leq \#w$ for any w , we have

$$|\text{Exp}(w)| + \sum_{z \in \text{Exp}_+(w)} |\text{Exp}(z)| \leq (\#w + |\text{Exp}_+(w)|) - \sum_{z \in \text{Exp}_+(w)} \#z + \sum_{z \in \text{Exp}_+(w)} \#z = \#w + |\text{Exp}_+(w)|.$$

Thus, the statement holds for $t = 1$. Suppose that the statement holds for some $t' \geq 1$.

$$\begin{aligned}
& |\text{Exp}(w)| + \sum_{i=1}^{t'+1} \sum_{z \in \text{Exp}_+^i(w)} |\text{Exp}(z)| \\
= & |\text{Exp}(w)| + \sum_{w' \in \text{Exp}_+(w)} \left(|\text{Exp}(w')| + \sum_{i=1}^{t'} \sum_{z \in \text{Exp}_+^i(w')} |\text{Exp}(z)| \right) \\
\leq & |\text{Exp}(w)| + \sum_{w' \in \text{Exp}_+(w)} \left(\#w' + \sum_{i=1}^{t'} |\text{Exp}_+^i(w')| \right) \quad (\text{by induction hypothesis}) \\
\leq & \left(\#w + |\text{Exp}_+(w)| - \sum_{w' \in \text{Exp}_+(w)} \#w' \right) + \sum_{w' \in \text{Exp}_+(w)} \#w' + \sum_{w' \in \text{Exp}_+(w)} \sum_{i=1}^{t'} |\text{Exp}_+^i(w')| \\
& \quad (\text{by Observation 23}) \\
= & \#w + |\text{Exp}_+(w)| + \sum_{w' \in \text{Exp}_+(w)} \sum_{i=1}^{t'} |\text{Exp}_+^i(w')| \\
\leq & \#w + |\text{Exp}_+(w)| + \sum_{i=2}^{t'+1} |\text{Exp}_+^i(w)| \\
= & \#w + \sum_{i=1}^{t'+1} |\text{Exp}_+^i(w)|
\end{aligned}$$

Thus, the statement holds for $t' + 1$. Therefore, the statement holds for any $t \geq 1$. \blacktriangleleft


Parallel Algorithm for Pattern Matching Problems Under Substring Consistent Equivalence Relations

Davaajav Jargalsaikhan ✉


Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Diptarama Hendrian ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Ryo Yoshinaka ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Ayumi Shinohara ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Abstract

Given a text and a pattern over an alphabet, the pattern matching problem searches for all occurrences of the pattern in the text. An equivalence relation \approx is a substring consistent equivalence relation (SCER), if for two strings X and Y , $X \approx Y$ implies $|X| = |Y|$ and $X[i : j] \approx Y[i : j]$ for all $1 \leq i \leq j \leq |X|$. In this paper, we propose an efficient parallel algorithm for pattern matching under any SCER using the “duel-and-sweep” paradigm. For a pattern of length m and a text of length n , our algorithm runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work, with $O(\tau_n^t + \xi_m^t \log^2 m)$ time and $O(\tau_n^w + \xi_m^w \cdot m \log^2 m)$ work preprocessing on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM), where τ_n^t , τ_n^w , ξ_m^t , and ξ_m^w are parameters dependent on SCERs, which are often linear in n and m , respectively.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases parallel algorithm, substring consistent equivalence relation, pattern matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.28

Related Version *Full Version*: <https://arxiv.org/abs/2202.13284>

Funding *Diptarama Hendrian*: JSPS KAKENHI Grant Number JP19K20208

Ryo Yoshinaka: JSPS KAKENHI Grant Numbers JP18K11150 and JP20H05703

Ayumi Shinohara: JSPS KAKENHI Grant Number JP21K11745

1 Introduction

The string matching problem is fundamental and widely studied in computer science. Given a text and a pattern, the string matching problem searches for all substrings of the text that match the pattern. Many matching functions that are used in different string matching problems, including exact [15], parameterized [4], order-preserving [14, 16] and cartesian-tree [18] matchings, fall under the class of *substring consistent equivalence relations* (SCERs) [17]. An equivalence relation on strings is an SCER, if two strings X and Y match under the equivalence relation, then they have equal length and $X[i : j]$ matches $Y[i : j]$, for all $1 \leq i \leq j < |X|$. Matsuoka et al. [17] generalized the KMP algorithm [15] for pattern matching problems under SCERs. They also investigated periodicity properties of strings under SCERs. Kikuchi et al. [13] proposed algorithms to compute the shortest and longest cover arrays for a given string under any SCER. Hendrian [9] generalized Aho-Corasick algorithm for the dictionary matching under SCERs.



© Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara; licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 28; pp. 28:1–28:21

Leibniz International Proceedings in Informatics



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

■ **Table 1** Summary of the encoding complexities for some SCER on P-CRCW PRAM.

	τ_n^t	τ_n^w	ξ_m^t	ξ_m^w
Exact	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Parameterized	$O(\log n)$	$O(n \log n)$	$O(1)$	$O(1)$
Cartesian-tree	$O(\log n)$	$O(n \log n)$	$O(\log m)$	$O(m \log m)$

Vishkin proposed two algorithms for exact pattern matching, pattern matching by dueling [19] and pattern matching by sampling [20]. Both algorithms match the pattern to a substring of the text from some positions which are determined by the property of the pattern, instead of its prefix or suffix as in, for instance, the KMP algorithm [15]. These algorithms are developed for parallel processing.

The dueling technique by Vishkin [19] has been proved to be useful for various kinds of pattern matching. Amir et al. [2] proposed a duel-and-sweep algorithm for two-dimensional exact matching, which is named “consistency and verification”. Cole et al. [8] extended it to two-dimensional parameterized matching. In addition, Jargalsaikhan et al. [11, 12] proposed serial and parallel duel-and-sweep algorithms for order-preserving matching.

In this paper, we propose an efficient parallel algorithm based on the dueling technique for the pattern matching problem under SCERs. Our parallel algorithm is the first to solve the problem under an arbitrary SCER in parallel. While Vishkin’s dueling algorithm for exact matching depends on the preferable properties of periods of strings, many of those do not hold with SCERs. Therefore, our algorithm involves new ideas and appears quite different from the original for exact pattern matching. For a pattern of length m and a text of length n , our algorithm runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work, with $O(\tau_n^t + \xi_m^t \log^2 m)$ time and $O(\tau_n^w + \xi_m^w \cdot m \log^2 m)$ work preprocessing on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM) [10]. Here, τ_n^t and τ_n^w are time and work respectively, needed on P-CRCW PRAM to encode in parallel a string X of length n under the SCER in concern. Given the encoding of X , ξ_m^t and ξ_m^w are time and work respectively to re-encode an element w.r.t. some suffix of X . Table 1 shows the encoding time and work complexities for some SCERs.

Due to the space restrictions, proofs of lemmas are relegated to Appendix B.

2 Preliminaries

We use Σ to denote an alphabet of symbols and Σ^* denotes the set of strings over the alphabet Σ . For a string $X \in \Sigma^*$, the length of X is denoted by $|X|$. The *empty string*, denoted by ε , is the string of length 0. For a string $X \in \Sigma^*$ of length n , $X[i]$ denotes the i -th symbol of X , $X[i : j] = X[i]X[i+1] \dots X[j]$ denotes a substring of X that begins at position i and ends at position j for $1 \leq i \leq j \leq n$. For $i > j$, $X[i : j]$ denotes the empty string.

► **Definition 1** (Substring consistent equivalence relation (SCER) [17]). *An equivalence relation $\approx \subseteq \Sigma^* \times \Sigma^*$ is a substring consistent equivalence relation (SCER) if for two strings X and Y , $X \approx Y$ implies $|X| = |Y|$ and $X[i : j] \approx Y[i : j]$ for all $1 \leq i \leq j \leq |X|$.*

For instance, while the parameterized matching [4] and order-preserving matching [16, 14] are SCERs, the permutation matching [6, 7] and function matching [1] are not.

Hereafter we fix an arbitrary SCER \approx . We say that a position i is the *tight mismatch position* if $X[1 : i-1] \approx Y[1 : i-1]$ and $X[1 : i] \not\approx Y[1 : i]$. For two strings X and Y , let $LCP(X, Y)$ be the length l of the longest prefixes of X and Y match. That is, l is the

greatest integer such that $X[1 : l] \approx Y[1 : l]$. Obviously, if i is the tight mismatch position for $X \not\approx Y$, then $LCP(X, Y) = i - 1$. The converse holds if $i \leq \min\{|X|, |Y|\}$. Similarly, for a string X and an integer $0 \leq a < |X|$, we define $LCP_X(a) = LCP(X, X[a + 1 : |X|])$. In other words, $LCP_X(a)$ is the length of the longest common prefix, when X is superimposed on itself with offset a . We say $X \approx\text{-matches } Y$ iff $X \approx Y$. Given a text T of length n and a pattern P of length m , a position i in T , $1 \leq i \leq n - m + 1$, is an $\approx\text{-occurrence}$ of P in T iff $P \approx T[i : i + m - 1]$.

► **Definition 2** ($\approx\text{-pattern matching}$).

Input: A text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length $m \leq n$.

Output: All $\approx\text{-occurrences}$ of P inside T .

In the remainder of this paper, we fix text T to be of length n and pattern P to be of length m . We also assume that $n = 2m - 1$. Larger texts can be cut into overlapping pieces of length that are less than or equal to $(2m - 1)$ and processed independently. That is, we search for pattern occurrences in each substring $T[1 : 2m - 1], T[m + 1 : 3m - 1], \dots, T[\lfloor \frac{n-1}{m} \rfloor \cdot m + 1 : n]$, independently. For an integer x with $1 \leq x \leq n - m + 1$, a *candidate* T_x is the substring of T starting from x of length m , i.e., $T_x = T[x : x + m - 1]$.

For SCER matchings often it is convenient to encode the strings where $\approx\text{-equivalence}$ is reduced to the identity. Amir and Kondratovsky [3] showed that every SCER admits an encoding satisfying the following property.¹

► **Definition 3** ($\approx\text{-encoding}$). Let Σ and Δ be alphabets. We say a function $f : \Sigma^* \rightarrow \Delta^*$ is an $\approx\text{-encoding}$ if (1) for any string $X \in \Sigma^*$, $|X| = |f(X)|$, (2) $f(X[1 : i]) = f(X)[1 : i]$, (3) for two strings X and Y of equal length k , $f(X)[i] = f(Y)[i]$ implies $f(X[j + 1 : k])[i - j] = f(Y[j + 1 : k])[i - j]$ for any $j < i \leq k$, and (4) $f(X) = f(Y)$ iff $X \approx Y$.

► **Proposition 4.** An equivalence relation \approx is an SCER if and only if it admits an $\approx\text{-encoding}$.

Standard encodings of SCERs often satisfy the above definition, such as the prev-encoding [4] for parameterized matching and parent-distance encoding [18] for cartesian-tree matching. However, the nearest neighbor encoding [14] for order-preserving matching violates the third condition. Our algorithm for $\approx\text{-pattern matching}$ proposed in this paper relies on the property of Definition 3 and does not work with the nearest neighbor encoding. Nonetheless, duel-and-sweep algorithms for order-preserving matching based on the encoding are possible by further elaboration [11, 12], but we will not discuss it in this paper.

Fixing an $\approx\text{-encoding } f$, we denote $f(X)$ by \tilde{X} for simplicity. In addition, we denote the encoding of $X[x : |X|]$ as $\tilde{X}_x = f(X[x : |X|])$. Thus $\tilde{X}_1 = \tilde{X}$. For a string X , we suppose that \tilde{X} can be computed in $\tau_{|X|}^t$ time and $\tau_{|X|}^w$ work in parallel on P-CRCW PRAM. Moreover, we assume that given \tilde{X} , x , and i such that $x + i - 1 \leq |X|$, to compute $\tilde{X}_x[i]$, i.e. re-encoding the element at position i with respect to suffix $X[x : |X|]$, takes $\xi_{|X|}^t$ time and $\xi_{|X|}^w$ work on P-CRCW PRAM. Note that, to compute $\tilde{X}_x[i]$ does not necessarily require computing the whole of \tilde{X}_x . Those parameters are often reasonably small. See Table 1 and Appendix A for the prev-encoding for parameterized matching and the parent-distance encoding for cartesian-tree matching.

Vishkin's dueling technique essentially depends on the preferable properties of periods of strings. Matsuoka et al. [17] have discussed in detail how the classical notion of periods and their properties can be generalized when considering SCER matching. Unfortunately,

¹ Lemma 12 in [3] does not explicitly mention the third property, but their proof entails it.

none of the generalizations yield a straightforward adaptation of Vishkin’s algorithm for SCER matching. Among those, the kind of periods involved in the duel-and-sweep algorithm discussed in this paper is *border-based period*.

► **Definition 5** (Border-based period). *Given a string X of length n , positive integer $p < n$ is called a border-based period of X if $X[1 : n - p] \approx X[p + 1 : n]$.*

Throughout the rest of the paper, we will refer to a border-based period as a *period*.

The family of models of computation used in this work is the priority concurrent-read concurrent-write (P-CRCW) PRAM [10]. This model allows simultaneous reading from the same memory location as well as simultaneous writing. In case of multiple writes to the same memory cell, the P-CRCW PRAM grants access to the memory cell to the processor with the smallest index.

3 Parallel algorithm for pattern matching under SCERs

We give an overview of the duel-and-sweep algorithm [2, 19]. The pattern is first preprocessed to obtain a *witness table*, which is later used to prune candidates during the pattern searching. As the name suggests, in the duel-and-sweep algorithm, the pattern searching is divided into two stages: the *dueling stage* and the *sweeping stage*. The pattern searching algorithm prunes candidates that cannot be pattern occurrences, first by performing “duels” between them, and then by “sweeping” through the remaining candidates to obtain pattern occurrences.

First, we explain the idea of dueling. Suppose P is superimposed on itself with an offset $a < m$ and the two overlapped regions of P do not match under \approx . Then it is impossible for two candidates T_x and T_{x+a} with offset a to match P simultaneously (see Figure 1). The dueling stage lets each pair of candidates with such offset a “duel” and eliminates one based on this observation, so that if candidate T_x gets eliminated during the dueling stage, then $T_x \not\approx P$. However, the opposite does not necessarily hold true: T_x surviving the dueling stage does not mean that $T_x \approx P$. On the other hand, it is guaranteed that if distinct candidates T_x and T_{x+a} that survive the dueling stage overlap, then the suffixes of T_x and P of length $m - a$ match if and only if so do the prefixes of T_{x+a} and P of the same length. The sweeping stage takes advantage of this property when checking whether surviving candidates and the pattern match, so that this stage can also be done quickly.

Prior to the dueling stage, the pattern is preprocessed to construct a *witness table* based on which the dueling stage decides which pair of overlapping candidates should duel and how they should duel. For each offset $0 \leq a < m$, when the overlapped regions obtained by superimposing P on itself with offset a do not match, we need only one position i to say that the overlapping regions do not match. We say that w is a *witness for the offset a* if $\tilde{P}_{a+1}[w] \neq \tilde{P}[w]$. We denote by $\mathcal{W}_P(a)$ the set of all witnesses for offset a . We say a witness w for offset a is *tight* if $w = \min \mathcal{W}_P(a)$. Obviously, $\mathcal{W}_P(a) = \emptyset$ if and only if $a = 0$ or a is a period of P . A *witness table* $W[0 : m - 1]$ is an array such that $W[a] \in \mathcal{W}_P(a)$ if $\mathcal{W}_P(a) \neq \emptyset$. When the overlap regions match for offset a , which implies that no witness exists for a , we express it as $W[a] = 0$.

More formally, in the dueling stage, we “duel” positions x and $x + a$ such that $\mathcal{W}_P(a) \neq \emptyset$ based on the following observation (see Figure 1).

► **Lemma 6.** *Suppose $w \in \mathcal{W}_P(a)$. Then,*

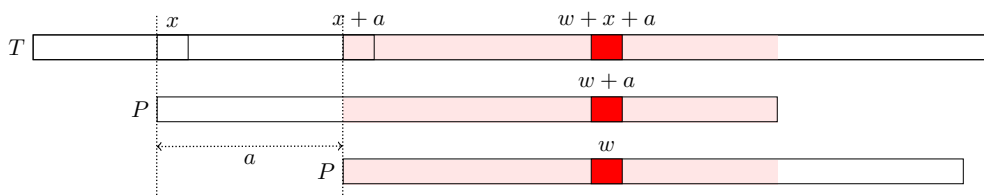
- *if $\tilde{T}_{x+a}[w] = \tilde{P}[w]$, then $T_x \not\approx P$,*
- *if $\tilde{T}_{x+a}[w] \neq \tilde{P}[w]$, then $T_{x+a} \not\approx P$.*

■ **Algorithm 1** Dueling with respect to S . There is one survivor assuming x is not consistent with y .

```

1 Function Dueling( $\tilde{S}, x, y$ )
2    $w \leftarrow W[y - x]$ ;
3   if  $\tilde{S}_y[w] = \tilde{P}[w]$  then return  $y$ ;
4   return  $x$ ;

```



■ **Figure 1** If $T_x \approx P \approx T_{x+a}$, then the overlapped regions of P superimposed on itself with offset a should match, i.e., $P_{a+1}[1 : m - a] \approx P[1 : m - a]$. If the overlapped region does not match, there must be a witness w such that $\tilde{P}_{a+1}[w] \neq \tilde{P}[w]$. Candidate positions x and $x + a$ perform a duel using the witness w based on Lemma 6.

Based on this lemma, we can safely eliminate either candidate T_x or T_{x+a} without looking into other positions. This process is called *dueling* (Algorithm 1). On the other hand, if the offset a has no witness, i.e. $P[1 : m - a] \approx P[a + 1 : m]$, no dueling is performed on them. We say that a position x is consistent with $x + a$ if $\mathcal{W}_P(a) = \emptyset$.

► **Lemma 7.** For any a, b, c such that $0 < a \leq b \leq c < m$, if a is consistent with b and b is consistent with c , then a is consistent with c .

