

31st Annual Symposium on Combinatorial Pattern Matching

CPM 2020, June 17–19, 2020, Copenhagen, Denmark

Edited by

Inge Li Gørtz

Oren Weimann



Editors

Inge Li Gørtz 

Technical University of Denmark, DTU Compute, Lyngby, Denmark
inge@dtu.dk

Oren Weimann 

University of Haifa, Israel
oren@cs.haifa.ac.il

ACM Classification 2012

Theory of computation → Design and analysis of algorithms; Theory of computation → Pattern matching;
Mathematics of computing → Discrete mathematics; Mathematics of computing → Information theory;
Mathematics of computing → Combinatoric problems; Information systems → Information retrieval;
Applied computing → Computational biology

ISBN 978-3-95977-149-8

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/978-3-95977-149-8>.

Publication date

June, 2020

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 3.0 Unported license (CC-BY 3.0):
<https://creativecommons.org/licenses/by/3.0/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.2020.0

ISBN 978-3-95977-149-8

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*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

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

■ Contents

Preface	
<i>Inge Li Gørtz and Oren Weimann</i>	0:ix
Program Committee	
.....	0:xi
Reviewers	
.....	0:xiii
Authors	
.....	0:xv–0:xvi

Invited Talk

Algebraic Algorithms for Finding Patterns in Graphs	
<i>Thore Husfeldt</i>	1:1–1:1

Regular Papers

Finding the Anticover of a String	
<i>Mai Alzamel, Alessio Conte, Shuhei Denzumi, Roberto Grossi, Costas S. Iliopoulos, Kazuhiro Kurita, and Kunihiko Wasa</i>	2:1–2:11
Double String Tandem Repeats	
<i>Amihood Amir, Ayelet Butman, Gad M. Landau, Shoshana Marcus, and Dina Sokol</i>	3:1–3:13
Efficient Tree-Structured Categorical Retrieval	
<i>Djamal Belazzougui and Gregory Kucherov</i>	4:1–4:11
Time-Space Tradeoffs for Finding a Long Common Substring	
<i>Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus</i>	5:1–5:14
On Two Measures of Distance Between Fully-Labelled Trees	
<i>Giulia Bernardini, Paola Bonizzoni, and Paweł Gawrychowski</i>	6:1–6:16
String Sanitization Under Edit Distance	
<i>Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering</i>	7:1–7:14
Counting Distinct Patterns in Internal Dictionary Matching	
<i>Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszypiński, Tomasz Waleń, and Wiktor Zuba</i>	8:1–8:15
Dynamic String Alignment	
<i>Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes</i>	9:1–9:13
Unary Words Have the Smallest Levenshtein k -Neighbourhoods	
<i>Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Tomasz Waleń, and Wiktor Zuba</i>	10:1–10:12

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann



Leibniz International Proceedings in Informatics

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

Summarizing Diverging String Sequences, with Applications to Chain-Letter Petitions <i>Patty Commins, David Liben-Nowell, Tina Liu, and Kiran Tomlinson</i>	11:1–11:15
Detecting k -(Sub-)Cadences and Equidistant Subsequence Occurrences <i>Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, and Ayumi Shinohara</i>	12:1–12:11
FM-Index Reveals the Reverse Suffix Array <i>Arnab Ganguly, Daniel Gibney, Sahar Hooshmand, M. Oğuzhan Külekci, and Sharma V. Thankachan</i>	13:1–13:14
On Indeterminate Strings Matching <i>Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau</i>	14:1–14:14
The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time <i>Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat</i>	15:1–15:15
Approximating Longest Common Substring with k mismatches: Theory and Practice <i>Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya</i>	16:1–16:15
String Factorizations Under Various Collision Constraints <i>Niels Grüttemeier, Christian Komusiewicz, Nils Morawietz, and Frank Sommer</i> ..	17:1–17:14
k -Approximate Quasiperiodicity under Hamming and Edit Distance <i>Aleksander Kędzierski and Jakub Radoszewski</i>	18:1–18:15
Longest Common Subsequence on Weighted Sequences <i>Evangelos Kipouridis and Kostas Tsichlas</i>	19:1–19:15
Parameterized Algorithms for Matrix Completion with Radius Constraints <i>Tomohiro Koana, Vincent Froese, and Rolf Niedermeier</i>	20:1–20:14
In-Place Bijective Burrows-Wheeler Transforms <i>Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara</i> ...	21:1–21:15
Genomic Problems Involving Copy Number Profiles: Complexity and Algorithms <i>Manuel Lafond, Binhai Zhu, and Peng Zou</i>	22:1–22:15
Compressed Orthogonal Search on Suffix Arrays with Applications to Range LCP <i>Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, and Masakazu Tateshita</i>	23:1–23:13
Text Indexing and Searching in Sublinear Time <i>J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich</i>	24:1–24:15
Chaining with Overlaps Revisited <i>Veli Mäkinen and Kristoffer Sahlin</i>	25:1–25:12
DAWGs for Parameterized Matching: Online Construction and Related Indexing Structures <i>Katsuhito Nakashima, Noriki Fujisato, Diptarama Hendrian, Yuto Nakashima, Ryo Yoshinaka, Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, and Masayuki Takeda</i>	26:1–26:14

On Extensions of Maximal Repeats in Compressed Strings <i>Julian Pape-Lange</i>	27:1–27:13
Faster Binary Mean Computation Under Dynamic Time Warping <i>Nathan Schaar, Vincent Froese, and Rolf Niedermeier</i>	28:1–28:13
Approximating Text-To-Pattern Distance via Dimensionality Reduction <i>Przemysław Uznański</i>	29:1–29:11

■ Preface

The Annual Symposium on Combinatorial Pattern Matching (CPM) is the international research forum in the areas of combinatorial pattern matching, string algorithms and related applications. The studied objects include strings as well as trees, regular expressions, graphs, and point sets, and the goal is to design efficient algorithms and data structures based on their properties, in order to design efficient algorithmic solutions for the addressed computational problems. The problems this conference deals with include those in bioinformatics and computational biology, coding and data compression, combinatorics on words, data mining, information retrieval, natural language processing, pattern matching and discovery, string algorithms, string processing in databases, symbolic computation, and text searching and indexing.

This volume contains the papers presented at the 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020) held on June 17-19. The conference was planned to be held in Copenhagen, but due to the Covid-19 pandemic the conference was instead held online using Zoom. The conference program includes 28 contributed papers and three invited talks by Barna Saha (University of California Berkeley), Karl Bringmann (Max-Planck-Institut für Informatik), and Thore Husfeldt (IT University of Copenhagen and Lund University). For the second time, CPM includes the “Highlights of CPM” special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this second edition we have invited Shay Golan to present his SODA 2020 paper “Locally consistent parsing for text indexing in small space”, Tomasz Kociumaka to present his STOC 2019 paper “String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure”, and Paweł Gawrychowski (University of Wrocław, Poland) to present his paper “Computing quartet distance is equivalent to counting 4-Cycles”.

The contributed papers were selected out of 49 submissions, corresponding to an acceptance ratio of about 57%. Each submission received at least three reviews. We thank the members of the Program Committee and all the additional external subreviewers that 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, and Pisa. 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), and 128 (CPM 2019). 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 in this year’s unusual circumstances.



■ Programme Committee

Inge Li Gørtz (co-chair)
Technical University of , Denmark

Oren Weimann (co-chair)
University of Haifa, Israel

Amir Abboud
IBM, USA

Amihood Amir
Bar-Ilan University and Johns Hopkins
University, Israel

Hideo Bannai
Kyushu University, Japan

Philip Bille
Technical University of Denmark, Denmark

Panagiotis Charalampopoulos
King's College London, UK

Debarati Das
University of Copenhagen, Denmark

Rolf Fagerberg
University of Southern Denmark, Denmark

Martin Farach-Colton
Rutgers University, USA

Gabriele Fici
Università di Palermo, Italy

Johannes Fischer
TU Dortmund, Germany

Elazar Goldenberg
The Academic College Of Tel Aviv-Yaffo,
Israel

Roberto Grossi
Università di Pisa, Italy

Danny Hermelin
Ben Gurion University of the Negev, Israel

Artur Jez
University of Wrocław, Poland

Dominik Kempa
University of California Berkeley, USA

Juha Kärkkäinen
University of Helsinki, Finland

Dominik Köppl
Kyushu University / JSPS, Japan

Avivit Levy
Shenkar College, Israel

Zsuzsanna Lipták
University of Verona, Italy

Giovanni Manzini
University of Eastern Piedmont, Italy

Shay Mozes
Interdisciplinary Center (IDC) Herzliya,
Israel

Kunsoo Park
Seoul National University, South Korea

Solon Pissis
Centrum Wiskunde & Informatica (CWI),
Netherlands

Nicola Prezza
University of Pisa, Italy

Jakub Radoszewski
University of Warsaw, Poland

Wojciech Rytter
University of Warsaw, Poland

Sharma Thankachan
University of Central Florida, USA

Przemyslaw Uznanski
University of Wrocław, Poland

Tomasz Waleń
University of Warsaw, Poland



■ Reviewers

Aditi Dudeja
Aleksander Łukasiewicz
Alexandru Popa
Anish Mukherjee
Arnab Ganguly
Arseny Shur
Arturs Backurs
Bojian Xu
Brian Brubach
Daniel Gibney
Dekel Tsur
Diptarka Chakraborty
Eitan Konratovsky
Evangelos Kipouridis
Garance Gourdel
Giovanna Rosone
Giovanni Pighizzini
Giulia Bernardini
Golnaz Badkobeh
Itai Boneh
Jonas Ellert
Julian Pape-Lange
Juliusz Straszyński
Kazuhiro Kurita
Manuel Lafond
Manuel Sorge
Marinella Sciortino
Mateusz Pawlik
Meirav Zehavi
Michał Gańczorz
Michelle Sweering
Pawel Gawrychowski
Ray Li
Rayan Chikhi
Ritu Kundu
Shay Golan
Shuhei Denzumi
Shunsuke Inenaga
Stéphane Vialette
Szymon Grabowski
Thierry Lecroq
Tomasz Kociumaka
Tomohiro I
Travis Gagie
Wiktor Zuba
Wing-Kai Hon
Yakov Nekrich
Yuto Nakashima

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann



Leibniz International Proceedings in Informatics

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

■ List of Authors

Aleksander Kedzierski
Alessio Conte
Amihood Amir
Arnab Ganguly
Ayelet Butman
Ayumi Shinohara
Binhai Zhu
Christian Komusiewicz
Daiki Hashimoto
Daniel Gibney
David Liben-Nowell
Dina Sokol
Diptarama Hendrian
Djamal Belazzougui
Dominik Köppl
Ely Porat
Evangelos Kipouridis
Frank Sommer
Gad M. Landau
Garance Gourdel
Giulia Bernardini
Gonzalo Navarro
Gregory Kucherov
Grigorios Loukides
Hideo Bannai
Huiqing Chen
Ian Munro
Jakub Radoszewski
Julian Pape-Lange
Juliusz Straszynski
Katsuhito Nakashima
Kiran Tomlinson
Kostas Tsichlas
Kristoffer Sahlin
Kunihiko Sadakane
Leen Stougie
M. Oguzhan Kulekci
Manal Mohamed
Manuel Lafond
Masayuki Takeda
Matan Kraus
Michelle Sweering
Mitsuru Funakoshi
Nadia Pisanti
Niels Grüttemeier
Nils Morawietz
Noriki Fujisato
Panagiotis Charalampopoulos
Paola Bonizzoni
Patty Commins
Pawel Gawrychowski
Peng Zou
Przemyslaw Uznański
Rolf Niedermeier
Ryo Yoshinaka
Sahar Hooshmand
Samah Ghazawi
Sharma V. Thankachan
Shay Golan
Shay Mozes
Shoshana Marcus
Shunsuke Inenaga

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann



Leibniz International Proceedings in Informatics

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

0:xvi **Authors**

Solon Pissis

Stav Ben Nun

Tatiana Starikovskaya

Tina Liu

Tomasz Kociumaka

Tomasz Waleń

Tomohiro Koana

Tsvi Kopelowitz

Veli Mäkinen

Vincent Froese

Wiktor Zuba

Wojciech Rytter

Yakov Nekrich

Yuto Nakashima

Algebraic Algorithms for Finding Patterns in Graphs

Thore Husfeldt 

IT University of Copenhagen, Denmark

Lund University, Sweden

thore@itu.dk

Abstract

I will give a gentle introduction to algebraic graph algorithms by showing how to determine if a given graph contains a simple path of length k . This is a famous problem admitting a beautiful and widely-known algorithm, namely the colour-coding method of Alon, Yuster and Zwick (1995). Starting from this entirely combinatorial approach, I will carefully develop an algebraic perspective on the same problem. First, I will explain how the colour-coding algorithm can be understood as the evaluation of a well-known expression (sometimes called the “walk-sum” of the graph) in a commutative algebra called the zeon algebra. From there, I will introduce the exterior algebra and present the algebraic framework recently developed with Brand and Dell (2018).

The presentation is aimed at a combinatorially-minded audience largely innocent of abstract algebra.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Mathematics of computing → Paths and connectivity problems; Mathematics of computing → Graph algorithms

Keywords and phrases paths, exterior algebra, wedge product, color-coding, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.1

Category Invited Talk

Funding *Thore Husfeldt*: Supported by the Swedish Research Council grant VR-2016-03855 “Algebraic Graph Algorithms” and the Villum Foundation grant 16582 “Basic Algorithms Research Copenhagen (BARC)”.

Acknowledgements Based on joint work with Cornelius Brand and Holger Dell.

References

- 1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.
- 2 Cornelius Brand, Holger Dell, and Thore Husfeldt. Extensor-coding. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*, page 151–164, New York, NY, USA, 2018. Association for Computing Machinery.



© Thore Husfeldt;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



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

Finding the Anticover of a String

Mai Alzamel 

Department of Informatics,
King's College London, UK
Department of Computer Science,
King Saud University, KSA
mai.alzamel@kcl.ac.uk

Shuhei Denzumi

Graduate School of Information Science and
Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Costas S. Iliopoulos

Department of Informatics,
King's College London, UK
costas.ilopoulos@kcl.ac.uk

Kunihiro Wasa


National Institute of Informatics, Tokyo, Japan
wasa@nii.ac.jp

Alessio Conte 

Dipartimento di Informatica,
Università di Pisa, Italy
conte@di.unipi.it

Roberto Grossi

Dipartimento di Informatica,
Università di Pisa, Italy
grossi@di.unipi.it

Kazuhiro Kurita 

IST, Hokkaido University, Sapporo, Japan
k-kurita@ist.hokudai.ac.jp

Abstract

A k -*anticover* of a string x is a set of pairwise distinct factors of x of equal length k , such that every symbol of x is contained into an occurrence of at least one of those factors. The existence of a k -anticover can be seen as a notion of non-redundancy, which has application in computational biology, where they are associated with various non-regulatory mechanisms. In this paper we address the complexity of the problem of finding a k -anticover of a string x if it exists, showing that the decision problem is NP-complete on general strings for $k \geq 3$. We also show that the problem admits a polynomial-time solution for $k = 2$. For unbounded k , we provide an exact exponential algorithm to find a k -anticover of a string of length n (or determine that none exists), which runs in $O^*(\min\{3^{\frac{n-k}{3}}, (\frac{k(k+1)}{2})^{\frac{n}{k+1}}\})$ time using polynomial space.

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

Keywords and phrases Anticover, String algorithms, Stringology, NP-complete

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.2

1 Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [16], [17]), because it is of paramount importance in several applications, not to talk about its theoretical aspects. Algorithms and data structures for finding repeating patterns or regularities in strings (see [9], [12]) are central to several fields of computer science including computational biology, pattern matching, data compression, and randomness testing. The nature and extent of periodicity in strings is also of immense combinatorial interest in its own right [17].

The notion of cover belongs to the area of quasiperiodicity, that is, a generalization of periodicity in which the occurrences of the period may overlap [3]. We call a proper factor u of a nonempty string y a *cover* of y , if every letter of y is within some occurrence of u in y . A cover u of y needs to be a border (i.e. a prefix and a suffix) of y . A cover of a string s is a string that covers all positions of s with its occurrences. Intuitively, s can be generated by overlapping/concatenating copies of its cover u . Covers in strings were already



© Mai Alzamel, Alessio Conte, Shuhei Denzumi, Roberto Grossi, Costas S. Iliopoulos, Kazuhiro Kurita, and Kunihiro Wasa;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 2; pp. 2:1–2:11

Leibniz International Proceedings in Informatics



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

extensively studied. A linear-time algorithm finding the shortest cover of a string was given by Apostolico et al. [4] and later on improved into an on-line algorithm by Breslauer [20]. A linear-time algorithm computing all the covers of a string was proposed by Moore and Smyth [19]. Afterwards an on-line algorithm for the all-covers problem was given by Li and Smyth [15]. Similar combinatorial covering problems have been studied on graphs [8, 21], and other types of quasiperiodicities include seeds [13], as well as variants including approximate and partial covers and seeds.

A power of order k is defined by a concatenation of k identical blocks of symbols, where k is at least 2: it is evident how a covers are generalizations of powers. Powers in various forms later came to be important structures in computational biology, where they are associated with various regulatory mechanisms and play an important role in genomic fingerprinting (for further reading see, e.g., [14] and references therein). Antipowers are an orthogonal notion to that of powers, that were introduced recently by Fici et al. in [6, 10]. In contrast to powers, antipowers insist instead on the diversity of consecutive blocks: an *antipower* (*antiperiod*) of order k is a concatenation of k pairwise distinct strings of equal length. A linear algorithm for computing the antiperiods was given in [1], and online algorithms are given in [2].

We define an *anticover* as a generalization of the notion of antipower: an anticover of a string x is a set of pairwise distinct factors of x of equal length, such that every symbol of x is within some occurrence of one of those factors. Equivalently, x can be generated by overlapping/concatenating a set of pairwise distinct strings of equal length. Some practical motivation for this problem can be found in Mincu and Popa [18], that considers the similar problem of *partitioning* a string into distinct factors: they show that this problem is motivated by an application in the DNA compositions, a short DNA fragment can be obtained that can be self-united into the desired DNA structure. They present that to produce the wanted DNA structure, it is mandatory that no two fragments are equal.

In this paper we show that the computation of a 2-anticover of a string x of length n over an alphabet Σ , if it exists, can be done in $O(n|\Sigma|)$ time and space. For the general case, given $k \geq 3$, we show that checking whether a string x has a k -anticover is NP-complete. Moreover, we provide an exact exponential algorithm to find a k -anticover of x (or determine that none exists), which runs in $O^*(\min\{3^{\frac{n-k}{3}}, (\frac{k(k-1)}{2})^{\frac{n}{k+1}}\})$ time using polynomial space.

In the literature Condon et al. [7] studied the complexity of partitioning problems for strings. In particular, they introduced the Equality-Free String Partition problem, which requires to partition a string x into factors $f_1 f_2 \dots f_\ell$, each factor f_i of length *at most* k , so that factors are pairwise different $f_i \neq f_j$ for $i \neq j$. Among the results, they proved that this problem is NP-complete for $k = 2$ and unbounded alphabet. We observe that our notion of k -anticover requires the factors to be of length *exactly* k , and thus the problem of finding an anticover is different from Equality-Free String Partition problem. First, checking if a partition of factors of length k is equality-free can be trivially done in nearly linear time. Second, there are strings that admit a solution for one of the two problems, but not the other (e.g., *ababa* for $k = 3$ admits the equality-free partition $ab \cdot a \cdot ba$, but not an anticover).

2 Preliminaries

Let Σ be a finite ordered alphabet. A *string* is defined as a sequence of zero or more symbols from Σ . An *empty string* is a string of length 0, denoted by ε . A string x of length n is represented by the sequence $x = x_1 x_2 \dots x_n$. We use the notation $x[i \dots j]$ as a shorthand for $x_i x_{i+1} \dots x_j$ and call it a *factor* or *substring* of x with length $j - i + 1$. We also say that a nonempty string s is a factor or substring of x with length $k \leq n$ if $s = x[i \dots i + k - 1]$ for an integer $i \in [1, n - k + 1]$; in that case, s occurs in x at position i . The factor $x[1 \dots j]$ is a *prefix* of x and the factor $x[j \dots n]$ is a *suffix* of x .

► **Definition 1.** Given an integer $k \geq 2$ and a string x of length $n \geq k$, let $S = \{i_1, i_2, \dots, i_\ell\}$ be an ordered set of positions in x chosen from $\{1, 2, \dots, n - k + 1\}$. We say that S is a k -anticover of x if

- (i) any two factors $x[i_j \dots i_j + k - 1]$ and $x[i_h \dots i_h + k - 1]$ are different, for $j \neq h$, and
- (ii) any position in x is covered, namely, $i_1 = 1$, $i_\ell = n - k + 1$, and $i_{j+1} - i_j \leq k$ for $1 \leq j \leq n - k$.

► **Example 2.** For $x = \text{abbbaaaaabab}$ and $k = 3$, the ordered set $S = \{1, 3, 5, 9, 11\}$ denotes a 3-anticover of x : abbbaaaaabab. We remark that the indices $i_1 = 1$ and $i_\ell = n - k + 1$ must be part of any k -anticover, as they represent the only ways of covering the first and last symbol of x .

In this paper we consider the following problem.

k -ANTICOVER

Input: A string $x = x_1x_2 \dots x_n$ and an integer $k \geq 2$, where $n \geq k$.

Output: Does a k -anticover S of x exist?

It is obvious that any k -length substring that only occurs *once* in x can be included “for free” in any k -anticover without risk of redundancy. We call *free factors* the corresponding factors, and remark that we can consider trivially covered the symbols that they span.

In the following, we identify $i_j \in S$ with its factor $x[i_j \dots i_j + k - 1]$, and sometimes we say that $x[i_j \dots i_j + k - 1]$ belongs to an anticover S , actually meaning that $i_j \in S$.

3 Hardness of k -Anticover for $k \geq 3$

In this section we show that solving k -ANTICOVER, namely, deciding whether a string x of length n has a k -anticover, is NP-complete for $k \geq 3$.

Firstly, observe that we can easily test in polynomial time whether S is a k -anticover, by checking that each pair of corresponding factors is distinct and, for each position $p \in \{1, \dots, n\}$, that S contains a factor that covers x (i.e., some $i_j \in \{p - k + 1, \dots, p\}$); thus k -ANTICOVER \in NP.

We prove its completeness for $k = 3$ by a polynomial time reduction from 3-SAT to 3-ANTICOVER, i.e., given a 3-CNF boolean formula \mathcal{F} , we build a string \mathcal{X} (in polynomial time) that admits a 3-anticover if and only if \mathcal{F} is satisfiable. For $k > 3$, we remark that the techniques utilized could be adapted to reduce a k -SAT instance to k -ANTICOVER, although we omit this analysis for space reasons.

More in detail, we focus on a peculiar variant of 3-SAT, still NP-complete, where each literal in \mathcal{F} is restricted to appear *at most* 3 times. This variant has been addressed in [22, Theorem 2.1], where it is shown that SAT “is NP-complete when restricted to instances with 2 or 3 variables per clause and at most 3 occurrences per variable”. Hence 3-SAT with at most 3 occurrences per variable is NP-complete.¹

In the following, let C_1, \dots, C_l be the clauses of \mathcal{F} , and v_1, \dots, v_m its variables. For a variable v_i , we refer to the 3 occurrences of its positive literal as v_i^1, v_i^2 , and v_i^3 , and the ones of its negative literal as $\neg v_i^1, \neg v_i^2$, and $\neg v_i^3$, meaning that each v_i^j (and each $\neg v_i^j$) appears at most once in \mathcal{F} .

¹ For the sake of completeness, we observe that 3-SAT with *exactly* 3 occurrences per variable is always satisfiable as consequence of [22, Theorem 2.4].

3.1 Overview of the reduction

We here introduce the structure and components of the reduction, which we then explain in detail.

The string \mathcal{X} is divided in two parts:

- The first one models the clauses of \mathcal{F} , where literals correspond to specific factors, and using a factor in the cover of \mathcal{X} means using the corresponding literal in \mathcal{F} . In essence, each clause contains three factors corresponding to the occurrences of its literals (where the three different occurrences of the same literal will correspond to different factors), and in order to cover all elements of a clause gadget we will need to use at least one of such factors.
- The second one contains *coherence gadgets*, which in essence enforce us to use factors in a way coherent with truth assignments (i.e., if factors i and j in the first part correspond to v_h and $\neg v_h$, then i and j cannot be used at the same time in the cover). Say we want to cover a clause using the factor corresponding to v_1^2 : the coherence gadgets will force us to use the strings corresponding to $\neg v_1^1$, $\neg v_1^2$, and $\neg v_1^3$ to cover the *second* part of the string, meaning they cannot be used anymore in the first one (or they would break the non-redundancy constraint of the anticover).

We present \mathcal{X} as a collection of smaller strings corresponding to gadgets, delimited by what we call *jolly characters*: these allow us to essentially ignore the order in which the pieces of the strings are re-combined and prevent any interaction between adjacent gadgets.

Jolly characters. To simplify the explanation, we use the jolly character “ \star ”: in essence, each single \star represents a unique character that does not appear anywhere else in the string, i.e., we can imagine that at the end of the reduction each \star is then iteratively replaced with a unique distinct character not appearing in the string.

Jolly characters give us 2 useful properties for $k = 3$:

- All factors including \star are free factors, so the $k - 1$ symbols preceding and succeeding a \star are trivially covered by free factors.
- In the string $A\star B$, then the k -length factors of A and B that are not free do not overlap: if \star occurs at $\mathcal{X}[i]$, the right-most factor of A and left-most of B that could be non-free are, respectively, at positions $i - k + 1$ and $i + 1$.
- As a corollary, $A\star B$ and $B\star A$ have the same answer to k -ANTICOVER. More in general, if we have a collection of strings of the form $\star A\star$ (starting and ending in \star), and we want to append them to create a single string (\mathcal{X}), the order we chose does not impact the answer of k -ANTICOVER on the string.

3.2 The clauses part

We now detail the clause gadget of \mathcal{X} . Firstly, let p_j^i and n_j^i be symbols in Σ representing, respectively, the literals v_i^j and $\neg v_i^j$.

Let C_h be an arbitrary clause of \mathcal{F} , say, $(\neg v_1^3 \vee v_5^1 \vee v_7^1)$, corresponding to the third occurrence of the negative literal of v_1 , and the first occurrences of the positive literals of v_5 and v_7 . Then the corresponding gadget is

$$C_h = \star \# \# h h n_1^3 \star \# \# h h p_5^1 \star \# \# h h p_7^1 \star$$

where

- n_1^3 , p_5^1 , and p_7^1 are the characters representing the corresponding occurrences of the literals, as described above.
- h is a character representing C_h (i.e., is different in every clause), whereas the character $\#$ is the same across all clauses, and \star are jolly characters.

Now observe that the only characters we need to cover are the first h of each pair hh : Indeed the 2 symbols following and preceding each \star can be trivially covered by free factors, as shown in the example below:

$$\star\#\#h\overline{hn_1^3}\star\#\#h\overline{hp_5^1}\star\#\#h\overline{hp_7^1}\star$$

As $k = 3$, we have 3 possible factors we can use to cover each of these h symbols; in particular, the first with $\{\#\#h, \#hh, hhn_1^3\}$, the second with $\{\#\#h, \#hh, hhp_5^1\}$, and the third with $\{\#\#h, \#hh, hhp_7^1\}$.

As h is clause-specific, the two strings $\#\#h$ and $\#hh$ in each set only appear here, and no constraint is imposed on their usage. However, as the anticover can contain each string at most once, it will have to include at least one string among hhn_1^3 , hhp_5^1 , and hhp_7^1 .

In essence, adding this occurrence of hhp_7^1 to the anticover will correspond to assigning “true” to v_7 .

The first part of \mathcal{X} will thus correspond to the gadgets corresponding to all clauses C_1, \dots, C_l appended after each other in sequence (as discussed above, the order is irrelevant).

In the second part of \mathcal{X} , we will then enforce coherence of assignments, i.e., using hhp_7^1 to cover C_h must forbid us from using the factors corresponding to the literal $\neg v_7$ in the rest of the clause part.

3.3 Auxiliary gadgets

In order to explain the coherence part, we first detail the auxiliary gadgets that compose it.

Gadget forbid(abc). Suppose we want to make sure that some string of length 3, say, abc , cannot be used to cover rest of the string. Then we can place the following gadget in \mathcal{X} :

$$\text{forbid}(abc) = \star\$abc\$ \star \$abc\$ \star \$abc\$ \star$$

where again the \star are jolly characters, but $\$$ is a gadget-specific character, i.e., each occurrence of a forbidding gadget has a unique character in place of the $\$$.

Similarly to above, we can observe that all characters are covered by free factors except the b characters in the middle, which can be covered by the strings $\$ab$, abc , $bc\$$: as we need to cover 3 characters, we must use all three of these strings. In turn, this means the string abc can *not* be used anywhere else in \mathcal{X} . We refer to this gadget as $\text{forbid}(abc)$.

An important observation is that all factors used except for abc are either free factors, or contain the character $\$$, meaning those strings will not appear anywhere else and thus not affect our choices while covering the rest of the string.

Gadget one-of(abc, def). Suppose now we have two strings abc and def , and we want to make sure that at most one of the two may be used in the cover of the rest of the string. Then we can place the following gadget in \mathcal{X} :

$$\text{one-of}(abc, def) = \text{forbid}(c\epsilon d) \star bc\epsilon de \star abc\epsilon def \star$$

2:6 Finding the Anticover of a String

Here too, ϵ is a gadget-specific character which is used only in this specific instance of this gadget (same as the $\$$ above, we differentiate so as to avoid confusion with the $\text{forbid}(c\epsilon d)$ gadget).

Let's analyze it from left to right. Firstly, we forbid the string $c\epsilon d$ so it cannot be used in the rest of the string. Then, to cover the ϵ in the central part $\dots \star bc\epsilon de \star \dots$, we must use either $bc\epsilon$ or ϵde . Finally, in the right part $\dots \star abc\epsilon def \star$ we need to cover the three symbols $\dots c\epsilon d \dots$: if $bc\epsilon$ was used in the central part, we cannot use it now, and since we cannot use $c\epsilon d$ either, to cover the c symbol we must use abc (while we can use ϵde to cover the remaining two symbols). Symmetrically, if we used ϵde in the central part of the gadget instead, we must use def to cover the right part.

It follows that to cover $\text{one-of}(abc, def)$ we must use either abc or def , meaning we can only use one of them in the rest of the string.

As above, all other factors used are either free or include ϵ , so they do not impact the rest of the string.

Gadget $\text{amplifier}(abc, def)$. Finally, the amplifier gadget is the core of our coherence enforcement. It corresponds to the following string:

$$\text{amplifier}(abc, def) = \text{forbid}(cde) \star abcdef \star$$

Differently from the gadgets above, this one does affect strings other than the input ones: we call bcd the *trigger* of the amplifier. Specifically this is the word made from the last two characters of the first string abc , and the first of the second string def . Furthermore, note how the string cde made of the third character of the first string, and the first two of the second, becomes forbidden.²

Focus now on the right part: in $\dots \star abcdef \star$ only the symbols $\dots cd \dots$ are not covered by free factors. Since cde is forbidden, to cover them we have two ways:

- use the trigger string bcd .
- use both abc and def .

As the name suggests, this gadget *amplifies* the effects of using the trigger cde elsewhere in \mathcal{X} , as it will then force us to use both abc and def to cover $\text{amplifier}(abc, def)$. Note that this gadget does create and affect factors that are not free and may occur somewhere else, so we will analyze its usage carefully.

3.4 The coherence part

Let v_i be a variable of \mathcal{F} . In the clauses part, its literals can appear in up to six clauses. Let w.l.o.g. the symbols $1 \dots 6$ represent the identifiers of these clauses: the factors representing the literals will thus be $\{11p_i^1, 22p_i^2, 33p_i^3\}$ for the positive ones, and $\{44n_i^1, 55n_i^2, 66n_i^3\}$ for the negative.³

As explained above, say that the gadget of clause C_h contains an occurrence of the factor $22p_i^2$: we want to say that using this occurrence of the factor in the anticover means that v_i is set to true (thus C_h is satisfied by v_i). In order to enforce coherence, and make the anticover correspond to a satisfying assignment, we must then make it impossible to use a factor corresponding to a negative value of v_i anywhere in the clauses part.

² Note that $\text{amplifier}(def, abc)$ is a different gadget: it will forbid fab and its trigger will be efa .

³ If a literal, say p_i^j , does not appear in a clause of \mathcal{F} , let it be represented by $\star \star p_i^j$.

More formally, we want to create a gadget for a variable v_i which, to be covered, forces us to use either all of $\{11p_i^1, 22p_i^2, 33p_i^3\}$, or all of $\{44n_i^1, 55n_i^2, 66n_i^3\}$ (this way, one set of strings is fully “burned” to cover this gadget, and only elements from the other set may be used in the rest of the string).

Gadget enforce(v_i). We do so by using the **one-of** gadget and a nested use of the **amplifier** gadget, with the following gadget made of 5 parts, which we call **enforce(v_i)**:

1. **amplifier**($11p_i^1, 2p_i^23$)
2. **amplifier**($22p_i^2, 33p_i^3$)
3. **amplifier**($44n_i^1, 5n_i^26$)
4. **amplifier**($55n_i^2, 66n_i^3$)
5. **one-of**($1p_i^12, 4n_i^15$)

Now, key observations are that $2p_i^23$ in gadget 1 is the trigger of gadget 2, while $5n_i^26$ in gadget 3 is the trigger of gadget 4, and finally, the arguments of gadget 5 are the triggers of gadgets 1 and 3.

In order to cover **one-of**($1p_i^12, 4n_i^15$) we must use (at least) one between $1p_i^12$ and $4n_i^15$. If we choose $1p_i^12$ to cover gadget 5, this *triggers* gadget 1, so to cover gadget 1 we must use both $11p_i^1$ and $2p_i^23$; in turn, this triggers gadget 2, forcing us to use both $22p_i^2$ and $33p_i^3$; on the other hand, gadgets 3 and 4 can be covered using their respective triggers, meaning that *all* strings corresponding to positive literals $\{11p_i^1, 22p_i^2, 33p_i^3\}$ are used by the cover, but it is not necessary to use any of the negative ones $\{44n_i^1, 55n_i^2, 66n_i^3\}$, which can be used in the clauses part. If, instead, we cover gadget 5 using $4n_i^15$, the situation is exactly symmetrical: we burn all the negative literals $\{44n_i^1, 55n_i^2, 66n_i^3\}$ on gadgets 3 and 4, but we may cover 1 and 3 using the triggers, and using the positive literals $\{11p_i^1, 22p_i^2, 33p_i^3\}$ in the clauses part.⁴ We have thus proven the following result.⁵

► **Theorem 3.** \mathcal{F} is satisfiable if and only if \mathcal{X} has a 3-anticover. As a consequence, problem k -ANTICOVER is NP-complete for $k \geq 3$.

4 Polynomial-Time Algorithm for $k = 2$

In this section, we show that 2-ANTICOVER can be reduced to 2-SAT in $O(n|\Sigma|)$ time and space, obtaining the following result.

► **Theorem 4.** Problem 2-ANTICOVER can be solved in $O(n|\Sigma|)$ time and space.

Proof. We run first a preliminary test to see if the input string x contains a factor of length 3 that occurs three or more times in it. A 2-anticover cannot exist a 2-ANTICOVER in this case, and we answer no. Indeed, let abc be a factor that occurs three or more times in x . Since the position corresponding to b can be covered either by the factors of length 2, ab or bc , when we find the third occurrence of abc , we cannot use ab or bc as it would be chosen twice. Hence, there is no way to cover the position of b in the third occurrence of abc in any 2-ANTICOVER. Running this test can be easily done in $O(n \log |\Sigma|)$ time [12].

⁴ For completeness, we remark that it is crucial to use **amplifier**($11p_i^1, 2p_i^23$) instead of **amplifier**($2p_i^23, 11p_i^1$): the latter one forbids the string 311, which does not contain \star nor characters representing the literal and might affect other coherence gadgets. Instead **amplifier**($11p_i^1, 2p_i^23$) forbids $p_i^12p_i^2$, which is safe as it may not appear in other coherence gadgets.

⁵ Again, we remark that all the gadgets in this reduction can be extended to any k rather than just 3, although we omit this for space reasons.

Instead, if this preliminary test is positive, we build a 2-SAT formula as follows. Let i be any position in the string x , and p_i the Boolean variable denoting whether or not the factor $x[i \dots i + 1]$ is chosen in the 2-ANTICOVER.

We have a first group of clauses C_i , for $1 \leq i \leq n$, where C_i says that the first ($i = 1$) and last ($i = n$) positions must be covered by the only possible factors $x[1 \dots 2]$ and $x[n - 1 \dots n]$, respectively, and any other position must be covered by the factor(s) of length $k = 2$ starting at position $i - 1$ or i :

$$C_1 = (p_1) \tag{1}$$

$$C_n = (p_{n-1}) \tag{2}$$

$$C_i = (p_{i-1} \vee p_i) \quad 2 \leq i \leq n - 1 \tag{3}$$

Furthermore, when $x[i \dots i + 1] = x[j \dots j + 1]$ for $i \neq j$, we should take at most one of them, and so we cannot take both, giving the second group B_{ij} of clauses:

$$B_{ij} = (\neg p_i \vee \neg p_j) \quad 1 \leq i < j \leq n \text{ such that } x[i \dots i + 1] = x[j \dots j + 1] \tag{4}$$

Let \mathcal{F} be the 2-SAT formula obtained by putting the clauses C_i and B_{ij} in logical \wedge . We observe that \mathcal{F} contains n clauses C_i and $O(n|\Sigma|)$ clauses B_{ij} . Recall that each factor of length 3 can occur at most twice in x . Thus, given any position i , we claim that there are at most $2|\Sigma| + 1$ positions $j \neq i$ such that $x[i \dots i + 1] = x[j \dots j + 1]$. Indeed, any other occurrence of $s = x[i \dots i + 1]$ is followed by a third symbol, say, c (unless that occurrence is a suffix of x). But sc is a factor of length 3, and can appear at most twice. Since we have at most $|\Sigma|$ choices for c plus the end of string case, this gives the desired upper bound. As there are at most n positions i , and for each of them there are at most $2|\Sigma| + 1$ positions j where the same factor of length 2 occurs, we conclude that there are $O(n|\Sigma|)$ clauses B_{ij} . Summing up, \mathcal{F} has $O(n|\Sigma|)$ size and it can be built $O(n|\Sigma|)$ time.

It is straightforward to see that \mathcal{F} is satisfied if and only if there is a 2-ANTICOVER for string x . Since 2-SAT can be solved in linear time in the size of the formula \mathcal{F} [5], we obtain the bounds stated in the theorem. \blacktriangleleft

5 Exact Exponential-Time algorithms for $k \geq 3$

In this section we consider a better algorithm than a brute-force algorithm for solving k -ANTICOVER. The task of k -ANTICOVER is finding a subset of positions satisfying the given constraint. By trying all subsets of positions, we can solve k -ANTICOVER. Since the number of subset of positions is $O(2^{n-k})$, the brute-force algorithm runs in $O^*(2^{n-k})$ time, where the $O^*(\cdot)$ notation ignores $poly(n)$ factors. Note that $|\Sigma|$ and k is bounded by n . Thus, $O^*(\cdot)$ notation ignores $poly(|\Sigma|)$ and $poly(k)$ factors. In this section, we give two exponential time algorithms. The former algorithm breaks the trivial 2^{n-k} -barrier for any $k \geq 3$. The latter algorithm is clearly better than the brute-force algorithm and, in addition, it outperforms the former algorithm when $k > 9$.

5.1 Breaking the trivial barrier

Let x be a string with length n and k be an integer. We consider a set of positions of x from 1 to $n - k + 1$. We partition this set as follows: Let $\mathcal{S} = \{S_1, \dots, S_\ell\}$ be a partition of positions $1, 2, \dots, n - k + 1$. For any two substring y, y' with length k starting from position j and j' respectively, both j and j' are in S_i if and only if $y = y'$. That is, each partition corresponds to some substring with length k in x . For example, given a string $abcabca$ and $k = 3$, $\mathcal{S} = \{\{1, 4\}, \{2, 5\}, \{3\}\}$.

We describe our proposed algorithm. Let $S_1 \in \mathcal{S}$ be the set containing position 1. We first pick position 1 to cover the 1-st character on x . Hence, after choosing 1, we need to solve at most $|S_1|$ subproblems. Note that for each subproblem, the first k letters are already covered, and thus, we have k options for covering the $(k+1)$ th letter in each subproblem.

Since we cannot pick positions which already are picked, the time complexity $T(\cdot)$ of this algorithm satisfies the following inequality: $T(n-k+1) \leq cT(n-k+1-c)$, where c is the size of partition from which we pick a substring. Since the sum of the size of the partitions is $n-k$, the time complexity is $O^*(c^{\frac{n-k+1}{c}})$.

It is known that this formula takes its maximum when $c = 3$ [11]. Hence, the time complexity of this algorithm is $O^*(3^{\frac{n-k+1}{3}}) = O^*(3^{\frac{n-k}{3}})$ time.

► **Theorem 5.** k -ANTICOVER can be solved in $O^*(3^{\frac{n-k}{3}})$ time and polynomial space.

5.2 A better upper bound for large k

In this subsection, we give a faster algorithm when k is large. Now we first introduce some terminologies. A set $S = \{s_1, \dots, s_\ell\}$ is a k -cover if $\bigcup_{i=1, \dots, \ell} \{s_i, \dots, s_i+k\} = \{1, \dots, n\}$. Hence, a trivial k -cover is $\{1, 1+k, 1+2k, \dots\}$. Note that each s_i corresponds to a position of x but a k -cover may have two positions which derives from the same substring. A k -cover S is *minimal* if there is no subset of S which is a k -cover. We say that s_i is *redundant* in S if $S \setminus \{s_i\}$ is also a k -cover.

► **Lemma 6.** Let x be a string and k be an integer. Then, if x has a k -anticover, then there is a minimal k -cover S such that S is also a k -anticover.

Proof. Let $S = \{s_1, \dots, s_\ell\}$ be a k -anticover. Since S is a k -anticover, $S \setminus \{s_i\}$ is also k -anticover if $S \setminus \{s_i\}$ is a k -cover of x . Hence, S becomes a minimal cover by removing redundant elements one by one. Hence, the statement holds. ◀

From Lemma 6, we can determine whether x has a k -anticover by enumerating all minimal k -covers. Hence, in the following, we propose an enumeration algorithm for all minimal k -covers.

We firstly give an upper bound of the number of all minimal k -covers of substrings with length $k+1$ such that each minimal k -cover has no redundant positions. Assume that by concatenating these $\lceil n/(k+1) \rceil$ substrings, we can reconstruct the input string. Let us consider the following problem COVER(x, k): given a string x of length $k+1$, the task is to enumerate all minimal k -covers in it under the assumption, for $0 \leq i \leq k-1$, that we can select the length of a first interval s_1 between 1 to k and we can pick the last $k-1$ characters. For example, we consider an instance $x = abcd$ and $k = 3$. The subproblem COVER($x, 3$) has the following six solutions: \overline{abcd} , \overline{abcd} , \overline{abcd} , \overline{abcd} , \overline{abcd} , and \overline{abcd} . The next lemma shows the upper bound:

► **Lemma 7.** Problem COVER(x, k) has at most $\frac{k(k+1)}{2}$ minimal k -covers.

Proof. Let i be the length of a first interval. Since we cover all characters, we have to choose a position 1. In addition, we pick the second position between 2 and $i+1$. Since the length of x is $k+1$, then it is a minimal k -cover. Hence, we have i choices for each i . Therefore, we have $\sum_{1 \leq i \leq k} i = \frac{k(k+1)}{2}$ solutions and the statement holds. ◀

From the above lemma, the number of solutions in each subproblem is at most $\frac{k(k+1)}{2}$. In addition, the number of subproblems is $\frac{n}{k+1} + 1$. Now, we can obtain all minimal k -covers which have no redundant positions between $c(k+1)+1$ to $(c+1)(k+1)$ for any non-negative integer c by trying all the combinations of concatenating solutions of all the subproblems. Because any k -anticover has no redundant positions, the following theorem holds.

■ **Table 1** The list of values of $\left(\frac{k(k+1)}{2}\right)^{\frac{1}{k+1}}$. We round the base of the exponent up to the fourth digit after the decimal point. Note that $3^{\frac{1}{3}}$ is approximately equal to 1.4423.

k	3	5	9	10	20	30
$\left(\frac{k(k+1)}{2}\right)^{\frac{1}{k+1}}$	1.5651	1.5705	1.4633	1.4396	1.2900	1.2192

► **Theorem 8.** *There is an algorithm solving k -ANTICOVER in $O^*\left(\left(\frac{k(k+1)}{2}\right)^{\frac{n}{k+1}}\right)$ time and polynomial space.*

From Theorem 8 and Table 1, the latter algorithm is better than the former algorithm if k is larger than 9. Combining two theorems, we obtain the following theorem.

► **Theorem 9.** *Problem k -ANTICOVER can be solved in $O^*\left(\min\left\{3^{\frac{n-k}{3}}, \left(\frac{k(k+1)}{2}\right)^{\frac{n}{k+1}}\right\}\right)$ time, using polynomial space.*

6 Concluding remarks

In this paper we proposed the k -ANTICOVER problem, a natural combinatorial problem on strings with applications to fields such as computational biology. We have shown that finding whether a string of length n can be covered by (possibly overlapping) distinct factors of length k is polynomial for $k = 2$, and NP-complete otherwise.

We have also shown how to design exact exponential algorithms for general k , which improve upon a trivial brute-force approach and get progressively more efficient for larger values of k .

A question that remains open is whether the proposed algorithms match the inherent computational complexity of the problem or whether faster solutions exist. Another is whether the problem remains NP-complete under natural restrictions, such as an alphabet of constant size.

References

- 1 Hayam Alamro, Golnaz Badkobeh, Djamel Belazzougui, Costas S Iliopoulos, and Simon J Puglisi. Computing the antiperiod (s) of a string. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 2 Mai Alzamel, Alessio Conte, Daniele Greco, Veronica Guerrini, Costas Iliopoulos, Nadia Pisanti, Nicola Prezza, Giulia Punzi, and Giovanna Rosone. Online algorithms on antipowers and antiperiods. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 175–188, Cham, 2019. Springer International Publishing.
- 3 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.
- 4 Alberto Apostolico, Martin Farach, and Costas S Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.
- 5 Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979. doi:10.1016/0020-0190(79)90002-4.
- 6 Golnaz Badkobeh, Gabriele Fici, and Simon J Puglisi. Algorithms for anti-powers in strings. *Information Processing Letters*, 137:57–60, 2018.
- 7 Anne Condon, Ján Maňuch, and Chris Thachuk. The complexity of string partitioning. *J. Discrete Algorithms*, 32:24–43, 2015. doi:10.1016/j.jda.2014.11.002.

- 8 Alessio Conte, Roberto Grossi, and Andrea Marino. Large-scale clique cover of real-world networks. *Information and Computation*, 270:104464, 2020. Special Issue on 26th London Stringology Days & London Algorithmic Workshop (LSD & LAW). doi:10.1016/j.ic.2019.104464.
- 9 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.
- 10 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q Zamboni. Anti-powers in infinite words. *Journal of Combinatorial Theory, Series A*, 157:109–119, 2018.
- 11 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.
- 12 Dan Gusfield. Algorithms on strings, trees, and sequences. 1997. *Computer Science and Computational Biology. New York: Cambridge University Press*, 1997.
- 13 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*, pages 1095–1112. SIAM, 2012.
- 14 Roman Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic acids research*, 31(13):3672–3678, 2003.
- 15 Yin Li and William F Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
- 16 M Lothaire. *Applied combinatorics on words*, volume 105. Cambridge University Press, 2005.
- 17 Monsieur Lothaire and M Lothaire. *Algebraic combinatorics on words*, volume 90. Cambridge university press, 2002.
- 18 Radu Stefan Mincu and Alexandru Popa. The maximum equality-free string factorization problem: Gaps vs. no gaps. In Alexander Chatzigeorgiou, Riccardo Dondi, Herodotos Herodotou, Christos Kapoutsis, Yannis Manolopoulos, George A. Papadopoulos, and Florian Sikora, editors, *SOFSEM 2020: Theory and Practice of Computer Science*, pages 531–543, Cham, 2020. Springer International Publishing.
- 19 Dennis Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, page 511–515, USA, 1994. Society for Industrial and Applied Mathematics.
- 20 Dennis Moore and William F Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994.
- 21 Robert Z Norman and Michael O Rabin. An algorithm for a minimum cover of a graph. *Proceedings of the American Mathematical Society*, 10(2):315–319, 1959.
- 22 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85–89, 1984. doi:10.1016/0166-218X(84)90081-7.

Double String Tandem Repeats

Amihood Amir

Department of Computer Science, Bar Ilan University, Ramat-Gan, 52900, Israel
<http://u.cs.biu.ac.il/~amir/>
amir@esc.biu.ac.il

Ayelet Butman

Department of Computer Science,
Holon Institute of Technology, Golomb St 52, Holon, 5810201, Israel
ayeletb@hit.ac.il

Gad M. Landau

Department of Computer Science, University of Haifa, Haifa 31905, Israel
NYU Tandon School of Engineering,
New York University, Six MetroTech Center, Brooklyn, NY 11201, USA
<http://www.cs.haifa.ac.il/~landau/>
landau@univ.haifa.ac.il

Shoshana Marcus

Department of Mathematics and Computer Science, Kingsborough Community College of the City
University of New York, 2001 Oriental Boulevard, Brooklyn, NY 11235, USA
shoshana.marcus@kbcc.cuny.edu

Dina Sokol

Department of Computer and Information Science, Brooklyn College and The Graduate Center,
City University of New York, Brooklyn, NY, USA
<http://www.sci.brooklyn.cuny.edu/~sokol>
sokol@sci.brooklyn.cuny.edu

Abstract

A *tandem repeat* is an occurrence of two adjacent identical substrings. In this paper, we introduce the notion of a *double string*, which consists of two parallel strings, and we study the problem of locating all tandem repeats in a double string. The problem introduced here has applications beyond actual double strings, as we illustrate by solving two different problems with the algorithm of the double string tandem repeats problem. The first problem is that of finding all corner-sharing tandems in a 2-dimensional text, defined by Apostolico and Brimkov. The second problem is that of finding all scaled tandem repeats in a 1d text, where a scaled tandem repeat is defined as a string UU' such that U' is discrete scale of U . In addition to the algorithms for exact tandem repeats, we also present algorithms that solve the problem in the inexact sense, allowing up to k mismatches. We believe that this framework will open a new perspective for other problems in the future.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms;
Theory of computation → Pattern matching

Keywords and phrases double string, tandem repeat, 2-dimensional, scale

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.3

Funding *Amihood Amir*: Partially supported by Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF) and Israel Science Foundation Grant 1475-18.

Gad M. Landau: Partially supported by Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF) and Israel Science Foundation Grant 1475-18.

Dina Sokol: Partially supported by Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF).



© Amihood Amir, Ayelet Butman, Gad M. Landau, Shoshana Marcus, and Dina Sokol;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 3; pp. 3:1–3:13

Leibniz International Proceedings in Informatics



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

1 Introduction

A *tandem repeat*, or *square*, is a string which consists of two consecutive identical occurrences of a substring or *root*, e.g. *abab*. Finding all tandem repeats in a given string is a well-studied problem with many applications in diverse areas such as biological sequence analysis and data compression. A *maximal run* in a string S is a substring of S that is periodic and cannot be extended at all to the right or left, e.g. *ababa* is a maximal run in the string *abaababac*. A maximal run in a string represents contiguous tandem repeats, all with periods conjugates of each other, and as such, maximal runs have been used to succinctly encode all tandem repeats. For example, the maximal run *ababa* represents consecutive tandem repeats with roots *ab* and *ba*.

In this paper, we consider the problem of finding tandem repeats in input that consists of two parallel strings. We define a *double string*, and introduce the corresponding notions of tandem repeat and run in a double string. Double strings are ubiquitous in nature, as molecules such as DNA come in pairs.¹ Hence, the problem considered is interesting from both a theoretical and practical perspective. However, the strength of this paper's contribution lies in its applicability to unrelated variants of the tandem repeats problem. We show how the solution to the double string tandem repeats problem can be used to solve two different problems. The first is finding 2D corner-sharing tandems, and the second is finding all scaled tandem repeats. We are confident that more applications of double string pattern matching will be discovered in the future.

In Section 2 we present precise definitions and examples of tandem repeats and runs in a double string, and then prove upper and lower bounds on the number of occurrences of such runs. In Section 3 we present an $O(n \log n)$ time algorithm for locating all double string tandem repeats. We then extend this algorithm to deal with double string tandem repeats while allowing k mismatches. In Section 4 we provide a reduction of the 2-dimensional (2D) *corner-sharing tandem problem* to the double string tandem repeats problem. We thus obtain a more efficient algorithm for locating all corner-sharing tandems in a 2D text, both with and without mismatches. Finally, in Section 5 we solve the *scaled tandem repeats problem* by reducing it to a tandem repeats problem on double strings.

2 Definition and Characterization of Double String Tandem Repeats

We use $S[i]$ to denote the i th character of a string S , and $S[i \dots j]$ to denote the substring of S from $S[i]$ through $S[j]$.

► **Definition 1.** A double string of length n consists of two parallel sequences over a given alphabet, each of length n , indexed by $1 \dots n$. We call the two strings S_1 and S_2 .

► **Example 1.** A double string of length 5, with $S_1 = abca$ and $S_2 = cbbba$.

1	2	3	4	5
a	a	b	c	a
c	c	b	b	a

► **Definition 2.** A double string tandem repeat (*2-str TR*) is a substring of S_1 and a substring of S_2 that are identical and consecutive. As in one string, we call the repeating

¹ In DNA there are specific relationships between corresponding bases, while our definition of a double string does not imply any such relationship.

substring the root or period of the 2-str TR. Specifically, a 2-str TR with length $2p$ beginning at location i in S_1 , implies that the substring $S_1[i \dots i + p - 1]$ is identical to the substring $S_2[i + p \dots i + 2p - 1]$. A 2-str TR beginning at location j in S_2 implies that $S_2[j \dots j + p - 1]$ is identical to the substring $S_1[j + p \dots j + 2p - 1]$.

► **Example 2.** A double string tandem repeat with root abc begins at location 2 in S_1 .

```

1 2 3 4 5 6 7 8
a a b c a a b b
c c b b a b c d

```

For the remainder of the paper, we assume that the 2-str TR begins in S_1 ; all lemmas and algorithms apply with minor modifications to indices for those beginning in S_2 .

► **Definition 3.** A 2-str run (i, j, p) in a double string (S_1, S_2) of length n , $1 \leq i \leq j \leq n - 2p + 1$, $1 \leq p \leq n/2$, is a sequence of one or more 2-str TR's with period size p beginning at each location $i \leq \ell \leq j$ in S_1 . The run is said to be maximal if it cannot be extended to the left or right, i.e. both $(i - 1, j, p)$ and $(i, j + 1, p)$ are not 2-str runs.

► **Example 3.** A maximal run with period size 3 occurs at locations $1 \dots 8$ in S_1 and $4 \dots 11$ in S_2 . It can be represented by the triple $(1, 6, 3)$, since 1 is the start of the leftmost tandem, 6 is the start of the rightmost tandem, and 3 is the period size.

```

1 2 3 4 5 6 7 8 9 10 11
a b c a b x y z z z z
a a a a b c a b x y z

```

Although all of the consecutive 2-str TR's in a 2-str run have the same period size, the actual characters in the periods can be different for different tandems in the same run, as is evident in Example 3. Thus, transitivity in equality of location i with location $i - p$ and $i + p$, for period p , which holds trivially for a run in a string, does **not** hold for a 2-str run. Nevertheless, 2-str runs can still be used as an efficient encoding of consecutive 2-str TR's, and as we show in the next subsection, there cannot be too many of them.

2.1 The number of maximal 2-str runs in a double string

► **Lemma 4.** Two distinct maximal 2-str runs in a double string, with the same period size, cannot overlap within S_1 or S_2 .

Proof. Let p be the period size of two distinct maximal 2-str runs in a given double string, and let j be the rightmost location of the 2-str run that has the leftmost starting location. Due to the maximality, there must be a mismatch following the first run, thus $S_1[j + 1] \neq S_2[j + p + 1]$, and location $j + 1$ cannot be included in any 2-str run with period p due to the mismatch. Therefore, the second 2-str run must start to the right of location $j + 1$ in S_1 and hence cannot overlap. ◀

Note that in one string, two maximal runs with the same period size may overlap, as long as the overlap is shorter than the period size, for e.g. $abcabcxabcx$.

► **Lemma 5.** There can be $O(n \log n)$ maximal 2-str runs in a double string of length n .

Proof. For a given period p there are no more than n/p maximal 2-str runs since they cannot overlap by Lemma 4. Since p can be $1 \dots n/2$, this yields $\sum_{p=1}^{n/2} n/p$, a harmonic series which is bound by $O(n \log n)$. ◀

3:4 Double String Tandem Repeats

► **Lemma 6.** *There can be $\Omega(n)$ maximal 2-str runs in a double string of length n .*

Proof. If we take any S_1 that contains $\Theta(n)$ runs (e.g. Fibonacci string [9]), and then set $S_1 = S_2$, we will get a double string with $\Theta(n)$ 2-str runs, since the first period of each run in S_1 will pair up with the second period in S_2 . ◀

Remark: We point out that the gap of $\log n$ between the upper and lower bound remains an open problem.

2.2 Primitivity in 2-str TR's and Runs

A string S is *primitive* if it cannot be expressed in the form $s = u^j$, for some integer $j > 1$ and some prefix u of S . For example, *ababa* is primitive, but *abab* is *non-primitive*. The notion of primitivity is very relevant to tandem repeats, since tandem repeats with primitive roots are really the only interesting tandem repeats. In fact, in a string, a maximal run with a primitive root encodes the information about all tandem repeats that span its substring, for e.g. *abababababa* encodes consecutive tandems with periods 2, 4, and 6. We can encode this output as a triple (i, j, p) , where i is the start location of the leftmost tandem, j is the start of the rightmost tandem, and p is the smallest period (in the above example it is $(1, 10, 2)$). This encoding is commonly used in algorithms that report all tandem repeats in a string.

On the other hand, the concept of primitivity in a 2-str TR is more subtle. We cannot say that we are only interested in TR's with primitive roots, as we will miss some TR's in the double string (see Example 4). Furthermore, a non-primitive TR may be a substring of a longer run as in Example 5. This non-primitivity certainly should not disqualify the run.

► **Example 4.** The 2-str TR beginning at location 1, of length 8, has non-primitive root *abab*. This is not implied by the 2-str TR at location 3 with primitive root *ab*.

```

1 2 3 4 5 6 7 8 9 10
a b a b c c c c c c
c c c c a b a b a b

```

► **Example 5.** The TR at location 1 has primitive root *xbab*, the TR's at locations 2, 3, and 4 have non-primitive roots *baba*, *abab*. There is also a 2-str run of period 2 beginning at location 4, which in a sense encodes the TR of period 4 beginning at location 4.

```

1 2 3 4 5 6 7 8 9 10 11
x b a b a b a b a b c
c c c c x b a b a b a

```

We conclude that since some non-primitive TR's must be reported, an algorithm that locates all 2-str TR's must search for these TR's. Hence, our algorithm finds and reports all 2-str TR's, including those that have non-primitive roots. For example, when searching the double string $S_1 = S_2 = a^n$, $\lfloor \frac{n}{2} \rfloor$ maximal 2-str runs will be found, one for each period size. If necessary, those that are not interesting can be filtered out by finding all 1d runs in each string, and merging this with the output of our algorithm, since every 2-str TR that has a non-primitive root will be part of a run in each individual string of the double string.

3 The Algorithm

A common idea used in algorithms that find tandem repeats in a string, is to search for all tandem repeats that cross a given point (see for e.g. [11, 15]). Instead of fixing the starting

point of a tandem, and searching for xx , the algorithm fixes certain points that the set of contiguous tandems must cross, and searches for all tandems that cross that point. We use this idea, searching for each period size separately, and reporting consecutive tandem repeats as a single run. We follow the framework of the Main-Lorentz algorithm [16] (see pseudocode in Algorithm 1). Given an input double string (S_1, S_2) of length n , in the first iteration, all runs that cross the center of the string are found. In the following iteration, (S_1, S_2) is split into two halves, and each one is searched individually. (To simplify the presentation we assume that n is a power of 2.) As implemented in Algorithm 1, this continues for $\log n$ iterations.

The runs that cross the center are classified into two groups. A *right* run has more than half of its characters to the right of the center of the string, and a *left* run has the majority of its characters to the left of the center. Algorithm 2, together with Figure 1, describes the procedure that finds all right runs; by symmetry, all left runs can be found.

The novel idea of Algorithm 2 is that computing the longest common extensions using *two different strings* yields the desired results. The forward comparisons are done with a substring of S_1 against a substring S_2 , and the same for the reverse comparisons. These extensions define which runs occur in the double string crossing the midpoint. The standard KMP algorithm [10] is used to compute all of the forward and reverse extensions in linear time, as done in [16]. The input pattern to KMP for the forward extensions is the string $S_1[\frac{n}{2} \dots n]$ and the text is $S_2[\frac{n}{2} + 1 \dots n]$. Conversely the reverses of $S_1[1 \dots \frac{n}{2} - 1]$ and $S_2[1 \dots \frac{n}{2}]$ are used as input to KMP for the reverse extensions. To ensure maximality, the 2-str runs that reach the end of the substring being processed can be discarded in every iteration other than the top level, since they are non-maximal and will be found in a different iteration. This will ensure in practice that each 2-str run will be found only once.

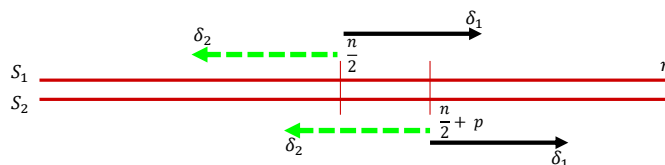
■ **Algorithm 1** Find Runs in a Double String.

Input: double string (S_1, S_2) of length n

Output: all runs that occur in the double string

```

for  $i = \log_2 n$  downto 1 do                                ▷ for  $\log n$  iterations of ML framework
  for  $\ell = 0$  to  $n/2^i - 1$  do                                ▷ for each piece of the input of width  $2^i$ 
    FindRightRuns( $(S_1, S_2)$ ,  $\ell 2^i + 1$ ,  $(\ell + 1)2^i$ )
    FindLeftRuns( $(S_1, S_2)$ ,  $\ell 2^i + 1$ ,  $(\ell + 1)2^i$ )
  end for
end for
    
```



■ **Figure 1** Computing right runs: The figure shows the first iteration, where $beg = 1$ and $end = |S_1|$. δ_1 is the length of the forward extension that results from matching $S_1[\frac{n}{2} \dots n]$ to $S_2[\frac{n}{2} + p \dots n]$. δ_2 is the length of the reverse extension of $S_1[1 \dots \frac{n}{2} - 1]$ and $S_2[1 \dots \frac{n}{2} + p - 1]$. If $\delta_1 + \delta_2 \geq p$, then there are tandem repeats with period size p beginning from location $S_1[\frac{n}{2} - \delta_2 \dots \frac{n}{2} + \delta_1 - 1]$. These are reported by the algorithm as a single run.

3:6 Double String Tandem Repeats

■ **Algorithm 2** FindRightRuns.

Input: double string (S_1, S_2) , beg , end (beginning and end indexes of substring to search)
Output: all right runs that occur in the double string that cross the midpoint.
 $n = end - beg + 1$
 $mid = (beg + end)/2$
for $p = 1$ to $n/2$ **do** ▷ find runs with period p

$\delta_1 =$ length of longest common prefix of $S_1[mid \dots end]$ and $S_2[mid + p \dots end]$ ▷ Forward Extension

$\delta_2 =$ length of longest common suffix of $S_1[beg \dots mid - 1]$ and $S_2[beg \dots mid + p - 1]$ ▷ Reverse Extension

if $\delta_1 + \delta_2 \geq p$ **then** ▷ Check length

if $\delta_1 < n/2$ AND $\delta_2 < n/2 - 1$ **then** ▷ Check for maximality

report run $(mid - \delta_2, mid + \delta_1 - 1, p)$

end if

end if

end for

► **Lemma 7.** *Algorithm 1 finds all 2-str runs in a double string (S_1, S_2) in $O(n \log n)$ time.*

Proof. Every run within (S_1, S_2) crosses the center of a substring of S_1 at some point in the algorithm. As proof of this, consider a run that does not cross the center of S_1 , and hence is not found in the first iteration. The run will be divided among different substrings at some point since in the final iteration the input strings are of length 1. In the step prior to its division, a given run must cross the center since the center becomes the splitting point of the following iteration. In each iteration, only one 2-str run of a given period can cross the center, since no two runs of the same period size can overlap by Lemma 4. Since the algorithm checks each possible period size, all 2-str runs will be found by the algorithm. Since there are $O(\log n)$ iterations, and each iteration takes $O(n)$ time, the total time complexity of Algorithm 1 is $O(n \log n)$. ◀

3.1 Tandem Repeats in a Double String with k -mismatches

The Hamming distance between two strings of equal length is the number of positions at which the corresponding characters are different. Allowing a Hamming distance up to k between the two occurrences of the root results in a k -mismatch 2-str TR. (The concept of a k -mismatch run applies as well, where a run includes consecutive k -mismatch tandem repeats, i.e. each repeat in the run has at most k mismatches, and overall the number of mismatches in the run is not relevant.) In this section we discuss a method for searching for 2-str TR's with up to k mismatches.

► **Example 6.** A double string tandem repeat with $k = 1$ mismatch begins at location 2 in S_1 .

1	2	3	4	5	6	7	8
a	a	b	c	a	a	b	b
c	c	b	b	b	b	c	d

Just as we were able to directly extend the Main and Lorentz idea in the previous section, we are able to extend the algorithm of [12] which solves the tandem repeats with k -mismatches

problem in 1 string. First, instead of using KMP to find the longest common extensions, the algorithm uses the “kangaroo method” that relies on suffix trees and Lowest Common Ancestor (LCA) queries to give the position of the first mismatch between strings [6].

Hence, suffix trees in both the forward and reverse direction must be constructed for each S_1 and S_2 , and preprocessed for LCA to allow constant time Longest Common Prefix (LCP) queries [8, 14].

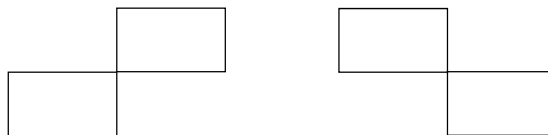
As in the previous algorithm, there are $O(\log n)$ iterations and in each iteration, the repeats that cross the center are found by using the forward and reverse extensions. However, in this case the comparisons are done *allowing up to k errors* in each direction. Specifically, each possible period p is searched for separately. For a given p , each LCP query returns a position of mismatch, and when the $k + 1$ st mismatch is encountered, we stop. Finally, the algorithm considers each pair, $(k', k - k')$ for $0 \leq k' \leq k$. For each pair, we check whether a 2-str TR exists when allowing k' mismatches in the reverse extension and $k - k'$ mismatches in the forward extension.

Time Complexity: The number of iterations is slightly smaller than in the previous algorithm, since for substrings with length $\leq k$, our algorithm should not be run, but a simple $O(k)$ time method should be used. In each iteration, there are $O(k)$ LCP queries done for each possible period size. In addition, before reporting in a particular period, we consider $O(k)$ pairs, allowing a number of mismatches to the left and right. Hence, each iteration takes $O(nk)$ time, while the overall runtime is $O(nk \log(n/k))$.

4 Application 1 - Corner Sharing Tandems

► **Definition 8.** A 2D corner-sharing tandem (cs-tandem) in a 2D array, is a configuration consisting of two occurrences of the same subarray that share one corner (see Figure 2).

In [2], Apostolico and Brimkov mention that all primitive corner-sharing tandems can be found in $O(n^4)$ time using similar techniques to their algorithm that they presented for side-sharing tandems. In this section, we reduce the problem of finding all corner-sharing tandems in a 2D array to the problem of finding tandems in a double string. We thus obtain an $O(n^3 \log n)$ time algorithm for this problem. Although the actual output may be of size $O(n^4)$ cs-tandems, we can reasonably represent the set of cs-tandems with the set of maximal cs-runs, which has size at most $O(n^3 \log n)$. For the special case of tandems that are square (i.e. of size $p \times p$), the algorithm achieves $O(n^2 \log n)$. Finally, the algorithm that allows mismatches in a 2-str TR is also extended to 2D cs-tandems with mismatches, as described in Section 4.3.



■ **Figure 2** The two configurations of a 2D cs-tandem.

► **Definition 9.** A 2D corner-sharing horizontal run (cs-run) is a sequence of one or more corner sharing tandems with the same period size occurring consecutively.

► **Lemma 10.** *There can be $O(n^3 \log n)$ and $\Omega(n^3)$ maximal cs-runs in a 2D array of size n^2 .*

Proof. The proof will be included in the full version of the paper. ◀

4.1 Reduction

The technique of *naming* in 1d is that of consistently replacing identical substrings with an integer called the *name*. We use the following 2D naming technique to reduce the 2D corner sharing tandem problem to the 2-str tandem problem. Given an input 2D text T , we construct $n/2$ 2D texts by naming all subcolumns of T . We create a new text called T_h for each $1 \leq h \leq n/2$, such that $T_h[r, c]$ is the name of the height h substring in column c beginning at row r .

Each two rows, i and $i + h$, in each text of names T_h , $1 \leq h \leq n/2$, is input as a double string to the algorithm that finds tandem repeats in a double string. Since we have a text of names for each height, every corner sharing tandem of height h' , will appear as a 2-str tandem in the text of names for $T_{h'}$. See Figure 3 for an example.

The time complexity for the reduction is $O(n^3)$ since we construct $O(n)$ texts, each in time linear to the size of T , as the naming can be done during construction of a suffix tree of all columns [17]. Algorithm 3 presents pseudocode for the corner-sharing tandem problem. Algorithm 1 is called $O(n)$ times for each of the $O(n)$ texts, and each running of Algorithm 1 takes $O(n \log n)$ time. Overall, the 2D corner-sharing tandem problem is solved in $O(n^3 \log n)$ time.

■ **Algorithm 3** Corner Sharing Tandems Algorithm in 2D Text.

Input: 2D text T of size $n \times n$

Output: all corner-sharing tandems in T

Preprocessing: Construct $n/2$ texts of names, T_h , $1 \leq h \leq n/2$

Text Scanning:

```

for  $h = 1$  to  $n/2$  do                                ▷ for each height  $h$ 
  for  $r = 1$  to  $n - 2h + 1$  do                            ▷ for each row  $r$  in  $T_h$ 
    call Algorithm 1 with rows  $r$  and  $r + h$  in  $T_h$  as  $S_1, S_2$  respectively.
  end for
end for

```

4.2 Corner-Sharing Square Tandems

If the problem of finding all corner-sharing tandems is limited to those tandems whose roots are of size $p \times p$, we can improve our algorithm to run in $O(n^2 \log n)$ time. We will have to show two things: 1: a transformation of the input 2D text into input to the double string problem in less time. 2. The search phase of the algorithm can be improved. The transformation can be done using the techniques of [7] for finding 2D palindromes, while the search phase can be shown to be faster using a counting trick. Details are omitted due to lack of space.

a	b	c	d	e	a	x	x	x	x
a	b	c	d	e	a	x	x	x	x
c	c	c	c	c	c	x	x	x	x
y	y	y	y	a	b	c	d	e	a
y	y	y	y	a	b	c	d	e	a
y	y	y	y	c	c	c	c	c	c

1	2	3	4	5	1	6	6	6	6
7	7	7	7	1	2	3	4	5	1

Figure 3 Input text T is shown on the left, containing a run with period size 3×4 , beginning at its upper left corner. The two corresponding rows in T_3 can be viewed as a double string, and the 2-str run with period 4 found with substring “123451” in S_1 directly corresponds to the cs-run in T .

4.3 Corner Sharing Tandems with k -mismatches

A 2D corner sharing tandem that allows up to k mismatches between copies is called a k -mismatch cs-tandem. A k -mismatch cs-run can be defined analogously as a set of contiguous k -mismatch cs-tandems, such that each individual cs-tandem contains at most k mismatches. The algorithm described in Section 3.1 searches for tandem repeats in a double string allowing k mismatches. The reduction of Section 4.1 can be used in a similar manner to reduce the k -mismatch cs-tandem problem to the k -mismatch double string problem. However, each mismatch between names in the 2D text may consist of one or more mismatches in the column, and will therefore need further investigation. Hence, we will need to process the mismatching columns when attempting to discover the actual tandems. Details will be included in the full version of the paper.

5 Application 2 - Scaled Tandem Repeats

5.1 Definitions and Properties

Denote the string $aa \dots a$, where a repeated r times, by a^r . Let $S = a_1^{r_1} a_2^{r_2} \dots a_j^{r_j}$ be a string for which $a_i \neq a_{i+1}$. Let $e \in N$, we say that $S^{[e]}$ is an e -scaling of S if $S^{[e]} = a_1^{r_1 \cdot e} a_2^{r_2 \cdot e} \dots a_j^{r_j \cdot e}$.

► **Definition 11.** A scaled tandem repeat is a string UU' where U' is an e -scaling of U for some integer e , i.e. $U' = U^{[e]}$. We call the period of a scaled tandem repeat the length of the first copy, i.e. $|U|$.

We say that a scaled tandem repeat is *sharp* if the the last letter of U is not equal to the first letter of U' . Similarly, we say that scaled tandem repeat UU' occurring within text T is a sharp occurrence, if the character in T prior to U differs from the first character of U , and the following character in T differs from the last character of U' . Using the techniques of [1] it is possible to show that any solution to the problem of finding sharp occurrences of sharp scaled tandem repeats yields a solution to the general scaled tandem problem with the same complexity. Thus, we solve the following problem.

Problem Definition. Given a 1-dimensional text $T = t_1 \dots t_n$, find all sharp occurrences of sharp scaled tandem repeats (SSTR) that are substrings of T .

3:10 Double String Tandem Repeats

We assume that the number of distinct characters in a sharp tandem repeat is at least two, otherwise it would not be sharp. We also assume that the scaled tandem repeats we are seeking are of scale $e > 1$, since for $e = 1$ this is the known case of regular tandem repeats.

Define T' as the run-length encoding (RLE) of the given text T , where each sequence of characters is replaced with a character and exponent. T'_{char} is the string of characters of the RLE of T , and T'_{exp} is the string of exponents of T' . In a similar manner to [5] we define the *quotient array* $S_Q[1..n-1]$ of array of numbers $S[1..n]$ as follows: $S_Q[i] = S[i+1]/S[i]$.

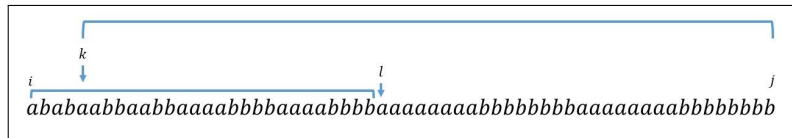
► **Lemma 12.** *No more than $O(n \log n)$ SSTR can occur in a string T of length n .*

Proof. Every SSTR in T must correspond to a tandem repeat in T'_{char} . By the Three Squares Lemma [4], each location in T'_{char} can have at most $O(\log n)$ tandem repeats. We conclude that there are $O(n \log n)$ SSTR's in T . ◀

The naive algorithm for the problem would consider every substring of the input text T and check whether it is an SSTR resulting in time $O(n^3)$. Known methods of using suffix trees and LCA's on the character and quotient arrays of the string (see e.g. [13]), allow checking in constant time, for every substring U of T , whether the subsequent substring of T is a scaled copy of U . Thus the time complexity of straight-forward improvements for finding all scaled tandem repeats would be $O(n^2)$.

In the next subsection we solve the SSTR problem in a more efficient way by reducing the problem into a tandem problem on double strings. To this end, we first generalize the definition of a *run* as a concatenated string of repeats, so that the problem can fit into the framework described in Sections 2 and 3.

► **Definition 13.** *Let T be a string, $1 \leq i < j \leq n$. We say that there is a scaled run from $T[i]$ to $T[j]$ if there are k, ℓ ; $i < k \leq \ell < j$, for which $\exists e, T[k \dots \ell] = T[i \dots \ell]^{[e]}$. e is called the scale of the run. The period of the run is the period of the leftmost scaled tandem repeat in the run. A scaled run with scale e is maximal if it cannot be extended by one character either to the right or the left, i.e. there are no scaled runs from $T[i]$ to $T[j+1]$, from $T[i-1]$ to $T[j]$, nor from $T[i-1]$ to $T[j+1]$ with scale e .*



■ **Figure 4** An example of a scaled run.

5.1.1 The Compact Region Idea for Scaling

In [3], Butman, Eres and Landau showed a linear-sized data structure of compact regions of text T that enables efficient work on scaled matching problems. The idea is to construct $n/2$ collections of strings $T_1, \dots, T_{n/2}$, where the sum of the lengths of the substrings in all T_i 's is $O(n)$. We will then seek dual tandems of each such substring S in the T_i 's and a substring of T whose length is $O(|S|)$.

We provide below the definition of the compact regions data structure, which is based upon the following observation.

► **Observation 1.** *If a substring S scaled to e occurs sharply in $\sigma_i^{j_i} \dots \sigma_k^{j_k}$ then j_i, \dots, j_k are multiples of e .*

Following the above observation, the compact regions structure computes for each scale e a compact text T_e in the following two steps:

Step 1: Locate all the regions in T where the symbols appear in scale e . Add the symbol $\$$ as a separator between the regions.

Step 2: Expand these regions to include the symbols on their boundaries. In order to simplify the computation of Stage 2, a symbol $t_j^{r_j}$ of T is replaced in T_e by $t_j^{\lfloor \frac{r_j}{e} \rfloor}$. Butman et al. [3] showed that the total length of all regions is $O(n)$, and that the compact regions data structure can be constructed in linear time.

$$\begin{aligned}
 T &= a^2b^5a^8c^6b^4a^8b^9a^3c^7b^2a^4c^8a^9b^1a^3b^6c^9b^9 \\
 T_2 &= a^1b^2\$b^2a^4c^3b^2a^4b^4\$b^4a^1c^3\$c^3b^1a^2c^4a^4\$a^4\$a^1b^3c^4\$c^4b^4\$b^4 \\
 T_3 &= \$b^1\$b^1a^2\$a^2c^3b^1\$b^1a^2\$a^2b^3a^1c^2\$c^2\$a^1\$a^1c^2\$c^2a^3\$a^1b^2c^3b^3 \\
 T_4 &= \$b^1\$b^1a^2c^1\$c^1b^1a^2b^2\$b^2\$c^1\$c^1a^1c^2\$a^2\$b^1\$b^1c^2\$c^2b^2\$b^2 \\
 T_5 &= \$b^1a^1\$a^1c^1\$c^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1a^1\$a^1\$b^1\$b^1c^1\$c^1b^1\$b^1 \\
 T_6 &= \$a^1\$a^1c^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1a^1\$a^1\$b^1c^1\$c^1b^1\$b^1 \\
 T_7 &= \$a^1\$a^1\$a^1\$a^1b^1\$b^1\$c^1\$c^1a^1\$a^1\$c^1\$c^1b^1\$b^1 \\
 T_8 &= \$a^1\$a^1b^1\$b^1\$c^1a^1\$a^1\$c^1\$c^1b^1\$b^1 \\
 T_9 &= \$b^1\$a^1\$c^1b^1
 \end{aligned}$$

■ **Figure 5** compact regions data structure example.

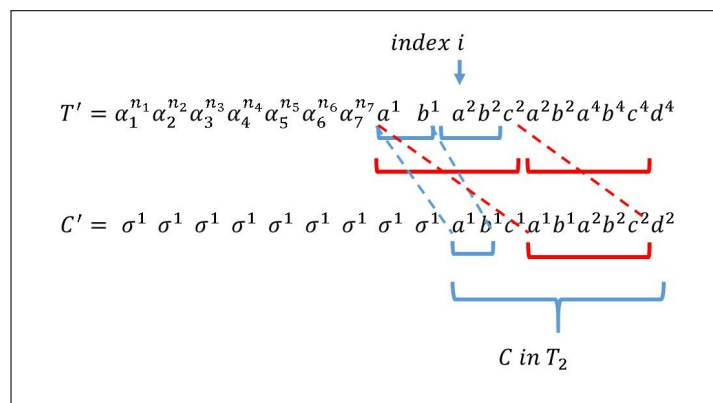
5.2 The Reduction

The reduction is based on the following lemma.

► **Lemma 14.** *Let T be a text and assume that there is a scaled tandem to scale $e > 1$ starting in index i of T , where the length of the period is p . Then the scaled part of the tandem is represented by a substring of a single compact region in T_e . In fact, the substring in T_e is precisely the period.*

Proof. Since the scale of the period is e , then e divides the exponent of every symbol in the scaled part of the tandem. We write in T_e the scales divided by e therefore what is written in T_e is precisely the period. ◀

Assume that a compact region C in T_e starts at location i of the RLE T' of T . Lemma 14 assures us that any scaled tandem whose scaled repetition occurs in C cannot start in any index smaller than $i - |C|$ and cannot end in any index larger than $i + |C|$. Let X be the string composed of $|C|$ occurrences of σ^1 , where σ is a symbol not in the alphabet. Let $C' = XC$. Then every double string tandem between the strings $T'[i - |C|..i + |C|]$ and C' is an e -scaled tandem in T . The figure below illustrates this. Both $abaabb$ and $abaabbccaabbaaaabbbbcccc$ are 2-scale tandems. They both appear as double string tandems between the appropriate substring of T' and C' .



■ **Figure 6** 2-scale tandems as double tandems.

Time: The compact regions data structure is created in time $O(n)$. For every region C the double string tandem repeats are found in time $O(|C| \log |C|)$. Since $\sum_{v_C} |C| = O(n)$ the total time is $O(n \log n)$.

References

- 1 Amihood Amir, Ayelet Butman, and Moshe Lewenstein. Real scaled matching. *Information Processing Letters*, 70(4):185–190, 1999. doi:10.1016/S0020-0190(99)00060-5.
- 2 Alberto Apostolico and Valentin E. Brimkov. Optimal discovery of repetitions in 2d. *Discrete Applied Mathematics*, 151(1-3):5–20, 2005. doi:10.1016/j.dam.2005.02.019.
- 3 Ayelet Butman, Revital Eres, and Gad M. Landau. Scaled and permuted string matching. *Information processing letters*, 92(6):293–297, 2004. doi:10.1016/j.ipl.2004.09.002.
- 4 Maxime Crochemore, Lucian Ilie, and Wojciech Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227–5235, 2009. Mathematical Foundations of Computer Science (MFCS 2007). doi:10.1016/j.tcs.2009.08.024.
- 5 Tali Eilam-Tsoreff and Uzi Vishkin. Matching patterns in strings subject to multilinear transformations. *Theoretical Computer Science*, 60(3):231–254, 1988. doi:10.1016/0304-3975(88)90112-0.
- 6 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.
- 7 Sara H. Gezhals and Dina Sokol. Finding maximal 2-dimensional palindromes. *Information and Computation*, 266:161–172, 2019. doi:10.1016/j.ic.2019.03.001.
- 8 David Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 9 Costas S. Iliopoulos, Dennis Moore, and W.F. Smyth. A characterization of the squares in a fibonacci string. *Theoretical Computer Science*, 172(1):281–291, 1997. doi:10.1016/S0304-3975(96)00141-7.
- 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 '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.
- 12 Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8:1–18, 2001. doi:10.1089/106652701300099038.