After the dueling stage, all surviving candidate positions are pairwise consistent. The dueling stage algorithm makes sure that no occurrence gets eliminated during the dueling stage. Taking advantage of the fact that surviving candidates from the dueling stage are pairwise consistent, the sweeping stage prunes them until all remaining candidates match the pattern. By ensuring pairwise consistency of the surviving candidates, the pattern searching algorithm reduces the number of comparisons at a position in the text during the sweeping stage.

Hereinafter, in our pseudo-codes we will use “ \leftarrow ” to note assignment operation into a local variable of a processor or assignment operation into a global variable which is accessed by a single processor at a time. We will use “ \Leftarrow ” to note assignment operation into a global variable which is accessible from multiple processors simultaneously. In case of a write conflict, the processor with the smallest index succeeds in writing into the memory.

3.1 Pattern preprocessing

The goal of the preprocessing stage is to compute a witness table $W[0 : m - 1]$, where $W[a] = 0$ if $\mathcal{W}_P(a) = \emptyset$, and $W[a] \in \mathcal{W}_P(a)$ otherwise. Algorithm 2 computes the tight mismatch position for X and Y , given \tilde{X} and \tilde{Y} .

► **Lemma 8.** For strings X and Y of equal length, given \tilde{X} and \tilde{Y} , Algorithm 2 computes the tight mismatch position in $O(1)$ time and $O(|X|)$ work on the P -CRCW PRAM.

■ **Algorithm 2** Check in parallel whether X and Y match, given \tilde{X} and \tilde{Y} . If they do not match, it returns the tight witness.

```

1 Function GetTightMismatchPos( $\tilde{X}, \tilde{Y}$ )
2    $w \leftarrow 0$ ;
3   for each  $i \in \{1, \dots, |X|\}$  do in parallel
4     if  $\tilde{X}[i] \neq \tilde{Y}[i]$  then  $w \leftarrow i$ ;
5   return  $w$ ;
```

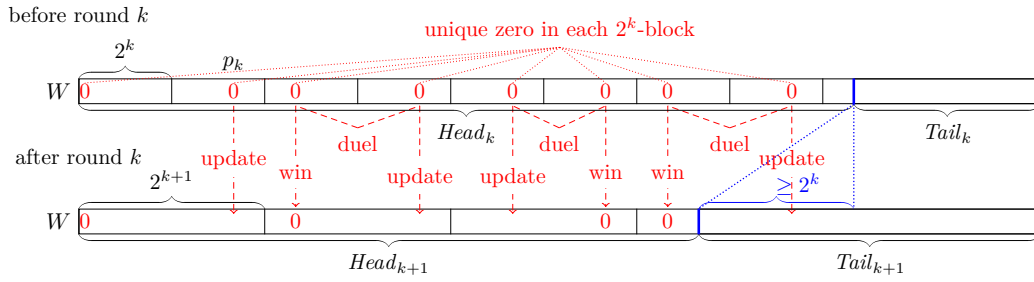
■ **Algorithm 3** Parallel algorithm for the pattern preprocessing.

```

1 Function PreprocessingParallel()
2    $tail \leftarrow m, k \leftarrow 0$ ;          /*  $tail$  is the starting position of  $Tail_k$  */
3   while  $2^k \leq tail$  do
4      $p \leftarrow \text{GetZeros}(2^k, 2^{k+1} - 1, k)[0]$ ;
5      $W[p] \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1 : m - p], \tilde{P}_{p+1}[1 : m - p])$ ;
6     if  $W[p] = 0$  then  $lcp \leftarrow m - p$ ;
7     else  $lcp \leftarrow W[p] - 1$ ;
8      $old\_tail \leftarrow tail$ ;
9      $tail \leftarrow \min(old\_tail - 2^k, m - lcp)$ ;
10    SatisfySparsity( $tail - 1, k$ );
11    FinalizeTail( $tail, old\_tail, p, k$ );
12     $k \leftarrow k + 1$ ;
```

One can compute a witness table naively inputting $\tilde{P}[1 : m - a]$ and $\tilde{P}_{a+1}[1 : m - a]$ for all the offsets $a < m$ to Algorithm 2. However, this naive method costs as much as $\Omega(\xi_m^w \cdot m^2)$ work. We will present a more efficient algorithm in this subsection.

Our pattern preprocessing algorithm is described in Algorithm 3 and its outline is illustrated in Figure 2. Initially, all entries of the witness table are set to zero. Throughout preprocessing, each element of W is updated at most once. Therefore, at any point of the execution of the preprocessing algorithm, if $W[i] \neq 0$, then it must hold $W[i] \in \mathcal{W}_P(i)$. We say that position i is *finalized* if $W[i] = 0$ implies $\mathcal{W}_P(i) = \emptyset$ and $W[i] \neq 0$ implies $W[i] \in \mathcal{W}_P(i)$. During the execution of Algorithm 3, the table is divided into two parts. The *head* is a prefix of a certain length and the *tail* is the rest suffix. Let us write the head and the tail at the round k of the while loop by $Head_k$ and $Tail_k$, respectively. The variable $tail$ in Algorithm 3 represents the starting position of the tail, or equivalently, the length of the head. Throughout the algorithm execution, the tail part is always finalized. On the other hand, though the zero entries of the head are not necessarily reliable, such zero positions become fewer and fewer. Consider partitioning the head into blocks of size 2^k . We will call each block a 2^k -block, with the last 2^k -block possibly being shorter than 2^k . That is, the 2^k -blocks are $W[i \cdot 2^k : (i + 1) \cdot 2^k - 1]$ for $i = 0, \dots, \lfloor h/2^k \rfloor - 1$ and $W[\lfloor h/2^k \rfloor \cdot 2^k : h - 1]$ where $h = |Head_k|$ is the size of the head. We say that $W[0 : x]$ is 2^k -sparse if every 2^k -block of $W[0 : x]$ contains exactly one zero entry possibly except that the last 2^k -block has no zero entry. We will guarantee that $Head_k$ is 2^k -sparse. Note that when the head is 2^k -sparse, the unique zero position of the first 2^k -block $W[0 : 2^k - 1]$ is always 0 ($W[0] = 0$) and $W[1 : 2^k - 1]$ contains no zeros.



■ **Figure 2** Illustration of the preprocessing invariant. W is partitioned into head and tail. The head is 2^k -sparse and the tail is finalized. The 2^k -sparsity is achieved by duels. The tail grows by at least 2^k at each round.

■ **Algorithm 4** Assuming that $W[0 : r]$ is 2^k -sparse, returns positions of zeros in $W[l : r]$.

```

1 Function GetZeros( $l, r, k$ )
2   create array  $A[0 : \lfloor r/2^k \rfloor - \lfloor l/2^k \rfloor]$  and initialize elements to  $-1$ ;
3   for each  $i \in \{l, \dots, r\}$  do in parallel
4     if  $W[i] = 0$  then  $A[\lfloor i/2^k \rfloor - \lfloor l/2^k \rfloor] \leftarrow i$ ;
5   return  $A$ ;

```

Initially, the entire table is the head and the size of the tail is zero: $Head_0 = W$ and $Tail_0 = \varepsilon$. The head is shrunk and the tail is extended by the following rule. Let the *suspected period* p_k at round k be the first zero position after the index 0, i.e., p_k is the unique position in the second 2^k -block such that $W[p_k] = 0$. Then, we let $Head_{k+1} = W[0 : m - x - 1]$ and $Tail_{k+1} = W[m - x : m - 1]$ for $x = |Tail_{k+1}| = \max(|Tail_k| + 2^k, LCP_P(p_k))$. When $|Head_k| < 2^k$, the 2^k -sparsity means that all the positions in the witness table are finalized. So, Algorithm 3 exits the while loop and halts. The goal of this subsection is to show the following theorem.

► **Theorem 9.** *Given \tilde{P} , the pattern preprocessing Algorithm 3 computes a witness table in $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on the P-CRCW PRAM.*

Proof. By Lemmas 13 and 15. ◀

In the remainder of this subsection, we explain how to maintain the 2^k -sparsity of the head and finalize the tail. Before going into the detail, we prepare a technical function **GetZeros**(l, r, k) in Algorithm 4, which returns positions $i \in \{l, \dots, r\}$ such that $W[i] = 0$ in an array, assuming that $W[0 : r]$ satisfies the 2^k -sparsity. Algorithm 4 runs in $O(1)$ time and $O(r)$ work on the P-CRCW PRAM.

Comparison with Vishkin’s algorithm

The preprocessing algorithm for exact matching by Vishkin [19] also constructs a witness table so that it satisfies the 2^k -sparsity, incrementing k , where it has no head/tail separation. Maintaining the 2^k -sparsity for the whole table is possible due to the periodicity property which holds for the exact identity but not for general SCERs. Let $p \leq \lfloor i/2 \rfloor$ be the shortest period of $P[: i]$ for some i . In exact matching if $P[i] \neq P[i + j - 1]$, $P[i - p] \neq P[i + j - 1]$ holds. Thus, we can update $W[j + p]$ by using $W[j]$ i.e., we may let $W[j + p] = W[j] - p$. However, this property does not hold on SCERs generally. Still, Vishkin’s technique for

■ **Algorithm 5** Satisfy 2^{k+1} -sparsity of $Head_{k+1} = W[0 : x]$.

```

1 Function SatisfySparsity( $x, k$ )
2    $A \leftarrow \text{GetZeros}(2^{k+1}, x, k)$ ;
3   for each  $i \in \{0, 1, \dots, \lfloor |A|/2 - 1 \rfloor\}$  do in parallel
4      $j_1 \leftarrow A[2i], j_2 \leftarrow A[2i + 1]$ ;
5     if  $j_1 \neq -1$  and  $j_2 \neq -1$  then
6        $surv \leftarrow \text{Dueling}(\tilde{P}, j_1, j_2)$ ;
7        $a \leftarrow j_2 - j_1$ ;
8       if  $surv = j_1$  then  $W[j_2] \leftarrow W[a]$ ;
9       if  $surv = j_2$  then  $W[j_1] \leftarrow W[a] + a$ ;

```

keeping the 2^k -sparsity can partially be applied to SCER cases under a certain condition (Lemma 10), which requires us to control the length of the head part carefully. Concerning the tail part, where Vishkin's technique does not work, we design a new efficient algorithm for computing witnesses.

Head invariant

First we discuss how the algorithm makes $Head_k$ 2^k -sparse. We maintain the head so that at the beginning of round k of Algorithm 3, it satisfies the following invariant properties.

- $Head_k$ is 2^k -sparse.
- For all positions i of $Head_k$,
 - $W[i] \neq 0$ implies $W[i] \in \mathcal{W}_P(i)$,
 - $W[i] \leq |Tail_k| + 2^k$.

The head maintenance procedure **SatisfySparsity** is described in Algorithm 5. Before calling the function **SatisfySparsity**, Algorithm 3 finalizes the suspected period p_k , the first position after 0 such that $W[p_k] = 0$. Due to the 2^k -sparsity, $2^k \leq p_k < 2^{k+1}$. Algorithm 3 finds the suspected period p_k at Line 4 and then finalizes the position p_k at Line 5.

Let us explain how Algorithm 5 works. The task of **SatisfySparsity**(x, k) is to make $W[0 : x]$ satisfy the 2^{k+1} -sparsity. In the case where the suspected period p_k is the smallest period of P , i.e., $\mathcal{W}_P(p_k) = \emptyset$, we have $tail = m - LCP_P(p_k) = p_k < 2^{k+1}$ when Algorithm 3 calls **SatisfySparsity**($tail - 1, k$). Then the array A obtained at Line 2 is empty and **SatisfySparsity**($tail - 1, k$) does nothing. After finalizing $Tail_{k+1}$, which will be explained later, the algorithm will halt without going into the next loop, since $|Head_{k+1}| \leq m - LCP_P(p_k) = p_k < 2^{k+1}$. At that moment all positions of W are finalized.

Hereafter we suppose that p_k is not a period of P . When **SatisfySparsity**($tail - 1, k$) is called, the value of $W[p_k]$ is the tight witness and the first 2^{k+1} -block contains no zeros except $W[0]$. At that moment, the other part of the head is 2^k -sparse. To make it 2^{k+1} -sparse, we perform duels between two zero positions i and j ($i < j$) within each of the 2^{k+1} -blocks of the head except for the first one. The witness used for the duel between i and j is $W[a]$ for $a = j - i$, which is in the first 2^{k+1} -block. The following two lemmas ensure that indeed such duels are possible. Suppose that the pattern is superimposed on itself with offsets i and j . Lemma 10 below claims that if we already know $w \in \mathcal{W}_P(a)$ and $j + w \leq m$, in other words, if the witness lies within the overlap region, then we can obtain a witness for one of the offsets i and j by dueling them using w , without looking into other positions. Lemma 11 ensures that indeed we have a witness $w = W[a]$ in our table such that $j + w \leq m$ holds.

► **Lemma 10.** *For two offsets i and $j = i + a$ with $a > 0$, suppose $w \in \mathcal{W}_P(a)$ and $j + w \leq m$. Then,*

1. *if the offset j survives the duel, i.e., $\tilde{P}_{j+1}[w] = \tilde{P}[w]$, then $w + a \in \mathcal{W}_P(i)$;*
2. *if the offset i survives the duel, i.e., $\tilde{P}_{j+1}[w] \neq \tilde{P}[w]$, then $w \in \mathcal{W}_P(j)$.*

► **Lemma 11.** *For round k , suppose the preprocessing invariant holds true and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, when SatisfySparsity is about to be called at Line 10 of Algorithm 3, for any two positions i and j of Head_{k+1} such that $0 < j - i < 2^{k+1}$, it holds that $j + W[j - i] \leq m$.*

Algorithm 5 updates the witness table in accordance with Lemma 10. In this way, the 2^k -sparsity of the head and the correctness of (non-zero) witnesses in the head are maintained. The invariants $W[i] \leq |\text{Tail}_k| + 2^k$ and $|\text{Head}_k| + \text{LCP}_P(p_k) \geq m$ are used in the proof of Lemma 11, which can be found in Appendix B.

► **Lemma 12.** *At the beginning of round k , for all $i \in \{0, \dots, 2^k - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 1$ and for all $i \in \{2^k, \dots, |\text{Head}_k| - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 2^k$.*

► **Lemma 13.** *In the round k of the while loop, Algorithm 5 updates the witness table so that Head_{k+1} is 2^{k+1} -sparse in $O(\xi_m^t)$ time and $O(\xi_m^w \cdot m/2^k)$ work on P-CRCW PRAM.*

Tail invariant

Next, we discuss how the algorithm finalizes Tail_{k+1} in the round k . This procedure is described in Algorithm 6. For the sake of convenience, we denote by \mathcal{T}_k the set of positions of Tail_k . Since Tail_k has already been finalized, it is enough to update $W[i]$ for $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$. We have two cases depending on how much the tail is extended.

The first case where $|\text{Tail}_{k+1}| = |\text{Tail}_k| + 2^k$ is handled naively. Since Head_k satisfies the 2^k -sparsity by the invariant, there are at most two zero positions in $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$. Algorithm 6 naively uses Algorithm 2 to finalize those positions.

Now, we consider the case $|\text{Tail}_{k+1}| = \text{LCP}_P(p_k) > |\text{Tail}_k| + 2^k$. The following lemma is a key to handle this case.

► **Lemma 14.** *Suppose $m - \text{LCP}_P(p) \leq b < m$. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \leq a \leq b$ and $a \equiv b \pmod{p}$.*

Let us partition $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$ into p_k subsets $\mathcal{S}_0, \dots, \mathcal{S}_{p_k-1}$ where $\mathcal{S}_s = \{i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k \mid i \equiv s \pmod{p_k}\}$, some of which can be empty. Lemma 14 implies that for each $s \in \{0, \dots, p_k - 1\}$, there exists a boundary offset b_s such that, for every $i \in \mathcal{S}_s$, $\mathcal{W}_P(i) = \emptyset$ iff $i > b_s$. Fortunately, for many s , one can find the boundary b_s very easily, unless $\mathcal{S}_s = \emptyset$. Let $q_s = \max \mathcal{S}_s$ for non-empty \mathcal{S}_s . Due to the 2^k -sparsity and the fact $p_k < 2^{k+1}$, it holds $W[q_s] \neq 0$ for all but at most three s . If $W[q_s] \neq 0$, then q_s is the boundary. By Lemma 14, $W[W[q_s] + q_s - i] \in \mathcal{W}_P(i)$ for all $i \in \mathcal{S}_s$. Accordingly, Algorithm 6 updates those values $W[i]$ in parallel in Lines 9–11.

On the other hand, for s such that $W[q_s] = 0$, Algorithm 7 uses binary search to find b_s and a witness $w \in \mathcal{W}_P(b_s)$ if it exists. Then, following Lemma 14, Algorithm 7 sets in parallel $W[i]$ to $w + (b_s - i)$ where $w \in \mathcal{W}_P(b_s)$ for $i \in \mathcal{S}_s$ such that $i \leq b_s$ (Line 10). If there is no boundary b_s in \mathcal{S}_s , then $\mathcal{W}_P(i) = \emptyset$ for all $i \in \mathcal{S}_s$. We do nothing in that case.

In Algorithm 7, the invariant is as follows. For $i \in \mathcal{S}_s$, $\mathcal{W}_P(i) \neq \emptyset$ if $i \leq l \cdot p_k + s$, and $\mathcal{W}_P(i) = \emptyset$ if $i \geq r \cdot p_k + s$. Each condition check of the binary search (Line 5) takes $O(\xi_m^t)$ time and $O(\xi_m^w \cdot m)$ work. Thus, the overall complexity of Algorithm 7 is $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work.