- 13 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.
- 14 Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989. doi:10.1016/0196-6774(89)90010-2.
- 15 J.J. Liu, G.S. Huang, and Y.L. Wang. A fast algorithm for finding the positions of all squares in a run-length encoded string. *Theoretical Computer Science*, 410(38):3942–3948, 2009. doi:10.1016/j.tcs.2009.05.032.
- 16 Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984. doi:10.1016/0196-6774(84)90021-X.
- 17 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

Efficient Tree-Structured Categorical Retrieval

Djamal Belazzougui¹

CAPA, DTISI, Centre de Recherche sur l'Information Scientifique et Technique, Algiers, Algeria
dbelazzougui@cerist.dzm

Gregory Kucherov 

CNRS and LIGM/Univ Gustave Eiffel, Marne-la-Vallée, France
Skolkovo Institute of Science and Technology, Moscow, Russia
Gregory.Kucherov@univ-mlv.fr

Abstract

We study a document retrieval problem in the new framework where D text documents are organized in a *category tree* with a pre-defined number h of categories. This situation occurs e.g. with taxonomic trees in biology or subject classification systems for scientific literature. Given a string pattern p and a category (level in the category tree), we wish to efficiently retrieve the t *categorical units* containing this pattern and belonging to the category. We propose several efficient solutions for this problem. One of them uses $n(\log \sigma(1 + o(1)) + \log D + O(h)) + O(\Delta)$ bits of space and $O(|p| + t)$ query time, where n is the total length of the documents, σ the size of the alphabet used in the documents and Δ is the total number of nodes in the category tree. Another solution uses $n(\log \sigma(1 + o(1)) + O(\log D)) + O(\Delta) + O(D \log n)$ bits of space and $O(|p| + t \log D)$ query time. We finally propose other solutions which are more space-efficient at the expense of a slight increase in query time.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching; Information systems \rightarrow Document representation; Information systems \rightarrow Information retrieval query processing; Theory of computation \rightarrow Data structures design and analysis

Keywords and phrases pattern matching, document retrieval, category tree, space-efficient data structures

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.4

Related Version <http://arxiv.org/abs/2006.01825>

Funding GK was partially funded by RFBR, project 20-07-00652, and joint RFBR and JSPS project 20-51-50007.

1 Introduction

Data is often structured using *category hierarchies* represented by trees. In many applications, such hierarchies play a crucial guiding role: for example, the International Classification of Diseases (ICD) provides a hierarchical classification of all human diseases and constitutes a common reference for diagnostics. In this paper, we are interested in sequence data, such as biological sequences or text documents, that are linked to a given hierarchy. More precisely, in our framework sequences are associated to leaves of a hierarchy, and tree nodes are mapped to several fixed levels, also called ranks.

This situation is common and occurs in several important applications. One is biology where species are classified according to the famous Linnaean taxonomy including eight common *taxonomic ranks*: species, genus, family, order, class, phylum, kingdom, domain. Then, given a set of sequences (DNA, RNA or protein) belonging to known species, one can associate them to the corresponding leaves of the taxonomic tree. Such a structure is used, for example, for phylogeny-based metagenomic classification where one considers the tree of

¹ Corresponding author



known genomic sequences as a reference for classifying sequences of a metagenomic sample, see e.g. [23]. A classification procedure may involve queries asking for the taxonomic units (i.e. internal nodes of the tree) of a certain rank whose sequences contain a given pattern, or similar type of queries.

Another example is provided by text documents such as scientific papers. The latter are usually annotated by subjects belonging to a fixed hierarchical nomenclature, such as ACM Computing Classification System (CCS) or Mathematics Subject Classification (MSC). Those subject hierarchies have a predefined number of levels: four levels for CCS and three for MSC. Given a corpus of scientific papers, one could ask about subject categories at a certain level whose documents contain a given pattern. This is a natural information retrieval scenario.

Here we study this problem from the stringology perspective (see e.g. [14, 8]). Assume we are given a set of D documents of total length n over an alphabet of size σ , organized in a tree of height h . The tree has D leaves, each associated with a distinct document, and the leaves are all at level h of the tree. The total number of nodes in the tree is denoted by Δ . The tree specifies a hierarchy of categories: each level of the tree corresponds to a category, and each internal node corresponds to a *categorical unit*.

The basic type of query we study in this paper is the following.

Given a pattern p , and a tree level (rank) $i \in [1..h]$, return all nodes (categorical units) d_1, \dots, d_t at level i that have at least one leaf (document) in their subtree that contains pattern p .

For example, given a large collection of genomic sequences organized in a taxonomic tree (for example, all known animal genomes), one may ask which animal families have a given sequence in the genomes of their members. Or, given a large hierarchy of documents (for example, all Computer Science papers), one may wonder in which subfields of Computer Science (corresponding to a certain level of the hierarchy) the term 'suffix tree' is used. This basic type of queries can be further extended in different ways. For example, one may impose an additional requirement of the minimum number of documents of the categorical unit containing the given pattern. In this first study, we focus on the basic query type.

In this work, we propose several algorithms for this problem. Our first solution (Section 3) is based on the approach of Muthukrishnan [16] to the document retrieval problem. By combining several algorithmic tools - efficient text index, colored range reporting queries, and level ancestor queries - we obtain a solution with $n(\log \sigma(1 + o(1)) + \log D + O(h)) + O(\Delta)$ bits of space and $O(|p| + t)$ query time, where t is the output size, i.e. the number of retrieved categorical units. To improve the space bound, in particular to get rid of the $O(nh)$ term which can be as big as $O(nD)$, we then develop a solution based on a wavelet tree built on top of the input category tree (Section 4). On this way, we first obtain a solution taking $n(\log \sigma + \log D) + O(D \log n)$ bits and $O(|p| + t \cdot h \log D)$ query time. We further improve it using the technique of heavy path decomposition, to obtain a solution in $n(\log \sigma(1 + o(1)) + \log D) + O(\Delta)$ bits of space and $O(|p| + t \log D)$ query time. In the final part of the paper (Section 5), we focus on solutions using succinct and compressed data structures, on top of the input data. That is, our main goal here is to replace the $n \log D$ bits by respectively $n \log \sigma$ or by $nH_0 + o(n \log \sigma)$ in representing the document array. We obtain memory-time trade-offs showing how this goal can be achieved at the price of a slight increase of query time.

We summarize our main results in the following table.

algorithm	space (bits)	query time
based on colored range queries (Sect. 3)	$n(\log \sigma(1 + o(1)) + \log D + O(h)) + O(\Delta)$	$O(p + t)$
based on wavelet tree (Sect. 4)	$n(\log \sigma(1 + o(1)) + O(\log D)) + O(\Delta) + O(D \log n)$	$O(p + t \log D)$
compact space (Sect. 5)	$O(n \log \sigma)$	$O(p + (t + 1) \cdot \log^\epsilon n(1 + \frac{h}{\log \sigma}))$
compressed space (Sect. 5)	$nH_k + o(n \log \sigma) + O(D \log n)$	$O(p + t \cdot h \log n (\log \log n)^2)$

2 Preliminaries

We first briefly present main algorithmic tools used by our algorithms.

2.1 Level ancestor queries on trees

Consider a rooted tree. To each node in the tree we associate its *level* so that the level of the root is 1, and the level of a child node is 1 more than the level of its parent. The height of a tree is defined as the maximal level of any node in the tree. We denote by ℓ_α the level of a node α .

We will use the implementation of level ancestor queries specified by the following lemma.

► **Lemma 1** ([19]). *There exists a data structure that represents a tree with n nodes within space $2n + o(n)$ and allows answering the following queries in constant time:*

1. *given a level ℓ and a node α at level at least ℓ , return the ancestor node β of α at level ℓ ,*
2. *given an integer i , return the node α where α is the leaf number i in left-to-right order.*

We denote by $\text{LAQ}(\alpha, i)$ the query which asks for the ancestor at level i of node α . We denote by $\text{leafselect}(i)$ the query which returns the i -th leaf of the tree in left to right order.

2.2 rank/select queries and wavelet trees

rank and **select** queries on sequences constitute basic building blocks of many succinct data structures [13]. Given a string $S[1..n]$ on an alphabet Σ , a query $\text{rank}_c(S, i)$, with $c \in \Sigma$ and $i \in [1..n]$, asks for the number of occurrences of c in $S[1..i]$ and $\text{select}_c(S, j)$ asks for the unique position i such that $S[i] = c$ and $\text{rank}_c(S, i) = j$.

Consider first the important case of binary sequences (bitvectors). The following result is well-known, see [18].

► **Lemma 2.** *A bitvector $B[1..n]$ can be represented using $n + o(n)$ bits of space, so that queries **rank** and **select** are answered in constant time.*

In the case of non-binary alphabet, **rank/select** queries can be efficiently answered using *wavelet trees*. The wavelet tree has been formally introduced in [9], but a similar structure has been used earlier [3]. Suppose we are given a sequence S of length n over an alphabet Σ .

The (*binary*) *wavelet tree* is a binary tree representation of S that is defined recursively as follows. Let $\Sigma_0 \neq \emptyset$ and $\Sigma_1 \neq \emptyset$ form a partition of Σ (that is, $\Sigma = \Sigma_0 \cup \Sigma_1$ and $\Sigma_0 \cap \Sigma_1 = \emptyset$). Then the root of the binary wavelet tree will contain a binary vector B , such that $B[i] = 0$ iff $S[i] \in \Sigma_0$. Let the sequence S_0 (resp., S_1) be formed by keeping only the elements of S that belong to Σ_0 (resp., Σ_1), in the same order. Then, the left (resp., right) child is defined

recursively using S_0 (resp., S_1) and a binary partition of Σ_0 (resp., Σ_1). The recursion stops whenever we reach a leaf that corresponds to a singleton subset of Σ . Such nodes will form the leaves of the wavelet tree. We refer the reader to the survey [17] for more details about wavelet trees. We will make use of the following lemma:

► **Lemma 3** ([9]). *The wavelet tree over the alphabet $[1..\sigma]$ can be represented using $n(\log \sigma + o(1)) + O(\sigma \log n)$ bits of space, supporting **rank** and **select** queries in $O(\log \sigma)$ time.*

The definition of binary wavelet tree can be readily generalized to the non-binary case. As in the binary case, to any node α labeled by an interval Σ_α is (implicitly) associated the sequence S_α which is the subsequence of $S[1..n]$ consisting of all characters belonging to Σ_α . If a node α of a wavelet tree has d children, then the alphabet interval $\Sigma_\alpha \subseteq [1..\sigma]$ assigned to α is partitioned into d disjoint subintervals instead of two, and α stores a sequence C_α over alphabet $[1..d]$ of length $|S_\alpha|$ such that $C_\alpha[i] = j$ iff $S_\alpha[j] \in \Sigma_{\alpha_j}$.

2.3 Text indexes

We assume familiarity with main text indexing structures: suffix trees, suffix arrays and BWT-indexes. Here we only recall some basic facts about them.

Given a text T over an alphabet $\Sigma = [1..\sigma]$, a suffix tree [22] is a tree data structure that stores in its leaves the suffixes of $T\$$, where $\$$ is a special character that does not appear in T and is lexicographically smaller than any character of T . Each suffix is associated with its starting position in $T\$$. Suffix tree allows answering basic string pattern matching queries: given a pattern p , return the set of starting positions of p in T .

The suffix array of T is a related but more space-efficient data structure defined as the array $\text{SA}[1..n+1]$ obtained by sorting all the suffixes of $T\$$ in lexicographic order and setting $\text{SA}[i] = j$ if and only if the suffix $T[j..n]\$$ has lexicographic rank i among all suffixes of $T\$$.

A suffix tree occupies $O(n \log n)$ bits of space and a matching query needs access to the original text T in addition to the suffix tree. The query time is $O(|p| \log \sigma)$. The suffix array [15] is an alternative to the suffix tree which occupies the same $O(n \log n)$ bits of space, but has lower constant factors in space and supports matching queries in $O(|p| + \log n)$ time.

The BWT-index (FM-index) is a space-efficient alternative to suffix arrays and suffix trees which uses $O(n \log \sigma)$ bits of space only. It was originally proposed in [4] and has seen many improvements. We will use the following version of BWT-index with alphabet-independent query time.

► **Lemma 4** ([1]). *Given a text T of length n over alphabet $[1..\sigma]$, we can build a BWT-index which occupies $n \log \sigma(1 + o(1))$ bits of space and supports computing the range of suffixes prefixed by a pattern p in time $O(|p|)$.*

Note that computing the range of suffixes answers also whether the pattern occurs in the text at all, and if so, reports the number of its occurrences (the size of the lexicographic order interval). For this reason, the query presented in the lemma above is usually referred to as a **count** query. The BWT-index is usually augmented with position information so that it becomes able to report the location of each occurrence of the pattern in addition to the number of occurrences. This can be achieved using for the example the compressed suffix array representation:

► **Lemma 5** ([10]). *Given a text T of length n over alphabet $[1..\sigma]$ and a constant $\epsilon > 0$, we can build a data structure which occupies $O(n \log \sigma)$ bits of space and that returns $\text{SA}[i]$ for any $i \in [1..n]$ in time $O(\log^\epsilon n)$.*

All the above-mentioned text indexes can trivially be extended to support the same type of queries on a collection of documents instead of a single document. More precisely, given a collection of texts T_1, T_2, \dots, T_D over the same alphabet Σ , the same queries can be supported by constructing an index of the string $T_1\$T_2\$ \dots T_D\$$.

2.4 Colored range reporting and document retrieval

Muthukrishnan [16] was the first to study the problem of efficiently retrieving documents containing a given string pattern. Through the use of a text index, he reduced the problem to the one of *color range reporting*, i.e. reporting all *distinct* values (“colors”) occurring in a given interval of an array. His data structure relies on the use of *range minimum query* data structures – a data structure that can find in constant time the smallest element in an sub-range of an array. His algorithm was subsequently improved in terms of space (Theorem 4 in [20]). We will use the following result on color range reporting, which can be obtained by using the optimal range-minimum query data structure [5] in the method of [20]:

► **Lemma 6.** *Given an array $A[1..n] \in [1..\sigma]^n$, we can build a static data structure that occupies $2n + o(n)$ bits that allows reporting all d distinct values occurring in a query interval $A[i..j]$ in time $O(d)$ ($O(1)$ time per reported value). The query will make read-only access to the data structure, read-only random access to elements of the array A and read-write access to a bitvector B of size σ . The bitvector needs to be initialized to zero before the first query and is reset to zero at the end of each query.*

In combination with text indexing, colored range reporting allows supporting document retrieval queries. More precisely, define the *document array* as follows: given a collection of D documents $T_1, T_2 \dots T_D$ of total length n , lexicographically sort all the suffixes of the text $T^* = T_1\$T_2\$ \dots T_D\$$, and set $A[i] = j$ iff the suffix of T^* of lexicographic rank i starts inside T_j (if the suffix starts with $\$$, then set $A[i] = 0$). Document array A can be easily obtained from a text index of $T^* = T_1\$T_2\$ \dots T_D\$$. For this, one can construct a bitmap of length $|T^*|$ with 1’s at positions of $\$$ in T^* and 0’s otherwise. Then $A[i] = \text{rank}_1(A, \text{SA}[i]) + 1$ for $i > D$ and $A[i] = 0$ for $i \leq D$. It is then immediate that using these data structures, Lemmas 4, 5, and 6 lead to solving the document retrieval problem in time $O(|p| + d \log^\epsilon n)$, where d is the number of resulting documents. For this, we can use the document alphabet-independent BWT index to compute the range $[i..j]$ of occurrences of p in $O(|p|)$ time and then report the d distinct documents that appear in the range $A[i..j]$ in $O(d \log^\epsilon n)$ time.

3 Solution based on Muthukrishnan’s data structure

Our first solution will be a combination of tools presented in the previous section. We first build a text index for the concatenation of documents $T_1\$T_2\$ \dots T_D\$$. More specifically, we build an instance of the text index of Lemma 4 which occupies $n \log \sigma(1 + o(1))$ bits and allows to locate the interval of all suffixes of the documents that start with p in time $O(|p|)$. We also build the document array $A[1..n]$, of size $n \log D$, indexed by the document suffixes sorted in lexicographic order and storing the documents each of the suffixes belongs to.

We further store h instances C_1, \dots, C_h of the data structure of Lemma 6, one instance per level of the tree, defined as follows. Consider d (virtual) arrays $A_i[1..n]$, one per level $i \in [1..h]$ of the tree, such that $A_i[j]$ stores the ancestor at level i of document $A[j]$. Then, each C_i is the data structure of Lemma 6 for supporting range reporting queries on array A_i . Thus, C_i allows to return, for any interval $[r..l]$, all distinct elements in $A_i[r..l]$ in constant time per element provided that a random-access to each element in A_i is supported in constant time.

Note that according to Lemma 6, a query will need to use D bits of working space² since it will need to use a temporary bitvector B of size $D_i \leq D$ where D_i is the number of nodes at level i of the tree³. By Lemma 6, each C_i occupies only $2n + o(n)$. Finally, in order to simulate constant-time random access to entries of arrays A_i , $1 \leq i \leq h$, we build a data structure for constant-time level ancestor queries on the category tree (Lemma 1). Notice that we can access cell $A_i[j]$ using the formula $A_i[j] = \text{LAQ}(\text{leafselect}(A[j]), i)$. The data structure will occupy $2\Delta + o(\Delta)$ bits of space, where Δ is the total number of nodes in the tree.

To answer a query consisting of a pattern p and level i , we proceed as follows. We first compute, in time $O(|p|)$, the interval $[\ell..r]$ of suffixes using the BWT-index (Lemma 4). The documents containing p are then those contained in $A[\ell..r]$. We then have to output all distinct ancestors at level i of documents $A[\ell..r]$, i.e. all distinct elements of $A_i[\ell..r]$. This is done in constant time per reported element using C_i , as follows from Lemma 6 and constant-time access to elements of A_i using LAQ and leafselect queries.

The document array occupies $n \log D$ bits of space. The text index is built on top of the $n \log \sigma(1 + o(1))$ bits. Each of the h instances of the data structure of Lemma 6 will occupy $2n + o(n)$ bits of space each for a total space of $2nh + o(hn)$ bits of space. The data structure built on top of the category tree occupies $2\Delta + o(\Delta)$ bits of space.

We thus have proved the following theorem:

► **Theorem 7.** *Given a collection of D documents of total length n over alphabet $[1..\sigma]$ so that the documents are organized in a hierarchy of documents represented by a tree of total size Δ and of height h , we can build a data structure of size $n(\log \sigma(1 + o(1)) + \log D + O(h)) + O(\Delta)$ bits of space that, given a pattern p , can find all t categories of documents at a given level i that have at least one document that contains the pattern in total time $O(|p| + t)$.*

This data structure will be good enough whenever h is small, for example, when $h = \log D$, which holds for example when each internal node in the tree has at least two children.

4 Wavelet-tree-based solution

If each node of our tree is branching, i.e. has two or more children, then $h = O(\log D)$ and the solution of Section 3 takes $O(n(\log \sigma + \log D))$ bits of space. (Recall that all leaves of our tree occur at level h) However, this may not be the case as the tree may have many non-branching (unary) nodes. In the extreme case, we may have $h = \Omega(D)$ and the space of Theorem 7 will become $\Omega(nD)$ which can be too large if D is large. In this section, we deal with this issue and present solutions based on wavelet trees.

As in Section 3, we assume that we first located an interval $[\ell..r]$ in the document array A that corresponds to the occurrences of the query pattern p . The goal is then to return all internal nodes at level i containing documents from $A[\ell..r]$ in their subtree. In Section 4.1, we present the first "warm-up" solution that we subsequently improve in Section 4.2.

² We define the working space as a writable space that is only used during queries and is restored to its initial state at the end of the query

³ We can use the same bitvector B (Lemma 6) of size D for all h levels: for a query on level i , the first D_i bits of B are initially set to zero and are reset to zero at the end of the query

4.1 Basic wavelet-tree-based solution

We build our wavelet tree on top of the input tree representing the hierarchy of the documents. Therefore, our initial wavelet tree is generally non-binary and non-balanced. As does the input tree, our wavelet tree has height h and $O(\Delta)$ nodes in total. To save space, we will eliminate unary nodes from the wavelet tree (such a node α stores a trivial sequence $C_\alpha = 1^{|S_\alpha|}$, see Section 2.2) and only encode $O(D)$ branching nodes. For each branching node α we store its depth denoted δ_α . Besides the wavelet tree, we will need a data structure for level ancestor queries (Lemma 1) for the input tree that occupies $O(\Delta)$ bits of space and answers queries in constant time.

Our alphabet Σ will be defined to be the set of documents $[1..D]$. The alphabet interval Σ_α assigned to a node α will be the indices of documents occurring in the subtree rooted at α . The string S for which the tree is built will be the document array $A[1..n]$.

Our wavelet tree may have nodes with more than two children and we implement them by local *binarization*. If a node has d children, we will encode it using a binary wavelet tree of $\log d$ levels, called a *local wavelet tree*. In total, the wavelet tree occupies $n(h \log D)$ bits, since the tree contains h levels and each of the n elements of the document array will contribute at most $\log D$ bits to each level.

Consider now a query which is defined by a pattern p and a level i in the input tree. Once we computed the document array interval corresponding to p , say $A[\ell..r]$, we use our wavelet tree to identify the desired nodes at level i . Starting from the root, we traverse the tree top-down through all the nodes α whose assigned sub-alphabet $\Sigma_\alpha \subseteq [1..D]$ intersects with elements of $A[\ell..r]$. This is done by recomputing the current interval for each traversed node. An invariant of this computation is that querying a node α with an interval $[i..j]$ ensures that all elements of $A[\ell..r] \cap \Sigma_\alpha$ are within $S_\alpha[i..j]$. Interval computation is done using **rank** queries on binary vectors B_α stored at nodes α of the wavelet tree, we refer to [7] where this computation is described in detail. We stop the traversal at a node α as soon as $\delta_\alpha \geq i$ and report its ancestor at level i using the level ancestor data structure.

The original tree has at most h levels and each node is replaced by a local wavelet tree with at most $\log D$ levels, therefore a root-to-leaf path in the wavelet tree has at most $h \log D$ nodes, and the total worst-case query time will be $O(h \log D)$ per reported node.

We now analyse the space usage of the data structure. Since the wavelet tree has D leaves and all nodes are branching, the total number of nodes is $O(D)$. Thus, the total space used by the wavelet trees is $n(h \log D)(1 + o(1)) + O(D \log n)$ bits (see Lemma 3). The space used by the BWT-index is $n \log \sigma(1 + o(1))$ (Lemma 4) and the space used by the document array is $n \log D$ bits. The space used by the data structure for level-ancestor queries is $O(\Delta)$ bits (Lemma 1). We thus proved the following theorem.

► **Theorem 8.** *Given a collection of D documents of total length n over alphabet $[1..\sigma]$ and so that the documents are organized in a hierarchy of documents represented by a tree of height h , we can build a data structure of size $n(\log \sigma + (h + 1) \log D)(1 + o(1)) + O(D \log n) + O(\Delta)$ bits of space that can, given a pattern p , find all t categories of documents at level i that have at least one document that contains the pattern in total time $O(|p| + t \cdot h \log D)$.*

4.2 Solutions based on heavy path decomposition

We now describe a more sophisticated solution based on the *heavy path decomposition* [21, 11] of the wavelet tree from the previous section. Here we present a high-level description of our algorithms, full details will be given in the extended version of the paper.

There are several variants of the definition of heavy path decomposition, with slight differences between the variants. In what follows we will use the following variant. With each node α of a given tree T , we associate a *weight* $w(\alpha)$ equal to the number of leaves in the subtree rooted at α . The *heavy child* β of α is the child of α with the greatest weight, with ties resolved arbitrarily. The other children of α are called *light*. The edge between α and its heavy child is called a *heavy edge*, whereas all the other edges from α to its children are called *light edges*.

The heavy path decomposition of a tree T is a decomposition of T into paths defined recursively as follows. We first compute the heavy path (i.e. a path consisting of heavy edges) from the root of T to a leaf, and then recursively apply the decomposition to all subtrees rooted at all light children of the heavy path nodes. An interesting property of the heavy path decomposition is that the number of light edges on any root-to-leaf path is at most $\log D$, where D is the number of leaves in the tree.

4.2.1 First solution based on heavy path decomposition

Our first solution will be neither space- nor time-optimal. For each heavy path starting at a node α for which the number of light children of nodes of the path is ℓ_α , the alphabet will be of size ℓ_α . We can order the nodes (light children) by increasing depths. The sequence S_α that is associated with a heavy path $\alpha = \alpha_1, \dots, \alpha_k$, will be of length n_α over alphabet $[1..\ell_\alpha]$, where n_α is the total number of occurrences of leaves (documents) in the subtree rooted at α in the document array A . That is, the sequence will be a subsequence of $A[1..n]$, where only the documents that belong to the leaves under α are kept, and the encoding of each element in the subsequence will be the index of the (light) children of the heavy path nodes under which the document appears. Let the depths of the nodes in the heavy path be denoted by $d_1 < \dots < d_k$. We additionally store a bitvector B_α marking the node depths of the different nodes. That is, we initialize the bitvector B_α by all zeros and then set $B_\alpha[d_i] = 1$ for every $i \in [1..k]$.

A query for level i will now proceed as follows. We traverse the tree top-down. For each heavy path, we do the following.

1. We first use the bitvector that marks the node depths to determine a subrange $[1..r]$ of the alphabet that will be used for the query (the light nodes included in the range will have depths at most i , whereas the nodes in the range $[r + 1..h]$ will have depth more than i).
2. We traverse the wavelet tree of the current heavy path. Such a query will spend time $O(t \log \ell_\alpha)$ for a heavy path with ℓ_α light children, in which t distinct light children appear in the sequence.

It is easy to see that the total space will be $O(n \log^2 D)$ bits, since the alphabet size is $O(\log \ell_\alpha)$ for each node α with n_α stored elements and each element of A will incur at most $\log D$ elements in the wavelet trees stored in the heavy paths of the tree. The query time can be bounded to be $O(\log^2 D)$ per reported document by a similar argument (we traverse $\log D$ heavy paths and each traversal costs $\log D$ time).

We thus obtain the following result.

► **Theorem 9.** *Given a collection of D documents of total length n over alphabet $[1..\sigma]$ so that the documents are organized in a hierarchy of documents represented by a tree of total size Δ , we can build a data structure of size $O(n \log^2 D + \Delta)$ bits of space that can, given a pattern p , find all t categories of documents at level i that have at least one document containing the pattern in total time $O(|p| + t \log^2 D)$.*

4.2.2 Second solution based on heavy path decomposition

Our second solution based on heavy path decomposition will rely on a more fine-grained encoding. We will make use of Huffman-shaped wavelet tree [6] for each heavy path, such that the wavelet tree node corresponding to a light node of relative weight w (the weight light node divided by weight of the root of heavy path) will be encoded using $\log(1/w) + O(1)$ bits and the corresponding wavelet tree leaf will be at depth $\log(1/w) + O(1)$. It is now easy to see that the encoding of each element of A will take $O(\log D)$ bits and, furthermore, the cost of a query can be upper-bounded by just $O(\log D)$. Both bounds rely on a telescoping argument. We have the following result.

► **Theorem 10.** *Given a collection of D documents of total length n over alphabet $[1..\sigma]$ and so that the documents are organized in a hierarchy of documents represented by a tree of total size Δ , we can build a data structure of size $n(\log \sigma(1 + o(1)) + O(\log D)) + O(\Delta) + O(D \log n)$ bits of space that can, given a pattern p , find all t categories of documents at a level i that have at least one document that contains the pattern in total time $O(|p| + t \log D)$.*

5 Compact and compressed data structures for categorical data queries

In this section we explore more space-efficient versions of the problem. More in detail, we are interested in studying the problem under succinct and compressed-space constraints. Namely, our aim is to use $O(n \log \sigma)$ bits for the succinct case and $nH_0 + o(n \log \sigma) + O(D \log n)$ bits of space for the compressed case. To achieve this, we will improve the solution of Section 3. More precisely, we avoid the storage of the document array and simulate direct access to the document array using Lemma 5. As a consequence, we can achieve time $O(\log^\epsilon n)$ to get the given document index $A[i]$ for any $i \in [1..n]$. This will reduce the space to represent the document array from $O(n \log D)$ to $O(n \log \sigma)$ bits. Now the space used by the range minimum query data structures will become the bottleneck. To reduce the space usage we will make use of sparsification. More precisely, we will divide the document array into blocks and sample just the values of the A array that are the smallest in each block. The space becomes $O(n/\alpha)$ bits where α is the sparsification factor. For details on how the sparsification is used to simulate the reporting of distinct documents that appear in interval $A[i..j]$, we refer the reader to [2, 12]. Here we just mention that the time per reported document becomes $O(\alpha \log^\epsilon n)$ and entails $O(\alpha)$ accesses to the document array, each of which requires $O(\log^\epsilon n)$ time. We thus have the following result.

► **Theorem 11.** *Given a parameter $\alpha \geq 1$ and a collection of D documents of total length n over alphabet $[1..\sigma]$ and so that the documents are organized in a hierarchy of documents represented by a tree of height h , we can build a data structure of size $O(n \log \sigma) + O(nh/\alpha)$ bits of space that can, given a pattern p , find all t categories of documents at level i that have at least one document that contains the pattern in total time $O(|p| + t \cdot \alpha \log^\epsilon n)$.*

By setting $\alpha = \lceil \frac{h}{\log \sigma} \rceil$ we get space $O(n \log \sigma)$ bits and query time $O(|p| + (t + 1) \log^\epsilon n \cdot (1 + \frac{h}{\log \sigma}))$. We thus have the following corollary.

► **Corollary 12.** *Given a parameter α and collection of D documents of total length n over alphabet $[1..\sigma]$ and so that the documents are organized in a hierarchy of documents represented by a tree of height h , we can build a data structure of size $O(n \log \sigma)$ bits of space that can, given a pattern p , find all t categories of documents at level i that have at least one document that contains the pattern in total time $O(|p| + (t + 1) \cdot \log^\epsilon n(1 + \frac{h}{\log \sigma}))$.*

Whenever $h = \log D$ (e.g. every internal node is branching), the query time simplifies to $O(|p| + (t+1) \cdot \log_{\sigma} D \cdot \log^{\epsilon} n) \in O(|p| + (t+1) \log^{1+\epsilon} n)$. We can also get compressed space. Namely, we can use a compressed suffix array [9] with query time $\log n \log \log n$ and space $nH_k + o(n)$ to represent the document array. We will combine the compressed suffix array with the alphabet-independent variant of BWT-index presented in [1]. We then get an index that uses space $nH_k + o(n \log \sigma)$ with query time $O(|p|)$ to find the suffix array interval of a pattern and $O(\log n \log \log n)$ time to access an element of the suffix array. Notice that we can translate access to a suffix array element to an access to a document array element using $O(D \log n)$ bits of space. Summing up, we get the following theorem.

► **Theorem 13.** *Given a parameter α and a collection of D documents of total length n over alphabet $[1..\sigma]$ and so that the documents are organized in a hierarchy of documents represented by a tree of height h , we can build a data structure of size $nH_k + o(n \log \sigma) + O(D \log n) + O(nh/\alpha)$ bits of space that can, given a pattern p , find all t categories of documents at level i that have at least one document that contains the pattern in total time $O(|p| + t \cdot \alpha \log n \log \log n)$.*

By setting $\alpha = h \cdot \log \log n$, we get space $nH_k + o(n \log \sigma) + O(D \log n)$ bits and query time $O(|p| + t \cdot h \log n (\log \log n)^2)$. The latter becomes $O(|p| + t \log D \log n (\log \log n)^2)$ whenever $h = O(\log D)$.

6 Conclusions

In this paper, we proposed several solutions for the problem of categorical retrieval. Possible extensions of our work include the case when the document hierarchy is a DAG rather than a tree. This situation occurs, for example, with phylogenetic networks. The solution in Section 3 could easily be extended to DAG structured categories if there was an efficient support for level ancestor queries on DAGs. Other possible extensions includes top-k queries in which categories are either ordered by a static order or by the total frequency of the pattern in the documents that belong to the reported categories.

References

- 1 Djamel Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):23, 2014.
- 2 Djamel Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms (JDA)*, 18:3–13, 2013.
- 3 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing (SICOMP)*, 17(3):427–462, 1988.
- 4 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.
- 5 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing (SICOMP)*, 40(2):465–492, 2011.
- 6 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms (TALG)*, 2(4):611–639, 2006.
- 7 Travis Gagie, Simon J Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–6. Springer, 2009.
- 8 Pawel Gawrychowski, Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Minimal discriminating words problem revisited. In *Proceedings of the 20th International Symposium*

- on *String Processing and Information Retrieval (SPIRE)*, volume 8214 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2013.
- 9 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.
 - 10 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing (SICOMP)*, 35(2):378–407, 2005.
 - 11 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing (SICOMP)*, 13(2):338–355, 1984.
 - 12 Wing-Kai Hon, Rahul Shah, Sharma V Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM (JACM)*, 61(2):1–36, 2014.
 - 13 G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.
 - 14 Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Computing discriminating and generic words. In L. Calderón-Benavides, C.N. González-Caro, E. Chávez, and N. Ziviani, editors, *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *Lecture Notes in Computer Science*, pages 307–317. Springer Verlag, 2012.
 - 15 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing (SICOMP)*, 22(5):935–948, 1993.
 - 16 S. Muthu Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.
 - 17 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms (JDA)*, 25:2–20, 2014.
 - 18 Gonzalo Navarro. *Compact data structures: a practical approach*. University of Cambridge, New York, NY, 2016.
 - 19 Gonzalo Navarro and Kunihiro Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16, 2014.
 - 20 Kunihiro Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms (JDA)*, 5(1):12–22, 2007.
 - 21 Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences (JCSS)*, 26(3):362–391, 1983.
 - 22 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.
 - 23 Derrick E. Wood and Steven L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, March 2014. doi:10.1186/gb-2014-15-3-r46.

Time-Space Tradeoffs for Finding a Long Common Substring

Stav Ben-Nun

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
stav.bennun@live.biu.ac.il

Shay Golan 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
golansh1@cs.biu.ac.il

Tomasz Kociumaka 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Matan Kraus

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
krausma@biu.ac.il

Abstract

We consider the problem of finding, given two documents of total length n , a longest string occurring as a substring of both documents. This problem, known as the LONGEST COMMON SUBSTRING (LCS) problem, has a classic $\mathcal{O}(n)$ -time solution dating back to the discovery of suffix trees (Weiner, 1973) and their efficient construction for integer alphabets (Farach-Colton, 1997). However, these solutions require $\Theta(n)$ space, which is prohibitive in many applications. To address this issue, Starikovskaya and Vildhøj (CPM 2013) showed that for $n^{2/3} \leq s \leq n$, the LCS problem can be solved in $\mathcal{O}(s)$ space and $\tilde{\mathcal{O}}(\frac{n^2}{s})$ time.¹ Kociumaka et al. (ESA 2014) generalized this tradeoff to $1 \leq s \leq n$, thus providing a smooth time-space tradeoff from constant to linear space. In this paper, we obtain a significant speed-up for instances where the length L of the sought LCS is large. For $1 \leq s \leq n$, we show that the LCS problem can be solved in $\mathcal{O}(s)$ space and $\tilde{\mathcal{O}}(\frac{n^2}{L \cdot s} + n)$ time. The result is based on techniques originating from the LCS WITH MISMATCHES problem (Flouri et al., 2015; Charalampopoulos et al., CPM 2018), on space-efficient locally consistent parsing (Birenzweig et al., SODA 2020), and on the structure of maximal repetitions (runs) in the input documents.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases longest common substring, time-space tradeoff, local consistency, periodicity

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.5

Funding Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.

1 Introduction

The LONGEST COMMON SUBSTRING (LCS) problem is a fundamental text processing problem with numerous applications; see e.g. [1, 39, 22]. Given *two* strings (documents) S_1, S_2 , the LCS problem asks for a longest string occurring in S_1 and S_2 . We denote the length of the longest common substring by $\text{lcs}(S_1, S_2)$.

The classic text-book solution to the LCS problem is to build the (generalized) suffix tree of the documents and find the node that corresponds to an LCS [40, 26, 17]. While this can be achieved in linear time, it comes at the cost of using $\Omega(n)$ words (of $\Theta(\log n)$

¹ The $\tilde{\mathcal{O}}$ notation hides $\log^{\mathcal{O}(1)} n$ factors.



bits each) to store the suffix tree. In applications with large amounts of data or strict space constraints, this renders the classic solution impractical. To overcome the space challenge of suffix trees, succinct and compressed data structures have been subject to extensive research [25, 35]. Nevertheless, these data structures still use $\Omega(n)$ bits of space in the worst-case. Starikovskaya and Vildhøj [36] showed that for $n^{2/3} \leq s \leq n$, the LCS problem can be solved in $\mathcal{O}(\frac{n^2}{s} + s \log n)$ time using $\mathcal{O}(s)$ space. Kociumaka et al. [31] subsequently improved the running time to $\mathcal{O}(\frac{n^2}{s})$ and extended the parameter range to $1 \leq s \leq n$.

These previous works also considered a generalized version of the LCS problem, where the input consists of m documents S_1, S_2, \dots, S_m (still of total length n) and an integer $2 \leq d \leq m$. The task there is to compute a longest string occurring as a substring of at least d of the m input documents. In this setting, Starikovskaya and Vildhøj [36] achieve $\mathcal{O}(\frac{n^2 \log^2 n}{s} (d \log^2 n + d^2))$ time and $\mathcal{O}(s)$ space for $n^{2/3} \leq s \leq n$, whereas Kociumaka et al. [31] showed a solution which takes $\mathcal{O}(\frac{n^2}{s})$ time and $\mathcal{O}(s)$ space for $1 \leq s \leq n$. The cost of this algorithm matches both a classic $\Theta(n)$ -space algorithm [27] and the time-space tradeoff for $d = m = 2$. Nevertheless, in this paper we focus on the LCS problem for *two* strings only.

Kociumaka et al. [31] additionally provided a lower bound which states that any deterministic algorithm using $s \leq \frac{n}{\log n}$ space must cost $\Omega(n \sqrt{\log(n/(s \log n))} / \log \log(n/(s \log n)))$ time. This lower bound is actually derived for the problem of distinguishing whether $\text{lcs}(S_1, S_2) = 0$, i.e., deciding if the two input strings have any character in common. This state of affairs naturally leads to a question of whether distinguishing between $\text{lcs}(S_1, S_2) < \ell$ and $\text{lcs}(S_1, S_2) \geq \ell$ gets easier as ℓ increases, or equivalently, whether $L := \text{lcs}(S_1, S_2)$ can be computed more efficiently when L is large. This case is relevant for applications since the existence of short common substrings is less meaningful for measuring string similarity.

1.1 Our Results

We provide new sublinear-space algorithms for the LCS problem optimized for inputs with a long common substring. The algorithms are designed for the word-RAM model with word size $w = \Theta(\log n)$, and they work for integer alphabets $\Sigma = \{1, 2, \dots, n^{\mathcal{O}(1)}\}$. Throughout the paper, the input strings reside in a read-only memory and any space used by the algorithms is a working space; furthermore, we represent the output by witness occurrences in the input strings so that it fits in $\mathcal{O}(1)$ machine words. Our main result is as follows:

► **Theorem 1.** *Given s with $1 \leq s \leq n$, the LCS problem with $L = \text{lcs}(S_1, S_2)$ can be solved deterministically in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n \log^* n}{s \cdot L} + n \log n)$ time,² and in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n}{s \cdot L} + n \log n)$ time with high probability using a Las-Vegas randomized algorithm.*

We remark that Theorem 1 improves upon the result of Kociumaka et al. [31] whenever $s < \frac{n}{\log n}$ and $L > \log n \log^* n$ (or $L > \log n$ if randomization is allowed).

We also show that the log factors can be removed from the running times in Theorem 1 if $s = \Theta(1)$. In fact, this yields an improvement upon Theorem 1 as long as $s < \log n \log^* n$.

► **Theorem 2.** *The LCS problem can be solved deterministically in $\mathcal{O}(1)$ space and $\mathcal{O}(\frac{n^2}{L})$ time, where $L = \text{lcs}(S_1, S_2)$.*

As a step towards our main result, we solve the LCS_ℓ problem defined below.

² The *iterated logarithm* function \log^* is formally defined with $\log^* x = 0$ for $x \leq 1$ and $\log^* x = 1 + \log^*(\log x)$ for $x > 1$. In other words $\log^* n$ is the smallest integer $k \geq 0$ such that $\log^{(k)} x \leq 1$.

LONGEST COMMON SUBSTRING WITH THRESHOLD ℓ (LCS_ℓ)

Input: Two strings S_1 and S_2 (of length at most n), an integer threshold $\ell \in [n]$

Output: A common substring G of S_1 and S_2 such that:

1. if $\ell \leq \text{lcs}(S_1, S_2) < 2\ell$, then $|G| = \text{lcs}(S_1, S_2)$;
2. if $\text{lcs}(S_1, S_2) \geq 2\ell$, then $|G| \geq 2\ell$.

If $\text{lcs}(S_1, S_2) < \ell$, then LCS_ℓ allows for an arbitrary common substring in the output.

► **Remark 3.** Note the following equivalent characterization of the output G of LCS_ℓ : for every common substring T with $\ell \leq |T| \leq 2\ell$, the common substring G is of length $|G| \geq |T|$.

► **Theorem 4.** *The LCS_ℓ problem can be solved deterministically in $\mathcal{O}(\frac{n \log^* n}{\ell})$ space and $\mathcal{O}(n \log n)$ time, and in $\mathcal{O}(\frac{n}{\ell})$ space and $\mathcal{O}(n \log n)$ time with high probability using a Las-Vegas randomized algorithm.*

1.2 Related work

The LCS problem has been studied in many other settings. Babenko and Starikovskaya [6], Flouri et al. [20], Thankachan et al. [38], and Kociumaka et al. [30] studied the LCS with k mismatches problem, where the occurrences in S_1 and S_2 can be at Hamming distance up to k . Charalampopoulos et al. [11] showed that this problem becomes easier when the strings have a long common substring with k mismatches (similarly to what we obtain for LCS with no mismatches). Thankachan et al. [37] and Ayad et al. [5] considered a related problem with edit distance instead of the Hamming distance. Alzamel et al. [2] proposed an $\tilde{\mathcal{O}}(n)$ -time algorithm for the Longest Common Circular Substring problem, where occurrences in S_1 and S_2 can be cyclic rotations of each other.

Amir et al. [3] studied the problem of answering queries asking for the LCS after a single edit in either of the two original input strings. Subsequently, Amir et al. [4] and Charalampopoulos et al. [12] considered a fully dynamic version of the problem, in which the edit operations are applied sequentially, ultimately achieving $\tilde{\mathcal{O}}(1)$ time per operation.

1.3 Algorithmic Overview

We first give an overview of the algorithm of Theorem 4. Then, we derive Theorem 1 in two steps, with an $\mathcal{O}(s)$ -space solution to the LCS_ℓ problem as an intermediate result.

An $\tilde{\mathcal{O}}(n/\ell)$ -space algorithm for the LCS_ℓ problem. In Section 3, we define an *anchored* variant of the LONGEST COMMON SUBSTRING problem (LCAS). In the LCAS problem, we are given two strings S_1, S_2 and sets of positions A_1 and A_2 , and we wish to find a longest common substring which can be obtained by extending (to the left and to the right) $S_1[p_1]$ and $S_2[p_2]$ for some $(p_1, p_2) \in A_1 \times A_2$. We then reduce the LCAS problem to the TWO STRING FAMILIES LCP problem, introduced by Charalampopoulos et al. [11] in the context of finding LCS with mismatches.

In Section 4, we show how to solve the LCS_ℓ problem by selecting positions in A_1 and A_2 so that *every* common substring T of S_1 and S_2 with $|T| \geq \ell$ can be obtained by extending $S_1[p_1]$ and $S_2[p_2]$ for some $(p_1, p_2) \in A_1 \times A_2$. To make this selection, we use *partitioning sets* by Birenzweig et al. [9], which consist of $\tilde{\mathcal{O}}(\frac{n}{\ell})$ positions chosen in a locally consistent manner. However, since partitioning sets do not select positions in long periodic regions, our algorithms use *maximal repetitions* (runs) [32, 7] and their *Lyndon roots* [13] to add $\mathcal{O}(\frac{n}{\ell})$ extra positions. Overall, we get an $\tilde{\mathcal{O}}(\frac{n}{\ell})$ -space and $\tilde{\mathcal{O}}(n)$ -time algorithm for the LCS_ℓ problem.

An $\mathcal{O}(s)$ space algorithm for the LCS_ℓ problem. In Section 5, we give a time-space tradeoff for the LCS_ℓ problem. The algorithm partitions the input strings into overlapping substrings, executes the algorithm of Section 4 for each pair of substrings, and returns the longest among the common substrings obtained from these calls. For a tradeoff parameter $1 \leq s \leq n$, the algorithm takes $\mathcal{O}(s)$ space and $\tilde{\mathcal{O}}(\frac{n^2}{s\ell} + n)$ time.

A solution to the LCS problem. In Section 6, we show how to search for LCS by repeatedly solving the LCS_ℓ problem with different choices of ℓ . We get an algorithm that takes $\mathcal{O}(s)$ space and $\tilde{\mathcal{O}}(\frac{n^2}{sL} + n)$ time, where $L = \text{lcs}(S_1, S_2)$, as stated in Theorem 1.

2 Preliminaries

For $1 \leq i < j \leq n$, denote the integer *intervals* $[i..j] = \{i, i+1, \dots, j\}$ and $[k] = \{1, 2, \dots, k\}$.

A string S of length $n = |S|$ is a finite sequence of characters $S[1]S[2] \cdots S[n]$ over an alphabet Σ ; in this paper, we consider polynomially-bounded integer alphabets $\Sigma = [1..n^{\mathcal{O}(1)}]$. The string $S^r = S[n]S[n-1] \cdots S[1]$ is called the *reverse* of the string S .

A string T is a *substring* of a string S if $T = S[x]S[x+1] \cdots S[y]$ for some $1 \leq x \leq y \leq |S|$. We then say that T *occurs* in S at position x , and we denote the *occurrence* by $S[x..y]$. We call $S[x..y]$ a *fragment* of S . A fragment $S[x..y]$ is a *prefix* of S if $x = 1$ and a *suffix* of S if $y = |S|$. These special fragments are also denoted by $S[.y]$ and $S[x.]$, respectively. A *proper fragment* of S is any fragment other than $S[1..|S|]$. A common prefix (suffix) of two strings S_1, S_2 is a string that occurs as a prefix (resp. suffix) of both S_1 and S_2 . The longest common prefix of S_1 and S_2 is denoted by $\text{LCP}(S_1, S_2)$, and the longest common suffix is denoted by $\text{LCP}^r(S_1, S_2)$. Note that $\text{LCP}^r(S_1, S_2) = (\text{LCP}(S_1^r, S_2^r))^r$.

An integer $k \in [|S|]$, is a *period* of a string S if $S[i] = S[i+k]$ for $i \in [|S| - k]$. The shortest period of S is denoted by $\text{per}(S)$. If $\text{per}(S) \leq \frac{1}{2}|S|$, we say that S is *periodic*. A periodic fragment $S[i..j]$ is called a *run* [32, 7] if it cannot be extended (to the left nor to the right) without increasing the shortest period. For a pair of parameters d and ρ , we say that a run $S[i..j]$ is a (d, ρ) -*run* if $|S[i..j]| \geq d$ and $\text{per}(S[i..j]) \leq \rho$. Note that every periodic fragment $S[i'..j']$ with $|S[i'..j']| \geq d$ and $\text{per}(S[i'..j']) \leq \rho$ can be uniquely extended to a (d, ρ) -run $S[i..j]$ while preserving the shortest period $\text{per}(S[i..j]) = \text{per}(S[i'..j'])$.

Tries and suffix trees. Given a set of strings \mathcal{F} , the compact trie [34] of these strings is the tree obtained by compressing each path of nodes of degree one in the trie [10, 21] of the strings in \mathcal{F} , which takes $\mathcal{O}(|\mathcal{F}|)$ space. Each edge in the compact trie has a label represented as a fragment of a string in \mathcal{F} . The suffix tree [40] of a string S is the compact trie of all the suffixes of S . The sparse suffix tree [29, 8, 28, 24] of a string S is the compact trie of selected suffixes $\{S[i..] : i \in B\}$ specified by a set of positions $B \subseteq [|S|]$.

3 Longest Common Anchored Substring problem

In this section, we consider an *anchored* variant of the LONGEST COMMON SUBSTRING problem. Let A_1 and A_2 be sets of distinguished positions, called *anchors*, in strings S_1 and S_2 , respectively. We say that a string T is a *common anchored substring* of S_1 and S_2 with respect to A_1 and A_2 if it has occurrences $S_1[i_1..j_1] = T = S_2[i_2..j_2]$ with a *synchronized pair of anchors*, i.e., with some anchors $p_1 \in A_1$ and $p_2 \in A_2$ such that $p_1 - i_1 = p_2 - i_2 \in [0, |T|]$.³

³ Note that the anchors could be at positions $p_1 = j_1 + 1$ and $p_2 = j_2 + 1$ (if $p_1 - i_1 = p_2 - i_2 = |T|$).

LONGEST COMMON ANCHORED SUBSTRING (LCAS)

Input: Two strings S_1, S_2 (of length at most n) and two sets of anchors $A_1 \subseteq [|S_1|]$, $A_2 \subseteq [|S_2|]$.

Output: A longest common anchored substring of S_1 and S_2 with respect to A_1, A_2 .

We utilize the following characterization of the longest common anchored substring.

► **Fact 5.** *The length of a longest common anchored substring of two strings S_1 and S_2 (with respect to anchors at A_1 and A_2) is*

$$\max\{|\text{LCP}(S_1[p_1..], S_2[p_2..])| + |\text{LCP}^r(S_1[..p_1-1], S_2[..p_2-1])| : p_1 \in A_1, p_2 \in A_2\}.$$

Moreover, for any $(p_1, p_2) \in A_1 \times A_2$ maximizing this expression, a longest common anchored substring is the concatenation $\text{LCP}^r(S_1[..p_1-1], S_2[..p_2-1]) \cdot \text{LCP}(S_1[p_1..], S_2[p_2..])$.

The characterization of Fact 5 lets us use the following problem, originally defined in a context of computing LCS with mismatches.

TWO STRING FAMILIES LCP (Charalampopoulos et al. [11])

Input: A compact trie $\mathcal{T}(\mathcal{F})$ of a family of strings \mathcal{F} and two sets $P, Q \subseteq \mathcal{F}^2$.

Output: The value $\text{maxPairLCP}(P, Q)$, defined as

$$\text{maxPairLCP}(P, Q) = \max\{|\text{LCP}(P_1, Q_1)| + |\text{LCP}(P_2, Q_2)| : (P_1, P_2) \in P, (Q_1, Q_2) \in Q\},$$

along with pairs $(P_1, P_2) \in P$ and $(Q_1, Q_2) \in Q$ for which the maximum is attained.

Charalampopoulos et al. [11] observed that the TWO STRING FAMILIES LCP problem can be solved using an approach by Crochemore et al. [14] and Flouri et al. [20].

► **Lemma 6** ([11, Lemma 3]). *The TWO STRING FAMILIES LCP problem can be solved in $\mathcal{O}(|\mathcal{F}| + N \log N)$ time using $\mathcal{O}(|\mathcal{F}| + N)$ space⁴, where $N = |P| + |Q|$.*

By Fact 5, the LCAS problem reduces to the TWO STRING FAMILIES LCP problem with:

$$\mathcal{F} = \{S_1[p..] : p \in A_1\} \cup \{(S_1[..p-1])^r : p \in A_1\} \\ \cup \{S_2[p..] : p \in A_2\} \cup \{(S_2[..p-1])^r : p \in A_2\}, \quad (1)$$

$$P = \{(S_1[p..], (S_1[..p-1])^r) : p \in A_1\}, \quad (2)$$

$$Q = \{(S_2[p..], (S_2[..p-1])^r) : p \in A_2\}. \quad (3)$$

The following theorem provides an efficient implementation of this reduction. The most challenging step, to construct the compacted trie $\mathcal{T}(\mathcal{F})$, is delegated to the work of Birenzweige et al. [9], who show that a sparse suffix tree of a length- n string S with $B \subseteq [n]$ can be constructed deterministically in $\mathcal{O}(n \log \frac{n}{|B|})$ time and $\mathcal{O}(|B| + \log n)$ space.

► **Theorem 7.** *The LONGEST COMMON ANCHORED SUBSTRING problem can be solved in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(|A_1| + |A_2| + \log n)$ space.*

⁴ The original formulation of [11, Lemma 3] does not discuss the space complexity. However, an inspection of the underlying algorithm, described in [14, 20], easily yields this additional claim.

Proof. We implicitly create a string $S = S_1\$S_1^r\$S_2\$S_2^r$ and construct a sparse suffix tree of S containing the following suffixes: $S_1[p..]\$S_1^r\$S_2\$S_2^r$ and $(S_1[. . p - 1])^r\$S_2\S_2^r for $p \in A_1$, as well as $S_2[p..]\$S_2^r$ and $(S_2[. . p - 1])^r$ for $p \in A_2$. We then trim this tree, cutting edges immediately above any $\$$ on their labels, which results in the compacted trie $\mathcal{T}(\mathcal{F})$ for the family \mathcal{F} defined in (1). We then build P and Q according to (2) and (3), respectively, and solve an instance of the TWO STRING FAMILIES LCP problem specified by $\mathcal{T}(\mathcal{F}), P, Q$. This yields pairs in P and Q for which $\max\text{PairLCP}(P, Q)$ is attained. We retrieve the underlying indices $p_1 \in A_1$ and $p_2 \in A_2$ and derive a longest common anchored substring of S_1 and S_2 according to Fact 5: $\text{LCP}^r(S_1[. . p_1 - 1], S_2[. . p_2 - 1]) \cdot \text{LCP}(S_1[p_1..], S_2[p_2..])$.

With the sparse suffix tree construction of [9] and the algorithm of Lemma 6 that solves the TWO STRING FAMILIES LCP problem, the overall running time is $\mathcal{O}(n \log \frac{n}{N} + N \log N) = \mathcal{O}(n \log n)$ and the space complexity is $\mathcal{O}(N + \log n)$, where $N = |A_1| + |A_2|$. ◀

4 Space-efficient $\tilde{\mathcal{O}}(n)$ -time algorithm for the $LC S_\ell$ problem

Our approach to solve the $LC S_\ell$ problem is via a reduction to the LCAS problem. For this, we wish to select anchors $A_1 \subseteq [|S_1|]$ and $A_2 \subseteq [|S_2|]$ so that every common substring T of length at least ℓ is a common anchored substring. In other words, we need to make sure that T admits occurrences $S_1[i_1..j_1] = T = S_2[i_2..j_2]$ with a synchronized pair of anchors.

As a warm-up, we describe a simple selection of $\mathcal{O}(n/\sqrt{\ell})$ anchors based on *difference covers* [33], which have already been used by Starikovskaya and Vildhøj [36] in a time-space tradeoff for the LCS problem. For every two integers τ, m with $1 \leq \tau \leq m$, this technique yields a set $DC_\tau(m) \subseteq [m]$ of size $\mathcal{O}(m/\sqrt{\tau})$ such that for every two indices $i_1, i_2 \in [m - \tau + 1]$, there is a shift $\Delta \in [0.. \tau - 1]$ such that $i_1 + \Delta$ and $i_2 + \Delta$ both belong to $DC_\tau(m)$. Hence, to make sure that every common substring of length at least ℓ is anchored, it suffices to select all $\mathcal{O}(n/\sqrt{\ell})$ positions in $DC_\ell(n)$ as anchors: $A_1 = [|S_1|] \cap DC_\ell(n)$ and $A_2 = [|S_2|] \cap DC_\ell(n)$.

We remark that such selection of anchors is *non-adaptive*: it does not depend on contents of the strings S_1 and S_2 , but only on the lengths of these strings (and the parameter ℓ). In fact, any non-adaptive construction needs $\Omega(n/\sqrt{\ell})$ anchors in order to guarantee that every common substring T of length at least ℓ is a common anchored substring. In the following, we show how adaptivity allows us to achieve the same goal using only $\tilde{\mathcal{O}}(n/\ell)$ anchors.

4.1 Selection of Anchors: the non-periodic case

We first show how to accommodate common substrings T of length $|T| \geq \ell$ that do not contain a $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run. The idea is to use *partitioning sets* by Birenzwise et al. [9].

► **Definition 8** (Birenzwise et al. [9]). *A set of positions $P \subseteq [n]$ is called a (τ, δ) -partitioning set of a length- n string S , for some parameters $\tau, \delta \in [n]$, if it has the following properties:*

Local Consistency: *For every two indices $i, j \in [1 + \delta.. n - \delta]$ such that $S[i - \delta.. i + \delta] = S[j - \delta.. j + \delta]$, we have $i \in P$ if and only if $j \in P$.*

Compactness: *If $p_i < p_{i+1}$ are two consecutive positions in $P \cup \{1, n + 1\}$ such that $p_{i+1} > p_i + \tau$, then $u = S[p_i.. p_{i+1} - 1]$ is periodic with period $\text{per}(u) \leq \tau$.*

Note that any (τ, δ) -partitioning set is also a (τ', δ') -partitioning set for any $\tau' \geq \tau$ and $\delta' \geq \delta$. The selection of anchors is based on an arbitrary $(\frac{1}{5}\ell, \frac{1}{5}\ell)$ -partitioning set P of the string $S_1 S_2$: for every position $p \in P$, p is added to A_1 (if $p \leq |S_1|$) or $p - |S_1|$ is added to A_2 (otherwise). Below, we show that this selection satisfies the advertised property.

► **Lemma 9.** *Let T be a common substring of length $|T| \geq \ell$ which does not contain a $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run. Then, T is a common anchored substring with respect to A_1, A_2 defined above.*

Proof. Let $S_1[i_1 \dots j_1]$ and $S_2[i_2 \dots j_2]$ be arbitrary occurrences of T in S_1 and S_2 , respectively. If there is a position $p_1 \in A_1$ with $p_1 \in [i_1 + \delta \dots j_1 - \delta]$, then the position $p_2 = i_2 + (p_1 - i_1)$ belongs to A_2 by the local consistency property of the underlying partitioning set, due to $S_1[p_1 - \delta \dots p_1 + \delta] = S_2[p_2 - \delta \dots p_2 + \delta]$. Hence, (p_1, p_2) is a synchronized pair of anchors and T is a common anchored substring with respect to A_1, A_2 .

If there is no such position $p_1 \in A_1$, then $S_1[i_1 + \delta \dots j_1 - \delta]$ is contained within a block between two consecutive positions of the partitioning set. The length of this block is at least $|T| - 2\delta \geq \frac{3}{5}\ell > \tau$, so the block is periodic by the compactness property of the partitioning set. Hence, $\text{per}(T[1 + \delta \dots |T| - \delta]) = \text{per}(S_1[i_1 + \delta \dots j_1 - \delta]) \leq \tau \leq \frac{1}{5}\ell$. A $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run in T can thus be obtained by maximally extending $T[1 + \delta \dots |T| - \delta]$ without increasing the shortest period. Such a run in T is a contradiction to the assumption. ◀

Birenzweige et al. [9] gave a deterministic algorithm that constructs a $(\tau, \tau \log^* n)$ -partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n \log \tau)$ time using $\mathcal{O}(\frac{n}{\tau} + \log \tau)$ space. Setting appropriate $\tau = \Theta(\ell / \log^* n)$, we get an $(\frac{1}{5}\ell, \frac{1}{5}\ell)$ -partitioning set of size $\mathcal{O}(\frac{n \log^* n}{\ell})$.

Furthermore, Birenzweige et al. [9] gave a Las-Vegas randomized algorithm that constructs a (τ, τ) -partitioning set of size $\mathcal{O}(\frac{n}{\tau})$ in $\mathcal{O}(n + \tau \log^2 n)$ time with high probability, using $\mathcal{O}(\frac{n}{\tau} + \log n)$ space. Setting $\tau = \frac{1}{5}\ell$, we get an $(\frac{1}{5}\ell, \frac{1}{5}\ell)$ -partitioning set of size $\mathcal{O}(\frac{n}{\ell})$.

4.2 Selection of anchors: the periodic case

In this section, for any parameters $d, \rho \in [n]$ satisfying $d \geq 3\rho - 1$, we show how to accommodate all common substrings containing a (d, ρ) -run by selecting $\mathcal{O}(\frac{n}{d})$ anchors. This method is then used for $d = \frac{3}{5}\ell$ and $\rho = \frac{1}{5}\ell$ to complement the selection in Section 4.1.

Let T be a common substring of S_1 and S_2 containing a (d, ρ) -run. We consider two cases depending on whether the run is a proper fragment of T or the whole T . In the first case, it suffices to select as anchors the first and the last position of every (d, ρ) -run.

► **Lemma 10.** *Let A_1 and A_2 contain the boundary positions of every (d, ρ) -run in S_1 and S_2 , respectively. If T is a common substring of S_1 and S_2 with a (d, ρ) -run r as a proper fragment, then T is a common anchored substring of S_1 and S_2 with respect to A_1, A_2 .*

Proof. In the proof, we assume that $r = T[i \dots j]$ with $i \neq 1$. The case of $j \neq |T|$ is symmetric.

Suppose that an occurrence of T in S_1 starts at position i_1 . The fragment matching r , i.e., $S_1[i_1 + i - 1 \dots i_1 + j - 1]$, is periodic, has length at least d and period at most ρ , so it can be extended to a (d, ρ) -run in S_1 . This run in S_1 starts at position $i_1 + i - 1$ due to $T[i - 1] \neq T[i + \text{per}(r) - 1]$, so $p_1 := i_1 + i - 1 \in A_1$. The same argument shows that $p_2 := i_2 + i - 1 \in A_2$ if T occurs in S_2 at position i_2 . Hence, (p_1, p_2) is a synchronized pair of anchors and T is a common anchored substring with respect to A_1, A_2 . ◀

We are left with handling the case when the whole T is a (d, ρ) -run, i.e., when T is periodic with $|T| \geq d$ and $\text{per}(T) \leq \rho$. In this case, we cannot guarantee that every pair of occurrences of T in A_1 and A_2 has a synchronized pair of anchors. For example, if $T = \mathbf{a}^d$ and $S_1 = S_2 = \mathbf{a}^n$ with $n \geq 2d$, this would require $\Omega(n/\sqrt{d})$ anchors. (There are $\Omega(n^2)$ pairs of occurrences, and each pair of anchors can accommodate at most $d + 1$ out of these pairs.)

Hence, we focus on the leftmost occurrences of T and observe that they start within the first $\text{per}(T)$ positions of (d, ρ) -runs. To achieve synchronization in these regions, we utilize the notion of the *Lyndon root* [13] $\text{lyn}(X)$ of a periodic string X , defined as the lexicographically smallest rotation of $X[1 \dots \text{per}(X)]$. For each (d, ρ) -run x , we select as anchors the leftmost two positions where $\text{lyn}(x)$ occurs within x (they must exist due to $d \geq 3\rho - 1$).

► **Lemma 11.** *Let A_1 and A_2 contain the first two positions where the Lyndon root occurs within each (d, ρ) -run of S_1 and S_2 , respectively. If T is a common substring of S_1 and S_2 such that the whole T is a (d, ρ) -run, then T is a common anchored substring of S_1 and S_2 .*

Proof. Let k be the leftmost position where $\text{lyn}(T)$ occurs in T and $T = S_1[i_1 \dots j_1]$ be the leftmost occurrence of T in S_1 . Since T is a (d, ρ) -run, $S_1[i_1 \dots j_1]$ can be extended to a (d, ρ) -run x in S_1 . Note that $S_1[i_1 \dots j_1]$ starts within the first $\text{per}(T)$ positions of x ; otherwise, T would also occur at position $i_1 - \text{per}(T)$. Consequently, position $i_1 + k - 1$ is among the first $2\text{per}(T)$ positions of x , and it is a starting position of $\text{lyn}(x) = \text{lyn}(T)$. As the subsequent occurrences of $\text{lyn}(x)$ within x start $\text{per}(T)$ positions apart, we conclude that $i_1 + k - 1$ is one of the first two positions where $\text{lyn}(x)$ occurs within x . Thus, $p_1 := i_1 + k - 1 \in A_1$. Symmetrically, $p_2 := i_2 + k - 1$ is added to A_2 . Hence, (p_1, p_2) is a synchronized pair of anchors and T is a common anchored substring with respect to A_1, A_2 . ◀

It remains to prove that Lemmas 10 and 11 yield $\mathcal{O}(\frac{n}{d})$ anchors and that this selection can be implemented efficiently. We use the following procedure as a subroutine:

► **Lemma 12** (Kociumaka et al. [19, Lemma 6]). *Given a string S , one can decide in $\mathcal{O}(|S|)$ time and $\mathcal{O}(1)$ space if S is periodic and, if so, compute $\text{per}(S)$.*

First, we bound the number of (d, ρ) -runs and explain how to generate them efficiently.

► **Lemma 13.** *Consider a string S of length n and positive integers ρ, d with $3\rho - 1 \leq d \leq n$. The number of (d, ρ) -runs in S is $\mathcal{O}(\frac{n}{d})$. Moreover, there is an $\mathcal{O}(1)$ -space $\mathcal{O}(n)$ -time deterministic algorithm reporting them one by one along with their periods.*

Proof. Consider all fragments $u_k = S[k\rho \dots (k+2)\rho - 1]$ with boundaries within $[n]$. Observe that each (d, ρ) -run v contains at least one of the fragments u_k : if $v = S[i \dots j]$, then u_k with $k = \lceil i/\rho \rceil$ starts at $k\rho \geq i$ and ends at $(k+2)\rho - 1 \leq i + \rho - 1 + 2\rho - 1 = i + 3\rho - 2 \leq i + d - 1 \leq j$. Moreover, if v contains u_k , then u_k is periodic with period $\text{per}(u_k) = \text{per}(v) \leq \rho = \frac{1}{2}|u_k|$ (the first equality is due to $|u_k| = 2\rho \geq 2\text{per}(v)$ and the periodicity lemma [18]), and v can be obtained by maximally extending u_k without increasing the shortest period.

This leads to a simple algorithm generating all (d, ρ) -runs, which processes subsequent integers k as follows: First, apply Lemma 12 to test if u_k is periodic and retrieve its period ρ_k . If this test is successful, then maximally extend u_k while preserving the period ρ_k , and denote the resulting fragment by v_k . If $|v_k| \geq d$, then report v_k as a (d, ρ) -run. We also introduce the following optimization: after processing k , skip all indices $k' > k$ for which $u_{k'}$ is still contained in v_k . (These indices k' are irrelevant due to $v_{k'} = v_k$ and they form an integer interval.)

The algorithm of Lemma 12 takes constant space and $\mathcal{O}(|u_k|)$ time, which sums up to $\mathcal{O}(n)$ across all indices k . The naive extension of u_k to v_k takes constant space and $\mathcal{O}(|v_k|)$ time. Due to the optimization, no two explicitly generated extensions v_k contain the same fragment $u_{k'}$. Hence, the total length of the fragments v_k (across indices k which were not skipped) is $\mathcal{O}(n)$. Thus, the overall running time is $\mathcal{O}(n)$ and the number of runs reported is $\mathcal{O}(\frac{n}{d})$. ◀

We conclude with a complete procedure generating anchors in the periodic case.

► **Proposition 14.** *There exists an $\mathcal{O}(1)$ -space $\mathcal{O}(n)$ -time algorithm that, given two strings S_1, S_2 of total length n , and parameters $d, \rho \in [n]$ with $d \geq 3\rho - 1$, outputs sets A_1, A_2 of size $\mathcal{O}(\frac{n}{d})$ satisfying the following property: If T is a common substring of S_1 and S_2 containing a (d, ρ) -run, then T is a common anchored substring of S_1 and S_2 with respect to A_1, A_2 .*

Proof. The algorithm first uses the procedure of Lemma 13 to retrieve all (d, ρ) -runs in S_1 along with their periods. For each (d, ρ) -run $S_1[i..j]$, Duval's algorithm [16] is applied to find the minimum cyclic rotation of $S_1[i..i + \text{per}(S_1[i..j]) - 1]$ in order to determine the Lyndon root $\text{lyn}(S_1[i..j])$ represented by its occurrence at position $i + \Delta$ of S_1 . Positions i , $i + \Delta$, $i + 2\Delta$, and j are reported as anchors in A_1 . The same procedure is repeated for S_2 resulting in the elements of A_2 being reported one by one.

The space complexity of this algorithm is $\mathcal{O}(1)$, and the running time is $\mathcal{O}(n)$ (for Lemma 13) plus $\mathcal{O}(\rho) = \mathcal{O}(d)$ per (d, ρ) -run (for Duval's algorithm). This sums up to $\mathcal{O}(n)$ as the number of (d, ρ) -runs is $\mathcal{O}(\frac{n}{d})$. For the same reason, the number of anchors is $\mathcal{O}(\frac{n}{d})$.

For each T , the anchors satisfy the required property due to Lemma 10 or Lemma 11, depending on whether the (d, ρ) run contained in T is a proper fragment of T or not. ◀

4.3 $\tilde{\mathcal{O}}(n/\ell)$ -space algorithm for arbitrary ℓ

The LCS_ℓ problem reduces to an instance of the LCAS problem with a combination of anchors for the non-periodic case and the periodic case. This yields the following result:

► **Theorem 15.** *The LCS_ℓ problem can be solved deterministically in $\mathcal{O}(\frac{n \log^* n}{\ell} + \log n)$ space and $\mathcal{O}(n \log n)$ time, and in $\mathcal{O}(\frac{n}{\ell} + \log n)$ space and $\mathcal{O}(n \log n + \ell \log^2 n)$ time with high probability using a Las-Vegas randomized algorithm.*

Proof. The algorithm first selects anchors A_1 and A_2 based on a $(\frac{1}{5}\ell, \frac{1}{5}\ell)$ -partitioning set, as described in Section 4.1 (the partitioning set can be constructed using a deterministic or a randomized procedure). Additional anchors A'_1 and A'_2 are selected using Proposition 14 with $d = \frac{3}{5}\ell$ and $\rho = \frac{1}{5}\ell$. Finally, the algorithm runs the procedure of Theorem 7 with anchors $A_1 \cup A'_1$ and $A_2 \cup A'_2$ and forwards the obtained result to the output.

With this selection of anchors, every common substring T of length $|T| \geq \ell$ is a common anchored substring. Depending on whether T contains a $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run or not, this follows from Proposition 14 and Lemma 9, respectively. Consequently, the solution to the LCAS problem is a common substring of length at least $|T|$.

Proposition 14 yields $\mathcal{O}(\frac{n}{\ell})$ anchors whereas a partitioning set yields $\mathcal{O}(\frac{n \log^* n}{\ell})$ or $\mathcal{O}(\frac{n}{\ell})$ anchors, depending on whether a deterministic or a randomized construction is used. Consequently, the space and time complexity is as stated in the theorem, with the cost dominated by both the partitioning set construction and the algorithm of Theorem 7. ◀

► **Remark 16.** Note that the algorithms of Theorem 15 return a longest common substring as long as $\text{lcs}(S_1, S_2) \geq \ell$ (and not just when $\ell \leq \text{lcs}(S_1, S_2) \leq 2\ell$ as LCS_ℓ requires).

4.4 $\mathcal{O}(1)$ -space algorithm for $\ell = \Omega(n)$

In Theorem 15, the space usage involves an $\mathcal{O}(\log n)$ term, which becomes dominant for very large ℓ . In this section, we design an alternative $\mathcal{O}(1)$ -space algorithm for $\ell = \Omega(n)$. Later, in Theorem 19, we generalize this algorithm to arbitrary ℓ , which lets us obtain an analog of Theorem 15 with the $\mathcal{O}(\log n)$ term removed from the space complexity.

Our main tool is a constant-space pattern matching algorithm.

► **Lemma 17** (Galil-Seiferas [23], Crochemore-Perrin [15]). *There exists an $\mathcal{O}(1)$ -space $\mathcal{O}(|P| + |T|)$ -time algorithm that, given a read-only pattern P and a read-only text T , reports the occurrences of P in T in the left-to-right order.*

► **Lemma 18.** *The LCS_ℓ problem can be solved deterministically in $\mathcal{O}(1)$ space and $\mathcal{O}(n)$ time for $\ell = \Omega(n)$.*

Proof. We show how to find $\mathcal{O}(1)$ anchors such that if T is a common substring of S_1 and S_2 of length $|T| \geq \ell$, then T is a common anchored substring of S_1 and S_2 .

We first use Proposition 14 with $d = \frac{3}{5}\ell$ and $\rho = \frac{1}{5}\ell$ to generate anchors A'_1 and A'_2 for the periodic case. The set of these anchors has a size of $\mathcal{O}(\frac{n}{d}) = \mathcal{O}(1)$, and, if T contains a $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run, then T is a common anchored substring of S_1 and S_2 with respect to A'_1, A'_2 .

In order to accommodate the case where T does not contain any $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run, we construct sets A_1 and A_2 as follows. Consider all the fragments $u_k = S_1[k\frac{\ell}{5} \dots (k+3)\frac{\ell}{5} - 1]$ with boundaries within $[n]$. For each such fragment, add $k\frac{\ell}{5}$ into A_1 . In addition, use Lemma 17 to find all occurrences of u_k in S_2 , and add all the starting positions of the occurrences to A_2 , unless the number of occurrences exceeds $\frac{n}{\ell/5}$ (then, $\text{per}(u_k) \leq \frac{\ell}{5}$). The number of fragments u_k is $\mathcal{O}(\frac{n}{\ell/5}) = \mathcal{O}(1)$, so the sets A_1 and A_2 contain $\mathcal{O}(1)$ elements.

Let $T = S_1[i_1 \dots j_1] = S_2[i_2 \dots j_2]$ be arbitrary occurrences of T in S_1 and S_2 , respectively. Then for $k = \lceil \frac{i_1}{\ell/5} \rceil$, the fragment u_k is contained within $S_1[i_1 \dots j_1]$. If $\text{per}(u_k) \leq \frac{1}{5}\ell$, then the occurrence of u_k in T can be extended to a $(\frac{3}{5}\ell, \frac{1}{5}\ell)$ -run in T (and that case has been accommodated using A'_1 and A'_2). Otherwise, $p_1 := k\frac{\ell}{5} \in A_1$ and all the positions where u_k occurs in S_2 , including $p_2 := i_2 + (k\frac{\ell}{5} - i_1)$, are in A_2 . Therefore, (p_1, p_2) is a synchronized pair of anchors and T is a common anchored substring with respect to A_1, A_2 .

The number of pairs $(p_1, p_2) \in (A_1 \cup A'_1) \times (A_2 \cup A'_2)$ is $\mathcal{O}(1)$. For each such pair, the algorithm computes $|\text{LCP}(S_1[p_1 \dots], S_2[p_2 \dots])| + |\text{LCP}^r(S_1[\dots p_1 - 1], S_2[\dots p_2 - 1])|$ naively, and returns the common substring corresponding to a maximum among these values. The computation for each pair takes $\mathcal{O}(n)$ time. By the argument above, the algorithm finds a common substring of length at least $|T|$ for every common substring T with $|T| \geq \ell$. \blacktriangleleft

5 Time-space tradeoff for the LCS_ℓ problem

In this section, we show how to use the previous algorithms in order to solve the LCS_ℓ problem in space $\mathcal{O}(s)$, where s is a tradeoff parameter specified on the input. Our approach relies on the following algorithm which, given a parameter $m \geq \ell$, reduces a single arbitrary instance of LCS_ℓ to $\mathcal{O}(\lceil \frac{n}{m} \rceil^2)$ instances of LCS_ℓ with strings of length $\mathcal{O}(m)$.

Algorithm 1 Self-reduction of LCS_ℓ to many instances on strings of length $\mathcal{O}(m)$.

```

1 foreach  $q_1 \in [|S_1|]$  s.t.  $q_1 \equiv 1 \pmod{m}$  and  $q_2 \in [|S_2|]$  s.t.  $q_2 \equiv 1 \pmod{m}$  do
2   | Solve  $LCS_\ell$  on  $S_1[q_1 \dots \min(q_1 + 3m - 1, |S_1|)]$  and  $S_2[q_2 \dots \min(q_2 + 3m - 1, |S_2|)]$ ;
3 return the longest among the common substrings reported;
    
```

Algorithm 1 clearly reports a common substring of S_1 and S_2 . Moreover, if T is a common substring of S_1 and S_2 satisfying $\ell \leq |T| \leq 2\ell$, then T is contained in one of the considered pieces $S_1[q_1 \dots \min(q_1 + 3m - 1, |S_1|)]$ (the one with $q_1 = 1 + m\lfloor \frac{i_1}{m} \rfloor$ if $T = S_1[i_1 \dots j_1]$) and $T = S_2[i_2 \dots j_2]$ is contained in one of the considered pieces $S_2[q_2 \dots \min(q_2 + 3m - 1, |S_2|)]$ (the one with $q_2 = 1 + m\lfloor \frac{i_2}{m} \rfloor$ if $T = S_2[i_2 \dots j_2]$). Thus, the common substring reported by Algorithm 1 satisfies the characterization of the LCS_ℓ problem given in Remark 3.

► Theorem 19. *The LCS_ℓ problem can be solved deterministically in $\mathcal{O}(1)$ space and $\mathcal{O}(\frac{n^2}{\ell})$ time.*

Proof. We apply the self-reduction of Algorithm 1 with $m = \ell$ to the algorithm of Lemma 18. The running time is $\mathcal{O}((\frac{n}{m})^2 m) = \mathcal{O}(\frac{n^2}{m}) = \mathcal{O}(\frac{n^2}{\ell})$ and the space complexity is constant. \blacktriangleleft

This result allows for the aforementioned improvement upon the algorithms of Theorem 15.

► **Theorem 4.** *The LCS_ℓ problem can be solved deterministically in $\mathcal{O}(\frac{n \log^* n}{\ell})$ space and $\mathcal{O}(n \log n)$ time, and in $\mathcal{O}(\frac{n}{\ell})$ space and $\mathcal{O}(n \log n)$ time with high probability using a Las-Vegas randomized algorithm.*

Proof. If $\ell \geq \frac{n}{\log n}$, we use the algorithm of Theorem 19, which costs $\mathcal{O}(\frac{n^2}{\ell}) = \mathcal{O}(n \log n)$ time. Otherwise, we use the algorithm of Theorem 15. The running time is $\mathcal{O}(n \log n + \ell \log^2 n) = \mathcal{O}(n \log n)$, and the space complexity is $\mathcal{O}(\frac{n \log^* n}{\ell} + \log n) = \mathcal{O}(\frac{n \log^* n}{\ell})$ or $\mathcal{O}(\frac{n}{\ell} + \log n) = \mathcal{O}(\frac{n}{\ell})$, respectively. ◀

A time-space tradeoff is, in turn, obtained using Algorithm 1 on top of Theorem 4.

► **Theorem 20.** *Given a parameter $s \in [1, n]$, the LCS_ℓ problem can be solved deterministically in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n \log^* n}{\ell \cdot s} + n \log n)$ time, and in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n}{\ell \cdot s} + n \log n)$ time with high probability using a Las-Vegas randomized algorithm.*

Proof. For a randomized algorithm, we apply the self-reduction of Algorithm 1 with $m = \ell \cdot s$ to the algorithm of Theorem 4. The space complexity is $\mathcal{O}(\frac{m}{\ell}) = \mathcal{O}(s)$, whereas the running time is $\mathcal{O}(n \log n)$ if $m \geq n$ and $\mathcal{O}((\frac{n}{m})^2 \cdot m \log m) = \mathcal{O}(\frac{n^2 \log m}{m}) = \mathcal{O}(\frac{n^2 \log n}{\ell \cdot s})$ otherwise.

A deterministic version relies on the algorithm of Theorem 19 for $s < \log^* n$, which costs $\mathcal{O}(\frac{n^2}{\ell}) = \mathcal{O}(\frac{n^2 \log^* n}{\ell \cdot s})$ time. For $s \geq \log^* n$, we apply the self-reduction of Algorithm 1 with $m = \frac{\ell \cdot s}{\log^* n}$ to the algorithm of Theorem 4. The space complexity is $\mathcal{O}(\frac{m \log^* n}{\ell}) = \mathcal{O}(s)$, whereas the running time is $\mathcal{O}(n \log n)$ if $m \geq n$ and $\mathcal{O}((\frac{n}{m})^2 \cdot m \log m) = \mathcal{O}(\frac{n^2 \log m}{m}) = \mathcal{O}(\frac{n^2 \log n \log^* n}{\ell \cdot s})$ otherwise. ◀

6 Time-space tradeoff for the LCS problem

In order to solve the LCS problem in time depending on $\text{lcs}(S_1, S_2)$, we solve LCS_ℓ for exponentially decreasing thresholds ℓ .

■ **Algorithm 2** A basic reduction from the LCS problem to the LCS_ℓ problem.

```

1  $\ell = n$ ;
2 do
3    $\ell = \ell/2$ ;
4    $T = LCS_\ell(S_1, S_2)$ ;
5 while  $|T| < \ell$ ;
6 return  $T$ ;
```

In Algorithm 2, as long as $\ell > \text{lcs}(S_1, S_2)$, LCS_ℓ clearly returns a common substring shorter than ℓ . In the first iteration when this condition is not satisfied, we have $\ell \leq \text{lcs}(S_1, S_2) < 2\ell$, so LCS_ℓ must return a longest common substring.

If the algorithm of Theorem 19 is used for LCS_ℓ , then the space complexity is $\mathcal{O}(1)$, and the running time is $\mathcal{O}(\sum_{i=1}^{\lceil \log \frac{n}{L} \rceil} \frac{n^2}{n/2^i}) = \mathcal{O}(\sum_{i=1}^{\lceil \log \frac{n}{L} \rceil} n \cdot 2^i) = \mathcal{O}(\frac{n^2}{L})$, where $L = \text{lcs}(S_1, S_2)$.