► **Lemma 15.** *In round k , Algorithm 6 finalizes Tail_{k+1} in $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work on P-CRCW PRAM.*

■ **Algorithm 6** Finalize $Tail_{k+1}$.

```

1 Function FinalizeTail(tail, old_tail, p, k)
2   if  $old\_tail - tail = 2^k$  then
3      $Z \leftarrow \text{GetZeros}(tail, old\_tail - 1, k);$  /*  $|Z| \leq 2$  */
4     for  $i = 0$  to  $|Z| - 1$  do
5        $z \leftarrow Z[i];$ 
6       if  $z \neq -1$  then
7          $W[z] \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1 : m - z], \tilde{P}_{z+1}[1 : m - z])$ 
8     else
9       for each  $i \in \{tail, \dots, old\_tail - 1\}$  do in parallel
10         $q \leftarrow j$  where  $j \in \{old\_tail - p, \dots, old\_tail - 1\}$  and  $j \equiv i \pmod{p}$ ;
11        if  $W[i] = 0$  and  $W[q] \neq 0$  then  $W[i] \leftarrow W[q] + q - i;$ 
12         $Z \leftarrow \text{GetZeros}(old\_tail - p, old\_tail - 1, k);$  /*  $|Z| \leq 3$  */
13        for  $i = 0$  to  $|Z| - 1$  do
14           $z \leftarrow Z[i];$ 
15          if  $z \neq -1$  then Finalize(tail, old_tail, p,  $z \bmod p$ );

```

■ **Algorithm 7** Finalize $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$ s.t. $i \equiv s \pmod{p_k}$.

```

1 Function Finalize(tail, old_tail, p, s)
2    $l \leftarrow \lceil (tail - s)/p \rceil - 1, r \leftarrow \lfloor (old\_tail - 1 - s)/p \rfloor + 1;$ 
3   while  $r - l > 1$  do
4      $i \leftarrow \lfloor (l + r)/2 \rfloor, j \leftarrow i \cdot p + s;$ 
5     if  $\text{GetTightMismatchPos}(\tilde{P}[1 : m - j], \tilde{P}_{j+1}[1 : m - j]) = 0$  then  $r \leftarrow i;$ 
6     else  $l \leftarrow i;$ 
7    $b_s \leftarrow l \cdot p + s;$ 
8    $w \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1 : m - b_s], \tilde{P}_{b_s+1}[1 : m - b_s]);$ 
9   for each  $i \in \{tail, \dots, b_s\}$  do in parallel
10    if  $W[i] = 0$  and  $i \equiv b_s \pmod{p}$  then  $W[i] \leftarrow w + b_s - i;$ 

```

3.2 Pattern searching

Our pattern searching algorithm prunes candidates in two stages: dueling and sweeping stages. During the dueling stage, candidate positions duel with each other, until the surviving candidate positions are pairwise consistent. During the sweeping stage, the surviving candidates from the dueling stage are further pruned so that only pattern occurrences survive. To keep track of the surviving candidates, we introduce a Boolean array $C[1 : m]$ and initialize every entry of C to *True*. If a candidate T_i gets eliminated, we set $C[i] = \text{False}$. The pattern searching algorithm updates C in such a way that $C[i] = \text{True}$ iff i is a pattern occurrence. Entries of C are updated at most once during the dueling and sweeping stages.

Comparison with Vishkin's algorithm

When considering exact matching, Vishkin [19] found that if the pattern is periodic, i.e., $P = Q^j Q'$ for some aperiodic string Q , a proper prefix Q' of Q , and $j \geq 2$, the problem can be reduced to finding occurrences of Q and Q' in the text. Then a position i is an occurrence

■ **Algorithm 8** Parallel algorithm for the dueling stage.

```

1 Function DuelingStageParallel()
2   for each  $j \in \{1, \dots, m\}$  do in parallel
3      $\mathcal{C}_{0,j}[1] \leftarrow j$ ;
4    $k \leftarrow 1$ ;
5   while  $k \leq \lceil \log m \rceil$  do
6     for each  $j \in \{1, \dots, \lceil m/2^k \rceil\}$  do in parallel
7        $\mathcal{A} \leftarrow \mathcal{C}_{k-1,2j-1}$ ,  $\mathcal{B} \leftarrow \mathcal{C}_{k-1,2j}$ ;
8        $\langle a, b \rangle \leftarrow \text{Merge}(\mathcal{A}, \mathcal{B})$ ;
9       Let  $\mathcal{C}_{k,j}$  be array of length  $(a + |\mathcal{B}| - b + 1)$ ;
10      for each  $i \in \{1, \dots, a\}$  do in parallel
11         $\mathcal{C}_{k,j}[i] \leftarrow \mathcal{A}[i]$ ;
12      for each  $i \in \{b, \dots, |\mathcal{B}|\}$  do in parallel
13         $\mathcal{C}_{k,j}[a + i - b + 1] \leftarrow \mathcal{B}[i]$ ;
14     $k \leftarrow k + 1$ ;
15  Initialize all elements of  $C$  to False;
16  for each  $i \in \{1, \dots, \lceil C_{\lceil \log m \rceil, 1} \rceil\}$  do in parallel
17     $C[\lceil C_{\lceil \log m \rceil, 1} \rceil[i]] \leftarrow \text{True}$ ;

```

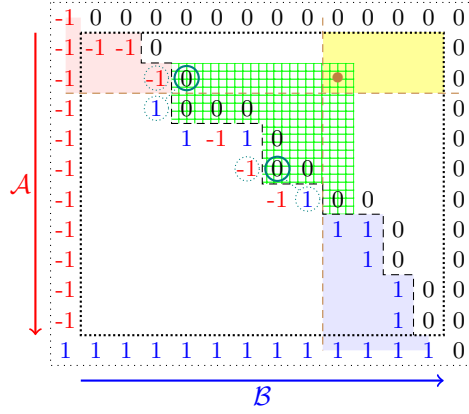
of P if and only if i is a starting position of j consecutive occurrences of Q followed by an occurrence of Q' . His dueling stage keeps the table C to be 2^k -sparse in the sense that $C[i] = \text{True}$ for at most one position i in every 2^k -block, incrementing k up to $\lfloor \log |Q|/2 \rfloor$. This can be done without violating the invariant since the occurrences of an aperiodic string Q are guaranteed to be sparse in the sense that the distance of two consecutive occurrences is bigger than $|Q|/2$. Then the sweeping stage naively checks whether those sparse surviving positions i with $C[i] = \text{True}$ are real occurrences. Apparently, this idea does not work at all in SCER matching. If P has a period p under an SCER, it does not mean that P is a repetition of $Q = P[1 : p]$ or that consecutive occurrences of Q form an occurrence of P . Our dueling and sweeping algorithms presented here are quite different from Vishkin's.

Dueling stage

The dueling stage is described in Algorithm 8. A set of positions is *consistent* if all elements in the set are pairwise consistent. During the round k , the algorithm partitions the candidate positions into blocks of size 2^k . Let $\mathcal{C}_{k,j} \subseteq \{(j-1)2^k + 1, \dots, j \cdot 2^k\}$ be the set of candidate positions in the j -th 2^k -block which have survived after the round k . The invariant of Algorithm 8 is as follows.

- At any point of execution of Algorithm 8, all pattern occurrences survive.
- For round k , each $\mathcal{C}_{k,j}$ is consistent.

Set $\mathcal{C}_{k,j}$ is obtained by “merging” $\mathcal{C}_{k-1,2j-1}$ and $\mathcal{C}_{k-1,2j}$. That is, $\mathcal{C}_{k,j}$ shall be a consistent subset of $\mathcal{C}_{k-1,2j-1} \cup \mathcal{C}_{k-1,2j}$ which contains all the occurrence positions in $\mathcal{C}_{k-1,2j-1} \cup \mathcal{C}_{k-1,2j}$. After the dueling stage, $\mathcal{C}_{\lceil \log m \rceil, 1}$ is a consistent set including all the occurrence positions. We then let $C[i] = \text{True}$ iff $i \in \mathcal{C}_{\lceil \log m \rceil, 1}$. In our algorithm, each set $\mathcal{C}_{k,j}$ is represented as an integer array, where elements are sorted in increasing order.



■ **Figure 3** Padded grid G given two consistent sets \mathcal{A} and \mathcal{B} . All the occurrence positions are inside the yellow area, where the brown dot indicates \hat{i} and \hat{j} . The red- and blue-shaded areas consist of -1 and 1 only, respectively (Lemma 17). Any point (i, j) in the green-shaded area, where $i \geq \hat{i}$, $j \leq \hat{j}$, and $G[i][j] = 0$, is satisfactory to output. Among those, our algorithm finds a point (i, j) such that $G[i][j] = 0$, $G[i][j-1] = -1$ and $G[i+1][j'] = 1$ for some j' (Lemma 18). For example, the coordinates marked with green circles may be output.

Let us consider merging two respectively consistent sets $\mathcal{A}(= \mathcal{C}_{k-1, 2j-1})$ and $\mathcal{B}(= \mathcal{C}_{k-1, 2j})$ where \mathcal{A} precedes \mathcal{B} , i.e., $\max \mathcal{A} < \min \mathcal{B}$. Sets \mathcal{A} and \mathcal{B} should be merged in such a way that the resulting set is consistent and contains all occurrences in \mathcal{A} and \mathcal{B} . That is, we must find a consistent set \mathcal{C} such that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{C} \subseteq \mathcal{A} \cup \mathcal{B}$, where $\hat{\mathcal{A}} = \{a \in \mathcal{A} \mid T_a \approx P\}$ and $\hat{\mathcal{B}} = \{b \in \mathcal{B} \mid T_b \approx P\}$ are the sets of occurrences in \mathcal{A} and \mathcal{B} , respectively. By the consistency property in Lemma 7, we can easily confirm that the following lemma holds.

► **Lemma 16.** *Suppose that we are given two respectively consistent position sets \mathcal{A} and \mathcal{B} such that \mathcal{A} precedes \mathcal{B} . If $a \in \mathcal{A}$ and $b \in \mathcal{B}$ are consistent, then $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ is also consistent, where $\mathcal{A}_{\leq a} = \{i \in \mathcal{A} \mid i \leq a\}$ and $\mathcal{B}_{\geq b} = \{j \in \mathcal{B} \mid j \geq b\}$.*

Therefore, it suffices to find $(a, b) \in \mathcal{A} \times \mathcal{B}$ such that a and b are consistent and $a \geq \max \hat{\mathcal{A}}$ and $b \leq \min \hat{\mathcal{B}}$. Then, $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ has the desired property.

To find such a pair (a, b) , let us consider a grid G of size $(|\mathcal{A}| + 2) \times (|\mathcal{B}| + 2)$. For $1 \leq i \leq |\mathcal{A}|$ and $1 \leq j \leq |\mathcal{B}|$, $G[i][j]$ represents the result of the duel between $\mathcal{A}[i]$ and $\mathcal{B}[j]$, which are the i -th and j -th smallest elements of \mathcal{A} and \mathcal{B} , respectively. We define $G[i][j] = 0$ if $W[d] = 0$ for $d = \mathcal{B}[j] - \mathcal{A}[i]$. If $W[d] \neq 0$ and $\mathcal{A}[i]$ wins the duel, then $G[i][j] = -1$. Otherwise, $\mathcal{B}[j]$ wins the duel and $G[i][j] = 1$. For the sake of technical convenience, we pad grid G with -1 s along the leftmost column, with 1 s along the bottom row, and with 0 s along the upper row and rightmost column. Specifically, $G[i][0] = -1$ for $i \in \{0, \dots, |\mathcal{A}|\}$, $G[|\mathcal{A}|+1][j] = 1$ for $j \in \{0, \dots, |\mathcal{B}|\}$, $G[i][|\mathcal{B}|+1] = 0$ for $i \in \{1, \dots, |\mathcal{A}|+1\}$, and $G[0][j] = 0$ for $j \in \{1, \dots, |\mathcal{B}|+1\}$. We will not compute the whole G , but this concept helps understanding the behavior of our algorithm. Figure 3 illustrates the grid, where elements of \mathcal{A} and \mathcal{B} are presented along the directions of rows and columns, respectively.

Lemma 16 implies that if $G[i][j] = 0$ then $G[i'][j'] = 0$ for any $i' \leq i$ and $j' \geq j$. Therefore, grid G can be divided into two regions: the upper-right region that consists of only 0 and the rest that consists of a mixture of -1 and 1 . The separation line looks like a step function (Figure 3). In terms of the grid representation, our goal is to find a coordinate (i, j) in the zero region which is to the lower left of (\hat{i}, \hat{j}) (brown dot in Figure 3) where $\hat{i} = \max(\{i' \mid \mathcal{A}[i'] \in \hat{\mathcal{A}}\} \cup \{0\})$ and $\hat{j} = \min(\{j' \mid \mathcal{B}[j'] \in \hat{\mathcal{B}}\} \cup \{|\mathcal{B}| + 1\})$. Those points are

■ **Algorithm 9** Merge two consistent sets \mathcal{A} and \mathcal{B} .

```

1 Function Merge( $\mathcal{A}, \mathcal{B}$ )
2    $l_A \leftarrow 0, r_A \leftarrow |\mathcal{A}| + 1, j \leftarrow 1;$ 
3   while  $r_A - l_A > 1$  do
4      $m_A \leftarrow \lfloor (l_A + r_A)/2 \rfloor, \text{observedOne} \leftarrow \text{False};$ 
5      $l_B \leftarrow 0, r_B \leftarrow |\mathcal{B}| + 1;$ 
6     while  $r_B - l_B > 1$  do
7        $m_B \leftarrow \lfloor (l_B + r_B)/2 \rfloor;$ 
8       if  $W[\mathcal{B}[m_B] - \mathcal{A}[m_A]] = 0$  then  $r_B \leftarrow m_B;$ 
9       else
10        if Dueling( $\tilde{T}, \mathcal{A}[m_A], \mathcal{B}[m_B]$ ) =  $\mathcal{A}[m_A]$  then  $l_B \leftarrow m_B;$ 
11        else
12           $\text{observedOne} \leftarrow \text{True};$ 
13          break;
14      if  $\text{observedOne}$  then  $r_A \leftarrow m_A;$ 
15      else  $l_A \leftarrow m_A, j \leftarrow r_B;$ 
16  return  $\langle l_A, j \rangle;$ 

```

shown as the green-shaded area in Figure 3. Then, $\mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$ has the desired property. The region $\{(i', j') \mid i' \leq \hat{i} \text{ and } j' \geq \hat{j}\}$ is shaded with yellow, which consists of only zeros by Lemma 16.

The following lemma helps us to find a desired point.

► **Lemma 17.** *If $i \leq \hat{i}$, then row i consists only of non-positive elements. Similarly, if $j \geq \hat{j}$, then column j consists only of non-negative elements.*

Our algorithm seeks for a coordinate (i, j) in the following lemma.

► **Lemma 18.** *There always exists a pair (i, j) such that $i \leq |\mathcal{A}|$, $j \geq 1$, $G[i][j] = 0$, $G[i][j-1] = -1$ and $G[i+1][j'] = 1$ for some j' . For such (i, j) , it holds that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$, assuming $\mathcal{A}_{\leq \mathcal{A}[0]} = \mathcal{B}_{\geq \mathcal{B}[|\mathcal{B}|+1]} = \emptyset$.*

Let us call a row i *low* if $G[i][j] = 0$ and $G[i][j-1] = -1$ for some j , and *high* if $G[i][j'] = 1$ for some j' . Note that, by Lemma 16 and the padding, each row is either low, high, or simultaneously low and high. At any point of Algorithm 9 execution, r_A is high and l_A is low, particularly $G[l_A][j] = 0$ and $G[l_A][j-1] = -1$. When $r_A = l_A + 1$, we are done by Lemma 18. In the inner while loop, we try to see whether the row $m_A = \lfloor (l_A + r_A)/2 \rfloor$ is high or low. As soon as we learn that m_A is high, we update r_A to be m_A . If m_A is revealed to be low, l_A is updated to be m_A .

► **Lemma 19.** *At any point of Algorithm 9 execution, (1) $G[l_A][j-1] = -1$ and $G[l_A][j] = 0$, (2) $G[r_A][j'] = 1$ for some j' , and (3) $G[m_A][l_B] = -1$ and $G[m_A][r_B] = 0$.*

► **Lemma 20.** *Given a witness table, \tilde{P} , and \tilde{T} , the dueling stage runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w m \log^2 m)$ work on P-CRCW-PRAM.*

■ **Algorithm 10** Parallel algorithm for the sweeping stage.

```

1 Function SweepingStageParallel()
2   create  $R[1 : m]$  and initialize elements of  $R$  to 0;
3    $k \leftarrow \lceil \log m \rceil$ ;
4   while  $k \geq 0$  do
5     create  $Piv[0 : \lfloor m/2^k \rfloor]$  and initialize its elements to  $-1$ ;
6     for each  $i \in \{1, \dots, m\}$  do in parallel
7       if  $C[i] = True$  and  $(i \bmod 2^k) > 2^{k-1}$  then  $Piv[\lfloor i/2^k \rfloor] \leftarrow i$ ;
8     for each  $b \in \{0, \dots, \lfloor m/2^k \rfloor\}$  do in parallel
9        $x \leftarrow Piv[b]$ ;
10      if  $x \neq -1$  then
11         $w \leftarrow \text{GetTightMismatchPos}(\tilde{P}[R[x] + 1 : m], \tilde{T}_x[R[x] + 1 : m])$ ;
12        if  $w = 0$  then  $R[x] \leftarrow m$ ;
13        else  $R[x] \leftarrow R[x] + w - 1$ ;
14      for each  $i \in \{1, \dots, m\}$  do in parallel
15         $x \leftarrow Piv[\lfloor i/2^k \rfloor]$ ;
16        if  $i \leq x$  and  $R[x] \leq m - (x - i) - 1$  then  $C[i] \leftarrow False$ ;
17        if  $i > x$  and  $C[i] = True$  then  $R[i] \leftarrow R[x] - (i - x)$ ;
18       $k \leftarrow k - 1$ ;

```

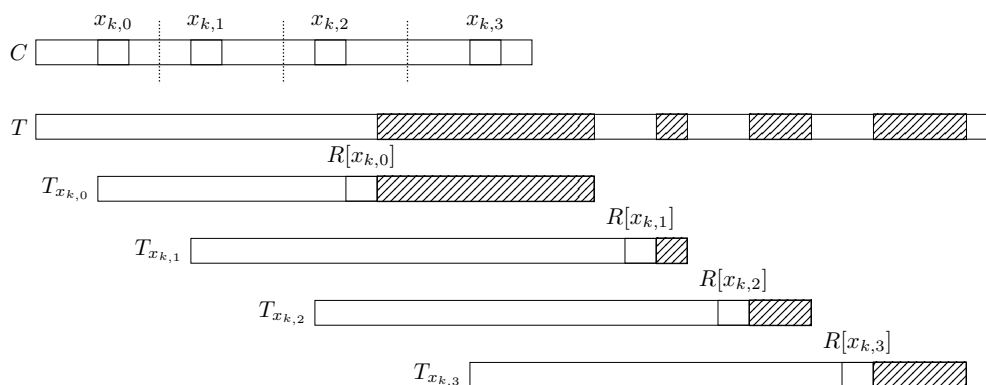
Sweeping stage

The sweeping stage is described in Algorithm 10. The sweeping stage updates C until $C[i] = True$ iff i is a pattern occurrence. All entries in C are updated at most once. Recall that all candidates that survived from the dueling stage are pairwise consistent. In addition to C , we will create a new integer array $R[1 : m]$. Throughout the sweeping stage, we have the following invariant properties:

- if $C[x] = False$, then $T_x \not\approx P$,
- if $C[x] = True$, then $LCP(T_x, P) \geq R[x]$.