► **Theorem 2.** *The LCS problem can be solved deterministically in $\mathcal{O}(1)$ space and $\mathcal{O}(\frac{n^2}{L})$ time, where $L = \text{lcs}(S_1, S_2)$.*

The $\mathcal{O}(1)$ -space solution is still used if the input space restriction is $s = \mathcal{O}(\log n)$. Otherwise, we start with $\ell = \Theta(\frac{n}{s})$ (in the randomized version) or $\ell = \Theta(\frac{n \log^* n}{s})$ (in the deterministic version) and a single call to the algorithm of Theorem 15. This is correct due to

Remark 16, the space complexity is $\mathcal{O}(s)$, and the running time is $\mathcal{O}(n \log n)$. In subsequent iterations, the procedure of Theorem 20 is used, and its running time is dominated by the first term: $\mathcal{O}(\frac{n^2 \log n}{\ell \cdot s})$ or $\mathcal{O}(\frac{n^2 \log n \log^* n}{\ell \cdot s})$, respectively. These values form a geometric progression for exponentially decreasing ℓ , dominated by the running time of the last iteration: $\mathcal{O}(\frac{n^2 \log n}{L \cdot s})$ and $\mathcal{O}(\frac{n^2 \log n \log^* n}{L \cdot s})$, respectively. This analysis yields our main result:

► **Theorem 1.** *Given s with $1 \leq s \leq n$, the LCS problem with $L = \text{lcs}(S_1, S_2)$ can be solved deterministically in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n \log^* n}{s \cdot L} + n \log n)$ time, and in $\mathcal{O}(s)$ space and $\mathcal{O}(\frac{n^2 \log n}{s \cdot L} + n \log n)$ time with high probability using a Las-Vegas randomized algorithm.*

References

- 1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.
- 2 Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Quasi-linear-time algorithm for longest common circular factor. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, pages 25:1–25:14, 2019. doi:10.4230/LIPIcs.CPM.2019.25.
- 3 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *LNCS*, pages 14–26. Springer, 2017. doi:10.1007/978-3-319-67428-5_2.
- 4 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common substring made fully dynamic. In *27th Annual European Symposium on Algorithms, ESA 2019*, pages 6:1–6:17, 2019. doi:10.4230/LIPIcs.ESA.2019.6.
- 5 Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with k -errors and applications. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *25th International Symposium on String Processing and Information Retrieval, SPIRE 2018*, volume 11147 of *LNCS*, pages 27–41. Springer, 2018. doi:10.1007/978-3-030-00479-8_3.
- 6 Maxim A. Babenko and Tatiana Starikovskaya. Computing the longest common substring with one mismatch. *Problems of Information Transmission*, 47(1):28–33, 2011. doi:10.1134/S0032946011010030.
- 7 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.
- 8 Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz, Benjamin Sach, and Hjalte Wedel Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):39:1–39:19, 2016. doi:10.1145/2836166.
- 9 Or Birenzweige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *31st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 607–626. Society for Industrial and Applied Mathematics, 2020. doi:10.1137/1.9781611975994.37.
- 10 Rene De La Briandais. File searching using variable length keys. In *Western Joint Computer Conference*, pages 295–298. ACM Press, 1959. doi:10.1145/1457838.1457895.
- 11 Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume

- 105 of *LIPICs*, pages 23:1–23:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CPM.2018.23.
- 12 Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2020*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.
 - 13 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.
 - 14 Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, and Marie-France Sagot. Longest repeats with a block of k don't cares. *Theoretical Computer Science*, 362(1-3):248–254, 2006. doi:10.1016/j.tcs.2006.06.029.
 - 15 Maxime Crochemore and Dominique Perrin. Two-way string matching. *Journal of the ACM*, 38(3):651–675, 1991. doi:10.1145/116825.116845.
 - 16 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.
 - 17 Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.
 - 18 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.1090/S0002-9939-1965-0174934-9.
 - 19 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In Nikhil Bansal and Irene Finocchi, editors, *23rd Annual European Symposium on Algorithms, ESA 2015*, volume 9294 of *LNCS*, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.
 - 20 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015. doi:10.1016/j.ipl.2015.03.006.
 - 21 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. doi:10.1145/367390.367400.
 - 22 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster queries for longest substring palindrome after block edit. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*, pages 27:1–27:13, 2019. doi:10.4230/LIPICs.CPM.2019.27.
 - 23 Zvi Galil and Joel I. Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983. doi:10.1016/0022-0000(83)90002-8.
 - 24 Paweł Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In Philip N. Klein, editor, *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 425–439. SIAM, 2017. doi:10.1137/1.9781611974782.27.
 - 25 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
 - 26 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
 - 27 Lucas Chi Kwong Hui. Color set size problem with applications to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *3rd Annual Symposium on Combinatorial Pattern Matching, CPM 1992*, volume 644 of *LNCS*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6_19.
 - 28 Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. Faster sparse suffix sorting. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects*

- of *Computer Science, STACS 2014*, volume 25 of *LIPICs*, pages 386–396. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPICs.STACS.2014.386.
- 29 Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In Jin-yi Cai and C. K. Wong, editors, *2nd Annual International Conference on Computing and Combinatorics, COCOON 1996*, volume 1090 of *LNCS*, pages 219–230. Springer, 1996. doi:10.1007/3-540-61332-3_155.
 - 30 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. doi:10.1007/s00453-019-00548-x.
 - 31 Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In Andreas S. Schulz and Dorothea Wagner, editors, *22th Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *LNCS*, pages 605–617. Springer, 2014. doi:10.1007/978-3-662-44777-2_50.
 - 32 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, 1999. doi:10.1109/SFFCS.1999.814634.
 - 33 Mamoru Maekawa. A \sqrt{n} algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985. doi:10.1145/214438.214445.
 - 34 Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. doi:10.1145/321479.321481.
 - 35 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007. doi:10.1145/1216370.1216372.
 - 36 Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In Johannes Fischer and Peter Sanders, editors, *24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, volume 7922 of *LNCS*, pages 223–234. Springer, 2013. doi:10.1007/978-3-642-38905-4_22.
 - 37 Sharma V. Thankachan, Chaitanya Aluru, Sriram P. Chockalingam, and Srinivas Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Benjamin J. Raphael, editor, *22nd Annual International Conference on Research in Computational Molecular Biology, RECOMB 2018*, volume 10812 of *LNCS*, pages 211–224. Springer, 2018. doi:10.1007/978-3-319-89929-9_14.
 - 38 Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k -mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016. doi:10.1089/cmb.2015.0235.
 - 39 Derrien Thomas, Estellé Jordi, Marco Sola Santiago, Knowles David G., Raineri Emanuele, Guigó Roderic, and Ribeca Paolo. Fast computation and applications of genome mappability. *PLOS ONE*, 7:1–16, January 2012. doi:10.1371/journal.pone.0030377.
 - 40 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT 1973*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society. doi:10.1109/SWAT.1973.13.

On Two Measures of Distance Between Fully-Labelled Trees

Giulia Bernardini 

University of Milano - Bicocca, Milan, Italy
giulia.bernardini@unimib.it

Paola Bonizzoni

University of Milano - Bicocca, Milan, Italy
paola.bonizzoni@unimib.it

Paweł Gawrychowski

Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Abstract

The last decade brought a significant increase in the amount of data and a variety of new inference methods for reconstructing the detailed evolutionary history of various cancers. This brings the need of designing efficient procedures for comparing rooted trees representing the evolution of mutations in tumor phylogenies. Bernardini et al. [CPM 2019] recently introduced a notion of the rearrangement distance for fully-labelled trees motivated by this necessity. This notion originates from two operations: one that permutes the labels of the nodes, the other that affects the topology of the tree. Each operation alone defines a distance that can be computed in polynomial time, while the actual rearrangement distance, that combines the two, was proven to be NP-hard.

We answer two open questions left unanswered by the previous work. First, what is the complexity of computing the permutation distance? Second, is there a constant-factor approximation algorithm for estimating the rearrangement distance between two arbitrary trees? We answer the first one by showing, via a two-way reduction, that calculating the permutation distance between two trees on n nodes is equivalent, up to polylogarithmic factors, to finding the largest cardinality matching in a sparse bipartite graph. In particular, by plugging in the algorithm of Liu and Sidford [ArXiv 2020], we obtain an $\tilde{O}(n^{4/3+o(1)})$ time algorithm for computing the permutation distance between two trees on n nodes. Then we answer the second question positively, and design a linear-time constant-factor approximation algorithm that does not need any assumption on the trees.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Approximation algorithms analysis; Theory of computation → Problems, reductions and completeness

Keywords and phrases Tree distance, Cancer progression, Approximation algorithms, Fine-grained complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.6

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 872539.



Giulia Bernardini: Partially supported by a research internship at CWI.

1 Introduction

Phylogenetic trees represent a plausible evolutionary relationship between the most disparate objects: natural languages in linguistics [22, 36, 44], ancient manuscripts in archaeology [12], genes and species in biology [25, 26]. The leaves of such trees are labelled by the entities they represent, while the internal nodes are unlabelled and stand for unknown or extinct items. A great wealth of methods to infer phylogenies have been developed over the



© Giulia Bernardini, Paola Bonizzoni, and Paweł Gawrychowski;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 6; pp. 6:1–6:16

Leibniz International Proceedings in Informatics



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

decades [19,42], together with various techniques to compare the output of different algorithms, e.g., by building a consensus tree that captures the similarity between a set of conflicting trees [11, 20, 27, 28] or by defining a metric between two trees [10, 16–18, 39, 40].

Fully-labelled trees, in opposition to classical phylogenies, may model an evolutionary history where the internal nodes, just like the leaves, correspond to extant entities. An important phenomenon that fits this model well is cancer progression [23, 37]. With the increasing amount of data and algorithms becoming available for inferring cancer evolution [6, 7, 29, 34, 46], there is a pressing need of methods to provide a meaningful comparison among the trees produced by different approaches. Besides the well-studied edit distance for fully-labelled trees [35, 38, 43, 47], a few recent papers proposed ad-hoc metrics for tumor phylogenies [13, 15, 21, 31]. Taking inspiration from the existing literature [4, 8, 14, 42] on phylogeny rearrangement, the study of an operational notion of distance for rearranging a fully-labelled tree is of great interest, and there are still many unexplored questions to be answered.

Following this line of research, we revisit the two notions of operational distance between fully-labelled trees recently introduced by Bernardini et al. [5]. We consider rooted trees on n nodes labelled with distinct labels from $[n] = \{1, 2, \dots, n\}$, and identify nodes with their labels. We recall the following two basic operations on such trees:

- **link-and-cut operation:** given u, v and w such that v is a child of u and w is not a descendant of v , the link-and-cut operation $v | u \rightarrow w$ consists of two suboperations: cut the edge (v, u) and add the edge (v, w) , effectively switching the parent of v from u to w .
- **permutation operation:** apply some permutation $\pi : [n] \rightarrow [n]$ to the nodes. If a node u was a child of v before the operation, then after the operation $\pi(u)$ is a child of $\pi(v)$.

The size $|\pi|$ of a permutation is the number of elements x s.t. $\pi(x) \neq x$. Two trees T_1 and T_2 are isomorphic if and only if one can reorder the children of every node so as to make the trees identical after disregarding the labels. The *permutation distance* $d_\pi(T_1, T_2)$ between two isomorphic trees is the smallest size $|\pi|$ of a permutation π that transforms T_1 into T_2 . Bernardini et al. [5] designed a cubic time algorithm for computing the permutation distance.

The size of a sequence of link-and-cut and permutation operations is the sum of the number of link-and-cut operations and the total size of all permutations. The *rearrangement distance* $d(T_1, T_2)$ between two (not necessarily isomorphic) trees with identical roots is the smallest size of any sequence of link-and-cut and permutation operations that, without permuting the root, transform T_1 into T_2 . Bernardini et al. [5] proved that computing the rearrangement distance is NP-hard, but for binary trees there exists a polynomial time 4-approximation algorithm.

We consider two natural open questions. First, what is the complexity of computing the permutation distance? Second, is there a constant-factor approximation algorithm for estimating the rearrangement distance between two arbitrary trees? For computing the permutation distance, in Section 3 we connect the complexity to that of calculating the largest cardinality matching in a sparse bipartite graph. By designing two-way reductions we show that these problems are equivalent, up to polylogarithmic factors. Due to the recent progress in the area of fine-grained complexity we now know, for many problems that can be solved in polynomial time, what is essentially the best possible exponent in the running time, conditioned on some plausible but yet unproven hypothesis [45]. For max-flow, and more specifically maximum matching, this is not the case yet, although we do have some understanding of the complexity of the related problem of computing the max-flow between all pairs of nodes [1, 2, 32]. So, even though our reductions don't tell us what is the best possible exponent in the running time, they do imply that it is the same

as for maximum matching in a sparse bipartite graph. In particular, by plugging in the asymptotically fastest known algorithm [33], we obtain an $\tilde{O}(n^{4/3+o(1)})$ time algorithm for computing the permutation distance between two trees on n nodes. The main technical novelty in our reduction from permutation distance is that, even though the natural approach would result in multiple instances of weighted maximum bipartite matching, we manage to keep the graphs unweighted.

For the rearrangement distance, in Section 4 we design a linear-time constant-factor approximation algorithm that does not assume that the trees are binary. The algorithm consists of multiple phases, each of them introducing more and more structure into the currently considered instance, while making sure that we don't pay more than the optimal distance times some constant. To connect the number of steps used in every phase with the optimal distance, we introduce a new combinatorial object that can be used to lower bound the latter inspired by the well-known algorithm for computing the majority [9].

2 Preliminaries

Let $[n] = \{1, 2, \dots, n\}$. We consider rooted trees and forests on nodes labelled with distinct labels from $[n]$, and identify nodes with their labels. The parent of u in F is denoted $p_F(u)$, and we use the convention that $p_F(u) = \perp$ when u is a root in F . $F|u$ denotes the subtree of F rooted at u , $\text{children}_F(u)$ stands for the set of children of a node u in F , and $\text{level}_F(u)$ is the level of u in F (with the roots being on level 0).

Two trees T_1 and T_2 are isomorphic, denoted $T_1 \equiv T_2$, if and only if there exists a bijection μ between their nodes such that, for every $u \in [n]$ with $p_{T_1}(u) \neq \perp$, it holds that $\mu(p_{T_1}(u)) = p_{T_2}(\mu(u))$, implying in particular that μ maps the root of T_1 to the root of T_2 . Let $\mathcal{I}(T_1, T_2)$ denote the set of all such bijections μ . Given two isomorphic trees T_1 and T_2 , we seek a permutation π with the smallest size that transforms T_1 into T_2 . This is equivalent to finding $\mu \in \mathcal{I}(T_1, T_2)$ that maximises the number of conserved nodes $\text{conserved}(\mu) = \{u : u = \mu(u)\}$, as these two values sum up to n .

When working on the rearrangement distance, for ease of presentation, instead of the link-and-cut operation we will work with the cut operation defined as follows:

- **cut operation:** let u, v be two nodes such that v is a child of u . The cut operation $(v \uparrow u)$ removes the edge (v, u) , effectively making v a root.

The size of a sequence of cut and permutation operations is defined similarly as for a sequence of link-and-cut and permutation operations. Since a permutation operation is essentially just renaming the nodes, we can assume that all permutation operations precede all link-and-cut (or cut) operations, or vice versa. Furthermore, multiple consecutive permutation operations can be replaced by a single permutation operation without increasing the total size.

This leads to the notion of rearrangement distance between two forests F_1 and F_2 . We write $F_1 \sim F_2$ to denote that, for every $u \in [n]$, at least one of the following three conditions holds: (i) $p_{F_1}(u) = p_{F_2}(u)$, (ii) $p_{F_1}(u) = \perp$, or (iii) $p_{F_2}(u) = \perp$. The rearrangement distance $\tilde{d}(F_1, F_2)$ is the smallest size of any sequence of cut and permutation operations that transforms F_1 into F'_1 such that $F'_1 \sim F_2$. This is the same as the smallest size of any sequence of cut and permutation operations that transforms F_2 into F'_2 such that $F_1 \sim F'_2$, as both sizes are equal to the minimum over all permutations π that fix the original root of the following expression

$$|\{u : \pi(u) \neq u\}| + |\{u : p_{F_1}(u) \neq p_{F_2}(\pi(u)) \wedge p_{F_1}(u) \neq \perp \wedge p_{F_2}(\pi(u)) \neq \perp\}|.$$

Consequently, \tilde{d} defines a metric. The original notion of rearrangement distance d between two trees was similarly defined as the smallest size of any sequence of link-and-cut and permutation operations that transforms T_1 into T_2 , under the additional assumption that

the roots of T_1 and T_2 are identical (so $d(T_1, T_2)$ is well-defined) and cannot participate in any permutation operation [5]. In Section 4 we connect $d(T_1, T_2)$ and $\tilde{d}(T_1, T_2)$, and then work with the latter.

A matching in a bipartite graph is a subset of edges with no two edges meeting at the same vertex. A maximum matching in an unweighted bipartite graph is a matching of maximum cardinality, whereas a maximum weight matching in a weighted bipartite graph is a matching in which the sum of weights is maximised. Given an unweighted bipartite graph with m edges, the well-known algorithm by Hopcroft and Karp [24] finds a maximum matching in $\mathcal{O}(m^{1.5})$ time. This has been recently improved by Liu and Sidford to $\tilde{\mathcal{O}}(m^{4/3+o(1)})$ [33].

A *heavy path decomposition* of a tree T is obtained by selecting, for every non-leaf node $u \in T$, its *heavy child* v such that $T|v$ is the largest: there will be some subtlety in how to resolve a tie in this definition that will be explained in detail later. This procedure decomposes the nodes of T into node-disjoint paths called *heavy paths*. Each heavy path p starts at some node, called its *head*, and ends at a leaf: $\text{head}_T(u)$ denotes the head of the heavy path containing a node u in T . An important property of such a decomposition is that the number of distinct heavy paths above any leaf (that is, intersecting the path from a leaf to the root) is only logarithmic in the size of T [41].

3 A Fast Algorithm for the Permutation Distance

Our aim is to find $\mu \in \mathcal{I}(T_1, T_2)$ that maximises $\text{conserved}(\mu)$, that is $\gamma(T_1, T_2) = \max\{\text{conserved}(\mu) : \mu \in \mathcal{I}(T_1, T_2)\}$. To make the notation less cluttered, we define $\gamma(x, y) = \gamma(T_1|x, T_2|y)$. Let us start by describing a simple polynomial time algorithm which follows the construction of [5]. We will then show how to improve it to obtain a faster algorithm that uses unweighted bipartite maximum matching. Finally, we will show a reduction from bipartite maximum matching to computing the permutation distance, establishing that these two problems are in fact equivalent, up to polylogarithmic factors.

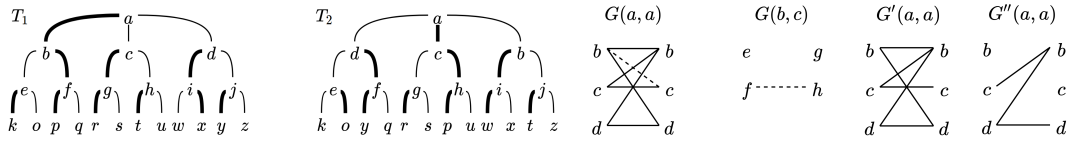
3.1 Polynomial Time Algorithm

We first run the folklore linear-time algorithm of [3] for determining if two rooted trees are isomorphic. Recall that this algorithm assigns a number from $\{1, 2, \dots, 2n\}$ to every node of T_1 and T_2 so that the subtrees rooted at two nodes are isomorphic if and only if their numbers are equal. The high-level idea is then to consider a weighted bipartite graph $G(u, v)$ for each $u, v \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(v)$ and $T_1|u \equiv T_2|v$. The vertices of $G(u, v)$ are $\text{children}_{T_1}(u)$ and $\text{children}_{T_2}(v)$, and there is an edge of weight $\gamma(u', v')$ between $u' \in \text{children}_{T_1}(u)$ and $v' \in \text{children}_{T_2}(v)$ if and only if $T_1|u' \equiv T_2|v'$ and $\gamma(u', v') > 0$. We call such graphs the *distance graphs* for T_1 and T_2 and denote them collectively by $\mathcal{G}(T_1, T_2)$.

$\gamma(u, v)$ is computed as follows, with $\mathcal{M}(G(u, v))$ denoting the weight of a (not necessarily perfect) maximum weight matching in $G(u, v)$, $\Gamma(u, v) = 1$ if $u = v$ and $\Gamma(u, v) = 0$ otherwise.

$$\gamma(u, v) = \begin{cases} \mathcal{M}(G(u, v)) + \Gamma(u, v) & \text{if } T_1|u \equiv T_2|v, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

The overall number of edges created in all graphs is $\mathcal{O}(n^2)$. Indeed, for each $u \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$, and for each pair of ancestors z of u in T_1 and w of u in T_2 such that $\text{level}_{T_1}(z) = \text{level}_{T_2}(w)$ and $T_1|z \equiv T_2|w$, we possibly add an edge (z, w) to the graph $G(p_{T_1}(z), p_{T_2}(w))$. Since there are up to n pairs of ancestors on the same level for each label, and the labels are n , there are $\mathcal{O}(n^2)$ edges overall.



■ **Figure 1** $G(a, a)$ (type 1), $G'(a, a)$, $G''(a, a)$ and $G(b, c)$ (type 3) for T_1 and T_2 on the left. The special edge in each graph is dashed.

We then start from the deepest level in both trees, and we move up level by level towards the roots in both trees simultaneously. For each level k , we consider all pairs of isomorphic subtrees rooted at level k , build the corresponding distance graphs, and use Equation (1) to weigh the edges. After having reached the roots, we return the value of $\gamma(T_1, T_2)$. The correctness of the algorithm follows directly from Lemma 13 of [5], stating that the permutation distance is equal to the minimum number of labels that are not conserved by any isomorphic mapping, i.e., $d_\pi(T_1, T_2) = n - \gamma(T_1, T_2)$. The running time is polynomial if we plug in any polynomial-time maximum weight matching algorithm.

In the next subsection we show how to obtain a better running time by constructing a different version of distance graphs, so that the total weight of their edges will be subquadratic, and replacing maximum weight matching with maximum matching.

3.2 Reduction to Bipartite Maximum Matching

We start by finding a heavy path decomposition of T_1 and T_2 , with some extra care in resolving a tie if there are multiple children with subtrees of the same size, as follows. Recall that we already know which subtrees of T_1 and T_2 are isomorphic, as the algorithm of [3] assigns the same number from $\{1, 2, \dots, 2n\}$ to nodes of T_1 and T_2 with isomorphic subtrees. For every $u, v \in [n]$ such that $T_1|u \equiv T_2|v$, we would like the heavy child u' of u in T_1 and v' of v in T_2 to be such that $T_1|u' \equiv T_2|v'$. This can be implemented in linear time: it suffices to group the nodes with isomorphic subtrees together, and then make the choice just once for every such group.

Consider a graph $G(u, v)$ for some $u, v \in [n]$: the edge corresponding to the heavy child u' of u in T_1 and the heavy child v' of v in T_2 is called *special* (note that this edge might not exist). The key observation is that the properties of heavy path decomposition allow us to bound the total weight of non-special edges by $\mathcal{O}(n \log n)$.

► **Lemma 1.** *The total weight of all non-special edges in $\mathcal{G}(T_1, T_2)$ is $\mathcal{O}(n \log n)$.*

Proof. Consider any $u \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$. For each pair of ancestors z of u in T_1 and w of u in T_2 such that $\text{level}_{T_1}(z) = \text{level}_{T_2}(w)$, $T_1|z \equiv T_2|w$ and either $\text{head}_{T_1}(z) = z$ or $\text{head}_{T_2}(w) = w$, u will contribute 1 to the weight of an edge (z, w) in $G(p_{T_1}(z), p_{T_2}(w))$. Because there are at most $\log n$ heavy paths above any node of T_1 or T_2 , each label $u \in [n]$ contributes 1 to the weight of at most $2 \log n$ non-special edges, making their total weight $\mathcal{O}(n \log n)$ overall. ◀

We divide the graphs in $\mathcal{G}(T_1, T_2)$ into three types: see Figure 1 for an example.

- Type 1:** graphs $G(u, v)$ with at least one non-special edge.
- Type 2:** graphs $G(u, v)$ with no non-special edges, and $\Gamma(u, v) = 1$.
- Type 3:** graphs $G(u, v)$ with no non-special edges, and $\Gamma(u, v) = 0$.

6:6 On Two Measures of Distance Between Fully-Labelled Trees

We will construct only the graphs of type 1 and 2, and extract from them the information that the graphs of type 3 would have captured. In what follows we show how to construct the graphs of type 1 and 2 in $\mathcal{O}(n \log^2 n)$ time.

Constructing the Graphs of Type 1 and 2. The first step is to find all pairs of nodes that correspond to graphs of type 1 or 2, and store them in a dictionary D implemented as a balanced search tree with $\mathcal{O}(\log n)$ access time. The second step is to find the non-special edges of these graphs, and store them in a separate dictionary, also implemented as a balanced search tree with $\mathcal{O}(\log n)$ access time. Note that the weights will be found at a later stage of the algorithm. We assume that both trees have been already decomposed into heavy paths, and we already know which subtrees are isomorphic. This can be preprocessed in $\mathcal{O}(n)$ time.

► **Lemma 2.** *All graphs of type 1 and 2 can be identified in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We consider every $u \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$ in two passes. In the first pass, we need to iterate over every ancestor z of u in T_1 and w of u in T_2 such that $\text{level}_{T_1}(z) = \text{level}_{T_2}(w)$, $T_1|z \equiv T_2|w$ and either $\text{head}_{T_1}(z) = z$ or $\text{head}_{T_2}(w) = w$, and if additionally $T_1|p_{T_1}(z) \equiv T_2|p_{T_2}(w)$ then designate $G(p_{T_1}(z), p_{T_2}(w))$ to be a graph of type 1 and insert it into D . As a non-special edge (z, w) of a graph $G(p_{T_1}(z), p_{T_2}(w))$ is such that either z or w are not on the same heavy path as their parents, this correctly determines all graphs of type 1.

To efficiently iterate over all such z and w given u , we assume that the nodes of every heavy path of a tree T are stored in an array, so that, given any node $u \in T$, we are able to access the node that belongs to the same heavy path as u and whose level is ℓ in constant time, if it exists. We denote such operation $\text{access}_T(u, \ell)$. Given two nodes $u \in T_1$ and $v \in T_2$ on the same level, the procedure below shows how to iterate over every ancestor z of u and w of v such that $\text{level}_{T_1}(z) = \text{level}_{T_2}(w)$ and either $\text{head}_{T_1}(z) = z$ or $\text{head}_{T_2}(w) = w$, in $\mathcal{O}(\log n)$ time, implying that all graphs of type 1 can be identified in $\mathcal{O}(n \log^2 n)$ time.

```

1 while  $u \neq \perp$  and  $v \neq \perp$  do
2   if  $\text{level}_{T_1}(\text{head}_{T_1}(u)) < \text{level}_{T_2}(\text{head}_{T_2}(v))$  then
3     output  $\text{access}_{T_1}(u, \text{level}_{T_2}(\text{head}_{T_2}(v)))$  and  $\text{head}_{T_2}(v)$ 
4      $v \leftarrow p_{T_2}(\text{head}_{T_2}(v))$ 
5   if  $\text{level}_{T_1}(\text{head}_{T_1}(u)) > \text{level}_{T_2}(\text{head}_{T_2}(v))$  then
6     output  $\text{head}_{T_1}(u)$  and  $\text{access}_{T_2}(v, \text{level}_{T_1}(\text{head}_{T_1}(u)))$ 
7      $u \leftarrow p_{T_1}(\text{head}_{T_1}(u))$ 
8   else
9     output  $\text{head}_{T_1}(u)$  and  $\text{head}_{T_2}(v)$ 
10     $u \leftarrow p_{T_1}(\text{head}_{T_1}(u))$ 
11     $v \leftarrow p_{T_2}(\text{head}_{T_2}(v))$ 

```

In the second pass, for each $u \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(u)$ and $T_1|u \equiv T_2|u$, we designate $G(u, u)$ to be a graph of type 2, unless it has been already designated to be a graph of type 1. ◀

► **Lemma 3.** *All graphs of type 1 and 2 can be populated with their edges in $\mathcal{O}(n \log^2 n)$ time.*

Proof. For each such graph $G(u, v)$ such that none of u, v is a leaf, let u' be the unique heavy child of u , and v' be the unique heavy child of v . We add the special edge (u', v') to $G(u, v)$. To find the non-special edges, we again consider every $u \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(u)$

and $T_1|u \equiv T_2|u$: we iterate over the ancestors z of u in T_1 and w of u in T_2 such that $\text{level}_{T_1}(z) = \text{level}_{T_2}(w)$, $T_1|z \equiv T_2|w$ and either $\text{head}_{T_1}(z) = z$ or $\text{head}_{T_2}(w) = w$, and if additionally $T_1|p_{T_1}(z) \equiv T_2|p_{T_2}(w)$ then add a non-special edge (z, w) to $G(p_{T_1}(z), p_{T_2}(w))$. This takes $\mathcal{O}(n \log^2 n)$ time overall. \blacktriangleleft

Processing the Graphs of Type 1 and 2. Having constructed the graphs of type 1 and 2 in $\mathcal{O}(n \log^2 n)$ time, we process them level by level. Consider $G(u, v)$: for each of its edges (u', v') corresponding to $u' \in \text{children}_{T_1}(u)$ and $v' \in \text{children}_{T_2}(v)$, we need to extract its weight $\gamma(u', v')$. If $G(u', v')$ is of type 1 or 2, the graph can be extracted from the dictionary in $\mathcal{O}(\log n)$ time. Otherwise, $G(u', v')$ is of type 3 and we need to make up for not having processed such graphs.

To this aim, we associate a sorted list of levels with each pair of heavy paths of T_1 and T_2 . The lists are stored in a dictionary indexed by the heads of the heavy paths. For every $u, v \in [n]$ such that $G(u, v)$ is of type 1 or 2, we append the levels of u and v to the lists associated with the respective heavy paths. The lists can be constructed in $\mathcal{O}(n \log^2 n)$ time by processing the graphs level by level, and allow us to efficiently use the following lemma.

► Lemma 4. *Consider $u, v \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(v)$ and $T_1|u \equiv T_2|v$, but $G(u, v)$ is of type 3. Either both u and v are leaves and $\gamma(u, v) = 0$, or the heavy child of u is u' , the heavy child of v is v' , and $\gamma(u, v) = \gamma(u', v')$.*

Proof. First observe that $u \neq v$, as otherwise $G(u, v)$ would be of type 2. Because $T_1|u \equiv T_2|v$, either both u and v are leaves or none of them is a leaf. In the former case, $G(u, v)$ is empty and $\gamma(u, v) = 0$. By how we resolve ties in the heavy path decomposition, in the latter case we have $T_1|u' \equiv T_2|v'$, where u' is the heavy child of u and v' is the heavy child of v . $G(u, v)$ consists of the unique special edge corresponding to the heavy child u' of u and v' of v , so $\mathcal{M}(G(u, v))$ is equal to the cost of the special edge, and by (1) we obtain that $\gamma(u, v) = \gamma(u', v')$. \blacktriangleleft

Given $u, v \in [n]$ such that $\text{level}_{T_1}(u) = \text{level}_{T_2}(v) = \ell$ and $T_1|u \equiv T_2|v$, we extract $\gamma(u, v)$ by accessing the sorted list associated with the heavy paths of u and v : we binary search for the smallest level $\ell' \geq \ell$ such that the heavy paths of u and v respectively contain a node u' and v' , both on level ℓ' , with $G(u', v')$ of type 1 or 2. Then Lemma 4, together with the fact that in our heavy path decomposition the subtrees rooted at the heavy children of two nodes with isomorphic subtrees are also isomorphic, implies that $\gamma(u, v) = \gamma(u', v')$.

It remains to describe how to compute $\mathcal{M}(G(u, v))$ for every graph $G(u, v)$ of type 1 and 2. We could have used any maximum weight matching algorithm, but this would result in a higher running time. Our goal is to plug in a maximum matching algorithm. This seems problematic as $G(u, v)$ is a weighted bipartite graph, but we will show that maximum weight matching can be reduced to multiple instances of maximum matching. However, bounding the overall running time will require bounding the total weight of all edges belonging to graphs of type 1 and 2. By Lemma 1 we already know that the total weight of all non-special edges is $\mathcal{O}(n \log n)$, but such bound doesn't hold for the special edges. Therefore, we proceed as follows. Let u' be the heavy child of u and v' be the heavy child of v . We construct $G'(u, v)$ by removing the special edge from $G(u, v)$. We also construct $G''(u, v)$ from $G(u, v)$ by removing all the edges incident to u' and v' (see Figure 1 for an example). Equation (1) can then be rewritten as follows:

$$\gamma(u, v) = \max\{\mathcal{M}(G'(u, v)), \mathcal{M}(G''(u, v)) + \gamma(u', v')\} + \Gamma(u, v). \quad (2)$$

This is because a maximum weight matching in $G(u, v)$ either includes the special edge (u', v') , implying that no other edges incident to u' and v' can be part of the matching and thus $\mathcal{M}(G(u, v)) = \mathcal{M}(G''(u, v)) + \gamma(u', v')$, or it does not include it, thus $\mathcal{M}(G(u, v)) = \mathcal{M}(G'(u, v))$. Since the graphs $G'(u, v)$ and $G''(u, v)$ contain only non-special edges, the overall weight of all edges in the obtained instances of maximum weight matching is $\mathcal{O}(n \log n)$.

We already know that constructing all the relevant graphs takes $\mathcal{O}(n \log^2 n)$ time. It remains to analyze the time to calculate the maximum weight matching in every $G'(u, v)$ and $G''(u, v)$. We first present a preliminary lemma that connects the complexity of calculating the maximum weight matching in a weighted bipartite graph to the complexity of calculating the maximum matching in an unweighted bipartite graph.

► **Lemma 5** ([30]). *Let G be a weighted bipartite graph, and let N be the total weight of all the edges of G . Calculating the maximum weight matching in G can be reduced in $\mathcal{O}(N)$ time to multiple instances of calculating the maximum matching in an unweighted bipartite graph, in such a way that the total number of edges in all such graphs is at most N .*

Proof. Using the decomposition theorem of Kao, Lam, Sung, and Ting [30], we can reduce computing the maximum weight matching in a weighted bipartite graph such that the total weight of all edges is N to multiple instances of calculating the largest cardinality matching in an unweighted bipartite graph. The total number of edges in all unweighted bipartite graphs is $\sum_i m_i = N$ and the reduction can be implemented in $\mathcal{O}(N)$ time by maintaining a list of edges with weight w , for every $w = 1, 2, \dots, N$. ◀

► **Theorem 6.** *Let $f(m)$ be the complexity of calculating the maximum matching in an unweighted bipartite graph on m edges, and let $f(m)/m$ be nondecreasing. The permutation distance can be computed in $\tilde{\mathcal{O}}(f(n))$ time.*

Proof. The total number of edges in all constructed graphs is $\mathcal{O}(n \log n)$, and the total time to construct the relevant graphs and extract the costs of their edges is $\mathcal{O}(n \log^2 n)$. Thus, the total running time is $\mathcal{O}(n \log^2 n)$ plus the time to compute the maximum weight matching in every graph of type 1 and type 2. Let N_i be the total weight of all non-special edges in the i -th of these graphs. By Lemma 1, $\sum_i N_i = \mathcal{O}(n \log n)$. Additionally, $N_i \leq n$. Let $m_{i,j}$ be the number of edges in the j -th instance of unweighted bipartite matching for the i -th graph. By Lemma 5, the overall time is hence $\sum_{i,j} f(m_{i,j})$, where $\sum_{i,j} m_{i,j} \leq \sum_i N_i = \mathcal{O}(n \log n)$ and $m_{i,j} \leq N_i \leq n$. We upper bound $\sum_{i,j} f(m_{i,j})$ using the assumption that $f(m)/m$ is nondecreasing as follows:

$$\sum_{i,j} f(m_{i,j}) = \sum_{i,j} m_{i,j} \cdot f(m_{i,j})/m_{i,j} \leq \sum_{i,j} m_{i,j} \cdot f(n)/n = \mathcal{O}(f(n) \log n). \quad \blacktriangleleft$$

► **Corollary 7.** *The permutation distance can be computed in $\tilde{\mathcal{O}}(n^{4/3+o(1)})$ time.*

3.3 Reduction from Bipartite Maximum Matching

We complement the algorithm described in Subsection 3.2 with a reduction from bipartite maximum matching to computing the permutation distance: see Figure 2 for an example.

► **Theorem 8.** *Given an unweighted bipartite graph on m edges, we can construct in $\mathcal{O}(m)$ time two trees with permutation distance equal to the cardinality of the maximum matching.*

Proof. We first modify the graph so that the degree of every node is at most 3. This can be ensured in $\mathcal{O}(m)$ time by repeating the following transformation: take a node u with neighbours v_1, v_2, \dots, v_k , $k \geq 4$. Replace u with u' and u'' both connected to a new node v ,

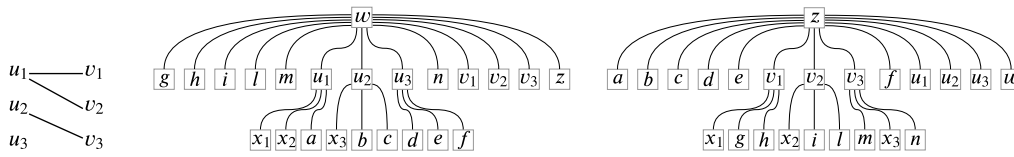


Figure 2 The two trees built for the graph on the left, according to Theorem 8.

connect u' to v_1, v_2, \dots, v_{k-2} and u'' to v_{k-1}, v_k . It can be verified that the cardinality of the maximum matching in the new graph is equal to that in the original graph increased by 1. By storing, for every node, the incident edges in a linked list, we can implement a single step of this transformation in constant time, and there are at most m steps.

We will now first construct two unlabelled trees and then explicitly assign appropriate labels to their nodes. Without loss of generality, let the nodes of the graph be u_1, u_2, \dots, u_m and v_1, v_2, \dots, v_m . In the first tree we create m nodes, labelled with u_1, u_2, \dots, u_m , connected to a common unlabelled root. In the second tree we do the same with nodes v_1, v_2, \dots, v_m . Then, for every edge (u_i, v_j) of the graph, we attach a new leaf to u_i in the first tree and to v_j in the second tree, and assign the same label to both of them. Finally, we attach enough unlabelled leaves to every u_i and v_j to make their degrees all equal to 3. To make both trees fully-labelled on the same set of labels, we further attach $1 + m + 3m - m = 3m + 1$ extra leaves to the roots of both trees. For every unlabelled leaf attached to u_1, u_2, \dots, u_m of the first tree, we choose an unlabelled extra leaf of the second tree, and assign the same label to both of them. We then assign the same label to the root of the first tree and an extra leaf of the second tree, and label the last m extra leaves of the second tree with u_1, u_2, \dots, u_m . We finally swap the trees and repeat the same procedure: see Figure 2 for an example.

The permutation distance between the two trees is equal to the cardinality of the maximum matching. Indeed, the trees are clearly isomorphic; moreover, any isomorphism must match extra leaves with extra leaves, and every u_i to a $v_{\pi(j)}$, for some permutation π on $[m]$. The extra leaves do not contribute to the number of conserved nodes, while u_i and $v_{\pi(j)}$ contribute 1 if and only if $(u_i, v_{\pi(j)})$ was an edge in the original graph. Thus, the distance is equal to the maximum over all permutations π of the number of edges $(u_i, v_{\pi(j)})$. This in turn is equal to the cardinality of the maximum matching in the original graph. ◀

4 A Constant-Factor Approximation Algorithm for the Rearrangement Distance

A linear-time algorithm that, given two trees T_1 and T_2 , approximates $d(T_1, T_2)$ within a constant factor, was known for the case where at least one of the trees is binary [5]: here we do not make any assumptions on the degrees. Throughout this section, we actually consider $\tilde{d}(F_1, F_2)$, and show how to approximate it within a constant factor. This allows us to approximate $d(T_1, T_2)$ within a constant factor using the following procedure. First, we add n leaves $n + 1, n + 2, \dots, n$ attached to the (identical) roots of T_1 and T_2 to obtain T'_1 and T'_2 , respectively. We call the resulting trees *anchored*. Because T_1 and T_2 are assumed to have the same root that cannot be permuted, we have $d(T_1, T_2) = d(T'_1, T'_2)$. We claim that $\tilde{d}(T'_1, T'_2) = d(T'_1, T'_2)$. Intuitively, in one direction it suffices to replace every link-and-cut operation $v | u \rightarrow w$ with a cut operation $(v \uparrow u)$; for the other direction, we argue that it does not make sense to permute the root, and every cut operation $(v \uparrow u)$ can be replaced by $v | u \rightarrow w$, where $w = p_{T_2}(v)$, and such link-and-cut operations are reordered so as not to make w a descendant of v .

► **Lemma 9.** For any two anchored trees T_1 and T_2 , $\tilde{d}(T_1, T_2) = d(T_1, T_2)$.

We can thus approximate $\tilde{d}(T'_1, T'_2)$ within a constant factor to obtain a constant factor approximation of $d(T_1, T_2)$. In the remaining part of this section we design an approximation algorithm for $\tilde{d}(F_1, F_2)$, where F_1 and F_2 are two arbitrary forests.

We start with describing the notation. Consider two forests F_1 and F_2 . For every $i \in [n]$, let $a[i] \in [n]$ be the parent of a non-root node i in F_1 , and $a[i] = 0$ if i is a root in F_1 . Formally, $a[i] = p_{F_1}(i)$ when $p_{F_1}(i) \neq \perp$ and $a[i] = 0$ otherwise; $b[i]$ is defined similarly but for F_2 . We think of a and b as vectors of length n .

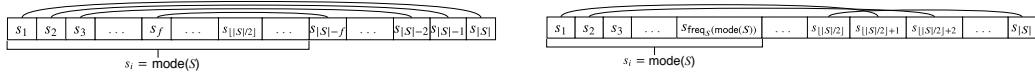
The algorithm consists of four steps, with step j transforming forest F_1^{j-1} into F_1^j by performing $\text{ALG}(j)$ operations, starting from $F_1^0 = F_1$. We will guarantee that $\text{ALG}(j) = \mathcal{O}(\tilde{d}(F_1^{j-1}, F_2))$. Then, by triangle inequality and symmetry, $\tilde{d}(F_1^j, F_2) \leq \tilde{d}(F_1^{j-1}, F_1^j) + \tilde{d}(F_1^{j-1}, F_2) \leq \text{ALG}(j) + \tilde{d}(F_1^{j-1}, F_2) = \mathcal{O}(\tilde{d}(F_1^{j-1}, F_2))$, so by induction $\tilde{d}(F_1^j, F_2) = \mathcal{O}(\tilde{d}(F_1, F_2))$. Consequently, $\text{ALG}(j) = \mathcal{O}(\tilde{d}(F_1, F_2))$, making the overall cost $\sum_j \text{ALG}(j) = \mathcal{O}(\tilde{d}(F_1, F_2))$. In the j -th step of the algorithm $a[i]$ refers to the parent of i in F_1^{j-1} . To analyse each step of the algorithm we will use the following two structures, the first of which is a streamlined version of family partitions defined in the previous paper [5].

► **Definition 10** (family partition). Given two forests F_1 and F_2 , the family partition $P(F_1, F_2)$ is the set $\{(a[i], b[i]) : a[i], b[i] \neq 0 \wedge a[i] \neq b[i]\}$.

► **Definition 11** (migrations graph). Given two forests F_1 and F_2 , the migrations graph $MG(F_1, F_2)$ consists of edges $\{(i, j) : a[i], a[j], b[i], b[j] \neq 0 \wedge a[i] = a[j] \wedge b[i] \neq b[j]\}$.

For a multiset S , let $|S|$ denote its cardinality, that is, the sum of multiplicities of all distinct elements of S . The mode of S , denoted $\text{mode}(S)$, is any element $s \in S$ with the largest multiplicity $\text{freq}_S(s)$. We will use the following combinatorial lemma.

► **Lemma 12.** Given any multiset S , let $f = \min\{|S| - \text{freq}_S(\text{mode}(S)), \lfloor |S|/2 \rfloor\}$. All $|S|$ elements of S can be partitioned into f pairs $(x_1, y_1), \dots, (x_f, y_f)$, $x_i \neq y_i$, for every $i \in [f]$, and the remaining $|S| - 2f$ elements.