The purpose of bookkeeping this information in R is to ensure that the sweeping stage algorithm uses $O(n)$ processors in each round. We do not want to access the same position of the text for each candidate covering the position. For two consistent candidate positions x and $x + a$ with $a > 0$, once we have calculated the value $r = LCP(T_x, P)$, we know that $LCP(T_{x+a}, P) \geq r - a$ for free, i.e., $\tilde{T}_{x+a}[1 : r - a] = \tilde{P}[1 : r - a]$. Then it suffices to check $\tilde{T}_{x+a}[r - a + 1 : m] = \tilde{P}[r - a + 1 : m]$. We keep the value $r - a$ in $R[x + a]$ for this trick, if $r - a \geq 0$. Throughout this section, we assume that a processor is attached to each position of C and T .

For each stage k , C is divided into 2^k -blocks. Unlike the preprocessing algorithm, k starts from $\lceil \log m \rceil$ and decreases with each round until $k = 0$. Let us look at each round in more detail. For the b -th 2^k -block of C , we pick as the “pivot” the smallest index $x_{k,b}$ in the second half of the 2^k -block such that $C[x_{k,b}] = True$. In Algorithm 10, we introduce array $Piv[0 : \lfloor m/2^k \rfloor]$ where $Piv[b] = x_{k,b}$. For each $x_{k,b}$, the algorithm computes $LCP(T_{x_{k,b}}, P)$ exactly and store the value in $R[x_{k,b}]$ on Lines 11–13. Suppose that $LCP(T_{x_{k,b}}, P) < m$, i.e., $T_{x_{k,b}} \not\approx P$ and $w = LCP(T_{x_{k,b}}, P) + 1$ is the tight mismatch position. Since all surviving candidate positions are pairwise consistent, if $T_{x_{k,b}} \not\approx P$, then, any candidate $T_{x_{k,b}-a}$ that “covers” w cannot match the pattern. Generally, we have the following.



■ **Figure 4** Illustrating the sweeping stage. The shaded regions of the text T are referenced during round k . Those referenced regions do not overlap.

► **Lemma 21.** *If two candidate positions x and $(x - a)$ with $a > 0$ are consistent and $LCP(T_x, P) \leq m - a - 1$, then $(x - a)$ is not an occurrence.*

Based on Lemma 21, Algorithm 10 updates $C[i]$ for indices i in the first half of each 2^k -block at Line 16. On the other hand, at Line 17, the algorithm updates the values of $R[i]$ for indices i in the second half of the block if $C[i] = True$. Since the surviving candidates are pairwise consistent, for candidate positions $(x_{k,b} + a)$ such that $a > 0$, $T_{x_{k,b}+a}[1 : r] \approx P[1 : r]$ for $r = R[x_{k,b}] - a$. In this way, the algorithm maintains the invariant properties. When $k = 0$, all the 2^k -blocks contain just one position x and $R[x]$ is set to be exactly $LCP(T_x, P)$ by Lines 11–13, unless $C[x] = False$ at that time. Then, if $R[x] < m$, then $C[x]$ will be *False* on Line 16. That is, when the algorithm halts, $C[x] = True$ iff $T_x \approx P$.

It remains to show the efficiency of the algorithm. We can prove the following lemmas.

► **Lemma 22.** *Each round of the while loop of Algorithm 10 can be performed in $O(\xi_m^t)$ time with $O(n)$ processors.*

► **Lemma 23.** *Given \tilde{P} and \tilde{T} , the sweeping stage algorithm finds all pattern occurrences in $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work on the P-CRCW PRAM.*

By Theorem 9 and Lemmas 20 and 23, we obtain the main theorem. Recall that when $n \geq 2m$, T is cut into overlapping pieces of length $(2m - 1)$ and each piece is processed independently.

► **Theorem 24.** *Given a witness table, \tilde{P} , and \tilde{T} , the pattern searching solves the pattern searching problem under SCER in $O(\xi_m^t \cdot \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work on the P-CRCW PRAM.*

4 Conclusion

Dueling [19] is a powerful technique, which enables us to perform pattern matching efficiently. In this paper, we have generalized the dueling technique for SCERs and have proposed a duel-and-sweep algorithm that solves the pattern matching problem for any SCER. Our algorithm is the first algorithm to solve any SCER pattern matching problem in parallel. Given a witness table, \tilde{P} , and \tilde{T} , we have shown that pattern searching under any SCER can be performed in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w n \log^2 m)$ work on P-CRCW PRAM. Given

\tilde{P} , a witness table can be constructed in $O(\xi_m^t \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on P-CRCW PRAM. The third condition of \approx -encoding in Definition 3 ensures the generality of our duel-and-sweep algorithm for SCERs. However, some standard encoding method of an SCER, namely the nearest neighbor encoding for order-preserving matching, does not fulfill the third condition. We do not know if there is an alternative encoding for order-preserving matching that fulfills the condition and is computationally as cheap as the nearest neighbor encoding. Nevertheless, Jargalsaikhan et al. [11, 12] succeeded in designing a parallel duel-and-sweep algorithm for order-preserving matching using the nearest neighbor encoding, which appears quite similar to the SCER algorithm proposed in this paper. In our future work, we would like to investigate the relation between the encoding function and the dueling technique and further generalize the definition of encoding so that it becomes more inclusive.

References

- 1 Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.
- 2 Amihood Amir, Gary Benson, and Martin Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994.
- 3 Amihood Amir and Eitan Konradovsky. Sufficient conditions for efficient indexing under different matchings. In *Proceedings of 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 4 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of computer and system sciences*, 52(1):28–42, 1996.
- 5 Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.
- 6 Ayelet Butman, Revital Eres, and Gad M. Landau. Scaled and permuted string matching. *Information processing letters*, 92(6):293–297, 2004.
- 7 Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In *Proceedings of the Prague Stringology Conference 2009*, pages 105–117, 2009.
- 8 Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms (TALG)*, 11(2):12, 2014.
- 9 Diptarama Hendrian. Generalized dictionary matching under substring consistent equivalence relations. In *Proceedings of the 14th International Workshop on Algorithms and Computation*, pages 120–132, 2020.
- 10 Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.
- 11 Davaajav Jargalsaikhan, Diptarama, Yohei Ueki, Ryo Yoshinaka, and Ayumi Shinohara. Duel and sweep algorithm for order-preserving pattern matching. In *Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2018)*, pages 624–635, 2018.
- 12 Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Parallel duel-and-sweep algorithm for the order-preserving pattern matching. In *Proceedings of the 46th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2020)*, pages 211–222, 2020.
- 13 Natsumi Kikuchi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Computing covers under substring consistent equivalence relations. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval*, pages 131–146, 2020.
- 14 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

- 15 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.
- 16 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
- 17 Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theoretical Computer Science*, 656:225–233, 2016.
- 18 Sung Gwan Park, Amihod Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching*, pages 16:1–16:14, 2019.
- 19 Uzi Vishkin. Optimal parallel pattern matching in strings. In *Proceedings of the 12th International Colloquium on Automata, Languages, and Programming*, pages 497–508, 1985.
- 20 Uzi Vishkin. Deterministic sampling — a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991.

A Examples of encoding

Prev-encoding for parameterized matching

For a string X of length n over $\Sigma \cup \Pi$, where Π is an alphabet of parameter symbols and Σ is an alphabet of constant symbols, the *prev-encoding* [4] for X , denoted by $prev_X$, is defined to be a string over $\Sigma \cup \mathbb{N}$ of length n such that for each $1 \leq i \leq n$,

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

► **Theorem 25.** *Given a string X of length n , $prev_X$ can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM. Moreover, given $prev_X$, $prev_{X[x:n]}[i]$ can be computed in $O(1)$ time and $O(1)$ work.*

Proof. Without loss of generality, we assume that Π forms a totally ordered domain. We will construct the following string X' from X . We define a new symbol, say ∞ , such that, for any element $\pi \in \Pi$, π is less than ∞ . For $1 \leq i \leq |X|$, $X'[i] = X[i]$ if $X[i] \in \Pi$ and $X'[i] = \infty$ if $X[i] \in \Sigma$. For X' , we construct $Lmax_{X'}$, which is defined as $Lmax_{X'}[i] = j$ if $X'[j] = \max_{k < i} \{X'[k] \mid X'[k] \leq X'[i]\}$. We use the rightmost (largest) j if there exist more than one such $j < i$. If there is no such j , then we define $Lmax_{X'}[i] = 0$. Suppose that $X[i] \in \Pi$ for $1 \leq i \leq |X|$. After computing $Lmax_{X'}$, $prev_X[i] = i - Lmax_{X'}[i]$ if $X[i] = X[Lmax_{X'}[i]]$. If $Lmax_{X'}[i] = 0$ or $X[i] \neq X[Lmax_{X'}[i]]$, then $X[i]$ is the first occurrence of this letter. Thus, $prev_X$ can be computed from $Lmax_{X'}$ in $O(1)$ time and $O(n)$ work. Since $Lmax_{X'}$ can be computed in $O(\log n)$ time and $O(n \log n)$ work [12], overall complexities are $O(\log n)$ time and $O(n \log n)$ work.

Given $prev_X$, $prev_{X[x:n]}[i]$ can be computed in the following manner in $O(1)$ time and $O(1)$ work.

$$prev_{X[x:n]}[i] = \begin{cases} 0 & \text{if } X[x+i-1] \in \Pi \text{ and } prev_X[x+i-1] \geq i, \\ prev_X[x+i-1] & \text{otherwise.} \end{cases} \blacktriangleleft$$

Parent-distance encoding for cartesian-tree matching

For a string X over a totally ordered alphabet, its parent-distance encoding [18] for cartesian-tree matching PD_X is defined as follows.

$$PD_X[i] = \begin{cases} i - \max_{1 \leq j < i} \{j \mid X[j] \leq X[i]\} & \text{if such } j \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

► **Theorem 26.** *Given a string X of length n , PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM. Moreover, given PD_X , $PD_{X[x:n]}[i]$ can be computed in $O(1)$ time and $O(1)$ work.*

Proof. For $1 \leq i \leq n$, $PD_X[i]$ is the nearest smaller value to the left of $X[i]$. Since the all-smaller-nearest-value problem can be solved in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM by Berkman et al. [5], PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM.

Given PD_X , $PD_{X[x:n]}[i]$ can be computed in the following manner in $O(1)$ time and $O(1)$ work.

$$PD_{X[x:n]}[i] = \begin{cases} 0 & \text{if } PD_X[x+i-1] \geq i, \\ PD_X[x+i-1] & \text{otherwise.} \end{cases} \quad \blacktriangleleft$$

B Proofs

► **Proposition 4.** *An equivalence relation \approx is an SCER if and only if it admits an \approx -encoding.*

Proof. It suffices to show the “if” direction. Suppose we have an \approx -encoding f . If $X \approx Y$, then $f(X) = f(Y)$ by (4) of Definition 3. In this case, we have $f(X[j:k])[i] = f(Y[j:k])[i]$ for any $1 \leq j \leq k \leq |X|$ and $1 \leq i \leq k-j+1$ by $f(X)[i+j-1] = f(Y)[i+j-1]$, (3), and (2). Hence, $X[j:k] \approx Y[j:k]$ by (4). ◀

► **Lemma 6.** *Suppose $w \in \mathcal{W}_P(a)$. Then,*

- if $\tilde{T}_{x+a}[w] = \tilde{P}[w]$, then $T_x \not\approx P$,
- if $\tilde{T}_{x+a}[w] \neq \tilde{P}[w]$, then $T_{x+a} \not\approx P$.

Proof. If $\tilde{T}_{x+a}[w] \neq \tilde{P}[w]$, then by the fourth property of the \approx -encoding (Definition 3), $T_{x+a} \not\approx P$. If $\tilde{T}_{x+a}[w] = \tilde{P}[w] \neq \tilde{P}_{a+1}[w]$, then by the third property of the \approx -encoding, $\tilde{T}_x[w+a] \neq \tilde{P}[w+a]$, so $T_x \not\approx P$. ◀

► **Lemma 27.** *Suppose that a and b are periods of X . If $a+b < |X|$, then $(b+a)$ is a period of X . If $a < b$, then $(b-a)$ is a period of $X[1:|X|-a]$.*

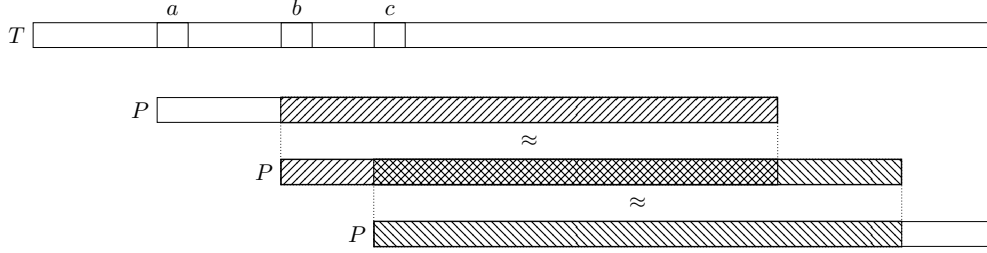
This lemma implies that if p is a period of X , then so is qp for every positive integer $q \leq \lfloor (|X|-1)/p \rfloor$.

Proof. Let $n = |X|$. Since a is a period of X , by the definition $X[1:n-a] \approx X[1+a:n]$. Thus, $X[1+b:n-a] \approx X[a+b+1:n]$. Similarly, since b is a period of X , by the definition $X[1:n-b] \approx X[1+b:n]$. Thus, $X[1:n-b-a] \approx X[1+b:n-a]$. Thus, $X[1+b:n-a] \approx X[a+b+1:n] \approx X[1:n-b-a]$, which means that $(b+a)$ is a period of X .

Since a and b are period of X , $X[1+b-a:n-a] \approx X[1+b:n]$ and $X[1:n-b] \approx X[1+b:n]$. Thus, by the transitivity property, $(b-a)$ is a period of $X[1:n-a]$. ◀



■ **Figure 5** Suppose that a and b are periods of X . If $a + b < |X|$, then $(b + a)$ is a period of X . If $a < b$, then $(b - a)$ is a period of $X[1 : |X| - a]$.



■ **Figure 6** For candidate positions $a < b < c$, if a is consistent with b and b is consistent with c , then a is consistent with c .

► **Lemma 7.** For any a, b, c such that $0 < a \leq b \leq c < m$, if a is consistent with b and b is consistent with c , then a is consistent with c .

Proof. By Lemma 27. ◀

► **Lemma 11.** For round k , suppose the preprocessing invariant holds true and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, when `SatisfySparsity` is about to be called at Line 10 of Algorithm 3, for any two positions i and j of Head_{k+1} such that $0 < j - i < 2^{k+1}$, it holds that $j + W[j - i] \leq m$.

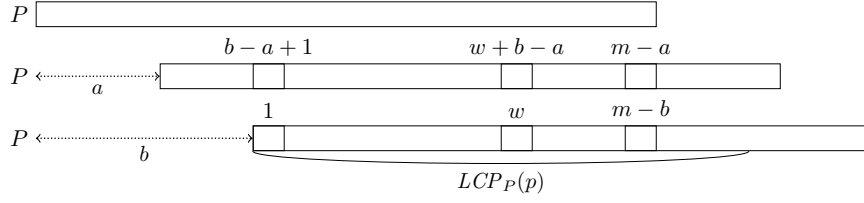
Proof. Let $a = j - i$ and $w = W[a]$. Recall that a belongs to the first 2^{k+1} -block and $W[a]$ is updated only if $a = p_k$. Suppose $a \neq p_k$. At the beginning of round k , by the invariant property, we have $w \leq |\text{Tail}_k| + 2^k$. Since $j < |\text{Head}_{k+1}| = m - |\text{Tail}_{k+1}|$, $j + w \leq j + |\text{Tail}_k| + 2^k < m - |\text{Tail}_{k+1}| + |\text{Tail}_k| + 2^k$. Since $|\text{Tail}_{k+1}| - |\text{Tail}_k| \geq 2^k$, $m - |\text{Tail}_{k+1}| + |\text{Tail}_k| + 2^k < m$. Thus, $j + w \leq m$.

If $a = p_k$, $w = W[p_k]$ is the tight witness for offset p_k , i.e., $w = \text{LCP}_P(p_k) + 1$. Since $|\text{Tail}_{k+1}| \geq \text{LCP}_P(p_k)$, $j + w \leq j + |\text{Tail}_{k+1}| + 1$. Since $j < |\text{Head}_{k+1}|$, $j + |\text{Tail}_{k+1}| + 1 \leq |\text{Head}_{k+1}| + |\text{Tail}_{k+1}| \leq m$. We have proved that $j + w \leq m$. ◀

► **Lemma 12.** At the beginning of round k , for all $i \in \{0, \dots, 2^k - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 1$ and for all $i \in \{2^k, \dots, |\text{Head}_k| - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 2^k$.

Proof. We show the lemma by induction on k . At the beginning of round 0, every element of W is zero and $|\text{Tail}_0| = 0$, thus, the claim holds. We will show that the lemma holds for $k + 1$ assuming that it is the case for k .

Suppose $i < 2^{k+1}$ and $i \neq p_k$. Then $W[i]$ is not updated. By induction hypothesis, $W[i] \leq |\text{Tail}_k| + 2^k \leq |\text{Tail}_{k+1}|$ holds. Suppose $i = p_k$. If $\mathcal{W}_P(p_k) = \emptyset$, the algorithm sets $W[p_k] = 0$ and thus the claim holds. If $\mathcal{W}_P(p_k) \neq \emptyset$, the algorithm sets $W[p_k]$ to the tight witness $\text{LCP}_P(p_k) + 1$. Thus, $W[p_k] = \text{LCP}_P(p_k) + 1 \leq |\text{Tail}_{k+1}| + 1$.



■ **Figure 7** For offsets a, b such that $m - LCP_P(p) \leq b < m$ and $a \equiv b \pmod{p}$, if $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$.

Suppose $2^{k+1} \leq i < |Head_{k+1}|$. If Algorithm 5 does not update $W[i]$, by the induction hypothesis, $W[i] \leq |Tail_k| + 2^k < |Tail_{k+1}| + 2^{k+1}$ holds. Suppose Algorithm 5 updates $W[i]$ or $W[j]$ by a duel between i and j , where $2^{k+1} \leq i < j < |Head_{k+1}|$ and $a = j - i < 2^{k+1}$. We have shown above that $W[a] \leq |Tail_{k+1}| + 1$. If i wins the duel, then $W[j] = W[a] \leq |Tail_{k+1}| + 1 \leq |Tail_{k+1}| + 2^{k+1}$. If j wins the duel, then $W[i] = W[a] + a \leq |Tail_{k+1}| + 1 + a \leq |Tail_{k+1}| + 2^{k+1}$. ◀

► **Lemma 14.** *Suppose $m - LCP_P(p) \leq b < m$. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \leq a \leq b$ and $a \equiv b \pmod{p}$.*

Proof. Figure 7 may help understanding the proof. Suppose $w \in \mathcal{W}_P(b)$, i.e., $\tilde{P}_{b+1}[w] \neq \tilde{P}[w]$. Since p is a period of $P[1 : LCP_P(p)]$ and $a \equiv b \pmod{p}$, by Lemma 27, $(b - a)$ is also a period of $P[1 : LCP_P(p)]$, i.e., $P[1 + b - a : LCP_P(p)] \approx P[1 : LCP_P(p) - (b - a)]$. Particularly for the position $w \leq m - b \leq m - a$, we have $\tilde{P}_{b-a+1}[w] = \tilde{P}[w]$. Then, $\tilde{P}_{b-a+1}[w] \neq \tilde{P}_{b+1}[w]$ by the assumption (Figure 7). By Property (3) of the \approx -encoding (Definition 3), $\tilde{P}_1[b - a + w] \neq \tilde{P}_{a+1}[b - a + w]$. That is, $(w + b - a) \in \mathcal{W}_P(a)$. ◀

► **Theorem 9.** *Given \tilde{P} , the pattern preprocessing Algorithm 3 computes a witness table in $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on the P-CRCW PRAM.*

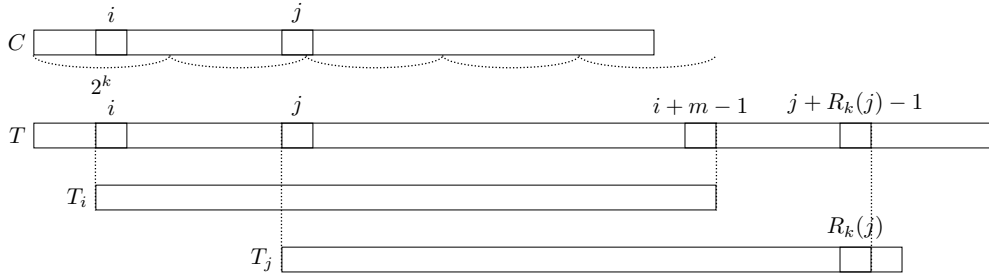
Proof. When the algorithm halts, by $2^k \leq tail$, the head size is at most 2^k . Therefore, the head is zero-free except for $W[0] = 0$ by the 2^k -sparsity. By the invariant, $W[i] \in \mathcal{W}_P(i)$ for all the positions of the head. On the other hand, every position of the tail is finalized and has a correct value in the witness table.

In Algorithm 3, the while loop runs $O(\log m)$ times, and each loop takes $O(\xi_m^t \log m)$ time and $O(\xi_m^t \cdot m \log m)$ work, by Lemmas 13 and 15. Thus, the overall complexity of Algorithm 3 is $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^t \cdot m \log^2 m)$ work. ◀

► **Lemma 17.** *If $i \leq \hat{i}$, then row i consists only of non-positive elements. Similarly, if $j \geq \hat{j}$, then column j consists only of non-negative elements.*

Proof. We prove the first half of the lemma. The second claim can be proven in the same way. Let us consider $i = \hat{i}$. Since \hat{i} is a pattern occurrence, for any j that \hat{i} is not consistent with, \hat{i} always wins the duel. Thus, if $G[\hat{i}][j] \neq 0$, then $G[\hat{i}][j] = -1$. Now, let us consider $i < \hat{i}$. For any $j \in \mathcal{B}$, $i < \hat{i} < j$. Since \mathcal{A} is a consistent set and \hat{i} is a pattern occurrence, if i is not consistent with j , i always wins any duel against j . Thus, if $G[i][j] \neq 0$, then $G[i][j] = -1$. ◀

► **Lemma 18.** *There always exists a pair (i, j) such that $i \leq |\mathcal{A}|$, $j \geq 1$, $G[i][j] = 0$, $G[i][j - 1] = -1$ and $G[i + 1][j'] = 1$ for some j' . For such (i, j) , it holds that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$, assuming $\mathcal{A}_{\leq \mathcal{A}[0]} = \mathcal{B}_{\geq \mathcal{B}[|\mathcal{B}|+1]} = \emptyset$.*



■ **Figure 8** Before round k , for two surviving candidates T_i and T_j such that $j - i \geq 2^k$, $i + m - 1 < j + R_k[j]$.

Proof. Let $i = \max\{i' \mid G[i'][j'] \leq 0 \text{ for all } j'\}$ and $j = \min\{j' \mid G[i][j'] = 0\}$. It is easy to see that those are well-defined and satisfy the desired condition.

Suppose (i, j) satisfies the condition. Since $G[i'][j'] \leq 0$ for all $i \leq \hat{i}$ and j' by Lemma 17, $G[i+1][j'] = 1$ means that $i+1 > \hat{i}$, i.e., $i \geq \hat{i}$. Similarly, $G[i][j-1] = -1$ means $j \leq \hat{j}$. ◀

► **Lemma 21.** *If two candidate positions x and $(x - a)$ with $a > 0$ are consistent and $LCP(T_x, P) \leq m - a - 1$, then $(x - a)$ is not an occurrence.*

Proof. Let $w = LCP(T_x, P) + 1$. Then $w \leq m - a$ and $T_x[1 : m - a] \not\approx P[1 : m - a]$. Since x is consistent with $(x - a)$, $P[a + 1 : m] \approx P[1 : m - a] \not\approx T_x[1 : m - a] \approx T_{x-a}[a + 1 : m]$, which means that $(x - a)$ is not a pattern occurrence. ◀

Lemma 22 follows from Lemma 28.

► **Lemma 28.** *After the round k , for two surviving candidate positions i and j with $i < j$ that do not belong to the same 2^{k-1} -block of C , $i + m \leq j + R[j]$.*

Proof. Before round $\lceil \log m \rceil$, which can be seen as after round $\lceil \log m \rceil + 1$, since all candidate positions belong to the same $2^{\lceil \log m \rceil}$ -block, the statement holds (base case). Assuming that the statement holds after the round $(k + 1)$, we prove that it also holds after the round k . Let R_{k+1} and R_k be the states of the array R after the rounds $(k + 1)$ and k , respectively. First, let us consider the case when surviving candidate positions i and j do not belong to the same 2^k -block of C . Obviously, i and j cannot belong to the same 2^{k-1} -block. By the induction hypothesis, $i + m \leq j + R_{k+1}[j]$. Since $R_k[j] \geq R_{k+1}[j]$, $i + m \leq j + R_k[j]$.

Now, let us consider the case when candidate positions i and j belong to the same 2^k -block of C . During round k , for each 2^k -block of C , Algorithm 10 chooses as surviving candidate position $x_{k,b}$ which is the smallest index in the second half of the 2^k -block. Thus, two surviving candidate positions i and j of the b -th 2^k -block belong to different 2^{k-1} -blocks iff $i < x_{k,b} \leq j$. For T_i to be a surviving candidate after round k , it must be the case that $m + i \leq LCP(T_{x_{k,b}}, P) + x_{k,b}$. For T_j , Algorithm 10 updates $R_k[j]$ to $LCP(T_{x_{k,b}}, P) - (j - x_{k,b})$. Substituting it into the previous inequality, we get $m + i \leq R_k[j] + (j - x_{k,b}) + x_{k,b} = R_k[j] + j$. ◀

► **Lemma 22.** *Each round of the while loop of Algorithm 10 can be performed in $O(\xi_m^t)$ time with $O(n)$ processors.*

Proof. Obviously it runs in constant time except for the computation at Line 11, where each processor attached to position i is used for re-encoding $\tilde{T}[i]$ into $\tilde{T}_x[i - x + 1]$ and comparing the value with $\tilde{P}[i - x + 1]$ for some x . Indeed, there is at most one b such that $x_{k,b} + R[x_{k,b}] \leq i < x_{k,b} + m$, since $x_{k,b-1} + m \leq x_{k,b} + R[x_{k,b}]$ for all $b \in \{1, \dots, \lceil m/2^k \rceil\}$ by Lemma 28. ◀

Efficient Construction of the BWT for Repetitive Text Using String Compression

Diego Díaz-Domínguez ✉

Department of Computer Science, University of Helsinki, Finland

Gonzalo Navarro ✉

CeBiB – Centre For Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Santiago, Chile

Abstract

We present a new semi-external algorithm that builds the Burrows–Wheeler transform variant of Bauer et al. (a.k.a., BCR BWT) in linear expected time. Our method uses compression techniques to reduce the computational costs when the input is massive and repetitive. Concretely, we build on induced suffix sorting (ISS) and resort to run-length and grammar compression to maintain our intermediate results in compact form. Our compression format not only saves space, but it also speeds up the required computations. Our experiments show important savings in both space and computation time when the text is repetitive. On average, we are 3.7x faster than the baseline compressed approach, while maintaining a similar memory consumption. These results make our method stand out as the only one (to our knowledge) that can build the BCR BWT of a collection of 25 human genomes (75 GB) in about 7.3 hours, and using only 27 GB of working memory.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases BWT, string compression, repetitive text

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.29

Supplementary Material *Software (Source Code)*: <https://github.com/ddiazdom/gr1BWT>
archived at `swh:1:dir:db4691b6162da5d456f6f3005daf8c23424b0120`

Funding *Diego Díaz-Domínguez*: Academy of Finland Grant 323233

Gonzalo Navarro: ANID Basal Funds FB0001 and Fondecyt Grant 1-200038, Chile

1 Introduction

The Burrows–Wheeler transform (BWT) [6] is a reversible string transformation that reorders the symbols of a text T according the lexicographical ranks of its suffixes. The features of this transform have turned it into a key component for text compression and indexing [32, 25]. In addition to being reversible, the reordering produced by the BWT reduces the number of equal-symbol runs in T , thus improving the compressibility. On the other hand, its combinatorial properties [10] enable the creation of self-indexes [9, 28] that support pattern matching in time proportional to the pattern length. Popular bioinformatic tools [18, 21] rely on the BWT to process data, as collections in this discipline are typically massive and repetitive, and the patterns to search for are short.

There are several algorithms in literature that produce the BWT in linear time [34, 1, 23, 8, 3]. Nevertheless, the computational resources their implementations require when the input is large are still too high for practical purposes. This problem is particularly evident in Genomics applications, where the amount of data is growing at an astronomical rate [35].

Although genomic collections are becoming more and more massive, the effective information they contain remains low compared to their sizes [27]. A promising solution to deal with this kind of data is then to design BWT algorithms that scale with the amount of information in the collection, not with its size.



© Diego Díaz-Domínguez and Gonzalo Navarro;
licensed under Creative Commons License CC-BY 4.0
33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 29; pp. 29:1–29:18

Leibniz International Proceedings in Informatics



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

Motivated by these ideas, some authors have developed new BWT algorithms that exploit text repetitions to reduce the computational requirements [14, 5, 13, 15, 4]. Their approach consists of extracting a set of representative strings from the text, perform calculations on them, and then extrapolate the results to the copies of those strings. For instance, the methods of Boucher et al. [5, 4] based on prefix-free parsing (PFP) use Karp–Rabin fingerprints [12] to create a dictionary of prefix-free phrases from T . Then, they create a parse by replacing the phrases in T with metasymbols, and finally construct the BWT using the dictionary and the parse. Similarly, Kempa et al. [14] consider a subset of positions in T that they call a string synchronizing set, from which they compute a partial BWT they then extrapolate to the whole text.

Although these repetition-aware techniques are promising, some of them are at a theoretical stage [14, 13, 15], while the rest [5, 4] have been empirically tested only under controlled settings, and their results depend on parameters that are not simple to tune. Thus, it is difficult to assess their performance under real circumstances.

Recently, Nunes et al. [31] proposed a method called GCIS that adapts the concept of *induced suffix sorting* (ISS) for compression. Their ideas are closely related to the linear-time BWT algorithm of Okanohara et al. [34]. Briefly, Okanohara et al. cut the text into phrases using ISS, assign symbols to the phrases, and then replace the phrases with their symbols. They apply this procedure recursively until all the symbols in the text are different. Then, when they go back from the recursions, they induce an intermediate BWT^i for the text of every recursion i using the previous BWT^{i+1} . On the other hand, GCIS stores the dictionaries that ISS generates in the recursions in a context-free grammar. The connection between these two methods is that GCIS captures in the grammar precisely the information that Okanohara et al. use to compute the BWT. Additionally, Díaz-Domínguez et al. [7] recently demonstrated that ISS-based compressors such as GCIS require much less computational resources than state-of-the-art methods like RePair [19] to encode the data, while maintaining high compression ratios. The simple construction of ISS makes it an attractive alternative to process high volumes of text. In particular, combining the ideas of Okanohara et al. with ISS-based compression is a promising alternative for computing big BWTs.

Our contribution. *Induced suffix sorting* (ISS) [17] has proved useful for compression [31, 7] and for constructing the BWT [34]. In this work, we show that compression can be incorporated in the internal stages of the BWT computation in a way that saves both working space and time. Okanohara et al. [34] use ISS to construct the BWT in recursive stages of parsing (which cuts the text into metasymbols) and partial construction of the BWT of the metasymbols; the final BWT is obtained when returning from the recursion. We use a technique similar to grammar compression to store the dictionaries of metasymbols, and run-length compression for the partial BWTs. This approach is shown not only to save the space required for those intermediate results, but importantly, the format we choose actually speeds up the computation of the final BWT as we return from the recursion, because the factorizations that help save space also save redundant computations. Unlike Okanohara et al., we receive as input a string collection and output its BCR BWT [1], a variant for string collections. The reason is that massive datasets usually contain multiple strings, in which case the BCR BWT variant is simpler to construct. Our experiments show that, when the input is a collection of human genomes (a repetitive dataset), our implementation requires 3.7x less computation time than `ropeBWT2` [21], an efficient implementation of the BCR BWT algorithm. Additionally, we use 7.6x less working memory than `pfp-ebwt` [4], a recent method that uses an strategy similar to ours. Under not so repetitive scenarios, our performance is competitive with `BCR_LCP_GSA` [1] or `gsufsort` [8].

2 Related Concepts

2.1 The Burrows–Wheeler Transform

Consider a string $T[1, n - 1]$ over alphabet $\Sigma[2, \sigma]$, and the sentinel symbol $\Sigma[1] = \$$, which we insert at $T[n]$. The *suffix array* [26] of T is a permutation $SA[1, n]$ that enumerates the suffixes $T[i, n]$ of T in increasing lexicographic order, $T[SA[i], n] < T[SA[i + 1], n]$, for $i \in [1, n - 1]$.

The *Burrows–Wheeler transform* (BWT) [6] is a reversible string transformation that stores in $BWT[i]$ the symbol that precedes the i th suffix of T in lexicographical order, i.e., $BWT[i] = T[SA[i] - 1]$ (assuming $T[0] = T[n] = \$$).

The mechanism to revert the transformation is the so-called LF mapping. Given an input position $BWT[j]$ that maps a symbol $T[i]$, $LF(j) = j'$ returns the index j' such that $BWT[j'] = T[i - 1]$ maps the preceding symbol of $T[i]$. Thus, spelling T reduces to continuously applying LF from $BWT[1]$, the symbol to the left of $T[n] = \$$, until reaching $BWT[j] = \$$.

The BCR BWT [1] is a variant of the original BWT that encodes a string collection $\mathcal{T} = \{T_1, T_2, \dots, T_k\}$ instead of a single string T . Briefly, if two (or more) symbols $a = T_x[k]$ and $b = T_y[k']$, from different strings $T_x, T_y \in \mathcal{T}$, are preceded by identical suffixes $T_x[k + 1..] = T_y[k' + 1..]$, the order of a and b in BWT is the same as the relative order of T_x and T_y in \mathcal{T} . The BCR BWT also appends sentinel symbols $\$$ to the strings of \mathcal{T} to detect their boundaries in the BWT. A position $BWT[j] = \$$ represents the start of a string T_u , and $BWT[LF(j)]$ maps the end of a string $T_{u'}$ $\in \mathcal{T}$ that is not necessarily T_u .

2.2 Grammar and Run-length Compression

Grammar compression [16] consists of encoding a text T as a small context-free grammar \mathcal{G} that only produces T . Formally, a grammar is a tuple $(V, \Sigma, \mathcal{R}, \mathbf{S})$, where V is the set of nonterminals, Σ is the set of terminals, \mathcal{R} is the set of replacement rules and $\mathbf{S} \in V$ is the start symbol. The right-hand side of $\mathbf{S} \rightarrow C \in \mathcal{R}$ is referred to as the compressed form of T . The size of \mathcal{G} is usually measured in terms of the number of rules, the sum of the lengths of the right-hand sides of \mathcal{R} , and the length of the compressed string.

Run-length compression encodes the equal-symbol runs of maximal length in T as pairs. More specifically, T becomes a sequence $(a_1, l_1), (a_2, l_2), \dots, (a_{n'}, l_{n'})$ of $n' \leq n$ pairs, where every (a_i, l_i) , with $i \in [1, n']$, stores the symbol $a_i \in \Sigma$ of the i th run and its length $l_i \geq 1$. For instance, let $T[i, j] = aaaa$ be a substring with four consecutive copies of a , where $T[i - 1] \neq a$ and $T[j + 1] \neq a$. Then $T[i, j]$ compresses to $(a, 4)$.