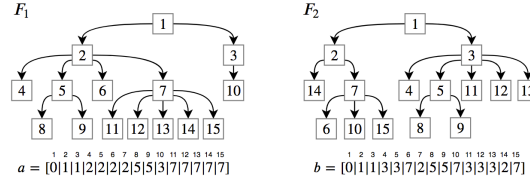


■ **Figure 3** Pairing in the case $f = |S| - \text{freq}_S(\text{mode}(S))$ (left) and $f = \lfloor |S|/2 \rfloor$ (right).

Proof. Number the elements of S so that $s_1 = \dots = s_{\text{freq}_S(\text{mode}(S))} = \text{mode}(S)$ and all of the others are sorted and numbered from $\text{freq}_S(\text{mode}(S)) + 1$ to $|S|$ accordingly. Then, if $f = |S| - \text{freq}_S(\text{mode}(S))$, pairs $(s_i, s_{|S|-i+1})$, $i \in [f]$ are s.t. $s_i \neq s_{|S|-i+1}$ (Figure 3, left); if $f = \lfloor |S|/2 \rfloor$, pairs $(s_i, s_{\lfloor |S|/2 \rfloor + i})$, $i \in [\lfloor |S|/2 \rfloor]$ are s.t. $s_i \neq s_{\lfloor |S|/2 \rfloor + i}$ (Figure 3, right). ◀

4.1 Step 1

Roughly speaking, the aim of the first step is to ensure that all nodes that might be possibly involved in a permutation, i.e., the nodes with different children in F_1 and F_2 , are roots. This is so that we do not need to worry about the relationship with their parents. For every $i \in [n]$ such that $a[i]$ and $b[i]$ are both defined and different, we cut the edges from $a[i]$ and



■ **Figure 4** F_1 and F_2 . The family partition is $P = \{(2, 3), (2, 7), (3, 7), (7, 3), (7, 2)\}$.

$b[i]$ to their parents in F_1 , thus making both of them roots. In other words, for every i such that $a[i], b[i] \neq 0$ and $a[i] \neq b[i]$, we cut edges $(a[i], a[a[i]])$ and $(b[i], a[b[i]])$. The resulting forest F_1^1 has the following property: for each $i \in [n]$ such that the parents of i in F_1^1 and in F_2 are both defined and different, $a[a[i]] = a[b[i]] = \perp$.

The number of cuts in this step is by definition at most twice the size of the family partition $P(F_1, F_2)$. Bernardini et al. [5] already showed that $|P(T_1, T_2)| \leq 2d(T_1, T_2)$ for two trees T_1 and T_2 . We show that this still holds for forests and \tilde{d} : for completeness, we provide a self-contained proof (cf. Lemma 16 in [5]).

► **Lemma 13.** $|P(F_1, F_2)| \leq 2\tilde{d}(F_1, F_2)$, implying $ALG(1) \leq 4\tilde{d}(F_1, F_2)$.

Proof. It is enough to verify that applying a single cut operation might decrease the size of the family partition by at most one, while applying a permutation operation π might decrease the size of the family partition by at most $2s$, where $s = |\{u : u \neq \pi(u)\}|$.

Consider a cut operation $(v \dagger u)$. The only change to a is that $a[v]$ becomes 0, so indeed the size of the family partition might decrease by at most one.

Now consider a permutation π . After applying π , an edge $(i, a[i])$ becomes $(\pi(i), \pi(a[i]))$, making $\pi(a[\pi^{-1}(i)])$ the parent of i . This transforms the family partition P into

$$P' = \{(\pi(a[i]), b[\pi(i)]) : a[i] \neq 0 \wedge b[\pi(i)] \neq 0 \wedge \pi(a[i]) \neq b[\pi(i)]\}.$$

To lower bound the size of $|P'|$, we first focus on the subset of P corresponding to the nodes that are fixed by π . We therefore define

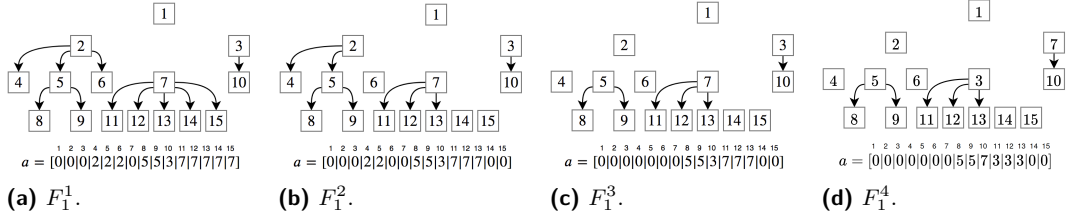
$$P_f = \{(a[i], b[i]) : a[i] \neq 0 \wedge b[i] \neq 0 \wedge a[i] \neq b[i] \wedge \pi(i) = i\}.$$

By definition, we can equivalently rewrite P_f as

$$P_f = \{(a[i], b[\pi(i)]) : a[i] \neq 0 \wedge b[\pi(i)] \neq 0 \wedge a[i] \neq b[\pi(i)] \wedge \pi(i) = i\}.$$

Now consider all pairs with the same second coordinate y in P_f : $(x_1, y), (x_2, y), \dots, (x_k, y)$, where $x_i \neq y$ for every $i \in [k]$. P' contains all pairs $(\pi(x_i), y)$ such that $\pi(x_i) \neq y$. If $\pi(y) = y$ then $\pi(x_i) = y$ cannot happen and P' contains all pairs with the second coordinate y from P_f ; otherwise, P' contains all such pairs except possibly one. Overall, $|P'| \geq |P_f| - s$, and $|P_f| \geq |P| - s$ so indeed $|P'| \geq |P| - 2s$. ◀

► **Example 14.** Consider F_1 and F_2 depicted in Figure 4. Step 1 consists of cut operations $(2 \dagger 1)$ (because, e.g., $a[4] \neq b[4]$ and $a[4] = 2$), $(3 \dagger 1)$ (because $b[4] = 3$) and $(7 \dagger 2)$ (because, e.g., $a[11] \neq b[11]$ and $a[11] = 7$). The resulting forest F_1^1 is shown in Figure 5a.



■ **Figure 5** The forests obtained after Step 1 (5a), Step 2 (5b), Step 3 (5c) and Step 4 (5d).

4.2 Step 2

Consider $u \in [n]$, and let $\text{children}_{F_1^1}(u) = \{v_1, \dots, v_k\}$. We define the multiset $B(u) = \{b[v_i] : b[v_i] \neq 0\}$ containing the parents in F_2 of the children of u in F_1^1 . Recall that $\text{mode}(B(u))$ is the most frequent element of $B(u)$ (ties are broken arbitrarily). We cut all edges (v_i, u) such that $b[v_i] \neq 0$ and $b[v_i] \neq \text{mode}(B(u))$, and define, for each $u \in [n]$, its representative $\text{rep}(u) = \text{mode}(B(u))$. Intuitively, $\text{rep}(u)$ is the node that might be convenient to replace u with using a permutation. Roughly speaking, in this step we get rid of all of the children of u that would be misplaced after permuting u and $\text{rep}(u)$, for each $u \in [n]$. The resulting forest F_1^2 has the following property: for each $u \in [n]$, for any child v of u in F_1^2 , either $b[v] = 0$ or $b[v] = \text{rep}(u)$, i.e., the children of each node u of F_1^2 have all the same parent $\text{rep}(u)$ in F_2 .

To bound the number of cuts in this step we first need a technical lemma relating the rearrangement distance of two forests and the size of any matching in their migrations graph.

► **Lemma 15.** *Consider two forests F_1 and F_2 and their migrations graph $MG(F_1, F_2)$. For any matching M in $MG(F_1, F_2)$ it holds that $|M| \leq \tilde{d}(F_1, F_2)$.*

Proof. By definition, there is an edge between i and j in $MG(F_1, F_2)$ if and only if $a[i] = a[j]$, but $b[i] \neq b[j]$. Let M be any matching in $MG(F_1, F_2)$. If $|M| > 0$ then $\tilde{d}(F_1, F_2) \geq 1$, so it is enough to show that, for a single operation transforming F_1 into F_1' , the graph $MG(F_1', F_2)$ contains a matching M' of size at least $|M| - s$, where $s = 1$ for a cut operation and $s = |\{u : u \neq \pi(u)\}|$ for a permutation operation π .

First, consider a cut operation $(v \dagger u)$. The only change in $MG(F_1', F_2)$ is removing all edges incident to v . M contains at most one edge incident to v , so we construct M' of size at least $|M| - 1$ from M by possibly removing a single edge. Second, consider a permutation operation π : we construct M' from M by removing every edge (v, w) such that $v \neq \pi(v)$ or $w \neq \pi(w)$. Because there is at most one edge incident to every u such that $u \neq \pi(u)$, M' contains at least $|M| - s$ edges. M' is a matching in $MG(F_1', F_2)$, as for every $(v, w) \in M'$ we have $p_{F_1'}(v) = p_{F_1}(v)$ and $p_{F_1'}(w) = p_{F_1}(w)$. ◀

► **Lemma 16.** $ALG(2) \leq 2\tilde{d}(F_1^1, F_2)$.

Proof. We consider each $u \in [n]$ separately. Let $m = \text{freq}_{B_u}(\text{mode}(B_u))$ and MG_u be the subgraph of $MG(F_1^1, F_2)$ induced by B_u . We will first construct a matching of appropriate size in every MG_u . We cut every (v_i, u) such that $b[v_i] \neq 0$ and $b[v_i] \neq \text{mode}(B_u)$, making $|B_u| - m$ cuts. Let $f = \min(|B_u| - m, \lfloor |B_u|/2 \rfloor)$. By Lemma 12, we can partition a subset of B_u into f pairs $(b[v_i], b[v_j])$ such that $b[v_i] \neq b[v_j]$. We add every edge (v_i, v_j) to the constructed matching. We claim that $|B_u| - m \leq 2f$. This holds because $|B_u| - m \leq 2(|B_u| - m)$ and $|B_u| - m \leq |B_u| - 1 \leq 2\lfloor |B_u|/2 \rfloor$ for nonempty B_u .

We take the union of all such matchings to obtain a single matching M . As argued above, the total number of cuts is at most $2|M|$. Together with Lemma 15, this implies that $ALG(2) \leq 2|M| \leq 2\tilde{d}(F_1^1, F_2)$. ◀

► **Example 17.** Consider again F_1 and F_2 of Figure 4. $B(7) = \{3, 3, 3, 2, 7\}$, thus we cut $(14 \dagger 7)$ and $(15 \dagger 7)$. $B(2) = \{3, 3, 7\}$, implying $(6 \dagger 2)$. The resulting F_1^2 is shown in Figure 5b.

4.3 Step 3

If after Step 2 all of the children of a node u of F_1 have the same parent $\text{rep}(u)$ in F_2 , it still may be the case where $\text{rep}(u) = \text{rep}(v)$ with $u \neq v$, i.e., all of the children of two distinct nodes of F_1 have the same parent in F_2 . In this case, it is not clear how to choose whether to replace u or v with $\text{rep}(u) = \text{rep}(v)$ in a permutation. This step aims at resolving this situation by cutting the ambiguous edges.

Consider thus $u \in [n]$, and let $\text{children}_{F_2}(u) = \{v_1, v_2, \dots, v_k\}$. We define the multiset $B'(u) = \{a[v_i] : a[v_i] \neq 0\}$ containing the parents in F_1^2 of the children of u in F_2 . We cut all edges $(v_i, a[v_i])$ such that $a[v_i] \neq 0$ and $a[v_i] \neq \text{mode}(B'(u))$, breaking ties arbitrarily, and define $\text{rep}'(u) = \text{mode}(B'(u))$. The resulting forest F_1^3 has the following property: for each $u \in [n]$, for any child v of u in F_2 , we have $a[v] = \perp$ or $a[v] = \text{rep}'(u)$.

We observe that the number of cuts performed by the above procedure is the same as if we had applied Step 2 on F_2 and F_1^2 . Therefore, Lemma 16 implies the following.

► **Lemma 18.** $ALG(3) \leq 2\tilde{d}(F_1^2, F_2)$.

► **Example 19.** Consider again F_1 and F_2 of Figure 4. We have $B'(3) = \{2, 2, 7, 7, 7\}$, we thus cut $(4 \dagger 2)$ and $(5 \dagger 2)$. The resulting forest F_1^3 is shown in Figure 5c.

4.4 Step 4

We summarize the properties of F_1^3 and F_2 :

1. For each $u \in [n]$ such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$, $a[u]$ and $b[u]$ are roots in F_1^3 .
2. For each $u \in [n]$ we can define $\text{rep}(u) \in [n]$ in such a way that, for any child v of u in F_1^3 , we have $b[v] = 0$ or $b[v] = \text{rep}(u)$.
3. For each $u \in [n]$ we can define $\text{rep}'(u) \in [n]$ in such a way that, for any child v of u in F_2 , we have $a[v] = 0$ or $a[v] = \text{rep}'(u)$.

To finish the description of the algorithm, we show how to find a permutation operation π of size $\mathcal{O}(\tilde{d}(F_1^3, F_2))$ that transforms F_1^3 into F_1^4 such that $F_1^4 \sim F_2$.

For every u such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$, we require that $\pi(a[u]) = b[u]$. Due to Property 1, for every such u we have ensured that $a[u]$ and $b[u]$ are roots of F_1^3 . So, if we can find a permutation π that satisfies all the requirements and does not perturb the non-roots of F_1^3 , then it will transform F_1^3 into F_1^4 such that $F_1^4 \sim F_2$. Furthermore, if for every x perturbed by π there exists u such that $a[u], b[u] \neq 0$ and $a[u] \neq b[u]$ with $x = a[u]$ or $x = b[u]$ then by Lemma 13 $|\pi| \leq 2|P(F_1^3, F_2)| \leq 4\tilde{d}(F_1^3, F_2)$ as required.

To see that there indeed exists such π , observe that due to Property 2 there cannot be two requirements $\pi(x) = y$ and $\pi(x) = y'$ with $y \neq y'$. Similarly, due to Property 3 there cannot be two requirements $\pi(x) = y$ and $\pi(x') = y$ with $x \neq x'$. Thinking of the requirements as a graph, the in- and out-degree of every node is hence at most 1, so we can add extra edges to obtain a collection of cycles defining a permutation π that does not perturb the nodes not participating in any requirement.

► **Example 20.** Consider F_1 and F_2 of Figure 4. $\pi = (3 \ 7)$ transforms F_1^3 into $F_1^4 \sim F_2$. The final F_1^4 is shown in Figure 5d.

References

- 1 Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemyslaw Uznanski, and Daniel Wolleb-Graf. Faster algorithms for all-pairs bounded min-cuts. In *46th ICALP*, pages 7:1–7:15, 2019.
- 2 Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. New algorithms and lower bounds for all-pairs max-flow in undirected graphs. In *31st SODA*, pages 48–61, 2020.
- 3 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- 4 Benjamin L Allen and Mike Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5(1):1–15, 2001.
- 5 Giulia Bernardini, Paola Bonizzoni, Gianluca Della Vedova, and Murray Patterson. A rearrangement distance for fully-labelled trees. In *30th CPM*, pages 28:1–28:15, 2019.
- 6 Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Mauricio Soto. Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In *8th ACM-BCB*, pages 1–10, 2017.
- 7 Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Mauricio Soto. Does relaxing the infinite sites assumption give better tumor phylogenies? an ilp-based comparative approach. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 16(5):1410–1423, 2018.
- 8 Magnus Bordewich and Charles Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics*, 8(4):409–423, 2005.
- 9 Robert S. Boyer and J. Strother Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–118. Kluwer Academic Publishers, 1991.
- 10 Gerth Støtting Brodal, Rolf Fagerberg, Thomas Mailund, Christian NS Pedersen, and Andreas Sand. Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In *24th SODA*, pages 1814–1832, 2013.
- 11 David Bryant. A classification of consensus methods for phylogenetics. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 61:163–184, 2003.
- 12 Peter Buneman. The recovery of trees from measures of dissimilarity. *Mathematics in the Archaeological and Historical Sciences*, 1971.
- 13 Simone Ciccolella, Giulia Bernardini, Luca Denti, Paola Bonizzoni, Marco Previtali, and Gianluca Della Vedova. Triplet-based similarity score for fully multi-labeled trees with poly-occurring labels, 2020. [arXiv:https://www.biorxiv.org/content/early/2020/04/14/2020.04.14.040550.full.pdf](https://www.biorxiv.org/content/early/2020/04/14/2020.04.14.040550.full.pdf).
- 14 Bhaskar DasGupta, Xin He, Tao Jiang, Ming Li, John Tromp, and Louxin Zhang. On distances between phylogenetic trees. In *8th SODA*, pages 427–436, 1997.
- 15 Zach DiNardo, Kiran Tomlinson, Anna Ritz, and Layla Oesper. Distance measures for tumor evolutionary trees. *Bioinformatics*, November 2019.
- 16 Annette J Dobson. Comparing the shapes of trees. In *Combinatorial Mathematics III*, pages 95–100. Springer, 1975.
- 17 Bartłomiej Dudek and Paweł Gawrychowski. Computing quartet distance is equivalent to counting 4-cycles. In *51st STOC*, pages 733–743, 2019.
- 18 George F Estabrook, FR McMorris, and Christopher A Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Systematic Zoology*, 34(2):193–200, 1985.
- 19 Joseph Felsenstein and Joseph Felsenstein. *Inferring phylogenies*, volume 2. Sinauer Associates Sunderland, MA, 2004.
- 20 Paweł Gawrychowski, Gad M. Landau, Wing-Kin Sung, and Oren Weimann. A faster construction of greedy consensus trees. In *45th ICALP*, pages 63:1–63:14, 2018.
- 21 Kiya Govek, Camden Sikes, and Layla Oesper. A consensus approach to infer tumor evolutionary histories. In *9th BCB*, pages 63–72, 2018.
- 22 Russell D Gray, Alexei J Drummond, and Simon J Greenhill. Language phylogenies reveal expansion pulses and pauses in pacific settlement. *Science*, 323(5913):479–483, 2009.

- 23 Iman Hajirasouliha, Ahmad Mahmoody, and Benjamin J Raphael. A combinatorial approach for analyzing intra-tumor heterogeneity from high-throughput sequencing data. *Bioinformatics*, 30(12):i78–i86, 2014.
- 24 John E. Hopcroft and Richard M. Karp. An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
- 25 Katharina T. Huber and Vincent Moulton. Phylogenetic networks from multi-labelled trees. *Journal of Mathematical Biology*, 52(5):613–632, 2006.
- 26 Daniel H Huson and David Bryant. Application of phylogenetic networks in evolutionary studies. *Molecular Biology and Evolution*, 23(2):254–267, 2006.
- 27 Jesper Jansson, Ramesh Rajaby, Chuanqi Shen, and Wing-Kin Sung. Algorithms for the majority rule (+) consensus tree and the frequency difference consensus tree. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 15(1):15–26, 2016.
- 28 Jesper Jansson, Chuanqi Shen, and Wing-Kin Sung. Improved algorithms for constructing consensus trees. *Journal of the ACM*, 63(3):1–24, 2016.
- 29 Wei Jiao, Shankar Vembu, Amit G Deshwar, Lincoln Stein, and Quaid Morris. Inferring clonal evolution of tumors from single nucleotide somatic mutations. *BMC bioinformatics*, 15(1):35, 2014.
- 30 Ming-Yang Kao, Tak Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. A decomposition theorem for maximum weight bipartite matchings. *SIAM J. Comput.*, 31(1):18–26, 2001.
- 31 Nikolai Karpov, Salem Malikic, Md Khaledur Rahman, and S Cenk Sahinalp. A multi-labeled tree dissimilarity measure for comparing “clonal trees” of tumor progression. *Algorithms for Molecular Biology*, 14(1):17, 2019.
- 32 Robert Krauthgamer and Ohad Trabelsi. Conditional lower bounds for all-pairs max-flow. *ACM Trans. Algorithms*, 14(4):42:1–42:15, 2018.
- 33 Yang P Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. *arXiv preprint arXiv:2003.08929*, 2020.
- 34 Salem Malikic, Farid Rashidi Mehrabadi, Simone Ciccolella, Md Khaledur Rahman, Camir Ricketts, Ehsan Haghshenas, Daniel Seidman, Faraz Hach, Iman Hajirasouliha, and S Cenk Sahinalp. Phiscs: a combinatorial approach for subperfect tumor phylogeny reconstruction via integrative use of single-cell and bulk sequencing data. *Genome Research*, 29(11):1860–1877, 2019.
- 35 Matt McVicar, Benjamin Sach, Cédric Mesnage, Jeffrey Lijffijt, Eirini Spyropoulou, and Tjil De Bie. Sumoted: An intuitive edit distance between rooted unordered uniquely-labelled trees. *Pattern Recognition Letters*, 79:52–59, 2016.
- 36 Luay Nakhleh, Tandy Warnow, Don Ringe, and Steven N Evans. A comparison of phylogenetic reconstruction methods on an indo-european dataset. *Transactions of the Philological Society*, 103(2):171–192, 2005.
- 37 Peter C Nowell. The clonal evolution of tumor cell populations. *Science*, 194(4260):23–28, 1976.
- 38 Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems*, 40(1):1–40, 2015.
- 39 David F Robinson and Leslie R Foulds. Comparison of weighted labelled trees. In *Combinatorial Mathematics VI*, pages 119–126. Springer, 1979.
- 40 David F Robinson and Leslie R Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131–147, 1981.
- 41 D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- 42 Mike Steel. *Phylogeny: discrete and random processes in evolution*. SIAM, 2016.
- 43 Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.
- 44 Robert S Walker, Søren Wichmann, Thomas Mailund, and Curtis J Atkisson. Cultural phylogenetics of the tupi language family in lowland south america. *PLOS One*, 7(4), 2012.

6:16 On Two Measures of Distance Between Fully-Labelled Trees

- 45 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *International Congress of Mathematicians*, 2018.
- 46 Ke Yuan, Thomas Sakoparnig, Florian Markowitz, and Niko Beerenwinkel. Bitphylogeny: a probabilistic framework for reconstructing intra-tumor phylogenies. *Genome Biology*, 16(1):36, 2015.
- 47 Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

String Sanitization Under Edit Distance

Giulia Bernardini 

University of Milano - Bicocca, Milan, Italy
giulia.bernardini@unimib.it

Huiping Chen

King's College London, UK
huiping.chen@kcl.ac.uk

Grigorios Loukides 

King's College London, UK
grigorios.loukides@kcl.ac.uk

Nadia Pisanti 

University of Pisa, Italy
ERABLE Team, Lyon, France
pisanti@di.unipi.it

Solon P. Pissis 

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France
solon.pissis@cwi.nl

Leen Stougie

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France
leen.stougie@cwi.nl

Michelle Sweering

CWI, Amsterdam, The Netherlands
michelle.sweering@cwi.nl

Abstract

Let W be a string of length n over an alphabet Σ , k be a positive integer, and \mathcal{S} be a set of length- k substrings of W . The ETFS problem asks us to construct a string X_{ED} such that: (i) no string of \mathcal{S} occurs in X_{ED} ; (ii) the order of all other length- k substrings over Σ is the same in W and in X_{ED} ; and (iii) X_{ED} has minimal edit distance to W . When W represents an individual's data and \mathcal{S} represents a set of confidential substrings, algorithms solving ETFS can be applied for utility-preserving string sanitization [Bernardini et al., ECML PKDD 2019]. Our first result here is an algorithm to solve ETFS in $\mathcal{O}(kn^2)$ time, which improves on the state of the art [Bernardini et al., arXiv 2019] by a factor of $|\Sigma|$. Our algorithm is based on a non-trivial modification of the classic dynamic programming algorithm for computing the edit distance between two strings. Notably, we also show that ETFS cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless the strong exponential time hypothesis is false. To achieve this, we reduce the edit distance problem, which is known to admit the same conditional lower bound [Bringmann and Künnemann, FOCS 2015], to ETFS.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases String algorithms, data sanitization, edit distance, dynamic programming, conditional lower bound

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.7

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 872539.

Giulia Bernardini: Partially supported by a research internship at CWI.

Huiping Chen: Supported by a Ph.D scholarship from China Scholarship Council.

Leen Stougie: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.

Michelle Sweering: Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.

Acknowledgements The authors would like to thank Takuya Mieno (Kyushu University) for proofreading the manuscript.



© Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 7; pp. 7:1–7:14



Leibniz International Proceedings in Informatics

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



1 Introduction

Strings are being used to represent individuals' data arising from a large range of domains e.g., transportation, web analytics, or molecular biology. For example, a string can represent: an individual's movement history, with each letter corresponding to a visited location [23, 20]; an individual's purchasing history in a retailer, with each letter corresponding to a purchased product [9]; or a fragment of the genome sequence of a patient, with each letter corresponding to a DNA base [17]. Analyzing such strings is thus necessary in many different applications. To support these applications, string data must often be disseminated beyond the party that has collected them. For example, transportation organizations disseminate their collected data to data analytics companies [14], retailers disseminate their data to marketing agencies [15], and hospitals disseminate DNA sequencing data to research institutions [17].

However, individuals' data dissemination has led to justified privacy concerns [22] due to the exposure of confidential information [3, 1, 6, 5]. To ease these concerns and comply with legislation such as HIPAA [8] in the US or GDPR [19] in the EU, it is essential to ensure that confidential information does not occur in the disseminated data; this process is called *sanitization*. At the same time, it is essential to preserve the *utility* of the original data, so that data processing and analysis tasks can be accurately performed on the disseminated data. For instance, a data analyst (resp. marketing researcher) should still be able to discover actionable patterns of locations (resp. purchased products) from transportation (resp. purchasing history) data [15, 14].

The *Combinatorial String Dissemination* (CSD) model was recently introduced in [3] to provide privacy and utility guarantees. In CSD, we are given a string W and the aim is to apply a sequence of edit operations to W , so that the resulting counterpart X of W satisfies a set of *privacy constraints* and a set of *utility properties*. Specifically, in [3] the authors considered the following CSD problem, referred to as TFS (Total order, Frequency, Sanitization). Given W of length n over an alphabet Σ , a positive integer k , and a set \mathcal{S} of *sensitive* length- k substrings of W modeling confidential information, construct the *minimal-length* string X such that: X does not contain any sensitive length- k substring (**C1**); and the order (and thus the frequency) of all other length- k substrings over Σ in W is the same as in X (**P1**). The constraint **C1** ensures that no sensitive length- k substring occurs in X . The property **P1** ensures that X incurs minimal utility loss for tasks based on the sequential nature of length- k non-sensitive substrings of W , as well as on their frequency. The authors of [3] proposed an $\mathcal{O}(kn)$ -time algorithm to solve the TFS problem, and showed that their algorithm is in fact worst-case optimal because the length of X is in $\Theta(kn)$.

The authors of [4] considered another CSD problem related to *edit distance*, that is, the minimum number of insertions, deletions or substitutions of letters needed to transform one string into another. The problem considered in [4] is referred to as ETFS (Edit distance, Total order, Frequency, Sanitization). Given W of length n over an alphabet Σ , a positive integer k , and a set \mathcal{S} of sensitive length- k substrings of W , ETFS asks us to construct a string X_{ED} that satisfies **C1** and **P1** while being at *minimal edit distance* from W . ETFS is the main problem we consider in this paper. Strings constructed by means of solving ETFS can be used, with minimal utility loss, in tasks that are based on edit distance as a similarity measure. Examples of such tasks are frequent pattern mining [21], clustering [12], entity extraction [24] and range query answering [16]. To solve ETFS, the authors of [4] proposed an $\mathcal{O}(k|\Sigma|n^2)$ -time algorithm. Their algorithm is based on solving a specific instance of approximate regular expression matching, essentially applying the algorithm of Myers and Miller [18] on an appropriate regular expression that models all strings satisfying **C1** and **P1** to finally pick the one that is at minimal edit distance to W .

Note that, to have a solution to TFS or ETFS, we may need to insert in W a letter $\# \notin \Sigma$. Indeed, inserting (or replacing letters of W with) any letter of Σ could violate **P1** and/or possibly create new occurrences of sensitive length- k substrings. We thus generally have that both X (the solution of TFS) and X_{ED} (a solution of ETFS) are over $\Sigma \cup \{\#\}$.

► **Example 1.** Let $W = \text{babaaaaabbbab}$, $\Sigma = \{\text{a, b}\}$, $k = 3$, and the set of sensitive substrings be $\{\text{aba, baa, aaa, aab, bba}\}$. Then $X = \text{babbb\#bab}$ and $X_{\text{ED}} = \text{bab\#aa\#abbb\#bab}$. Note that both X and X_{ED} satisfy **C1** and **P1**. X is the *shortest* such string and X_{ED} is a string *closest* to W in terms of edit distance.

In Section 3, we show the following theorem improving the result of [4] by a factor of $|\Sigma|$.

► **Theorem 2.** *The ETFS problem can be solved in $\mathcal{O}(kn^2)$ time.*

Our algorithm is based on a non-trivial modification of the classic dynamic programming algorithm for computing the edit distance between two given strings. In particular, the modification is based on the fact that in ETFS we are given a *single* string W , and we are asked to *construct* a string X_{ED} that satisfies **C1** and **P1** and that is closest to W . We thus actually fill in the dynamic programming matrix that computes the minimum edit distance between W and a regular expression that is a suitable abstraction of X_{ED} ; our algorithm encodes in its recurrence formulae the choices that specify the instance of the regular expression that we eventually output.

In Section 4, we also show that ETFS cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis (SETH) [11, 10] is false. This is the most technically involved part of the paper.

► **Theorem 3.** *The ETFS problem cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless SETH is false.*

To arrive at this theorem, we reduce the weighted edit distance problem, which is known to admit the same conditional lower bound [2, 7], to the ETFS problem. In particular, given two strings P and Q of length $\Theta(n)$, we construct an instance of ETFS of length $\mathcal{O}(n)$ from the output of which we can infer the insertions corresponding to some optimal alignment of P and Q with respect to the weighted edit distance. Using another suitable instance of ETFS, we can determine the corresponding deletions. That gives us an optimal alignment of P and Q , from which we can compute the weighted edit distance of P and Q in $\mathcal{O}(n)$ time.

2 Definitions and Notation

Let $T = T[0]T[1] \dots T[n-1]$ be a *string* of length $|T| = n$ over a finite alphabet Σ of size $|\Sigma| = \sigma$. By Σ^* we denote the set of all strings over Σ , and by Σ^k the set of all length- k strings over Σ . For two indices $0 \leq i \leq j \leq n-1$, $T[i..j] = T[i] \dots T[j]$ is the *substring* of T that starts at position i and ends at position j of T . By ε we denote the *empty string* of length 0. A *prefix* of T is a substring of the form $T[0..j]$, and a *suffix* of T is a substring of the form $T[i..n-1]$. A prefix (resp. suffix) of a string is *proper* if it is not equal to the string itself. The *reverse* $T[n-1]T[n-2] \dots T[0]$ of T is denoted by T^R . Given two strings U and V we say that U has a *suffix-prefix overlap* of length $\ell > 0$ with V if and only if the length- ℓ suffix of U is equal to the length- ℓ prefix of V , i.e., $U[|U| - \ell .. |U| - 1] = V[0 .. \ell - 1]$.

We fix a string W of length n over an alphabet Σ and an integer $0 < k < n$. We refer to a length- k string or a *pattern* interchangeably. An occurrence of a pattern is uniquely defined by its starting position. Let \mathcal{S} be the set representing the sensitive patterns as positions

over $\{0, \dots, n - k\}$ with the following closure property: for every $i \in \mathcal{S}$, any j for which $W[j..j+k-1] = W[i..i+k-1]$, must also belong to \mathcal{S} . That is, if an occurrence of a pattern is in \mathcal{S} , then all its occurrences are in \mathcal{S} . A substring $W[i..i+k-1]$ of W is called *sensitive* if and only if $i \in \mathcal{S}$; \mathcal{S} is thus the complete set of occurrences of sensitive patterns. The difference set $\mathcal{I} = \{0, \dots, n - k\} \setminus \mathcal{S}$ is the set of occurrences of *non-sensitive* patterns.

For any string U , we denote by \mathcal{I}_U the set of occurrences in U of non-sensitive length- k strings over Σ . (We have that $\mathcal{I}_W = \mathcal{I}$.) We call an occurrence i the *t-predecessor* of another occurrence j in \mathcal{I}_U if and only if i is the largest element in \mathcal{I}_U that is less than j . This relation induces a *strict total order* on the occurrences in \mathcal{I}_U . We call a subset \mathcal{J} of \mathcal{I}_U a *t-chain* if for all elements in \mathcal{J} except the minimum one, their t-predecessor is also in \mathcal{J} . For two strings U and V , chains \mathcal{J}_U and \mathcal{J}_V are *equivalent*, denoted by $\mathcal{J}_U \equiv \mathcal{J}_V$, if and only if $|\mathcal{J}_U| = |\mathcal{J}_V|$ and $U[u..u+k-1] = V[v..v+k-1]$, where u is the j -th smallest element of \mathcal{J}_U and v is the j -th smallest of \mathcal{J}_V , for all $j \leq |\mathcal{J}_U|$.

Given two strings U and V the *edit distance* $d_E(U, V)$ is defined as the minimum number of elementary edit operations (letter insertion, deletion, or substitution) that transform one string into the other. Each edit operation can also be associated with a cost: a fixed positive value. Given two strings U and V the *weighted edit distance* $d_{WE}(U, V)$ is defined as the minimal total cost of a sequence of edit operations to transform one string into the other.

We now formally define ETFS, the main problem considered in this paper.

► **Problem 1** (ETFS). *Given a string W of length n , an integer $k > 1$, and a set \mathcal{S} (and thus set \mathcal{I}), construct a string X_{ED} which is at minimal (weighted) edit distance from W and satisfies the following:*

C1 X_{ED} does not contain any sensitive pattern.

P1 $\mathcal{I}_W \equiv \mathcal{I}_{X_{ED}}$, i.e., the t-chains \mathcal{I}_W and $\mathcal{I}_{X_{ED}}$ are equivalent.

We also provide the following auxiliary definitions from [18]. The set of *regular expressions* over Σ is defined recursively as follows: (i) $a \in \Sigma \cup \{\varepsilon\}$ is a regular expression. (ii) If E and F are regular expressions, then so are EF , $E|F$, and E^* , where EF denotes the set of strings obtained by concatenating a string in E and a string in F , $E|F$ is the union of the strings in E and F , and E^* consists of all strings obtained by concatenating zero or more strings from E . Parentheses are used to override the natural precedence of the operators, which places the operator $*$ highest, the concatenation next, and the operator $|$ last. We say that a string T *matches* a regular expression E , if T is equal to one of the strings in E .

3 ETFS-DP: An $\mathcal{O}(kn^2)$ -time Algorithm for ETFS

In this section we describe ETFS-DP, a dynamic programming algorithm that solves ETFS faster than the algorithm proposed in [4]. We describe our algorithm for the unweighted edit distance model for simplicity, but it should be clear that it can be extended to the weighted edit distance model in a straightforward way and with no additional cost. Intuitively, since we are looking for a string X_{ED} that contains all the non-sensitive patterns of W , and in the same order, for each pair (U, V) of non-sensitive patterns of W such that U is the t-predecessor of V , we can (i) *merge* U and V into $U \cdot V[k-1]$ when U and V have a suffix-prefix overlap of length $k-1$; or (ii) *interleave* U and V constructing a string UYV , where Y is a carefully selected string over $\Sigma \cup \{\#\}$, where $\# \notin \Sigma$.

Let us start by defining a regular expression gadget \oplus , which encodes all candidate strings that can be used to interleave two non-sensitive patterns while respecting **C1**, and two similar gadgets \ominus and \otimes . We will make use of the following regular expression:

$$\Sigma^{<k} = \underbrace{((a_1|a_2|\dots|a_{|\Sigma|}|\varepsilon)\dots(a_1|a_2|\dots|a_{|\Sigma|}|\varepsilon))}_{k-1 \text{ times}}$$

where $\Sigma = \{a_1, a_2, \dots, a_{|\Sigma|}\}$ is the alphabet of W . Given a letter $\# \notin \Sigma$, we define

$$\oplus = \#(\Sigma^{<k}\#)^*, \quad \ominus = (\Sigma^{<k}\#)^*, \quad \otimes = (\# \Sigma^{<k})^*.$$

Let $N_0, N_1, \dots, N_{|\mathcal{I}|-1}$ be the sequence of non-sensitive patterns as they occur in W from left to right. In [3], X_{ED} was built by finding an optimal alignment between W and a regular expression R constructed as follows. First, set $R = \ominus N_0$ and then process pairs of non-sensitive patterns N_i and N_{i+1} , for all $i \in \{1, \dots, |\mathcal{I}| - 2\}$: in the i -th step, if N_i and N_{i+1} can be merged, append $(N_{i+1}[k-1] \mid \oplus N_{i+1})$ to R . Otherwise, append $\oplus N_{i+1}$ to R . After processing all pairs, conclude by appending \otimes to R . The length of R is in $\mathcal{O}(k|\Sigma|n)$.

The general idea in Algorithm ETFS-DP is to simulate the alignment of W to R without constructing R explicitly. Instead, we use a string $T = \boxminus N_0 \boxplus N_1 \cdots \boxplus N_{|\mathcal{I}|-1} \boxtimes$, where \boxminus , \boxplus and \boxtimes are length-1 *placeholders* for \ominus , \oplus and \otimes , respectively. The length of T is thus only $(k+1)|\mathcal{I}| + 1 = \mathcal{O}(kn)$, leading to an $\mathcal{O}(kn^2)$ -time algorithm when aligned to W , $|W| = n$.

3.1 Dynamic Programming

In a preprocessing phase, we compute a binary array M of length $|\mathcal{I}|$ so that $M[\ell] = 1$ if the ℓ -th and the $(\ell - 1)$ -th non-sensitive patterns (in the order given by their occurrences in W) can be merged. We set $M[0] = 0$ for completeness. More formally, for all $0 < \ell \leq |\mathcal{I}| - 1$, $M[\ell] = 1$ if $N_{\ell-1}[1..k-1] = N_\ell[0..k-2]$, and $M[\ell] = 0$ otherwise.

We then solve ETFS in a dynamic programming fashion by filling in an $(|\mathcal{I}|(k+1) + 1) \times (|W| + 1)$ matrix E . The rows of E correspond to string T , and the columns to string W . We denote by $E[i][\cdot]$ and $E[\cdot][j]$ the i -th row and the j -th column of E , respectively.

Entry $E[i][j]$, for all $0 \leq i \leq |\mathcal{I}|(k+1)$ and $0 \leq j \leq |W|$, contains the edit distance between (the regular expression corresponding to) $T[0..i]$ and $W[0..j-1]$. Rows corresponding to \boxplus , i.e., rows with index $i = \ell(k+1)$ for some $\ell \in [1, |\mathcal{I}| - 1]$, implicitly represent a regular expression gadget and must be filled in with ad hoc rules; we will refer to them as *gadgets rows*. In turn, we will name *possibly mergeable* the rows with index $i = \ell(k+1) - 1$ for some $\ell \in [1, |\mathcal{I}| - 1]$, as they must be filled in taking into account the option of merging the corresponding pattern with the preceding one, should it be possible. All other rows of E will be called *ordinary*. In what follows, I is a function such that $I[T[i] \neq W[j-1]] = 1$ if $T[i] \neq W[j-1]$, and 0 otherwise. We give below the recursive formulae that constitute the core of our dynamic programming algorithm.

Initialization. Entry $E[0][j]$ contains the edit distance between \ominus and $W[0..j-1]$ for $j \geq 1$, while $E[0][0] = 0$. Because of the definition of \ominus , it is only possible to match up to $k-1$ consecutive letters, after which a mismatch due to $\#$ occurs, and hence $E[0][j] = \lceil j/k \rceil$. $E[i][0]$ stores the edit distance between $T[0..i]$ and the empty prefix ε of W . This distance is minimized by the shortest possible string in each regular expression prefix, obtained by always merging when allowed, and picking the shortest possible string encoded by \oplus when not. This leads to the following formula, where $\ell \in [0, |\mathcal{I}| - 1]$.

$$E[i][0] = \begin{cases} E[i-k-1][0] + 1, & \text{if } i = (\ell+1)(k+1) - 1 \wedge M[\ell] = 1 \text{ (merge)} \\ E[i-1][0] + 1, & \text{otherwise (no merge)} \end{cases} \quad (1)$$

Ordinary Rows: $i \not\equiv 0 \pmod{k+1}$ and $i \not\equiv -1 \pmod{k+1}$. The formula is the same as in the standard algorithm for edit distance [13]: recall that $E[\cdot][j]$ correspond to $W[j-1]$.

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i][j-1] + 1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)} \end{cases} \quad (2)$$

Possibly Mergeable Rows: $i \equiv -1 \pmod{k+1}$. These rows correspond to the last letter of a non-sensitive pattern. The first three options of Equation 3 encode the case where we do not merge, regardless of the value of $M[\ell]$. The last two options, instead, require $M[\ell] = 1$, as a merge does take place. This means that the letters corresponding to the k rows above will not appear in the output string X_{ED} , and thus play no role in the edit distance computation. We thus read the values of row $i - k - 1$, corresponding to the last letter of the previous non-sensitive pattern.

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i][j-1] + 1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)} \\ E[i-k-1][j] + 1, & \text{if } M[\ell] = 1 \text{ (insert and merge)} \\ E[i-k-1][j-1] + I[T[i] \neq W[j-1]], & \text{if } M[\ell] = 1 \text{ (match or sub. and merge)} \end{cases} \quad (3)$$

Gadget Rows: $i \equiv 0 \pmod{k+1}$. A gadget row encodes the possibility of interleaving two non-sensitive patterns with a string that preserves **C1** and **P1** and minimizes the edit distance. Because of the form of the regular expression gadgets, a **#** can either be inserted or substituted directly after a non-sensitive pattern, or be preceded by another **#** no more than k positions earlier. This results in the following formula:

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i-1][j-1] + 1, & \text{(substitute)} \\ E[i][j-1] + 1, \dots, E[i][\max\{0, j-k\}] + 1, & \text{(delete or extend gadget)} \end{cases} \quad (4)$$

The following lemma states that the above formulae correctly compute the edit distance between prefixes of T and prefixes of W .