2.3 Induced Suffix Sorting

Induced suffix sorting (ISS) [17] computes the lexicographical ranks of a subset of suffixes in T and then uses the result to induce the order of the rest. This method is the underlying procedure in several algorithms that build the suffix array [30, 29, 22] and the BWT [34, 5] in linear time. The ISS idea introduced by the suffix array algorithm SA-IS of Nong et al. [30] is of interest to this work. The authors give the following definitions:

► **Definition 1.** A symbol $T[i]$ is called *L-type* if $T[i] > T[i + 1]$ or if $T[i] = T[i + 1]$ and $T[i + 1]$ also *L-type*. On the other hand, $T[i]$ is said to be *S-type* if $T[i] < T[i + 1]$ or if $T[i] = T[i + 1]$ and $T[i + 1]$ is also *S-type*. By definition, symbol $T[n]$, the one with the sentinel, is *S-type*.

► **Definition 2.** A symbol $T[i]$, with $i \in [1, n]$, is called *leftmost S-type*, or *LMS-type*, if $T[i]$ is S-type and $T[i - 1]$ is L-type.

► **Definition 3.** An LMS substring is (i) a substring $T[i, j]$ with both $T[i]$ and $T[j]$ being LMS symbols, and there is no other LMS symbol in the substring, for $i \neq j$; or (ii) the sentinel itself.

SA-IS is a recursive method. In every recursion i , it initializes an empty suffix array A^i for the input text T^i ($i=1$). Then, it scans T^i from right to left to classify the symbols as L-type, S-type or LMS-type. As it moves through the text, the algorithm records the text positions of the LMS substrings in A^i . More specifically, if $T^i[j] = a$ is the first symbol of an LMS substring, it inserts j in the right-most empty position in the bucket a of A^i . After scanning T^i , SA-IS sorts the LMS substrings in A^i using ISS. This procedure only requires two linear scans of A^i (we refer the reader to Nong et al. [30] for further detail).

ISS sorts the LMS substrings in a way that is slightly different from lexicographic ordering, we refer to it as \prec_{LMS} ordering. In particular, if an LMS substring $T^i[a, b]$ is a prefix of another LMS substring $T^i[a', b']$, then $T^i[a, b]$ gets higher order. However, the higher rank of $T^i[a, b]$ implies that the suffix $T^i[a..]$ is lexicographically greater than the suffix $T^i[a'..]$. The cause of this property is explained in Section 2 of Ko and Aluru [17].

The idea now is to use the sorted LMS substrings to induce the order of the suffixes in T^i that are not prefixed by LMS substrings. Still, LMS substring with the same sequence are still unsorted in A^i . Nong et al. solve this problem by creating a new string T^{i+1} in which they replace the distinct LMS substrings with their orders in A^i , and use T^{i+1} as input for another recursive call $i + 1$. The base case for the recursion is when all the suffixes in A^i are prefixed by different symbols, in which case they return A^i without further processing.

When the $(i + 1)$ th recursive call ends, the suffixes of T^i prefixed by the same LMS substrings are completely sorted in A^{i+1} , so SA-IS proceeds to complete A^i . For doing so, it resets A^i , inserts the LMS substrings arranged as they respective symbols appear in A^{i+1} , and performs ISS again to reorder the unsorted suffixes of T^i . Once it finishes, it passes A^i to the previous recursion $i - 1$. The final array A^1 is the suffix array for T .

3 Methods

3.1 Definitions

We consider a collection $\mathcal{T} = \{T_1, T_2, \dots, T_k\}$ of k strings over the alphabet $\Sigma[2, \sigma]$. The input for our algorithm is thus the sequence $T = T_1\$T_2 \dots T_k\$$ of total length $n = |T|$ that represents the concatenation of \mathcal{T} . The symbol $\$$ is a sentinel that we use as a boundary between consecutive strings in T . We map $\$ = \Sigma[1]$ to the smallest symbol in the alphabet.

Let $\mathcal{D} = \{D_1, D_2, \dots, D_y\}$ be a string set that is not suffix-free. A suffix $D_j[u..]$, with $D_j \in \mathcal{D}$, is *proper* if $1 < u \leq |D_j|$. Additionally, we consider a suffix $D_j[u..]$, with $D_j \in \mathcal{D}$, to be *left-maximal* if there is at least one other suffix $D_{j'}[u'..]$, with $D_{j'} \in \mathcal{D}$, such that (i) $j \neq j'$, (ii) $D_{j'}[u'..] = D_j[u..]$, and (iii) both $D_{j'}[u'..]$ and $D_j[u..]$ are proper suffixes with $D_{j'}[u' - 1] \neq D_j[u - 1]$ or one of them is not a proper suffix.

3.2 Overview of Our Algorithm

We call our algorithm for computing the BCR BWT of \mathcal{T} **grlBWT**. This method relies on the ideas developed by Nong et al. in the SA-IS algorithm (Section 2.3), but includes elements of grammar and run-length compression (Section 2.2). These new features reduce the space usage of the temporal data that **grlBWT** maintains in memory, thus decreasing both working memory and computing time. We now give a brief overview of our approach.

Our method works in two phases: the *parsing* phase and the *induction* phase. The parsing phase is similar to the recursive steps of SA-IS. In every iteration i (or parsing round), we first scan the input string T^i ($T^1 = T$) to build a dictionary D^i with the phrases that occur as LMS substrings. We also record the frequency of every phrase, i.e., the number of times it occurs as an LMS substring in T^i . Subsequently, we use the phrases in D^i and their frequencies to construct a preliminary BWT for T^i ($pBWT^i$), which we complete in the induction phase. We say $pBWT^i$ is partial because it has empty spaces we can not fill just with the information in D^i . To make the completion more efficient during the induction phase, we encode $pBWT^i$ using run-length compression and D^i using a technique similar to grammar compression. Finally, we store $pBWT^i$ and D^i on disk, and create a new text T^{i+1} for the next parsing round $i + 1$. We construct T^{i+1} by replacing the LMS substrings of T^i with their associated symbols in D^i . The parsing phase finishes when no new dictionary phrases can be extracted from the input text T^i (see Section 3.3).

Let h be the number of iterations the parsing phase of grlBWT incurred with T . The induction phase starts by building the BWT for T^h . After obtaining BWT^h , we start a new iterative process in which we revisit the data we dumped to disk during the parsing phase in reverse order (i.e., from round $h - 1$ to round 1). In every iteration i , the BWT^{i+1} of T^{i+1} is already computed, and we use it along with compressed version of D^i to induce the order of the symbols in the empty entries of $pBWT^i$. Once we finish the induction, we compact $pBWT^i$ using run-length encoding to create BWT^i . The final BCR BWT for T is thus in BWT^1 .

3.3 The Parsing Phase

In this section, we explain the steps we perform during the i th iteration of the parsing phase of grlBWT. Assume we receive as input a text T^i over the alphabet $\Sigma^i = [1, \sigma^i]$. We first initialize a hash table H^i that we will use to construct the dictionary D^i . The keys in H^i will be the phrases that occur as LMS substrings in T^i while the values of H^i will be the frequencies of the keys, i.e., the number of times the keys occur as LMS substrings in T^i .

The basic idea to fill H^i consists of scanning T^i from right to left to classify its symbols according the definitions of Section 2.3, and hash an LMS substring every time we reach an LMS-type symbol. This mechanism is almost the same as the one described by Nong et al. [30] to detect the LMS substrings (except for the hashing). However, we add an extra consideration. The detection of LMS substrings is oblivious of the fact that T^1 encodes a string collection rather than a single string. More precisely, in any parsing round $i > 1$, we could have an LMS substring of T^i whose *expansion*¹ produces a substring of T^1 that covers two or more strings of \mathcal{T} . These border phrases make the computation of the BCR BWT a bit more difficult as we need to treat them differently. We avoid this problem by maintaining a bit vector $B^i[1, \sigma^i]$ that marks which symbols in T^i expand to suffixes of strings in \mathcal{T} . Thus, during the right-to-left scan of T^i , each time we reach a position $T^i[j]$, such that $B^i[T^i[j]] = 1$, we truncate the active LMS substring. We record the phrase $F = T^i[j + 1, j']$ in H^i , where $T[j']$ is the last LMS-type symbol we accessed, and start a new phrase from position $T^i[j]$. Notice that the truncated strings are not LMS substrings by definition, but

¹ Let $T^i[j, j']$ be a substring of T^i . We define the *expansion* of $T^i[j, j']$ as the string in Σ^1 we obtain by recursively replacing every symbol $T^i[j] \in \Sigma^i$, for $j \in [j, j']$, with its corresponding phrases in the dictionaries $D^{i-1}, D^{i-2}, \dots, D^1$.

they do not affect our algorithm (the reasons are explained in Definition 4 and Lemma 3 of Díaz-Domínguez et al. [7]). We receive B^i as input along with T^i at the beginning of the parsing round i , and we compute the next $B^{i+1}[1, \sigma^{i+1}]$ when we finish the round.

For practical reasons, we change the representation of D^i , encoded in H^i for the moment, to a more convenient data structure. First, we concatenate all the keys of D^i in one single vector R^i . We mark the boundaries of consecutive phrases in R^i with a bit vector L^i in which we set $L^i[j] = 1$ if $R^i[j]$ is the first symbol of a phrase, and set $L^i[j] = 0$ otherwise. We also augment L^i with a data structure that supports rank_1 queries [33] to map each symbol $D^i[j]$ to its corresponding phrase. We store the values of D^i in another vector $N^i[1, |D^i|]$. We maintain the relative order so that the value $N^i[o]$ maps the o th phrase we inserted into R^i . For simplicity, we will refer to the representation (R^i, L^i, N^i) just like D^i . We still need H^i to construct the parse T^{i+1} , so we do not discard it but store it into disk.

The next step is to build $pBWT^i$ from D^i . For that purpose, we use the following observations:

► **Lemma 4.** *Let $X[1, x]$ and $Y[1, y]$ be two different strings over the alphabet Σ^i , with lengths $x > 1$ and $y > 1$ (respectively). Assume both occur as suffixes in one or more phrases of D^i . Let \mathcal{X} be the list of positions in T^i where X occurs as a suffix of an LMS substring. More specifically, each $j \in \mathcal{X}$ is a position such that $T^i[j, j + x - 1]$ is an occurrence of X and $T^i[j - j', j + x - 1]$, with $j' \geq 0$, is an LMS substring. Let us define a list \mathcal{Y} equivalent to \mathcal{X} , but for Y . If $X \prec_{LMS} Y$ (see Section 2.3), then all the suffixes of T^i starting at positions in \mathcal{X} are lexicographically greater than the suffixes starting at positions in \mathcal{Y} .*

Proof. Assume first that X is not a prefix of Y (and vice versa). We compare the sequences of these strings from left to right until we find a mismatching position u (i.e., $X[u] \neq Y[u]$). We know that symbols $X[u]$ and $Y[u]$ define the lexicographical order of the suffixes in \mathcal{X} relative to the suffixes in \mathcal{Y} . In the other scenario, when one string is a prefix of the other, we can not use this mechanism as we will not find a mismatching position $X[u] \neq Y[u]$. For this case, we resort to the symbol types of Section 2.3. We assume for this proof that X is a prefix of Y , but the other way is equivalent. We know that $X[x]$ and $Y[x]$ have different types. $X[x]$ is LMS type because X is a suffix of an LMS substring. On the other hand, $Y[x]$ is L type because if it were S type, then it would also be LMS type, and thus $Y[1, x]$ would be an occurrence for X . This observation is due to $Y[x - 1] = X[x - 1]$ is L type. Given the types of $X[x]$ and $Y[x]$, the occurrences of X in \mathcal{X} are always followed in T^i by symbols that are greater than $Y[x + 1]$, meaning that the suffixes of T^i starting at positions in \mathcal{X} are lexicographically greater than the suffixes starting at positions in \mathcal{Y} . This observation does not hold when X or Y have length one: $X[x]$ equals $Y[1]$ and both are LMS type, so there is not enough information to decide the lexicographical order of the suffixes in \mathcal{X} and \mathcal{Y} . ◀

The consequence of Lemma 4 is that the suffixes of length > 1 in D^i induce a partition over SA^i (the suffix array of T^i):

► **Lemma 5.** *Let $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ be the set of strings of length > 1 that occur as suffixes in the phrases of D^i . Additionally, let $\mathcal{O} = \{O_1, O_2, \dots, O_k\}$ be the set of occurrences in T^i for the strings in \mathcal{S} . For every $S_u \in \mathcal{S}$, its associated list $O_u \in \mathcal{O}$ stores each position j such that $T^i[j, j + |S_j| - 1]$ is an occurrence of S_u and $T^i[j - j', j + |S_j| - 1]$, with $j' \geq 0$, is an LMS substring. It holds that \mathcal{O} induces a partition over the suffix array of T^i (SA^i) as the lexicographical sorting places the elements of each $O_u \in \mathcal{O}$ in a consecutive range of SA^i .*

Proof. We demonstrate the lemma by showing that the lexicographical sorting does not interleave suffixes of T^i in SA^i that belong to different lists of \mathcal{O} . Assume a string $S_u \in \mathcal{S}$, associated with the list $O_u \in \mathcal{O}$, is a prefix in another string $S_{u'} \in \mathcal{S}$, which in turn is associated with the list $O_{u'} \in \mathcal{O}$. Even though we do not know the symbols that occur to the right of S_u in its occurrences of O_u , we do know that both S_u and $S_{u'}$ are suffixes of LMS substrings, and by Lemma 4, we know that all the suffixes of T^i in O_u are lexicographically greater than the suffixes in $O_{u'}$. Hence, the interleaving of suffixes in SA^i from different lists of \mathcal{O} is not possible, even if \mathcal{S} is not a prefix-free set. \blacktriangleleft

Lemma 5 gives us a simple way to construct the preliminary BWT for T^i ($pBWT^i$). We consider for the moment $pBWT^i$ to be a vector of lists to simplify the explanations. We first sort the strings of \mathcal{S} in \prec_{LMS} order. Then, for every *oth* string $S \in \mathcal{S}$ in \prec_{LMS} order, we insert in the list $pBWT^i[o]$ the symbols that occur to the left of S in D^i . There are three cases to consider for this task:

► **Lemma 6.** *Let $S \in \mathcal{S}$ be the string with \prec_{LMS} order o among the other strings in \mathcal{S} . If S is left-maximal in D^i , then the list $pBWT^i[o]$ contains more than one distinct symbol, and it is not possible to decide the relative order of those symbols with the information of D^i .*

Proof. Let X and Y be two phrases of D^i where S occurs as a suffix. Assume the left symbol of S in X is $x \in \Sigma^i$ and the left symbol in Y is $y \in \Sigma^i$. In this scenario, the relative order of x and y is not decided by S , but for the sequences that occur to the right of X and Y in T^i . However, those sequences are not accessible directly from D^i . Hence, it is not possible to decide the order of x and y in $pBWT^i[o]$. \blacktriangleleft

► **Lemma 7.** *Consider the string $S \in \mathcal{S}$ of Lemma 6. When S occurs as a non-proper suffix in a phrase $F \in D^i$, it is not possible to complete the sequence of symbols for $pBWT^i[o]$.*

Proof. The symbols that occur to the left of S in T^i are stored in the LMS substrings that precede F in T^i . However, it is not possible to know from D^i which are those substrings. \blacktriangleleft

We now describe the information of $pBWT^i$ that we can extract from D^i :

► **Lemma 8.** *Let $S \in \mathcal{S}$ be the string of Lemma 6. Additionally, let $O \in \mathcal{O}$ be the list of occurrences of S in T^i as described in Lemma 5. If all the suffixes of T^i in O are preceded by the same symbol $s \in \Sigma^i$ (i.e., S is not left-maximal), then $pBWT^i[o] = (s, l)$ is an equal-symbol run of length $l = |O|$, where o is the \prec_{LMS} order of S in \mathcal{S} .*

Proof. By Lemma 5, we know that the suffixes of T^i in O are prefixed by S , and that they form a consecutive range $SA^i[j, j']$. Additionally, the symbols that occur to the left of the suffixes in $SA^i[j, j']$ are those for the list of $pBWT^i[o]$. However, we still have not resolved the relative order of the suffixes in $SA^i[j, j']$, so (in theory) we do not know how rearrange the symbols in $pBWT^i[o]$. The suffixes of T^i in O are preceded by the same symbol s , so it is no necessary to further sort $SA^i[j, j']$ because the outcome for $pBWT^i[o]$ will be always an equal-symbol run for s of length $l = |O|$. \blacktriangleleft

The problem is that we do not store O , so we do not know value for l in (s, l) . Nevertheless, we do have the frequencies of the phrases in D^i , in the vector N^i . In this way, we can compute l by summing the frequencies in N^i for the phrases of D^i where S occurs as a suffix.

Now that we have covered all the theoretical aspects of the parsing phrase, we proceed to describe our procedure to build $pBWT^i$.

3.3.1 Constructing the Preliminary BWT for the Parsing Round

The computation of $pBWT^i$ starts with the construction of a *generalized* suffix array SA_{D^i} for D^i . We say SA_{D^i} is generalized because it only considers the suffixes of the dictionary phrases. If a string $S \in \mathcal{S}$ appears as a suffix in two or more phrases, those occurrences maintain in SA_{D^i} the relative order in which their enclosing phrases appear in D^i . In practice, the values we store in SA_{D^i} are the positions in R^i , the vector storing the concatenated phrases of D^i (see the encoding of D^i in Subsection 3.3).

We compute SA_{D^i} using a modified version of the ISS method mentioned in Subsection 2.3. The first difference is that, in step one, we insert in SA_{D^i} the position in R^i of the last symbol of each phrase. Put it another way, suppose $R^i[j]$, with $L^i[j+1] = 1$, is the last symbol of a phrase F , then we insert j in the right-most available cell in the bucket $R^i[j]$ of SA_{D^i} . The step one in the original ISS puts LMS-type symbols at the end of the buckets. In our case, the last symbol of a phrase is, by definition, LMS type in T^i , so the operation is homologous. The second difference of our ISS variation is that, during step two and three, we skip each position $SA_{D^i}[u]$ representing the start of a phrase ($L^i[SA_{D^i}[u]] = 1$) as they do not induce suffixes.