► **Lemma 4.** $E[i][j] = d_E(T[0..i], W[0..j-1])$, for all $0 \leq i < |\mathcal{I}|(k+1)$ and $0 < j \leq |W|$, and $E[i][0] = d_E(T[0..i], \varepsilon)$.

Proof. The correctness of the equations that describe how to fill in entries $E[0][j]$ and $E[i][0]$ follows from the explanation in paragraph “Initialization”, and the correctness of Equation 2 follows from the standard dynamic programming algorithm for edit distance [13]. Let us focus on the case of possibly mergeable rows (Equation 3): when merging is not possible, the equation is the same as in the standard algorithm, and therefore it is correct. When merging is possible, we must pick the minimum value among all possible edit operations when we actually choose to merge and among all possible operations when we do not merge, even if we could. The first three rows of Equation 3 correspond to the three possible operations when we do not merge, and are again the same possibilities as the standard algorithm for edit distance; the last two rows correspond to the case where we merge. When we merge, we append the letter corresponding to the possibly mergeable row to the previous non-sensitive pattern. If we were to run the standard algorithm for computing the edit distance between such string and W , the row above, where we had to read the values for insertion and match or substitution, would be the one corresponding to the last letter of the previous non-sensitive pattern. These are precisely the values of the last two rows of Equation 3, that are therefore correct.

Consider now the gadget rows. An entry $E[i][j]$ on a gadget row should contain the value of an optimal alignment between $W[0..j-1]$ and a prefix of X_{ED} that ends with a **#**: since $\# \notin \Sigma$, it cannot match with any letter of W , therefore $I[T[i] \neq W[j-1]] = 1$ always holds. As previously observed, a **#** can either be inserted or substituted directly after a non-sensitive

pattern, or be preceded by another $\#$ no more than k positions earlier. Moreover, it is easy to see that, if an optimal alignment between W and the regular expression R involves a local alignment between $W[i..j]$ and $\#S\#$ with $|S| = j - i - 1 < k$, then $S = W[i + 1..j - 1]$: this is because any alignment with $S \neq W[i + 1..j - 1]$ can be improved by replacing S with $W[i + 1..j - 1]$. Equation 4 follows from the two observations above: the first two lines compute the cost of appending a $\#$ directly after a non-sensitive pattern, that always entails either an insertion or a substitution.

The third row of the equation considers the possibility of interleaving two non-sensitive patterns with a whole string encoded by \oplus , or deleting $\#$. ◀

Note that Lemma 4 refers to rows $0 \leq i < |\mathcal{I}|(k + 1)$. Let us now look at the last row: even if it was filled in like any other gadget row, since it corresponds to \otimes instead of \oplus , its values need to be interpreted in a different way. Namely, the value stored in $E[|\mathcal{I}|(k + 1)][j]$, for all $0 \leq j \leq |W|$, is the cost of an optimal alignment between $W[0..j + e_j - 1]$ and a string in R whose length- $(e_j + 1)$ suffix is $\#W[j..j + e_j - 1]$, where $e_j = \min\{k - 1, |W| - j\}$.

Unlike in the standard edit distance algorithm [13], the edit distance between W and any string matching the regular expression R is not necessarily found in its bottom-right entry $E[|\mathcal{I}|(k + 1)][|W|]$. Instead, it is found among the rightmost k entries of the last row (in case X_{ED} ends with a string in \otimes), and the rightmost entry of the second-last row (when X_{ED} ends with the last letter of the last non-sensitive pattern). We thus obtain the following.

► **Lemma 5.** *Let X_{ED} be a solution to ETFS. Then*

$$d_E(X_{\text{ED}}, W) = \min \left\{ E[|\mathcal{I}|(k + 1) - 1][|W|], E[|\mathcal{I}|(k + 1)][|W|], \right. \\ \left. E[|\mathcal{I}|(k + 1)][|W| - 1], \dots, E[|\mathcal{I}|(k + 1)][|W| - k + 1] \right\}. \quad (5)$$

3.2 Construction of X_{ED}

Once we have computed the edit distance d according to Lemma 5, we need to construct a string X_{ED} that matches R and is at edit distance d from W . To do so, when computing each entry $E[i][j]$ of the matrix for $i, j \geq 1$, we store, in an array A , a *pointer* $\langle i', j' \rangle$ to an entry from which the minimum value for $E[i][j]$ was obtained. We then build X_{ED}^R by following any path from an entry $E[\bar{i}][\bar{j}]$ where the global optimum is stored to $E[0][0]$.

At any step of the construction, let $E[i'][j']$ be the endpoint of the pointer stored for $E[i][j]$ currently considered, i.e., $A[i][j] = \langle i', j' \rangle$. If $\bar{i} = |\mathcal{I}|(k + 1)$, i.e., if the minimum is in the last row of E , we initialize X_{ED}^R with $W[\bar{j}..|W| - 1]^R$; otherwise, we just initialize it with the empty string ε . We then enforce the following rules:

If $i' < i$, we append $T[i]$ to X_{ED}^R when i is not a gadget row and $\#$ otherwise. Indeed, the condition is fulfilled when the edge in the path is either diagonal (a match or a substitution in the alignment) or vertical (an insertion in W). Moreover, i' can either be equal to $i - 1$ or to $i - k - 1$ (when we merge two non-sensitive patterns).

If $i' = i$ and $i \equiv 0 \pmod{k + 1}$, we append $\#$ to X_{ED}^R followed by $W[j'..j - 2]^R$. Because this happens when we follow a horizontal edge on a gadget row, the solution must include the corresponding substring, that is composed of $\#$ and $j - j' - 1$ letters of W .

If none of the two cases above happens, we do not write anything, because a horizontal edge in the path corresponds to a deletion in W . We denote the above procedure by Algorithm X_{ED} -construct. Lemma 6 guarantees that this construction produces a string that satisfies **C1** and **P1**.

► **Lemma 6.** *X_{ED} returned by Algorithm X_{ED} -construct satisfies **C1** and **P1**.*

Proof. Let us start by proving that Algorithm $X_{\text{ED-construct}}$ satisfies **C1**. X_{ED}^R (and thus X_{ED}) is obtained by appending either consecutive letters of T^R (case $i' < i$ for all but gadget rows) or a letter $\#$ (all cases for gadget rows) or a number of consecutive letters of W^R (case $i' = i$ for gadget rows and initialization of X_{ED}^R when the minimum is on the last row of E): since T does not contain any sensitive patterns by construction and $\# \notin \Sigma$, we only need to verify that no more than $k - 1$ consecutive letters read directly from W^R can ever be appended to X_{ED}^R . Inspect case $i' = i$ for gadget rows: $j - j' - 1$ is the number of entries between entry $E[i][j]$ and the endpoint of the corresponding horizontal pointer $A[i][j]$. The last line of Equation 4 exhibits the only possibilities for a pointer to point a non-adjacent entry on the same row, thus $j' \geq j - k$ and consequently $j - j' - 1 \leq k - 1$. Since both when the path leaves a gadget row and when it goes on on a gadget row a $\#$ is appended to X_{ED}^R , no sensitive patterns can be created and therefore X_{ED} satisfies **C1**.

Let us now show that **P1** is satisfied as well, i.e., $N_0, N_1, \dots, N_{|\mathcal{I}|-1}$ occur in X_{ED} in the same order as they appear in W , and no other length- k string over Σ is a substring of X_{ED} . Consider a letter $N_\ell[h] = T[i]$. If $0 \leq h < k - 1$, i is an ordinary row. Since any optimal path goes from the entry of E where the minimum is stored to $E[0][0]$, and by construction to leave a row the pointers can only point to an entry in the row directly above the current one (ordinary rows) or in the $(k + 1)$ -th row above (merge case in the possibly mergeable rows, see Equations 2 and 3), there are only two possibilities: either the path goes through row i , i.e., there exists j such that $A[i][j] = \langle i - 1, j' \rangle$ is part of the optimal path, or row i is skipped by the path, and thus there exists j such that $A[i + k - h - 1][j] = \langle i - h - 2, j' \rangle$. Let us observe that in the latter case, all of the rows from $i - h - 1$ to $i + k - h - 2$ are skipped by the path, while in the first case $A[i + k - h - 1][j] = \langle i + k - h - 2, j' \rangle$ and no rows are skipped up to $i - h - 2$. In the first case, Algorithm $X_{\text{ED-construct}}$ will append $N_\ell[h]$ to X_{ED}^R after $N_\ell[h + 1]$ for all $0 \leq h \leq k - 1$, then a $\#$ right after $N_\ell[0]$, that prevents the making of spurious length- k strings over Σ in X_{ED} . In the second case, $N_\ell[h]$ is not explicitly appended to the string: instead, after appending $N_\ell[k - 1]$ to X_{ED}^R , the algorithm goes to row $i - h - 2$, corresponding to $N_{\ell-1}[k - 1]$. Nevertheless, this only happens when the merge condition is satisfied, i.e., when $N_{\ell-1}[1..k - 1] = N_\ell[0..k - 2]$, implying that $N_\ell[h] = N_{\ell-1}[h + 1]$ will be appended next to $N_\ell[h + 1] = N_{\ell-1}[h + 2]$ after $k - h - 1$ steps. The order in which $N_0, N_1, \dots, N_{|\mathcal{I}|-1}$ appear in X_{ED} is by construction the same as they appear in T , which in turn is the same as the order they appear in W . In no other parts of the algorithm a length- k string over Σ is created in X_{ED} . It follows that **P1** is preserved. \blacktriangleleft

3.3 Wrapping up

► **Lemma 7.** *Algorithm ETFS-DP runs in $\mathcal{O}(kn^2)$ time.*

Proof. We first construct string T and array M in $\mathcal{O}(kn)$ time and initialize the first row and the first column of matrix E in $\mathcal{O}(kn)$ time. There are $\mathcal{O}(kn)$ “ordinary”, $\mathcal{O}(n)$ “possibly mergeable” and $\mathcal{O}(n)$ “gadget rows”, each of size $\mathcal{O}(n)$. Each entry (and its corresponding pointer) on the “ordinary” and “possibly mergeable” rows takes constant time to compute, while the entries (and pointers) on the gadget rows require $\mathcal{O}(k)$ time each. Thus, we can compute all entries and pointers in $\mathcal{O}(kn^2)$ time. Tracing back the pointers and constructing string X_{ED} takes again $\mathcal{O}(kn^2)$ time. This results in a total time complexity of $\mathcal{O}(kn^2)$. \blacktriangleleft

Lemmas 4-7 imply Theorem 2.

4 A Conditional Lower Bound for ETFS

We prove that, assuming SETH introduced in [11] and [10], ETFS cannot be solved in strongly subquadratic time. We do so by a reduction from the classical edit distance problem, and using the following known conditional lower bound for it: for all $\delta > 0$, the edit distance d_E between two strings of length $\Omega(n)$ cannot be computed in $\mathcal{O}(n^{2-\delta})$ time without violating SETH [2], and hence the well-known quadratic-time solution of [13] for computing the edit distance between two strings of length $\mathcal{O}(n)$ is optimal up to subpolynomial factors. Bringmann and Künnemann [7] proved that this is also true for weighted edit distance, where each operation (insertion, deletion, substitution and match) has a corresponding fixed non-negative cost (respectively c_i, c_d, c_s, c_m), and the following conditions, which we will call the BK conditions, hold: (i) $c_i + c_d > c_m$, (ii) $c_i + c_d > c_s$, and (iii) $c_m \neq c_s$.

Let P and Q be two arbitrary strings over Σ , both of length $\Theta(n)$, and without loss of generality $1 \leq |P| \leq |Q|$. We would like to compute the weighted edit distance between P and Q with the following associated costs: $c_i = 2.5, c_d = 2.5, c_s = 1, c_m = 0$. These costs satisfy the BK conditions. Let $c = (c_i, c_d, c_s, c_m)$ and d_c be the weighted edit distance with associated costs c . Assuming SETH is true, there is no algorithm for computing $d_{(2.5, 2.5, 1, 0)}(P, Q)$ in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$ [7]. In order to prove that ETFS cannot be solved in strongly subquadratic time either, we will compute $d_{(2.5, 2.5, 1, 0)}(P, Q)$, by solving two instances of ETFS on a string of length $\mathcal{O}(n)$ and using an additional $\mathcal{O}(n)$ number of operations. Thus if ETFS is solvable in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, SETH is false.

Let us now show the first instance of the ETFS. We define a new alphabet $\Sigma' = \Sigma \sqcup \{a, b, c_1, c_2, c_3, d, e, f, g\}$ and a new string $U(P, Q) = F_1 F_2 F_3 F_4$ over Σ' as follows:

$$F_1 = (\text{aab})^{2x+1} \text{aae}, \quad F_2 = \prod_{i=0}^{|P|-1} c_1 d P[i] c_2 c_3, \quad F_3 = (\text{aae})^{2x-1} \text{aa}, \quad F_4 = \prod_{i=0}^{|Q|-1} c_1 f Q[i] c_2 c_3$$

where $x = 2|Q|$, and the product denotes the concatenation operation on strings. We also set $k = 5$ and define the set \mathcal{I} of non-sensitive pattern occurrences over U as follows:

$$\mathcal{I} = \{0, 3, 6, 9, \dots, 6x\} \cup \{6x + 6, 6x + 11, 6x + 16, \dots, 6x + 1 + 5|P|\}.$$

In particular, $U(P, Q)$ is the string input to the first instance of ETFS. The construction above gives us the following sequence of *non-sensitive* patterns:

$$\begin{aligned} & \text{aabaa, aabaa, aabaa, } \dots, \text{aabaa} && (2x + 1 \text{ occurrences}) \\ & c_1 d P[0] c_2 c_3, c_1 d P[1] c_2 c_3, c_1 d P[2] c_2 c_3, \dots, c_1 d P[|P| - 1] c_2 c_3 && (|P| \text{ occurrences}). \end{aligned}$$

It is easy to verify that the set \mathcal{I} of occurrences of non-sensitive patterns (and thus the complementary set \mathcal{S}) has the closure property requested by ETFS. The resulting regular expression R is

$$R = \ominus \text{aabaa} \oplus \text{aabaa} \oplus \dots \oplus \text{aabaa} \oplus c_1 d P[0] c_2 c_3 \oplus c_1 d P[1] c_2 c_3 \oplus \dots \oplus c_1 d P[|P| - 1] c_2 c_3 \otimes.$$

We will prove that it is optimal to align the first $x + 1$ patterns with F_1 , a gadget \oplus with F_2 , the next x patterns with F_3 and the final $|P|$ patterns with F_4 . Then, we will show that the alignment of those last patterns with F_4 corresponds to an alignment of P and Q .

We call the occurrences of **aabaa** and $c_1 d P[i] c_2 c_3$ in the regular expression R , or in any string in the regular language corresponding to R , *AB-patterns* and *P-patterns*, respectively. Notice that these non-sensitive patterns are substrings of F_1 and F_2 and that we cannot merge any two consecutive non-sensitive patterns.

7:10 String Sanitization Under Edit Distance

Recall that the output X_{ED} of ETFS is a string with minimal edit distance to U in that language. One alignment of U and R , which we denote by $\mathcal{A}_{U/R}$ and that we will later show to be optimal under unit cost for insertion, deletion and substitution and zero cost for match, is as follows:

- We align F_1 with the first $x + 1$ AB-patterns interleaved by #’s as illustrated below. The cost of this alignment is $x + 1$ substitutions.

```
aabaabaabaabaabaabaabaab...aabaae
aabaa#aabaa#aabaa#aabaa#...aabaa#
```

- We align F_2 with a single gadget \oplus suitably expanded as shown below. The cost of this alignment is $|P|$ substitutions. Recall that we have to use a # after every $k - 1 = 4$ letters, so as not to introduce any new length- k substrings that would violate property **P1**.

```
c1dP[0]c2c3c1dP[1]c2c3c1dP[2]c2c3c1dP[3]c2c3...c1dP[|P| - 1]c2c3
c1dP[0]c2# c1dP[1]c2# c1dP[2]c2# c1dP[3]c2# ...c1dP[|P| - 1]c2#
```

- We align F_3 with the remaining x AB-patterns interleaved by #’s as illustrated below. The cost of this alignment is $2x - 1$ substitutions.

```
aaeaaeaaeaaeaaeaaeaae...aaeaa
aabaa#aabaa#aabaa#aabaa#...aabaa
```

- We align F_4 with the final $|P|$ P-pattern occurrences according to an optimal alignment $\mathcal{A}_{P/Q}$ of P and Q with respect to cost c . Let \mathbf{p} and \mathbf{q} denote placeholders for letters of P and Q , respectively. For each edit operation in $\mathcal{A}_{P/Q}$ (insertion of \mathbf{q} , deletion of \mathbf{p} , substitution or match between \mathbf{p} and \mathbf{q}), we align in $\mathcal{A}_{U/R}$ the corresponding fragment of F_4 and the P-pattern of R as follows.

Insertion	Deletion	Substitution or Match
$c_1 \mathbf{f} \mathbf{q} c_2 c_3$	- - - - -	- $c_1 \mathbf{f} \mathbf{q} c_2 c_3$
$\# \mathbf{f} \mathbf{q} c_2 c_3$	$\# c_1 \mathbf{d} \mathbf{p} c_2 c_3$	$\# c_1 \mathbf{d} \mathbf{p} c_2 c_3$

When inserting a letter of Q , rather than paying 5 consecutive gaps opposite to fragment $c_1 \mathbf{f} \mathbf{q} c_2 c_3$ of F_4 , we extend the gadget \oplus of R with $\# \mathbf{f} \mathbf{q} c_2 c_3$, to pay only one (unavoidable) substitution for $\#$. Deleting a letter of P , instead, results in 6 gaps in $\mathcal{A}_{U/R}$. Finally, substitutions and matches in $\mathcal{A}_{P/Q}$ result in the same alignment in $\mathcal{A}_{U/R}$, with the cost being, respectively, 3 and 2 according to whether $\mathbf{q} = \mathbf{p}$ or not. Therefore, it turns out that the cost of this last fragment of alignment $\mathcal{A}_{U/R}$ equals $d_{(1,6,3,2)}(P, Q)$.

We next show that it is possible to express $d_{(1,6,3,2)}(P, Q)$ in terms of $d_{(2,5,2,5,1,0)}(P, Q)$, because symmetry will greatly simplify things later on, when we swap P and Q .

► **Lemma 8.** *Let c and c' be two costs. We write $c \sim c'$ if for any alphabet Σ and for all $P, Q \in \Sigma^*$, the set of optimal alignments of P and Q with respect to cost c is equal to the set of optimal alignments of P and Q with respect to cost c' . Then*

1. $c \sim \alpha c$ for all $\alpha \in \mathbb{R}_{>0}$.
2. $c \sim (c_i + \alpha, c_d, c_s + \alpha, c_m + \alpha)$ for all $\alpha \in \mathbb{R}$.
3. $c \sim (c_i, c_d + \alpha, c_s + \alpha, c_m + \alpha)$ for all $\alpha \in \mathbb{R}$.

Proof. Let the number of insertions, deletions, substitutions and matches in some alignment of P and Q be n_i , n_d , n_s and n_m respectively. We know that $n_i + n_s + n_m = |Q|$ and $n_d + n_s + n_m = |P|$. So the transformations 1, 2, and 3 of c given in the lemma statement change the costs of alignments from d to αd , $d + \alpha|Q|$ and $d + \alpha|P|$ respectively. The costs of alignments are all strictly increasing in d , so the optimal alignments are preserved. ◀

By applying transformation 2 of Lemma 8 with $\alpha = 1.5$ and then transformation 3 of Lemma 8 with $\alpha = -3.5$, we obtain

$$d_{(1,6,3,2)}(P, Q) = d_{(2.5,2.5,1,0)}(P, Q) - 1.5|Q| + 3.5|P|. \quad (6)$$

By summing up the costs of the alignment $\mathcal{A}_{U/R}$ detailed above and using Equation 6, we get

$$d_E(U, X_{ED}) \leq 4.5(|P| + |Q|) + d_{(2.5,2.5,1,0)}(P, Q), \quad (7)$$

which we can bound by $3|P| + 7|Q|$, because $d_{(2.5,2.5,1,0)}(P, Q) \leq 2.5(|Q| - |P|) + |P|$, corresponding to the cost of deleting the $(|Q| - |P|)$ extra letters of Q (recall that $|P| \leq |Q|$) and substituting the remaining $|P|$ letters. In Lemma 9 we prove that alignment $\mathcal{A}_{U/R}$ is indeed optimal and equality holds in Equation 7.

► **Lemma 9.** *Alignment $\mathcal{A}_{U/R}$ is optimal. Moreover, from any output X_{ED} of ETFS on U we can obtain a supersequence P' of P in $\mathcal{O}(|Q|)$ time such that $d_c(P, Q) = |P'| - |P| + d_c(P', Q)$ and there exists an optimal alignment of P' and Q , which does not use any insertions.*

The reader can probably share the intuition that alignment $\mathcal{A}_{U/R}$ is optimal, at least for the part $F_1F_2F_3$ of string U . We prove that indeed no AB-pattern is aligned to any part of F_4 and that no P -pattern is aligned to $F_1F_2F_3$ (see Example 10). The proof of Lemma 9 consists of a case analysis combined with basic counting and bounding arguments, for which we refer the reader to the full version of this paper.

► **Example 10.** Let $P = \text{KITTEN}$ and $Q = \text{SITTING}$ over $\Sigma = \{\text{E, G, I, K, N, S, T}\}$. We define a new alphabet $\Sigma' = \Sigma \sqcup \{\mathbf{a, b, c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{d, e, f, g}\}$ and a new string $U(P, Q) = F_1F_2F_3F_4$ over Σ' as follows (recall that $x = 2|Q|$, so $2x + 2 = 4Q + 2 = 30$):

$$\begin{aligned} F_1 &= \mathbf{aabaabaabaabaabaabaabaab \dots aabaae} \\ F_2 &= \mathbf{c_1dKc_2c_3c_1dIc_2c_3c_1dTc_2c_3c_1dTc_2c_3c_1dEc_2c_3c_1dNc_2c_3} \\ F_3 &= \mathbf{aaeaaeaaeaaeaaeaaeaaeaae \dots aaeaa} \\ F_4 &= \mathbf{c_1fSc_2c_3c_1fIc_2c_3c_1fTc_2c_3c_1fTc_2c_3c_1fIc_2c_3c_1fNc_2c_3c_1fGc_2c_3} \end{aligned}$$

We also set $k = 5$ and define the set \mathcal{I} of non-sensitive pattern occurrences over U as follows:

$$\mathcal{I} = \{0, 3, 6, 9, \dots, 6x\} \cup \{6x + 6, 6x + 11, 6x + 16, \dots, 6x + 1 + 5|P|\}.$$

We thus have the following sequence of occurrences of non-sensitive patterns:

$$\begin{aligned} &\mathbf{aabaa, aabaa, aabaa, \dots, aabaa} && (29 \text{ occurrences}) \\ &\mathbf{c_1dKc_2c_3, c_1dIc_2c_3, c_1dTc_2c_3c_1dTc_2c_3, c_1dEc_2c_3, c_1dNc_2c_3} && (6 \text{ occurrences}). \end{aligned}$$

Therefore, the corresponding regular expression R is

$$R = \ominus \mathbf{aabaa} \oplus \dots \oplus \mathbf{aabaa} \oplus \mathbf{c_1dKc_2c_3} \oplus \mathbf{c_1dIc_2c_3} \oplus \mathbf{c_1dTc_2c_3c_1dTc_2c_3} \oplus \mathbf{c_1dEc_2c_3} \oplus \mathbf{c_1dNc_2c_3} \otimes.$$

We now show the crucial fragment of alignment $\mathcal{A}_{U/R}$: how F_4 is aligned with the P -patterns.

```
-c_1fSc_2c_3-c_1fIc_2c_3-c_1fTc_2c_3-c_1fTc_2c_3-c_1fIc_2c_3-c_1fNc_2c_3c_1fGc_2c_3
#c_1dKc_2c_3#c_1dIc_2c_3#c_1dTc_2c_3#c_1dTc_2c_3#c_1dEc_2c_3#c_1dNc_2c_3# fGc_2c_3
```

Observe that the cost of the above alignment under unit cost equals to 15: the cost of 4 P -pattern matches (8), plus the cost of 2 P -pattern substitutions (6), plus the cost of 1 gadget insertion (1). It can be readily verified that $d_{(1,6,3,2)}(\text{KITTEN}, \text{SITTING}) = 15$.

7:12 String Sanitization Under Edit Distance

Lemma 9 implies the following result.

► **Corollary 11.** $d_E(U, X_{\text{ED}}) = 4.5(|P| + |Q|) + d_{(2.5, 2.5, 1, 0)}(P, Q)$.

Given that constructing U takes $\mathcal{O}(n)$ time, Corollary 11 tells us that, if we can compute $d_E(U, X_{\text{ED}})$ in strongly subquadratic time, then we can also compute d_c between any two strings in strongly subquadratic time contradicting SETH. In fact, to prove that the *output string* of ETFS cannot be computed in strongly subquadratic time either, we show that $d_{(2.5, 2.5, 1, 0)}$ can be obtained by solving ETFS twice and $\mathcal{O}(n)$ additional operations.

By Lemma 9, from the output X_{ED} of the ETFS algorithm, we can obtain a supersequence P' of P in $\mathcal{O}(n)$ time such that $d_c(P, Q) = d_c(P, P') + d_c(P', Q)$ and no insertions are required to optimally align P' and Q . There also exists a supersequence Q' of Q such that $d_c(P, Q) = d_c(P, P') + d_c(Q', Q) + d_c(P', Q')$ and some optimal alignment of P' and Q' which aligns each $P'[i]$ with $Q'[i]$ through either a match or a substitution. One such Q' is the string obtained by taking the alignment of P' and Q given by ETFS and inserting aligned letters of P' into the gaps of Q . The edit distance of Q and P is

$$d_c(P, P') + d_c(Q, Q') + d(P', Q') = |P'| - |P| + |Q'| - |Q| + \sum_{i=0}^{|P'|-1} I[P'[i] \neq Q'[i]], \quad (8)$$

which can be computed in $\mathcal{O}(n)$ time once we know P' and Q' .

Note that by using ETFS on $U(Q, P')$, we could find a supersequence Q'' of Q such that $d_c(P, Q) = d_c(P, P') + d_c(Q, Q'') + d_c(P', Q'')$ and no deletions are required to optimally align P' and Q'' . It is not necessarily the case that we do not need any more insertions, though, as optimal alignments are not unique. We now show that we can still compute an appropriate Q' by changing c .

Let \mathcal{Q}_c be the set of supersequences Q'' of Q with minimal $d_c(Q'', Q) + d_c(P', Q'')$ and no deletions needed in the alignment of P' and Q'' . Note that there exists some $Q' \in \mathcal{Q}_c$ such that $|P'| = |Q'|$. Increasing the cost of deletion by ϵ , $d_c(Q'', Q) + d_c(P', Q'')$ increases by at least $\epsilon(|P'| - |Q|)$ with equality if and only if $|Q''| = |P'|$. Since $|Q'| = |P'|$, no deletions implies no insertions. Therefore it suffices to find the insertions, when aligning P' and Q with weights $c' = (2.5, 2.5 + \epsilon, 1, 0)$ for some $\epsilon > 0$. We find these insertions by running the ETFS algorithm on $U(G(Q), G(P'))$ with $k = 5$, where $G(V) = \prod_{i=0}^{|V|-1} (V[i]g)$ for any string $V \in (\Sigma \sqcup \{\mathbf{a}, \mathbf{b}, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{d}, \mathbf{e}, \mathbf{f}\})^*$, and with the set of non-sensitive pattern occurrences

$$\mathcal{I} = \{0, 3, 6, 9, \dots, 6x'\} \cup \{6x' + 6, 6x' + 11, 6x' + 16, \dots, 6x' + 1 + |Q|\},$$

where $x' = 2|G(P')|$. The solution to this new problem corresponds to an optimal alignment of $G(Q)$ and $G(P')$ with $c = (2.5, 2.5, 1, 0)$, which in its turn corresponds to an optimal alignment of Q and P' with weights $c' = (5, 5, 1, 0) \sim (2.5, 2.5 + 5, 1, 0)$ by Lemma 8. We first carefully define what properties such a corresponding alignment should satisfy, and then prove that all optimal alignments of $G(Q)$ and $G(P')$ are indeed of this form.

► **Definition 12.** *The alignment of $G(P')$ and $G(Q)$ corresponding to an alignment $\mathcal{A}_{P'/Q}$ of P' and Q is defined as follows:*

- *If $P'[i]$ is aligned with $Q[i]$ in $\mathcal{A}_{P'/Q}$, then $G(P')[2i]$ and $G(P')[2i + 1]$ are aligned with $G(Q)[2i]$ and $G(Q)[2i + 1]$, respectively.*
- *If $P'[i]$ is deleted in $\mathcal{A}_{P'/Q}$, then $G(P')[2i]$ and $G(P')[2i + 1]$ are deleted.*
- *If $Q[i]$ is inserted in $\mathcal{A}_{P'/Q}$, then $G(Q)[2i]$ and $G(Q)[2i + 1]$ are inserted.*

► **Lemma 13.** *Let $P', Q \in \Sigma^*$ such that there exists an optimal alignment of P' and Q which does not include any insertions. Each optimal alignment of $G(P')$ and $G(Q)$ with respect to cost $c = (2.5, 2.5, 1, 0)$ corresponds to an optimal alignment of P' and Q with weights $c' = (5, 5, 1, 0)$.*

Proof. Let the number of insertions, deletions, substitutions and matches in some optimal alignment $\mathcal{A}_{P'/Q}$ of P' and Q be w , x , y and z respectively. The cost of $\mathcal{A}_{P'/Q}$ with respect to c' is $5w + 5x + y$. The corresponding alignment of $G(P')$ and $G(Q)$ has $2w$ insertions, $2x$ deletions, y substitutions and $2z + y$ matches, and its cost with respect to c is $(2w) \cdot 2.5 + (2x) \cdot 2.5 + y \cdot 1 + (2z + y) \cdot 0 = 5w + 5x + y$. Therefore $d_c(G(P'), G(Q)) \leq d_{c'}(P', Q)$. It remains to be shown that equality holds.

Consider an optimal alignment $\mathcal{A}_{G(P')/G(Q)}$ of $G(P')$ and $G(Q)$. We will show that we can transform $\mathcal{A}_{G(P')/G(Q)}$ into one corresponding to an alignment of P' and Q without increasing the edit distance. Consider the rightmost $P'[i]$ and $Q[j]$ where the corresponding alignment fails, and call them x and y . There are 13 possibilities for their alignment:

1	2	3	4	5	6	7	8	9	10	11	12	13
--xg	-xg	-x-g	-xg	-xg-	x-g	xg	xg-	x--g	x-g	x-g-	xg-	xg--
yg x g	y g g	y x g g	y x g	y x g g	y g g	y g	y g g	·y g g	·y g	·y g g	·y g	··y g

Blue letters are original letters of $G(Q)$, red letters are deleted letters from $G(P')$, dots are arbitrary strings and dashes denote gaps. Note that configurations 1, 7 and 13 are already properly aligned. Moreover, the cost can be reduced for configurations 2, 3, 4, 5, 6, 8, 9, 11 and 12 by deleting red letters and shifting blue ones. This only leaves configuration 10. Here there are 3 subcases:

- If x is aligned with an x , there must be a g between x and y . We can align this g with $G(P')[2i]$ and move to configuration 13 without increasing the cost nor changing letters.
- If x is aligned with an x , we move an adjacent inserted letter to this place and reduce the cost, which is a contradiction.
- Otherwise, x is aligned with a different letter. In this case we can realign it with y without increasing the cost or changing letters.

Since there is a corresponding alignment for the output string, equality holds. ◀

Therefore the output string is equal to $G(Q')$ for some $Q' \in \mathcal{Q}_c$. We can infer Q' in $\mathcal{O}(n)$ time and compute $d_c(P, Q)$ using Equation 8. However, since $d_c(P, Q)$ could not be computed in strongly subquadratic time given SETH, we conclude that ETFS cannot be computed in strongly subquadratic time either, unless SETH is false, thus proving Theorem 3.

5 Final Remarks

The following questions remain unanswered. Can ETFS be solved in $\mathcal{O}(n^2)$ time? Can ETFS be solved in strongly subquadratic time when $|\mathcal{S}| = \mathcal{O}(1)$?

References

- 1 O. Abul, F. Bonchi, and F. Giannotti. Hiding sequential and spatiotemporal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 22(12):1709–1723, 2010.
- 2 A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *47th ACM Annual Symposium on Theory of Computing (STOC)*, pages 51–58, 2015.
- 3 G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. Pissis, and G. Rosone. String sanitization: A combinatorial approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, pages 627–644, 2019.

- 4 G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. Pissis, G. Rosone, and M. Sweering. Combinatorial algorithms for string sanitization. *arXiv*, 2019.
- 5 G. Bernardini, H. Chen, G. Fici, G. Loukides, and S. P. Pissis. Reverse-safe data structures for text indexing. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 199–213, 2020.
- 6 L. Bonomi, L. Fan, and H. Jin. An information-theoretic approach to individual sequential data sanitization. In *9th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 337–346, 2016.
- 7 K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 79–97, 2015.
- 8 U.S. Department of Health & Human Services. Health Insurance Portability and Accountability Act. <https://aspe.hhs.gov/report/health-insurance-portability-and-accountability-act-1996>, 1996.
- 9 R. Gwadera, A. Gkoulalas-Divanis, and G. Loukides. Permutation-based sequential pattern hiding. In *13th IEEE International Conference on Data Mining (ICDM)*, pages 241–250, 2013.
- 10 R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and Systems Sciences*, 62(2):367–375, 2001.
- 11 R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and Systems Sciences*, 63(4):512–530, 2001.
- 12 L. Jin, C. Li, and R. Vernica. SEPIA: estimating selectivities of approximate string predicates in large databases. *The VLDB Journal*, 17(5):1213–1229, 2008.
- 13 V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.
- 14 A. Liu, K. Zhengy, L. Liz, G. Liu, L. Zhao, and X. Zhou. Efficient secure similarity computation on encrypted trajectory data. In *31st IEEE International Conference on Data Engineering (ICDE)*, pages 66–77, 2015.
- 15 G. Loukides and R. Gwadera. Optimal event sequence sanitization. In *SIAM International Conference on Data Mining (SDM)*, pages 775–783, 2015.
- 16 W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. Efficiently supporting edit distance based string similarity search using B^+ -trees. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):2983–2996, 2014.
- 17 B. Malin and L. Sweeney. Determining the identifiability of DNA database entries. In *American Medical Informatics Association Annual Symposium (AMIA)*, pages 537–541, 2000.
- 18 E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.
- 19 European Parliament. General Data Protection Regulation. <http://data.consilium.europa.eu/doc/document/ST-9565-2015-INIT/en/pdf>.
- 20 G. Poulis, S. Skiadopoulos, G. Loukides, and A. Gkoulalas-Divanis. Apriori-based algorithms for km-anonymizing trajectory data. *Transactions on Data Privacy*, 7:165–194, 2014.
- 21 J. Shang, J. Peng, and J. Han. MACFP: Maximal Approximate Consecutive Frequent Pattern Mining under edit distance. In *SIAM International Conference on Data Mining (SDM)*, pages 558–566, 2016.
- 22 H. J. Smith, T. Dinev, and H. Xu. Information privacy research: An interdisciplinary review. *MIS Quarterly*, 35(4):989–1015, 2011.
- 23 M. Terrovitis, G. Poulis, N. Mamoulis, and S. Skiadopoulos. Local suppression and splitting techniques for privacy preserving publication of trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 29(7):1466–1479, 2017.
- 24 Z. Wen, D. Deng, R. Zhang, and R. Kotagiri. 2ED: An Efficient Entity Extraction Algorithm using two-level Edit-Distance. In *35th IEEE International Conference on Data Engineering (ICDE)*, pages 998–1009, 2019.

Counting Distinct Patterns in Internal Dictionary Matching

Panagiotis Charalampopoulos 

King's College London, UK
University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Manal Mohamed 

London, UK
manalabd@gmail.com

Wojciech Rytter 

University of Warsaw, Poland
rytter@mimuw.edu.pl

Tomasz Waleń 


University of Warsaw, Poland
walen@mimuw.edu.pl

Tomasz Kociumaka 


Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Jakub Radoszewski 

University of Warsaw, Poland
Samsung R&D, Warsaw, Poland
jrad@mimuw.edu.pl

Juliusz Straszyński 

University of Warsaw, Poland
jks@mimuw.edu.pl

Wiktor Zuba 

University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

We consider the problem of preprocessing a text T of length n and a dictionary \mathcal{D} in order to be able to efficiently answer queries $\text{COUNTDISTINCT}(i, j)$, that is, given i and j return the number of patterns from \mathcal{D} that occur in the *fragment* $T[i..j]$. The dictionary is *internal* in the sense that each pattern in \mathcal{D} is given as a fragment of T . This way, the dictionary takes space proportional to the number of patterns $d = |\mathcal{D}|$ rather than their total length, which could be $\Theta(n \cdot d)$. An $\tilde{O}(n+d)$ -size¹ data structure that answers $\text{COUNTDISTINCT}(i, j)$ queries $\mathcal{O}(\log n)$ -approximately in $\tilde{O}(1)$ time was recently proposed in a work that introduced internal dictionary matching [ISAAC 2019]. Here we present an $\tilde{O}(n+d)$ -size data structure that answers $\text{COUNTDISTINCT}(i, j)$ queries 2-approximately in $\tilde{O}(1)$ time. Using range queries, for any m , we give an $\tilde{O}(\min(nd/m, n^2/m^2) + d)$ -size data structure that answers $\text{COUNTDISTINCT}(i, j)$ queries exactly in $\tilde{O}(m)$ time. We also consider the special case when the dictionary consists of all square factors of the string. We design an $\mathcal{O}(n \log^2 n)$ -size data structure that allows us to count distinct squares in a text fragment $T[i..j]$ in $\mathcal{O}(\log n)$ time.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases dictionary matching, internal pattern matching, squares

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.8

Related Version Full version at <https://arxiv.org/abs/2005.05681>.

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 677651).

Tomasz Kociumaka: Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.

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 Polish National Science Center, grant no. 2018/31/D/ST6/03991.

¹ The $\tilde{O}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors for inputs of size n .



1 Introduction

Internal Dictionary Matching was recently introduced in [5] as a generalization of Internal Pattern Matching. In the classical Dictionary Matching problem, we are given a dictionary \mathcal{D} consisting of d patterns, and the goal is to preprocess \mathcal{D} so that, presented with a text T , we can efficiently compute the occurrences of the patterns from \mathcal{D} in T . In Internal Dictionary Matching, the text T is given in advance, the dictionary \mathcal{D} is a set of fragments of T , and the Dictionary Matching queries can be asked for any fragment of T .

The Internal Pattern Matching problem consists in preprocessing a text T of length n so that we can efficiently compute the occurrences of a fragment of T in another fragment of T . A data structure of nearly linear size that allows for sublogarithmic-time Internal Pattern Matching queries was presented in [15], while a linear-size data structure allowing for constant-time Internal Pattern Matching queries in the case that the ratio between the lengths of the two factors is constant was presented in [18]. Other types of internal queries have been also studied; we refer the interested reader to [17].

In [5], several types of Internal Dictionary Matching queries about fragments $T[i..j]$ in a string T were considered: $\text{EXISTS}(i, j)$, $\text{REPORT}(i, j)$, $\text{REPORTDISTINCT}(i, j)$, $\text{COUNT}(i, j)$, $\text{COUNTDISTINCT}(i, j)$. Data structures of size $\tilde{O}(n + d)$ and query time $\tilde{O}(1 + \text{output})$ were shown for answering each of the first four queries, with COUNT queries requiring most advanced techniques. For COUNTDISTINCT queries, only a data structure answering these queries $\mathcal{O}(\log n)$ -approximately was shown. In this work, we focus on more efficient data structures for such queries. COUNTDISTINCT queries are formally defined as follows.

COUNTDISTINCT

Input: A text T of length n and a dictionary \mathcal{D} consisting of d patterns, each given as a fragment $T[a..b]$ of T (represented only by integers a, b).

Query: $\text{COUNTDISTINCT}(i, j)$: Count all distinct patterns $P \in \mathcal{D}$ that occur in $T[i..j]$.

Observe that the input size is $n + d$, while the total length of strings in \mathcal{D} could be $\Theta(n \cdot d)$.

We also consider a special case of this problem when the dictionary \mathcal{D} is the set of all squares (i.e., strings of the form UU) in T . The case that \mathcal{D} is the set of palindromes in T was considered by Rubinchik and Shur in [20].

► **Example 1.** Let us consider the following text:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	d	a	a	a	a	b	a	a	b	b	a	a	c

For the dictionary $\mathcal{D} = \{\text{aa}, \text{aaaa}, \text{abba}, \text{c}\}$, we have:

$$\text{COUNTDISTINCT}(5, 12) = 2, \quad \text{COUNTDISTINCT}(2, 6) = 2, \quad \text{COUNTDISTINCT}(2, 12) = 3.$$

In particular, $T[5..12]$ contains two distinct patterns from \mathcal{D} : **aa** (two occurrences) and **abba**. When the dictionary \mathcal{D} represents all squares in T , we have

$$\text{COUNTDISTINCT}(5, 12) = 3, \quad \text{COUNTDISTINCT}(2, 6) = 2, \quad \text{COUNTDISTINCT}(2, 12) = 4.$$

In particular, $T[5..12]$ contains three distinct squares: **aa** (two occurrences), **bb** and **aabaab**.

Let us note that one could answer $\text{COUNTDISTINCT}(i, j)$ queries in time $\mathcal{O}(j - i)$ by running $T[i..j]$ over the Aho–Corasick automaton of \mathcal{D} [1] or in time $\tilde{O}(d)$ by performing Internal Pattern Matching [18] for each element of \mathcal{D} individually. Neither of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

Our results and a roadmap. We start with preliminaries in Section 2 and an algorithmic toolbox in Section 3. Our results for the case of a static dictionary are summarized in Table 1. Our solutions exploit string periodicity using runs and use data structures for variants of the (colored) orthogonal range counting problem and for auxiliary internal queries on strings.