The next step is to scan SA_{D^i} from left to right to compute $pBWT^i$. From now on, we consider $pBWT^i$ to be a run-length compressed vector instead of a vector of lists. As we move throughout the suffix array, we search for every range $SA_{D^i}[j, j']$, with $j' - j + 1 \geq 1$, that encode suffixes with the same sequence². Nevertheless, we consider only the ranges that either represent suffixes of length > 1 or suffixes of length 1 that expand to suffixes of \mathcal{T} . Recall that the left-most symbol of an LMS substring is the same as the right-most symbol of the LMS substring that precedes it. Hence, considering all the suffixes in D^i will produce a redundant (and incorrect) BWT. The only exception to this rule are the LMS substrings at the beginning of the strings of \mathcal{T} as they do not share a symbol with the LMS substring to their left. This kind of substrings only appear when we cross from T_{u+1} to T_u in T^i , with $T_u, T_{u+1} \in \mathcal{T}$. We can detect this situation using B^i , the bit vector marking the symbols in Σ^i that expand to suffixes of strings in \mathcal{T} (see Section 3.3).

We define the length of $SA_{D^i}[j, j']$ as $l = \sum_{u=j}^{j'} N^i[\text{rank}(L^i, SA_{D^i}[u])]$. This value is the sum of the frequencies of the phrases where the suffixes in $SA_{D^i}[j, j']$ occur.

If all the suffixes of $SA_{D^i}[j, j']$ are followed by the same symbol $s \in \Sigma^i$, we append (s, l) to $pBWT^i$ (see Lemma 8). Otherwise we append $(*, l)$. The symbol $*$ represents an empty entry and it is out of Σ^i . We will resolve $(*, l)$ in the next phase of grlBWT (see Lemmas 6 and 7). After scanning SA_{D^i} , we store $pBWT^i$ into disk and discard SA_{D^i} .

3.3.2 Grammar Compression and Next Parsing Round

Once we finish constructing $pBWT^i$, the next step in the parsing round i is to store D^i in a compact form to use it later during the induction phase of grlBWT. We first explain why we need D^i during the induction phase and then describe the format we choose to encode it.

Broadly speaking, the induction process consists of scanning BWT^{i+1} from left to right, mapping every symbol $BWT^{i+1}[j] \in \Sigma^{i+1}$ back to the phrase $F \in D^i$ from which it originated, and then checking which of the proper suffixes of F produced empty entries in $pBWT^i$ (see Lemmas 6 and 7). Assume the suffix $F[u..] = S \in \mathcal{S}$ produced an empty entry, then we append $F[u-1]$ in the BWT range associated with S (see Lemma 5).

² In practice, we compute every distinct range $SA_{D^i}[j, j']$ during the construction of the suffix array. We reserve the least significant bit in the cells of SA_{D^i} to mark every position $SA_{D^i}[j]$. We flag these positions during the execution of our modified version of ISS (Section 2.3).

The process described above requires D^i and a mechanism to map the left-maximal suffixes in D^i back to the empty entries they produce in $pBWT^i$. We solve the problem by encoding D^i with a representation that is similar to grammar compression (Section 2.2).

We start by discarding N^i and the rank_1 data structure, as they are no longer necessary (see the current encoding of D^i in Section 3.3). For our method to work, we also need each string $S \in \mathcal{S}$ associated with an empty entry $(*, l)$ of $pBWT^i$ to be a member of D^i . This property might not hold when the string S that produced an empty entry meets Lemma 6. The problem arises if S always appears as a proper suffix in D^i , not as a full phrase. If that is the case, we *create*³ a new independent entry for S in D^i .

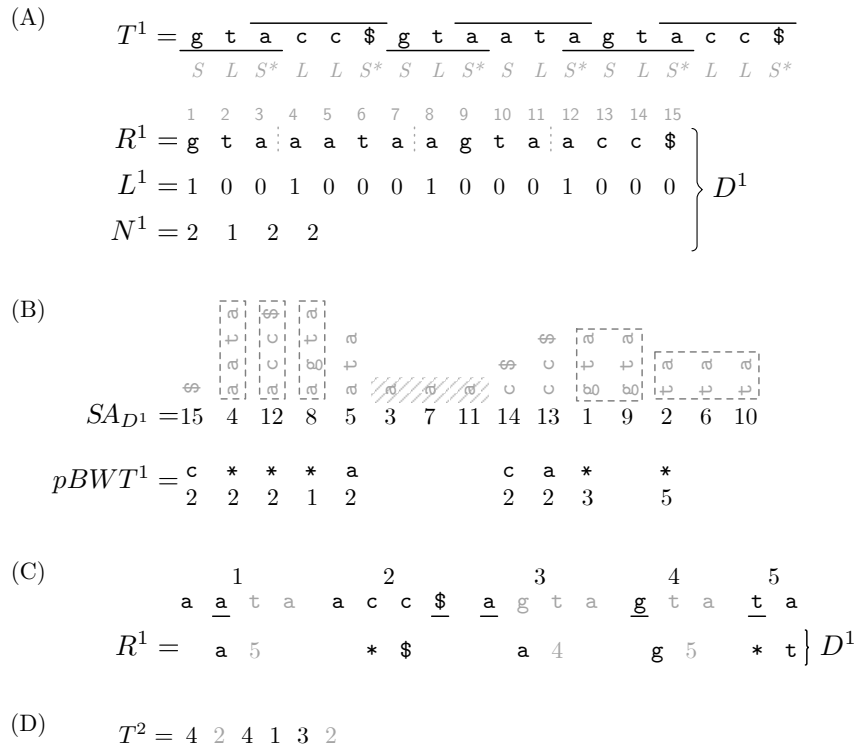
After expanding D^i , we create a hash table M^i in which we insert each phrase $F \in D^i$ occurring as a left-maximal suffix. If F has \prec_{LMS} rank b in D^i , then we insert the pair (F, b) into M^i , where F is the key and b is the value. Once we construct M^i , we reorder the way in which the phrases of D^i are concatenated in R^i according to their \prec_{LMS} ranks.

The next step consists of *compressing* D^i . We scan R^i from left to right, and for every $F = R^i[j, j'] \in D^i$, with $L^i[j] = 1$ and $L^i[j' + 1] = 1$, we search for the longest proper suffix $F[u..]$ that exists in M^i as a key. If such key exists, then we replace F with $R[j, j + 1] = F[u - 1] \cdot b'$, where b' is the value associated with $F[u..]$ in M^i . If no proper suffix of F exists in M^i , then we replace F as $R[j, j + 1] = * \cdot F[|F| - 1]$, where $*$ is a dummy symbol. After updating the sequence of F , we mark the symbols in $R[j + 2, j']$ as discarded if $j' - j + 1 = |F| \geq 2$. When we finish the scan of the dictionary, we left-contract R^i by removing the discarded symbols. This process reduces the phrases in D^i to strings of length two, so the vector L^i is no longer necessary.

Now we explain the rationale of our encoding. We develop our argument as a chain of implications. Consider again the phrase F , which we replaced with the sequence $F[u - 1] \cdot b'$. We obtained $b' \in \Sigma^{i+1}$ when we performed a lookup operation of $S = F[u..]$ in M^i during the compression of D^i . The value b' that the lookup returned is the \prec_{LMS} order of S in D^i . The membership of S to the keys of M^i implies that S appears as a left-maximal suffix in D^i , which in turn implies that S is a full phrase in D^i too (we enforced this property when we expanded the dictionary). Additionally, the left-maximal condition of S implies that there were at least two suffix occurrences of S preceded by different symbols. This is why S produces an empty entry in $pBWT^i$. Now, recall that we sorted the phrases of D^i in \prec_{LMS} order in R^i . Therefore, if we want to access S , we have to go to the substring $R^i[2b' - 1, 2b']$. This substring does not encode the full sequence of S , but its longest left-maximal suffix $R^i[2b] \in \Sigma^{i+1}$ (which is also a left-maximal suffix of F) along with the left-context symbol for that suffix ($R^i[2b - 1] \in \Sigma^i$). Recursively, the longest left-maximal symbol of S is not a sequence either, but a pointer to another position of R^i . We access this nested left-maximal suffix by setting $b' = R^i[2b']$ and updating the values $R^i[2b' - 1, 2b']$. We continue applying this idea until we reach a range $R^i[2b' - 1, 2b']$ where $R^i[2b'] \in \Sigma^i$, which implies that we reached the last suffix of F . This last range will store the sequence $* \cdot F[|F| - 1]$. Notice that the right symbol in this case is not the last symbol of F but its left context $F[|F| - 1]$. This is because the LMS substrings overlap by one character in T^i , so $F[|F|]$ is redundant as it also appears as a prefix in another phrase. However, we need $F[|F| - 1]$, as we will append it to one of the empty entries of $pBWT^i$ in the next phase of grlBWT . The only exception to this rule is when F expands to a suffix of a string in \mathcal{T} . In that case, we store $F[|F|]$ instead of $F[|F| - 1]$ in $R[2b' - 1, 2b']$ as $F[|F|] = R^i[2b']$ is not a prefix in any other phrase. On the

³ It means we append the sequence of F at the end of R^i and expand L^i accordingly.

29:10 Efficient Construction of the BWT Using String Compression



■ **Figure 1** Parsing round $i = 1$ of grlBWT for the collection $\mathcal{T} = \{\text{gtacc}, \text{gtaatagtacc}\}$, with $T = \text{gtacc}\$ \text{gtaatagtacc}\$$. (A) Construction of the dictionary $D^1 = \{R^1, L^1, N^1\}$ from $T^1 = T$. The sequence in gray below T^1 stores the symbol types: L-type is L , S-type is S and LMS-type is S^* . The dashed vertical lines in R^1 mark the boundaries between dictionary phrases. (B) Constructing $pBWT^1$ from the suffix array SA_{D^1} of D^1 . The ranges of SA_{D^1} enclosed by dashed boxes produce empty entries in $pBWT^i$. Notice we do not use the entries $SA_{D^1}[5, 7] = 3, 7, 11$ for computing $pBWT^1$ as their corresponding symbols are redundant. For instance, consider $R^1[3] = a$, which is a suffix in $R^1[1, 3] = \text{gta}$. The last symbol in the occurrences of gta in T^1 overlaps the first symbol of $\text{acc}\$$ or aata . Therefore, the symbol $R^1[3] = a$ of gta is covered by $\text{acc}\$$ or aata in $pBWT^1$. We represent $pBWT^1$ as a sequence of equal-symbol runs. The upper row depicts the run symbols while the lower row show the run lengths. (C) Compressing D^1 . The strings at the top are the dictionary phrases sorted in \prec_{LMS} order. We add the string ta to the dictionary as it appears as a left-maximal suffix in D^i , and hence, produces an empty entry in $pBWT^1$. The sequences in gray are the longest left-maximal suffixes of the phrases. The underlined symbols are the left contexts of those suffixes. Notice we compress $F = \text{acc}\$$ directly to $*\$$ as F does not have a left-maximal suffix. Besides, as F does not overlap other phrases in T^1 (because of $\$$), we store $F[|F|] = \$$ instead of $F[|F| - 1] = c$. A different situation occurs with $S = \text{ta}$. This phrase does not have left-maximal suffixes of length > 1 . However, in this case, we store $S[|S| - 1] = t$ instead of $S[|S|] = a$ as S overlaps other phrases in T^1 . (D) The parse T^2 we obtain by replacing the phrases of T^1 with their \prec_{LMS} orders in D^i . The gray symbols expand to suffixes of strings in \mathcal{T} .

other hand, we need the dummy symbol $*$ to maintain the invariant that all the phrases encoded in R^i have length two. Once we finish the compression, we store R^i on disk. From now on, we use D^i to refer to R^i .

The final step for the parsing round i is to create the new text T^{i+1} . We first reload from disk the hash table H^i we produced at the beginning of the iteration, and replace its values with the keys' \prec_{LMS} orders. More specifically, if a key $F \in D^i$ of H^i has \prec_{LMS} order

b among the other strings that produced empty entries in $pBWT^i$, then we update the value of F in H^i to b . Notice that the strings we stored as keys in H^i are not the same as those we have now in D^i because we compress them. Therefore, we can not lookup the phrases of D^i in the keys of H^i to update the hash table values. Still, we can overcome this problem if we modify H^i after sorting D^i in \prec_{LMS} order but before we compress it.

Once we update H^i , we construct T^{i+1} by scanning T^i again and replacing the LMS substrings with their associated values in H^i . If T^{i+1} has length k (the number of strings in \mathcal{T}), then we stop the parsing phase as all the strings in \mathcal{T} are now compressed to one symbol. An example of the parsing step is depicted in Figure 1.

3.4 The Induction Phase

The induction phase starts with the computation of BWT^h , the BCR BWT for the text T^h of the last parsing round h . This step is trivial as each symbol in T^h encodes a full string of \mathcal{T} (see the ending condition of the parsing phase). Hence, the left context of every symbol is the symbol itself. BCR BWT maintains the relative order of the strings in \mathcal{T} (see Section 2.1), so BWT^h is T^h itself.

We now describe the steps we perform during every induction step $i < h$. In this case, assume we receive as input (i) BWT^{i+1} from the previous phase, (ii) D^i , and (iii) $pBWT^i$. Before explaining our procedure, we describe some important properties of BWT^{i+1} .

► **Lemma 9.** *Let $BWT^{i+1}[j]$ and $BWT^{i+1}[j']$ be two symbols at different positions j and j' , with $j < j'$, whose mapping phrases in D^i are F and F' , respectively. Also, let the proper suffixes $F[u..] = F'[u'..] = S \in \mathcal{S}$ (see Lemma 5 for the description of \mathcal{S}) be two occurrences in T^i of a string S that appears as a left-maximal suffix in D^i . The suffix of T^i prefixed by $F[u..]$ precedes in SA^i (the suffix array of T^i) the suffix of T^i prefixed by $F'[u'..]$.*

Proof. As $F[u..]$ and $F'[u'..]$ are equal, their relative orders are decided by the right contexts in T^i of the occurrences $BWT^{i+1}[j]$ and $BWT^{i+1}[j']$ of F and F' (respectively). By induction, we know that BWT^{i+1} is complete, and as $BWT^{i+1}[j]$ precedes $BWT^{i+1}[j']$ in the BWT, the right context of $F[u..]$ has a smaller order in SA^i than the right context of $F'[u'..]$. ◀

If we generalize Lemma 9 to $x \geq 1$ occurrences of S , then we can use the following lemma to compute the sequence for the empty entry of $pBWT^i$ generated by S :

► **Lemma 10.** *Let S be a string of \mathcal{S} . Additionally, let $J = \{j_1, j_2, \dots, j_x\}$ be a list of strictly increasing positions of BWT^{i+1} . Every $BWT^{i+1}[j_o]$, with $j_o \in J$, is a symbol $b \in \Sigma^{i+1}$ generated from a phrase $F \in D^i$ where $S = F[u..]$ occurs as a proper suffix. The symbols of B^{i+1} referenced by J are not necessarily equal, and hence, their associated phrases in D^i are not necessarily the same. However, these phrases of D^i are all suffixed by S . Assume we scan J from left to right, and for every j_o , we extract the symbol $F[u-1] \in \Sigma^i$ that precedes S and append it to a list L_S . The resulting list $L_S \in \Sigma^{i*}$ has the same sequence of symbols as the BWT^i range that maps the block for S in the partition of \mathcal{S} .*

Proof. Because of Lemma 9, the suffix of T^i prefixed by the occurrence $BWT^{i+1}[j_o]$ of S precedes the suffix of T^i prefixed by the occurrence $BWT^{i+1}[j_{o+1}]$. This property holds for every j_o , with $o \in [1, x-1]$. Hence, the suffixes of T^i prefixed by S are already sorted J . ◀

Our compressed representation for D^i (see Section 3.3.2) has precisely the information we need to construct L_S as described in the procedure of Lemma 10. Still, the idea only works when S always appears as a proper suffix in the phrases of D^i . When S matches a full phrase

(see Lemma 7), there is no left-context symbol for S we can extract from D^i . Nevertheless, there is only one phrase $F \in D^i$ where S can be a non-proper suffix, and because S comes from BWT^{i+1} , F has to be an LMS substring in T^i . This observation implies that F maps to a symbol $b \in \Sigma^{i+1}$ in T^{i+1} . Hence, we can extract the left-context symbols of S from the range in BWT^{i+1} that corresponds to the b th bucket of SA^{i+1} . We explain how to carry out this process in the next subsection.

3.5 The Induction Algorithm

Let p be the sum of the lengths in the empty entries of $pBWT^i$. These lengths correspond to the second field in the run-length representation of $pBWT^i$. We start the induction by creating a vector $P^i[1, p]$, which we logically divide into σ^{i+1} buckets (recall that σ^{i+1} matches the number of empty entries in $pBWT^i$). Every bucket b of P^i will be of size l_b , the length of the b th empty entry (from left to right) of $pBWT^i$. Subsequently, we perform a scan over BWT^{i+1} from left to right. For each symbol $BWT^{i+1}[j] = b \in \sigma^{i+1}$, we first check if its associated LMS substring $F \in D^i$ (the string from which we obtain the symbol b during the parsing round i) exists as a suffix in other phrases of D^i . This information is already encoded in a bit vector $V^i[1, \sigma^{i+1}]$ we constructed during the parsing round i . When F occurs as an LMS substring and as a proper suffix in other dictionary phrases ($V^i[b] = 1$), we append a dummy symbol in the bucket b of P . This is the situation we described at the end of the previous subsection. After processing b , we decompress the left-maximal suffixes of its phrase F from the compressed representation of D^i .

The decompression of F begins by accessing the range $D^i[2b - 1, 2b]$ (see Section 3.3.2). If $o = D^i[2b]$ belongs to Σ^{i+1} , then the symbol o encodes a string $F[u..] = S$, with $u > 1$, whose sequence is a left-maximal suffix in D^i . During the parsing phase of `grlBWT`, we inserted S to D^i as an independent string as it yields an empty entry for $pBWT^i$ (see Lemma 6). The order of S in D^i is precisely o , its \prec_{LMS} rank among the other phrases of D^i . On the other hand, the left-context symbol of S is $D^i[2b - 1] \in \Sigma^i$. With this information, we apply Lemma 10 by appending the symbol $D^i[2b - 1]$ to the bucket o of P . Then, we move to the next left-maximal suffix of F by setting $b = o$ and updating the range $D^i[2b - 1, 2b]$.

The decompression of F stops when $D^i[2s]$ belongs to Σ^i , which means we reach the last symbol of F . For the moment, we do not know for which phrase of D^i $D^i[2s]$ is its left context. Hence, we set $BWT^{i+1}[j] = D^i[2s]$ and leave this position on hold to process it later. After finishing the scan of BWT^{i+1} , its symbols are now over the alphabet Σ^i . These values are the ones we have to insert in the dummy positions of P^i . Notice that the entries in P^i and BWT^{i+1} are already sorted by their right contexts in T^i . Hence, the completion of the dummy positions reduces to a merge of two sorted lists.

The last step in the induction round i consists of merging $pBWT^i$, BWT^{i+1} and P^i in BWT^i . We scan $pBWT^i$ and we append its entries to BWT^i as long as they are not empty. Then, when we reach an empty entry $(*, l)$, we proceed as follows: assume the current pair $(*, l)$ is the b th empty entry of $pBWT^i$. Then, we check if the phrase $F \in D^i$ that produced this entry (the one with \prec_{LMS} order b) only occurs as a full LMS substring in T^i ($V^i[b] = 0$). If that is the case, we append the next l symbols of BWT^{i+1} into BWT^i . On the other hand, when F occurs as an LMS substring, but also as a proper suffix in other phrases of D^i ($V^i[b] = 1$), the next l symbols of BWT^i are a mix of entries from the bucket b of P^i and BWT^{i+1} . We append symbols from the bucket b of P^i as long as they are not dummy. When we reach a dummy symbol in P^i , we change the list, and append the next x symbols of BWT^{i+1} into BWT^i , where x is the number of consecutive dummy symbols we saw in P^i . Once we process the x entries of BWT^{i+1} , we go back to P and continue back and forth between P and BWT^i until we process all the symbols in the bucket b of P .

The last case we have to cover for the merge is when F always occurs as a proper suffix in D^i (i.e., it is not an LMS substring of T^i). This situation is simple as we marked F in V^i ($V^i[b] = 1$). Hence, we just copy the content of the bucket b of P into BWT^i . Notice this bucket will not have dummy entries as b does not appear in B^{i+1} as a symbol. We obtain occurrences of b while decompressing other phrases of D^i whose \prec_{LMS} ranks do appear as symbols in B^{i+1} , and where F is a proper left-maximal suffix. Once we complete all the induction rounds, the final BCR BWT is in BWT^i .

3.5.1 Speeding up the Induction with Compression

If we run-length encode BWT^{i+1} , the induction becomes more efficient. Every position $BWT^{i+1}[j]$ is not a symbol b , but a pair (l, b) that represents l consecutive copies of b . Thus, instead of decompressing l times the left-maximal suffixes of the phrase associated with b , we decompress them only once, and copy the result l times to the different buckets of P^i .

Maintaining P^i as a run-length encoded sequence also improves efficiency. A compact representation of P^i reduces the working memory, which in turn reduces the number of cache misses. The only problem with this idea is that we do not know before the induction how many equal-symbols runs P^i will have. There are two solutions to this problem. First, we could represent P^i as a dynamic vector [2]. Its initial size would be σ^{i+1} , and then we expand the buckets as we insert new equal-symbol runs into them. The second option is to perform a preliminary scan of BWT^{i+1} to compute the size of the run-length compressed version of P^i , then we scan BWT^{i+1} again to perform the induction.

3.6 Complexity of our Method

We show that the construction of the BWT remains linear, even though we perform compression during the intermediate steps.

► **Theorem 11.** *Let $\mathcal{T} = \{T_1, T_2, \dots, T_k\}$ be a collection with k strings and n symbols. The algorithm `grlBWT` constructs the BCR BWT of \mathcal{T} in expected $O(n)$ time and requires $O(n)$ bits of working space.*

Proof. The complexities in the theorem were already proved for the linear-time algorithms that construct the suffix array [30] and the BWT [34] using ISS. We show that these complexities are not altered by our compression scheme. Let $n^i = |T^i|$ be the length of the input text we receive at parsing round i . Hashing the dictionary phrases from T^i runs in $O(n^i)$ expected time and requires $O(n^i)$ bits of space. The construction of SA_{D^i} runs in $O(n^i)$ time and space as we use ISS to build it, and the number of symbols in D^i is never greater than n^i . The extra steps of the parsing round only require a constant number of linear scans over SA_{D^i} . During the induction phase, we only perform linear scans over BWT^i and $pBWT^i$. We still have the cost of accessing the left-maximal suffixes of D^i when we scan BWT^{i+1} during the induction phase. However, our simple compressed representation for D^i (Section 3.3.2) supports random access in $O(1)$ time to the symbols, and the number of left-maximal suffixes we visit during the scan of BWT^{i+1} is no more than n^i . In every parsing round, the size of T^{i+1} is at most half the size of T^i , so the sum of the text lengths n^1, n^2, \dots, n^h is $O(n)$ (see Nong et al. [30]). This property also implies that `grlBWT` never visits more than $O(n)$ left-maximal suffixes during its induction phase. ◀

■ **Table 1** Datasets. The upper rows are the Illumina reads while the lower rows are the human genomes. Columns four and five are the minimum and average string length (respectively) in the collection. The value for r is the number of equal-symbol runs in the BCR BWT of the collection.

Dataset	σ	Number of strings	Max. length	Avg. length	n	n/r
ILL1	5	84,006,956	151	151	12,769,057,312	3.18
ILL2	5	160,285,798	151	151	24,363,441,296	4.07
ILL3	5	235,805,550	151	151	35,842,443,600	4.67
ILL4	5	305,931,740	151	151	46,501,624,480	5.03
ILL5	5	377,453,488	151	151	57,372,930,176	5.33
HGA05	16	334,065	248,956,422	42,715	14,269,998,434	4.82
HGA10	16	759,341	250,522,664	39,025	29,634,170,092	8.76
HGA15	16	835,485	250,522,664	53,918	45,048,695,199	12.02
HGA20	16	874,235	250,522,664	68,650	60,017,146,889	15.67
HGA25	16	899,424	250,522,664	83,447	75,055,723,570	19.42

4 Experiments

We implemented `grlBWT` as a C++ tool, also called `grlBWT`. This software uses the `SDSL-lite` library [11] to operate with bit vectors and rank data structures. Our source code is available at <https://github.com/ddiazdom/grlBWT>. We compared the performance of `grlBWT` against other tools that compute BWTs for string collections:

- `ropebwt2`⁴: a variation of the original BCR algorithm of Bauer et al. [1] that uses rope data structures [2]. This method is described in Heng Lee [20].
- `pfp-eBWT`⁵: the eBWT algorithm of Boucher et al. [4] that builds on PFP and ISS.
- `BCR_LCP_GSA`⁶: the current implementation of the semi-external BCR algorithm [1].
- `egap`⁷: a semi-external algorithm of Edigi et al. [8] that builds the BCR BWT.
- `gsufsort`⁸: an in-memory method proposed by Louza et al. [23] that computes the BCR BWT and (optionally) other data structures.

We also considered the tool `bwt-lcp-em` [3] for the experiments. Still, by default it builds both the BWT and the LCP array, and there is no option to turn off the LCP array, so we discarded it. We compiled all the tools according to their authors' description. For `grlBWT`, we used the compiler flags `-O3 -msse4.2 -funroll-loops`.

We considered two common types of genomic data for the experiments: short reads and assembled genomes. We downloaded five collections of Illumina reads produced from different human genomes⁹. We concatenated the strings so that our dataset 1 had one read collection, dataset 2 had two collections, and so on. We named the files `ILLX`, where `X` is the number of read collections concatenated. We also downloaded from NCBI¹⁰ 25 collections of fully-assembled human genomes. Like with the reads, we created the inputs

⁴ <https://github.com/lh3/ropebwt2>

⁵ <https://github.com/davidecenzato/PFP-eBWT>

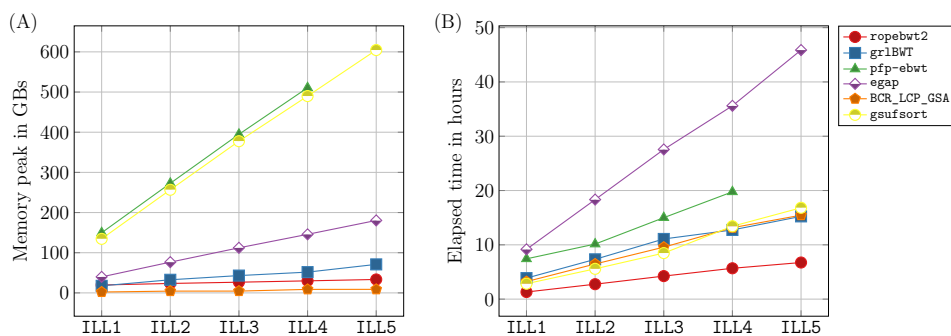
⁶ https://github.com/giovannarosone/BCR_LCP_GSA

⁷ <https://github.com/felipelouza/egap>

⁸ <https://github.com/felipelouza/gsuftsort>

⁹ <https://www.internationalgenome.org/data-portal/data-collection/hgdp>

¹⁰ <https://www.ncbi.nlm.nih.gov/assembly>



■ **Figure 2** Memory peak usage (GBs) and elapsed time (in hours) for the Illumina reads.

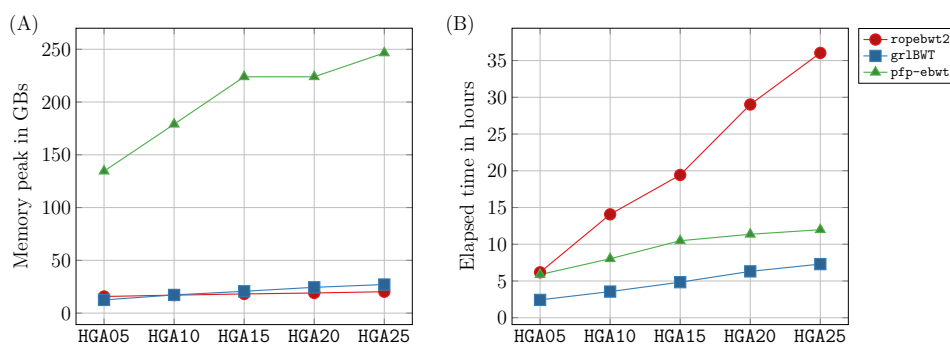
for our experiments so that every dataset has five more genomes than the previous one. This setup aims to increase the repetitiveness as the collection size increases. We named each file using the prefix `HGA` concatenated with the number of genomes it had. The only preprocessing step we performed on the genomes was to put every chromosome in one line and set all the characters to upper case. All our inputs are described in Table 1.

We also investigated the effect of page cache [24, Ch. 16] in `gr1BWT`. In every parsing round i , we keep T^i on disk, and linearly scan its file by loading from disk to RAM one data chunk of 8 MB at the time. Similarly, we keep a buffer of 8 MB in RAM for T^{i+1} , which we dump into disk every time it gets full. We manipulate BWT^i (respectively, BWT^{i+1}) in the same way. We used the function `posix_fadvise` to turn off the page cache for T^i , T^{i+1} , BWT^i , and BWT^{i+1} . Then we assessed the performance of `gr1BWT` on the assembled genomes using `posix_advice` and not using it. We did not evaluate the effect of the page cache in the other tools.

We limited the RAM usage of `egap` to three times the input size. For `BCR_LCP_GSA`, we turned off the construction of the data structures other than the BCR BWT and left the memory parameters by default. In the case of `gsufsort`, we used the flag `-bwt` to build only the BWT. For `ropebwt2`, we set the flag `-L` to indicate that the data was in one-sequence-per-line format, and the flag `-R` to avoid considering the DNA reverse strands in the BWT. We ran the experiments on the Illumina reads using one thread in all programs as not all support multi-threading. For this purpose, we set the extra flag `-P` to `ropebwt2` to indicate single-thread execution. We tested the human genomes only on `ropebwt2`, `gr1BWT` and `pfp-ebwt`. By default, `ropebwt2` uses four working threads, so we set the same number of threads for `gr1BWT` and `pfp-ebwt`. We did not report results for `pfp-ebwt` with dataset ILL25 as the execution crashed. We carried out the experiments on a machine with Debian 4.9, 736 GB of RAM, and processor Intel(R) Xeon(R) Silver @ 2.10GHz, with 32 cores.

5 Results and Discussion

We summarize our experiments in Figures 2 and 3. The results we report for `gr1BWT` do not consider the use of `posix_advice` to turn off the page cache. In Illumina reads, the fastest method was `ropeBWT2`, with a mean elapsed time of 4.14 hours. It is then followed by `BCR_LCP_GSA`, `gsufsort`, `gr1BWT`, `pfp-bwt`, and `egap`, with mean elapsed times of 9.58, 9.43, 10.05, 13.08, and 27.30 hours, respectively (Figure 2B). We notice that `gr1BWT` is competitive with `BCR_LCP_GSA` and `gsufsort`. However, it gets slightly faster than them from input ILL4 onward. We expected this behaviour since the largest datasets are more repetitive.



■ **Figure 3** Memory peak usage (GBs) and elapsed time (in hours) for the assembled genomes.

Regarding the working space, the most efficient was BCR_LCP_GSA, with an average memory peak of 5.73 GB. It is then followed by `ropebwt2`, with an average memory peak of 26.64 GB. In both cases, the memory consumption increases slowly with the input size. In the case of `gr1BWT`, the memory peak is more considerable; 42.20 GB on average, with a memory consumption that grows faster than the previous methods (see Figure 2A). However, `egap`, `gsufsort`, and `pfp-ebwt` are far more expensive, and their memory consumption grow even faster. The tool `egap` uses 110.94 GBs on average. On the other hand, `pfp-ebwt` and `gsufsort` have similar average memory peaks: 331.98 and 372.68 GBs, respectively.

In the repetitive datasets (human genomes), the results changed drastically (see Figure 3). Our tool `gr1BWT` outperformed `ropebwt2` and `pfp-ebwt` in elapsed time, with an average of 4.89 hours versus 20.95 and 9.55 hours of `ropebwt2` and `pfp-ebwt`, respectively. As expected, the time for `gr1BWT` grows smoothly with the input size, while the time for `ropebwt2` grows fast. The time function for `pfp-ebwt` also grows smoothly, but the results are still slower than those of `gr1BWT` (see Figure 3B). Regarding memory peak, `ropebwt2` is the most efficient tool, with a mean of 18.05 GB. Still, `gr1BWT` obtained competitive results, with an average peak of 20.38 GB. In this case, the memory consumption growth in `gr1BWT` is slightly steeper than in `ropebwt2`, but it remains smooth. In contrast, `pfp-ebwt` has a more dramatic growth in memory consumption, with an average memory peak of 156.74 GB (Figure 3A).

We observe that `ropebwt2` and `pfp-ebwt` performed well in one measure, but not in both. In contrast, `gr1BWT` maintained a low footprint for both measures, elapsed time and memory consumption. This result demonstrates that our strategy of keeping the intermediate data of the BWT algorithm in compressed format works well when the text is repetitive.

Our experiments on the page cache showed there is an average slowdown of 19% in `gr1BWT` when the cache is disabled with the function `posix_advice`. This slowdown factor increases with the input size, being the lowest with HGA05 (12%) and the highest with HGA20 (26%). This result is expected as we are only using a static buffer of 8 MB. A simple solution would be to set a dynamic buffer that uses, say, 0.5% of the input instead of the fixed 8 MB.

6 Concluding Remarks

We introduced a method for building the BCR BWT that maintains the data of intermediate stages in compressed form. The representation we chose not just reduces space usage, but also reduces computation time. Our experimental results showed that our algorithm is competitive with the state-of-the-art tools under not so repetitive scenarios, and that it greatly reduces the computational requirements when the input becomes more repetitive, standing out as the most efficient tool to date (and to our knowledge) in this context.

References

- 1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.
- 2 Hans-J Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
- 3 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the multi-string BWT and LCP array in external memory. *Theoretical Computer Science*, 862:42–58, 2021.
- 4 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the original eBWT faster, simpler, and with less memory. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 129–142, 2021.
- 5 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, 2019. Article 13.
- 6 Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 7 Diego Díaz-Domínguez and Gonzalo Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. 31st Data Compression Conference (DCC)*, pages 83–92, 2021.
- 8 Lavinia Egidi, Felipe A. Louza, Giovanni Manzini, and Guilherme P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14:6, 2019.
- 9 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.
- 10 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.
- 11 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.
- 12 Richard Karp and Michael Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- 13 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1344–1357, 2019.
- 14 Dominik Kempa and Tomasz Kociumaka. String Synchronizing Sets: Sublinear-Time BWT Construction and Optimal LCE Data Structure. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767, 2019.
- 15 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In *Proc. 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013. IEEE, 2020.
- 16 John C. Kieffer and En Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.
- 17 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.
- 18 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3), 2009. Article R25.
- 19 Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

- 20 Heng Li. Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.
- 21 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.
- 22 Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, 678(1):22–39, 2017.
- 23 Felipe A. Louza, Guilherme P. Telles, Simon Gog, Nicola Prezza, and Giovanna Rosone. gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections. *Algorithms for Molecular Biology*, 15, 2020. Article 18.
- 24 Robert Love. *Linux kernel development*. Pearson Education, 2010.
- 25 Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, and Alexandru Tomescu. *Genome-Scale Algorithm Design*. Camb. U. Press, 2015.
- 26 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 27 Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*, 54(2), 2021. Article 29.
- 28 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39:article 2, 2007.
- 29 Ge Nong. Practical linear-time $O(1)$ -workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):1–15, 2013.
- 30 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. 19th Data Compression Conference (DCC)*, pages 193–202, 2009.
- 31 Daniel Saad Nogueira Nunes, Felipe A. Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. 28th Data Compression Conference (DCC)*, pages 42–51, 2018.
- 32 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
- 33 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70. SIAM, 2007.
- 34 Daisuke Okanohara and Kunihiko Sadakane. A linear-time Burrows–Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009.
- 35 Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: astronomical or genetical? *PLoS Biology*, 13(7):e1002195, 2015.