■ **Table 1** Our results for COUNTDISTINCT queries. Here, m is an arbitrary parameter.

Space	Preprocessing time	Query time	Variant	Section
$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$	2-approximation	4
$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.1
$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.2
$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\log n)$	$\mathcal{D} = \text{squares}$, exact	6

For the case of a dynamic dictionary, where queries are interleaved with insertions and deletions of patterns in the dictionary, it was shown in [5] that the product of the time to process an update and the time to answer an EXISTS(i, j) query cannot be $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$, unless the Online Boolean Matrix-Vector Multiplication conjecture [13] is false. In the full version of this paper, we outline a general scheme that adapts our data structures for the case of a dynamic dictionary. In particular, we show how to answer COUNTDISTINCT(i, j) queries 2-approximately in $\tilde{\mathcal{O}}(m)$ time and process each update in $\tilde{\mathcal{O}}(n/m)$ time, for any m .

2 Preliminaries

We begin with basic definitions and notation. Let $T = T[1]T[2] \cdots T[n]$ be a *string* of length $|T| = n$ over a linearly sortable alphabet Σ . The elements of Σ are called *letters*. By ε we denote an *empty string*. For two positions i and j on T , we denote by $T[i..j] = T[i] \cdots T[j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $j < i$). A fragment is called *proper* if $i > 1$ or $j < n$. A fragment of T is represented in $\mathcal{O}(1)$ space by specifying the indices i and j . A *prefix* of T is a fragment that starts at position 1 and a *suffix* is a fragment that ends at position n . By UV and U^k we denote the concatenation of strings U and V and k copies of the string U , respectively. A *cyclic rotation* of a string U is any string V such that $U = XY$ and $V = YX$ for some strings X and Y .

Let U be a string of length m with $0 < m \leq n$. We say that U is a *factor* of T if there exists a fragment $T[i..i+m-1]$, called an *occurrence* of U in T , that matches U . We then say that U occurs at the *starting position* i in T .

A positive integer p is called a *period* of T if $T[i] = T[i+p]$ for all $i = 1, \dots, n-p$. We refer to the smallest period as *the period* of the string, and denote it by $\text{per}(T)$. A string is called *periodic* if its period is no more than half of its length and *aperiodic* otherwise. The weak version of the periodicity lemma [9] states that if p and q are periods of a string T and satisfy $p+q \leq |T|$, then $\text{gcd}(p, q)$ is also a period of T . A string T is called *primitive* if it cannot be expressed as U^k for a string U and an integer $k > 1$.

The elements of the dictionary \mathcal{D} are called *patterns*. Henceforth, we assume that $\varepsilon \notin \mathcal{D}$, i.e., that the length of each $P \in \mathcal{D}$ is at least 1. We also assume that each pattern of \mathcal{D} is given by the starting and ending positions of its occurrence in T . Thus, the size of the dictionary $d = |\mathcal{D}|$ refers to the number of patterns in \mathcal{D} and not their total length. A *compact trie* of \mathcal{D} is the trie of \mathcal{D} in which all non-terminal nodes with exactly one child become implicit. The path-label $\mathcal{L}(v)$ of a node v is defined as the path-ordered concatenation of the string-labels of the edges in the root-to- v path. We refer to $|\mathcal{L}(v)|$ as the *string-depth* of v .

3 Algorithmic Tools

3.1 Modified Suffix Trees

A \mathcal{D} -modified suffix tree [5], denoted as $\mathcal{T}_{T,\mathcal{D}}$, of a given text T of length n and a dictionary \mathcal{D} is obtained from the trie of $\mathcal{D} \cup \{T[i..n] : 1 \leq i \leq n\}$ by contracting, for each non-terminal node u other than the root, the edge from u to the parent of u . As a result, all the nodes of $\mathcal{T}_{T,\mathcal{D}}$ (except for the root) correspond to patterns in \mathcal{D} or to suffixes of T . For $1 \leq i \leq n$, the node representing $T[i..n]$ is labelled with i ; see Figure 1. For a dictionary \mathcal{D} whose patterns are given as fragments of a text T , we can construct $\mathcal{T}_{T,\mathcal{D}}$ in $\mathcal{O}(|\mathcal{D}| + |T|)$ time [5].

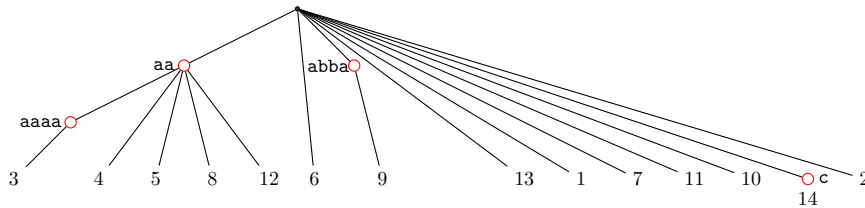


Figure 1 Example of a \mathcal{D} -modified suffix tree for text $T = \text{adaaaaabaabbaac}$ and dictionary $\mathcal{D} = \{\text{aa}, \text{aaaa}, \text{abba}, \text{c}\}$ (figure from [5]).

Let us denote by $\text{Occ}(\mathcal{D})$ the set of all occurrences of dictionary patterns in T , that is, the set of all fragments of T that match a pattern in \mathcal{D} . Using $\mathcal{T}_{T,\mathcal{D}}$, the set $\text{Occ}(\mathcal{D})$ can be computed in time $\mathcal{O}(n + d + |\text{Occ}(\mathcal{D})|)$.

We say that a tree is a *weighted tree* if it is a rooted tree with an integer weight on each node v , denoted by $\omega(v)$, such that the weight of the root is zero and $\omega(u) < \omega(v)$ if u is the parent of v . We say that a node v is a *weighted ancestor at depth ℓ* of a node u if v is the top-most ancestor of u with weight of at least ℓ .

► **Theorem 2** ([2, Section 6.2.1]). *After $\mathcal{O}(n)$ -time preprocessing, weighted ancestor queries for nodes of a weighted tree \mathcal{T} of size n can be answered in $\mathcal{O}(\log \log n)$ time per query.*

The \mathcal{D} -modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$ is a weighted tree with the weight of each node defined as the length of the corresponding string. We define the *locus* of a fragment $T[i..j]$ in $\mathcal{T}_{T,\mathcal{D}}$ to be the weighted ancestor of the leaf i at string-depth $j - i + 1$.

3.2 Auxiliary Internal Queries

In a *Bounded LCP* query, one is given two fragments U and V of T and needs to return the longest prefix of U that occurs in V ; we denote such a query by $\text{BoundedLCP}(U, V)$. Kociumaka et al. [18] presented several tradeoffs for this problem, including the following.

► **Lemma 3** ([18],[17, Corollary 7.3.4]). *Given a text T of length n , one can construct in $\mathcal{O}(n\sqrt{\log n})$ time an $\mathcal{O}(n)$ -size data structure that answers Bounded LCP queries in $\mathcal{O}(\log^\epsilon n)$ time, for any constant $\epsilon > 0$.*

Recall that $\text{COUNT}(i, j)$ returns the number of all occurrences of all the patterns of \mathcal{D} in $T[i..j]$. The following result was proved in [5].

► **Lemma 4** ([5]). *The $\text{COUNT}(i, j)$ queries can be answered in $\mathcal{O}(\log^2 n / \log \log n)$ time with an $\mathcal{O}(n + d \log n)$ -size data structure, constructed in $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$ time.*

3.3 Geometric Toolbox

For a set of n points in 2D, a range counting query returns the number of points in a given rectangle.

► **Theorem 5** (Chan and Pătraşcu [4]). *Range counting queries for n integer points in 2D can be answered in time $\mathcal{O}(\log n / \log \log n)$ with a data structure of size $\mathcal{O}(n)$ that can be constructed in time $\mathcal{O}(n\sqrt{\log n})$.*

A quarterplane is a range of the form $(-\infty, x_1] \times (-\infty, x_2]$. By reversing coordinates we can also consider quarterplanes with some dimensions of the form $[x_i, \infty)$. Let us state the following result on orthant color range counting due to Kaplan et al. [14] in the special case of two dimensions.

► **Theorem 6** ([14, Theorem 2.3]). *Given n colored integer points in 2D, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$ -size data structure that, given any quarterplane Q , counts the number of distinct colors with at least one point in Q in $\mathcal{O}(\log n)$ time.*

We show how to apply geometric methods to a special variant of the COUNTDISTINCT problem, where we are interested in a small subset of occurrences of each pattern.

Let $\mathcal{D} = \{P_1, P_2, \dots, P_d\}$ and \mathcal{S} be a family of sets S_1, \dots, S_d such that $S_k \subseteq \text{Occ}(P_k)$, where $\text{Occ}(P_k)$ is the set of positions of T where P_k occurs. Let $\|\mathcal{S}\| = \sum_k |S_k|$. For each pattern P_k , we call the positions in the set S_k the *special positions* of P_k . Counting distinct patterns occurring at their special positions in $T[i..j]$ is called $\text{COUNTDISTINCT}_{\mathcal{S}}(i, j)$.

► **Lemma 7.** *The $\text{COUNTDISTINCT}_{\mathcal{S}}(i, j)$ queries can be answered in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ that can be constructed in $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ time.*

Proof. We assign a different integer color c_k to every pattern $P_k \in \mathcal{D}$. Then, for each fragment $T[a..b] = P_k$ such that $a \in S_k$, we add point (a, b) with color c_k in an initially empty 2D grid \mathcal{G} . A $\text{COUNTDISTINCT}_{\mathcal{S}}(i, j)$ query reduces to counting different colors in the range $[i, \infty) \times (-\infty, j]$ of \mathcal{G} . The complexities follow from Theorem 6. ◀

3.4 Runs

A *run* (also known as a *maximal repetition*) is a periodic fragment $R = T[a..b]$ which can be extended neither to the left nor to the right without increasing the period $p = \text{per}(R)$, i.e., $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective positions exist. If \mathcal{R} is the set of all runs in a string T of length n , then $|\mathcal{R}| \leq n$ [3] and \mathcal{R} can be computed in $\mathcal{O}(n)$ time [19]. The *exponent* $\text{exp}(R)$ of a run R with period p is $|R|/p$. The sum of exponents of runs in a string of length n is $\mathcal{O}(n)$ [3, 19].

The *Lyndon root* of a periodic string U is the lexicographically smallest rotation of its $\text{per}(U)$ -length prefix. If L is the Lyndon root of a periodic string U , then U may be represented as (L, r, a, b) ; here $U = L[|L| - a + 1..|L|]L^rL[1..b]$, and r is called the *rank* of U . Note that the minimal rotation of a fragment of a text can be computed in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ -time preprocessing [16].

For a periodic fragment U , let $\text{run}(U)$ be the run with the same period that contains U .

► **Lemma 8** ([3, 7, 17]). *For a periodic fragment U , $\text{run}(U)$ and its Lyndon root are uniquely determined and can be computed in constant time after linear-time preprocessing.*

We use runs in 2-approximate $\text{COUNTDISTINCT}(i, j)$ queries and in counting squares.

4 Answering CountDistinct 2-Approximately

4.1 CountDistinct for Extended or Contracted Fragments

For two positions ℓ and r , we define $\text{Pref}_{\mathcal{D}}(\ell, r)$ as the longest prefix of $T[\ell..r]$ that matches some pattern $P \in \mathcal{D}$; the length of such prefix is at most $r - \ell + 1$. Let us show how to compute the locus of $\text{Pref}_{\mathcal{D}}(\ell, r)$ in the \mathcal{D} -modified suffix tree $\mathcal{T}_{T, \mathcal{D}}$. To this end, we preprocess $\mathcal{T}_{T, \mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T, \mathcal{D}}$ a pointer $p(v)$ to the nearest ancestor u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$. To return $\text{Pref}_{\mathcal{D}}(\ell, r)$, we find the locus u of $T[\ell..r]$ in the \mathcal{D} -modified suffix tree. We return $p(u)$ if $|\mathcal{L}(u)| = |T[\ell..r]|$ and $p(v)$, where v is the parent of u , otherwise.

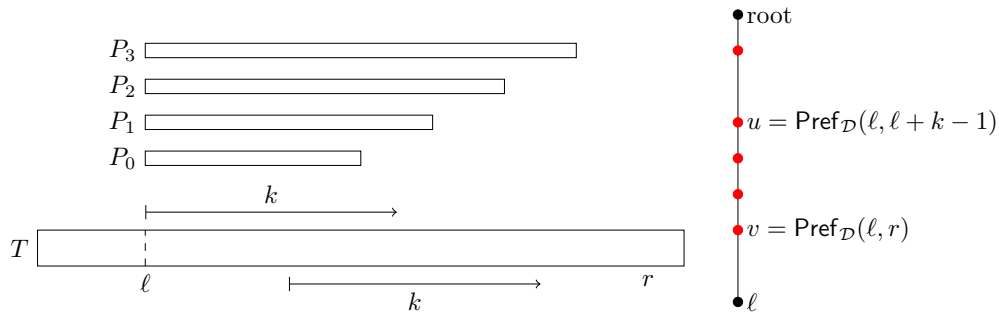
Lemma 9 applies the \mathcal{D} -modified suffix tree to the problem of maintaining the count of distinct patterns occurring in a fragment subject to extending or shrinking the fragment.

► **Lemma 9.** *For any constant $\epsilon > 0$, given $\text{COUNTDISTINCT}(i, j)$, one can compute $\text{COUNTDISTINCT}(i \pm 1, j)$ and $\text{COUNTDISTINCT}(i, j \pm 1)$ in $\mathcal{O}(\log^\epsilon n)$ time with an $\mathcal{O}(n + d)$ -size data structure that can be constructed in $\mathcal{O}(n\sqrt{\log n} + d)$ time.*

Proof. We only present a data structure for $\text{COUNTDISTINCT}(i \pm 1, j)$ queries. Queries $\text{COUNTDISTINCT}(i, j \pm 1)$ can be handled analogously by building the same data structure for the reverses of all the strings in scope.

We show how to compute the number of patterns $P \in \mathcal{D}$ whose only occurrence in some fragment $T[\ell..r]$ starts at position ℓ . The computation of $\text{COUNTDISTINCT}(i \pm 1, j)$ follows directly by setting $j = r$ and ℓ equal to $i - 1$ or i .

Data structure. We preprocess T for Bounded LCP queries (Lemma 3) and construct the \mathcal{D} -modified suffix tree $\mathcal{T}_{T, \mathcal{D}}$ of text T and dictionary \mathcal{D} . In addition, we preprocess $\mathcal{T}_{T, \mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T, \mathcal{D}}$ the number $\#(v)$ of the ancestors u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$.



■ **Figure 2** The setting of Lemma 9. Left: text T . Right: the path from the root of $\mathcal{T}_{T, \mathcal{D}}$ to the leaf with path-label $T[\ell..r]$. The nodes of the path whose path-labels match some patterns from \mathcal{D} are drawn in red. Here, P_0 is the longest pattern that occurs at ℓ and also has an occurrence in $T[\ell + 1..r]$; its locus in $\mathcal{T}_{T, \mathcal{D}}$ is $u = \text{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$. The patterns that occur in $T[\ell..r]$ only at position ℓ are P_1, P_2 and P_3 . The locus of P_3 is $v = \text{Pref}_{\mathcal{D}}(\ell, r)$. Then, $\#(v) - \#(u) = 5 - 2 = 3$.

Query. We want to count patterns longer than $k = |\text{BoundedLCP}(T[\ell..r], T[\ell + 1..r])|$. Let $u = \text{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$ and $v = \text{Pref}_{\mathcal{D}}(\ell, r)$. The desired number of patterns is equal to $\#(v) - \#(u)$. See Figure 2 for a visualization. ◀

4.2 Auxiliary Operation

Two fragments $U = T[i_1..j_1]$ and $V = T[i_2..j_2]$ are called *consecutive* if $i_2 = j_1 + 1$. We denote the overlap $T[\max\{i_1, i_2\}.. \min\{j_1, j_2\}]$ of U and V by $U \cap V$.

3-FRAGMENTS-COUNTING

Input: A text T of length n and a dictionary \mathcal{D} consisting of d patterns

Query: Given three consecutive fragments F_1, F_2, F_3 in T such that $|F_1| = |F_3|$ and $|F_2| \geq 8 \cdot |F_1|$, count distinct patterns P from \mathcal{D} that have an occurrence starting in F_1 and ending in F_3 and do not occur in either F_1F_2 or F_2F_3

Let us fix $|F_1| = |F_3| = x$ and $|F_2| = y \geq 8x$. Additionally, let us call an occurrence of $P \in \mathcal{D}$ that starts in fragment F_a and ends in fragment F_b an (F_a, F_b) -occurrence. We will call an (F_1, F_3) -occurrence an *essential occurrence*.

We say that a string S is *highly periodic* if $\text{per}(S) \leq \frac{1}{4}|S|$. We first consider the case that all patterns in \mathcal{D} are not highly periodic.

► **Lemma 10.** *If each $P \in \mathcal{D}$ is not highly periodic, then*

$$\begin{aligned} 3\text{-FRAGMENTS-COUNTING}(F_1, F_2, F_3) = \\ \text{COUNT}(F_1F_2F_3) - \text{COUNT}(F_1F_2) - \text{COUNT}(F_2F_3) + \text{COUNT}(F_2). \end{aligned}$$

Proof. Let us start with the following claim.

▷ **Claim 11.** Any $P \in \mathcal{D}$ that has an essential occurrence occurs exactly once in $F_1F_2F_3$.

Proof. We have $|F_1F_2F_3| = x + y + x = 2x + y$. String P has an essential occurrence, so $|P| \geq y$. Therefore, if there are two occurrences of P in $F_1F_2F_3$, then they overlap in

$$2|P| - (2x + y) \geq 2|P| - \left(\frac{1}{4}|P| + |P|\right) = \frac{3}{4}|P|$$

positions. This implies that P is highly periodic, which is a contradiction. ◁

Claim 11 shows that $3\text{-FRAGMENTS-COUNTING}(F_1, F_2, F_3)$ is equal to the number of essential occurrences. Let us prove that the stated formula does not count any (F_a, F_b) -occurrences other than (F_1, F_3) -occurrences.

- Each (F_1, F_2) -occurrence is registered when we add $\text{COUNT}(F_1F_2F_3)$ and unregistered when we subtract $\text{COUNT}(F_1F_2)$. Similarly for (F_2, F_3) -occurrences.
- Each (F_2, F_2) -occurrence is registered when we add $\text{COUNT}(F_1F_2F_3)$, $\text{COUNT}(F_2)$ and unregistered when we subtract $\text{COUNT}(F_1F_2)$, $\text{COUNT}(F_2F_3)$.
- Each (F_1, F_1) -occurrence is registered when we add $\text{COUNT}(F_1F_2F_3)$ and unregistered when we subtract $\text{COUNT}(F_1F_2)$. Similarly for (F_3, F_3) -occurrences. ◀

We now proceed with answering 3-FRAGMENTS-COUNTING queries for the dictionary of highly periodic patterns.

► **Lemma 12.** *If F_2 is aperiodic, then there are no essential occurrences of highly periodic patterns. Otherwise, all essential occurrences of highly periodic patterns are generated by the same run, that is, $\text{run}(F_2)$.*

Proof. The first claim follows from the fact that such an occurrence of a pattern $P \in \mathcal{D}$ has an overlap of length at least $2\text{per}(P)$ with F_2 and hence $\text{per}(P) \leq \frac{1}{2}|F_2|$ is a period of F_2 .

As for the second claim, it suffices to show that, for any pattern $P \in \mathcal{D}$ that has an essential occurrence, we have $\text{per}(P) = \text{per}(F_2)$. The inequalities $|F_2| \geq 2\text{per}(F_2)$ and $|F_2| \geq 2\text{per}(P)$ imply $|F_2| \geq \text{per}(F_2) + \text{per}(P)$. Hence, by the periodicity lemma, $q = \gcd(\text{per}(P), \text{per}(F_2))$ is a period of F_2 . As $q \leq \text{per}(F_2)$, we conclude that $q = \text{per}(F_2)$. Thus, $\text{per}(F_2)$ divides $\text{per}(P)$, and therefore $\text{per}(P) = \text{per}(F_2)$. This concludes the proof. ◀

For a periodic factor U of T , let $\text{PERIODIC}(U)$ denote the set of distinct patterns from \mathcal{D} that occur in U and have the same shortest period. Let us make the following observation.

► **Observation 13.** *If all $P \in \mathcal{D}$ are highly periodic, F_2 is periodic, and $R = \text{run}(F_2)$, then*

$$3\text{-FRAGMENTS-COUNTING}(F_1, F_2, F_3) = |\text{PERIODIC}(F_1F_2F_3 \cap R)| - |\text{PERIODIC}(F_1F_2 \cap R) \cup \text{PERIODIC}(F_2F_3 \cap R)|.$$

Next we now show how to efficiently evaluate the right-hand side of the formula in the observation above, using Theorem 5 for efficiently answering range counting queries in 2D.

We group all highly periodic patterns by Lyndon root and rank; for a Lyndon root L and a rank r , we denote by $\mathcal{D}_{L,r}^p$ the corresponding set of patterns. Then, we build the data structure of Theorem 5 for the set of points obtained by adding the point (a, b) for each $(L, r, a, b) \in \mathcal{D}_{L,r}^p$. We refer to the 2D grid underlying this data structure as $\mathcal{G}_{L,r}$. Note that the total number of points in the data structures over all Lyndon roots and ranks is $\mathcal{O}(d)$.

Each occurrence of a pattern (L, r, a, b) lies within some run in \mathcal{R} with Lyndon root L . Let us state a simple fact.

► **Fact 14.** *A periodic string (L, r, a, b) occurs in a periodic string (L, r', a', b') if and only if at least one of the following conditions is met:*

- (1) $r = r'$, $a \leq a'$, and $b \leq b'$;
- (2) $r = r' - 1$ and $a \leq a'$;
- (3) $r = r' - 1$ and $b \leq b'$;
- (4) $r \leq r' - 2$.

► **Lemma 15.** *One can compute $|\text{PERIODIC}(U)|$ for any periodic fragment U in time $\mathcal{O}(\log n / \log \log n)$ using a data structure of size $\mathcal{O}(n + d)$ that can be constructed in time $\mathcal{O}(n + d\sqrt{\log n})$.*

Proof. For $U = (L, r, a, b)$, we count points contained in at least one of the rectangles

- (1) $(-\infty, a] \times (-\infty, b]$ in $\mathcal{G}_{L,r}$,
- (2) $(-\infty, a] \times (-\infty, |L|]$ in $\mathcal{G}_{L,r-1}$,
- (3) $(-\infty, |L|] \times (-\infty, b]$ in $\mathcal{G}_{L,r-1}$,

and we add to the count the number of patterns of the form (L, r', a, b) with $r' < r - 1$. For the latter term, it suffices to store an array $X_L[1..t]$ such that $X_L[r] = \sum_{i=1}^r |\mathcal{D}_{L,i}^p|$, where t is the maximum rank of a pattern with Lyndon root L . The total size of these arrays is $\mathcal{O}(n)$ by the linearity of the sum of exponents of runs in a string [3, 19]. ◀

► **Remark 16.** In particular, in the proof of the above lemma, we count points that are contained within at least one out of a constant number of rectangles. Therefore, not only we can easily compute $|\text{PERIODIC}(U)|$, but similarly we are able to compute $|\text{PERIODIC}(U_1) \cup \text{PERIODIC}(U_2)|$ for some periodic factors U_1, U_2 of T .

We are now ready to prove the main result of this subsection.

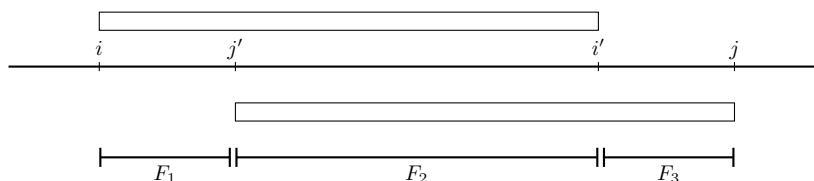
► **Lemma 17.** *The 3-FRAGMENTS-COUNTING(F_1, F_2, F_3) queries can be answered in time $\mathcal{O}(\log^2 n / \log \log n)$ with a data structure of size $\mathcal{O}(n + d \log n)$ that can be constructed in $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$ time.*

Proof. By Lemma 10, in order to count the patterns that are not highly periodic, it suffices to perform three COUNT queries. To this end, we employ the data structure of Lemma 4 which answers COUNT queries in $\mathcal{O}(\log^2 n / \log \log n)$ time, occupies space $\mathcal{O}(n + d \log n)$, and can be constructed in time $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$.

We now proceed to counting highly periodic patterns. First, we check whether F_2 is periodic; this can be done in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$ -time preprocessing of T [18, 17]. If F_2 is not periodic, then by Lemma 12 no highly periodic pattern has an essential occurrence, and we are thus done. If F_2 is periodic, three $|\text{PERIODIC}(U)|$ queries suffice to obtain the answer due to Observation 13. They can be efficiently answered due to Lemma 15 and Remark 16; the complexities are dominated by those for building the data structure for COUNT queries. ◀

4.3 Approximation Algorithm

Let us fix $\delta = \frac{1}{9}$. A fragment of length $\lfloor (1 + \delta)^p \rfloor$ for any positive integer p will be called a *p-basic fragment*. Our data structure stores $\text{COUNTDISTINCT}(i, j)$ for every basic fragment $T[i..j]$. Using Lemma 9, these values can be computed in $\mathcal{O}(n \log^{1+\epsilon} n + d)$ time with a sliding window approach. The space requirement is $\mathcal{O}(n \log n + d)$.



■ **Figure 3** A 2-approximation of $\text{COUNTDISTINCT}(i, j)$ is achieved using precomputed counts for basic factors $T[i..i']$ and $T[j'..j]$.

In order to answer an arbitrary $\text{COUNTDISTINCT}(i, j)$ query, let $T[i..i']$ and $T[j'..j]$ be the longest prefix and suffix of $T[i..j]$ being a basic factor; see Figure 3. We sum up $\text{COUNTDISTINCT}(i, i')$ and $\text{COUNTDISTINCT}(j', j)$ and the result of a 3-FRAGMENTS-COUNTING query for $F_1 = T[i..j' - 1]$, $F_2 = T[j'..i']$, $F_3 = T[i' + 1..j]$. (Note that $(|F_1| + |F_2|) \cdot (1 + \delta) > |F_1| + |F_2| + |F_3|$ implies $\delta(|F_1| + |F_2|) > |F_3|$, and since $|F_1| = |F_3|$, we have that $|F_1| = |F_3| \leq \frac{1}{8}|F_2|$.) Now, a pattern $P \in \mathcal{D}$ is counted at least once if and only if it occurs in $T[i..j]$. Also, a pattern $P \in \mathcal{D}$ is counted at most twice (exactly twice if and only if it occurs in both F_1F_2 and F_2F_3). The above discussion and Lemma 17 yield the following result.

► **Theorem 18.** *The $\text{COUNTDISTINCT}(i, j)$ queries can be answered 2-approximately in time $\mathcal{O}(\log^2 n / \log \log n)$ with a data structure of size $\mathcal{O}((n + d) \log n)$ that can be constructed in time $\mathcal{O}(n \log^{1+\epsilon} n + d \log^{3/2} n)$ for any constant $\epsilon > 0$.*

5 Time-Space Tradeoffs for Exact Counting

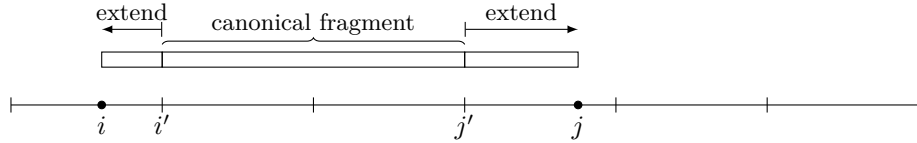
5.1 Tradeoff for Large Dictionaries

The following result is yet another application of Lemma 9.

► **Theorem 19.** For any $m \in [1, n]$ and any constant $\epsilon > 0$, the $\text{COUNTDISTINCT}(i, j)$ queries can be answered in $\mathcal{O}(m \log^\epsilon n)$ time using an $\mathcal{O}(n^2/m^2 + n + d)$ -size data structure that can be constructed in $\mathcal{O}((n^2 \log^\epsilon n)/m + n\sqrt{\log n} + d)$ time.

Proof. A fragment of the form $T[c_1m+1 \dots c_2m]$ for integers c_1 and c_2 will be called a *canonical fragment*. Our data structure stores $\text{COUNTDISTINCT}(i', j')$ for every canonical fragment $T[i' \dots j']$ and the data structure of Lemma 9. Hence the space complexity $\mathcal{O}(n^2/m^2 + n + d)$.

We can compute in $\mathcal{O}(n \log^\epsilon n)$ time $\text{COUNTDISTINCT}(i', j)$ for a given i' and all j using Lemma 9. There are $\mathcal{O}(n/m)$ starting positions of canonical fragments and hence the counts for all canonical fragments can be computed in $\mathcal{O}((n^2 \log^\epsilon n)/m)$ time. Additional preprocessing time $\mathcal{O}(n\sqrt{\log n} + d)$ originates from Lemma 9.



■ **Figure 4** An illustration of the setting in the query algorithm underlying Theorem 19.

We can answer a $\text{COUNTDISTINCT}(i, j)$ query in $\mathcal{O}(m \log^\epsilon n)$ time as follows. Let $T[i' \dots j']$ be the maximal canonical fragment contained in $T[i \dots j]$. We retrieve $\text{COUNTDISTINCT}(i', j')$ for $T[i' \dots j']$. Then, we apply Lemma 9 $\mathcal{O}(m)$ times; each time we extend the fragment for which we count, until we obtain $\text{COUNTDISTINCT}(i, j)$. See Figure 4. ◀

5.2 Tradeoff for Small Dictionaries

We call a set of strings \mathcal{H} a *path-set* if all elements of \mathcal{H} are prefixes of its longest element. We now show how to efficiently handle dictionaries that do not contain large path-sets.

► **Lemma 20.** If \mathcal{D} does not contain any path-set of size greater than k , then we can construct in $\mathcal{O}(kn \log n)$ time an $\mathcal{O}(kn \log n)$ -size data structure that answers $\text{COUNTDISTINCT}(i, j)$ queries in $\mathcal{O}(\log n)$ time.

Proof. Let $\mathcal{D} = \{P_1, \dots, P_d\}$ and $\mathcal{S} = \{\text{Occ}(P_1), \dots, \text{Occ}(P_d)\}$. Every position of T contains at most k occurrences of patterns from \mathcal{D} . This implies that $\|\mathcal{S}\| \leq kn$. We can obviously treat a $\text{COUNTDISTINCT}(i, j)$ query as a $\text{COUNTDISTINCT}_{\mathcal{S}}(i, j)$ query. The complexities follow from Lemma 7. ◀

A proof of the following lemma is rather standard and is included in the full version of the paper.

► **Lemma 21.** For any $k \in [1, n]$, we can compute a maximal family \mathcal{F} of pairwise-disjoint path-sets in \mathcal{D} , each consisting of at least k elements, in $\mathcal{O}(n + d)$ time.

We now combine Lemmas 3, 20 and 21 to get the main result of this section.

► **Theorem 22.** For any $m \in [1, n]$ and any constant $\epsilon > 0$, the $\text{COUNTDISTINCT}(i, j)$ queries can be answered in $\mathcal{O}(m \log^\epsilon n + \log n)$ time using an $\mathcal{O}((nd \log n)/m + d)$ -size data structure that can be constructed in $\mathcal{O}((nd \log n)/m + d)$ time.

Proof. We first apply Lemma 21 for $k = \lceil d/m \rceil$. We then have a decomposition of \mathcal{D} to a family \mathcal{F} of at most m path-sets and a set \mathcal{D}' with no path-set of size greater than $\lceil d/m \rceil$. We directly apply Lemma 20 for \mathcal{D}' . In order to handle path-sets, we build the data

structure of Lemma 3. Then, upon a $\text{COUNTDISTINCT}(i, j)$ query, for each path-set $\mathcal{H} \in \mathcal{F}$, we compute the longest pattern in \mathcal{H} that occurs in $T[i..j]$ using a Bounded LCP query followed by a predecessor query [24] in a structure that stores the lengths of the elements of \mathcal{H} , with the lexicographic rank in \mathcal{H} stored as satellite information. The data structure of [24] is randomized, but it can be combined with deterministic dictionaries [21] using a simple two-level approach (see [23]), resulting in a deterministic *static* data structure. ◀

► **Remark 23.** Let us fix the query time to be $\mathcal{O}(m \log^\epsilon n)$ for $m = \Omega(\log n)$. Then, Theorem 22 outperforms Theorem 19 in terms of the required space for $d = o(n/(m \log n))$. For example, for $m = d = n^{1/4}$, the data structure of Theorem 22 requires space $\tilde{\mathcal{O}}(n)$ while the one of Theorem 19 requires space $\tilde{\mathcal{O}}(n\sqrt{n})$.

6 Internal Counting of Distinct Squares

The number of occurrences of squares could be quadratic, but we can construct a much smaller $\mathcal{O}(n \log n)$ -size subset of these occurrences (called *boundary occurrences*) that, from the point of view of COUNTDISTINCT queries, gives almost the same answers. This is the main trick in this section. Distinct squares with a boundary occurrence in a given fragment can be counted in $\mathcal{O}(\log n)$ time due to Lemma 7. The remaining squares can be counted based on their structure: we show that they are all generated by the same run.

Now, the dictionary \mathcal{D} is the set of all squares in T . By the following fact, $d = \mathcal{O}(n)$ and \mathcal{D} can be computed in $\mathcal{O}(n)$ time.

► **Fact 24** ([7, 8, 10, 12]). *A string T of length n contains $\mathcal{O}(n)$ distinct square factors and they can all be computed in $\mathcal{O}(n)$ time.*

We say that an occurrence of a square U^2 is *induced* by a run R if it is contained in R and the shortest periods of U and R are the same. Every occurrence of a square is induced by exactly one run.

We need the following fact (note that it is false for the set of *all* runs; see [11]).

► **Fact 25.** *The sum of the lengths of all highly periodic runs is $\mathcal{O}(n \log n)$.*

Proof. We will prove that each position in T is contained in $\mathcal{O}(\log n)$ highly periodic runs. Let us consider all highly periodic runs R containing some position i , such that $m \leq \text{per}(R) < \frac{3}{2}m$ for some even integer m . Suppose for the sake of contradiction that there are at least 5 such runs. Note that each such run fully contains one of the fragments $T[i - 3m + 1 + t..i + t]$ for $t \in \{0, m, 2m, 3m\}$. By the pigeonhole principle, one of these four fragments is contained in at least two runs, say R_1 and R_2 . In particular, the overlap of these runs is at least $3m \geq \text{per}(R_1) + \text{per}(R_2)$, which is a contradiction by the periodicity lemma. ◀

We define a family of occurrences $\mathcal{B} = B_1, \dots, B_d$ such that, for each square U_i^2 , the set B_i contains the leftmost and the rightmost occurrence of U_i^2 in every run. We call these *boundary occurrences*. Boundary occurrences of squares have the following property.

► **Lemma 26.** $|\mathcal{B}| = \mathcal{O}(n \log n)$ and the set family \mathcal{B} can be computed in $\mathcal{O}(n \log n)$ time.

Proof. Let us define the *root* of a square U^2 to be U . A square is *primitively rooted* if its root is a primitive string. Let *p-squares* be primitively rooted squares, *np-squares* be the remaining ones. The number of occurrences of p-squares in a string of length n is $\mathcal{O}(n \log n)$ and they can all be computed in $\mathcal{O}(n \log n)$ time; see [6, 22].

We now proceed to np-squares. Note that for any highly periodic run R , the leftmost occurrence of each np-square induced by R starts in one of the first $\text{per}(R)$ positions of R ; a symmetric property holds for rightmost occurrences and last $\text{per}(R)$ positions. In addition, it can be readily verified that such a position is the starting (resp. ending) position of at most $\text{exp}(R)$ squares induced by R . It thus suffices to bound the sum of $\text{exp}(R) \cdot \text{per}(R)$ over all highly periodic runs R . The fact that $\text{exp}(R) \cdot \text{per}(R) = |R|$ concludes the proof of the combinatorial part by Fact 25.

For the algorithmic part, it suffices to iterate over the $\mathcal{O}(n)$ runs of T . ◀

► **Lemma 27.** *If $T[i..j]$ is non-periodic, $\text{COUNTDISTINCT}(i, j) = \text{COUNTDISTINCT}_{\mathcal{B}}(i, j)$.*

Proof. Let us consider an occurrence of a square U^2 inside $T[i..j]$. Let R be the run that induces this occurrence. By the assumption of the lemma, R does not contain $T[i..j]$. Then at least one of the boundary occurrences of U^2 in R is contained in $T[i..j]$. ◀

For a periodic fragment F of T , by $\text{RunSquares}(F)$ we denote the number of distinct squares that are induced by F (being a run if interpreted as a standalone string). The value $\text{RunSquares}(F)$ can be computed in $\mathcal{O}(1)$ time, as it was shown in e.g. [7].

Let F_1 be a prefix and F_2 be a suffix of a periodic fragment F , such that each of F_1 and F_2 is of length at most $\text{per}(F)$ – and hence they are disjoint. By $\text{BSq}(F, F_1, F_2)$ (“bounded squares”) we denote the number of distinct squares induced by F which have an occurrence starting in F_1 or ending in F_2 .

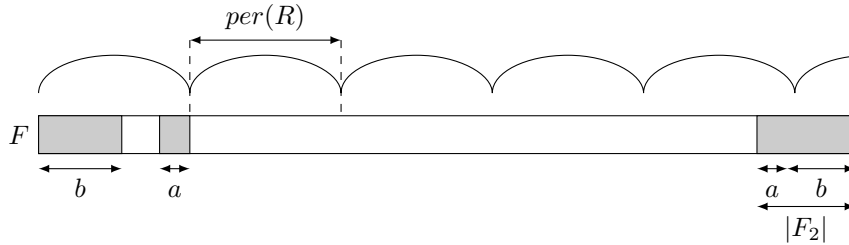
► **Lemma 28.** *Given $\text{per}(F)$, the $\text{BSq}(F, F_1, F_2)$ queries can be answered in $\mathcal{O}(1)$ time.*

Proof. We are to count distinct squares induced by F that start in F_1 or end in F_2 .

We introduce an easier version of BSq queries. Let $\text{BSq}'(F, F_1) = \text{BSq}(F, F_1, \varepsilon)$ be the number of squares induced by F which start in its prefix F_1 of length at most $p := \text{per}(F)$.

Reduction of BSq to BSq' . First, observe that the set of squares induced by F starting at some position $q \in [1, p]$ and the set of squares induced by F ending at some position $q' \in [|F| - p + 1, |F|]$ are equal if $q \equiv q' + 1 \pmod{p}$ and disjoint otherwise. Also note that $F_2 = UV$ for some prefix V and some suffix U of $F[p]F[1..p-1]$; we consider this rotation of $F[1..p]$ to offset the $+1$ factor in the above modular equation. Let $|U| = a$ and $|V| = b$.

Then, by the aforementioned observation, we are to count distinct squares that start in some position in the set $[1, |F_1|] \cup [1, b] \cup [p - a + 1, p]$; see Figure 5.



■ **Figure 5** Reduction of BSq to BSq' ; the case that $|F_1| \leq b$.

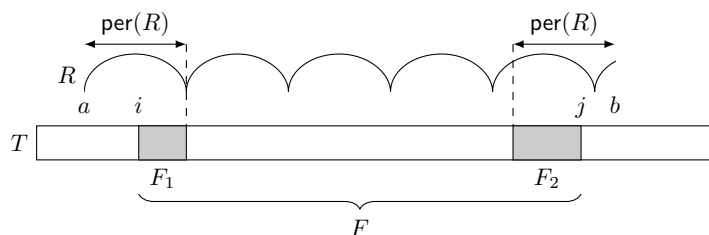
Hence the computation of $\text{BSq}(F, F_1, F_2)$ is reduced to at most two instances of the special case when F_2 is the empty string.

Computation of $BSq'(F, F_1)$. The number of squares induced by F starting at $F[i]$ is $\lfloor (|F| - i + 1)/(2p) \rfloor$. Consequently, $BSq'(F, F_1) = \sum_{i=1}^{|F_1|} \lfloor (|F| - i + 1)/(2p) \rfloor = |F_1| \cdot t - \max\{0, |F_1| - k - 1\}$, where $t = \lfloor |F|/(2p) \rfloor$ and $k = |F| \bmod (2p)$. ◀

► **Lemma 29.** *Assume that $F = T[i..j]$ is periodic and $R = T[a..b] = \text{run}(T[i..j])$. Let $F_1 = T[i..a + p - 1]$ and $F_2 = T[b - p + 1..j]$, where $\text{per}(R) = p$. Then:*

$$\text{COUNTDISTINCT}(i, j) = \text{COUNTDISTINCT}_{\mathcal{B}}(i, j) + \text{RunSquares}(F) - BSq(F, F_1, F_2). \quad (1)$$

Proof. In the sum $\text{COUNTDISTINCT}_{\mathcal{B}}(i, j) + \text{RunSquares}(F)$, all squares are counted once except for squares whose boundary occurrences are induced by R , which are counted twice. They are exactly counted in the term $BSq(F, F_1, F_2)$; see Figure 6. ◀



■ **Figure 6** The setting in Lemma 29. Note that F_1 is empty if $i \geq a + \text{per}(R)$; similarly for F_2 .

► **Theorem 30.** *If \mathcal{D} is the set of all square factors of T , then $\text{COUNTDISTINCT}(i, j)$ queries can be answered in $\mathcal{O}(\log n)$ time using a data structure of size $\mathcal{O}(n \log^2 n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We precompute the set \mathcal{B} in $\mathcal{O}(n \log n)$ time using Lemma 26 and perform $\mathcal{O}(n \log^2 n)$ time and space preprocessing for $\text{COUNTDISTINCT}_{\mathcal{B}}(i, j)$ queries.

In order to answer a $\text{COUNTDISTINCT}(i, j)$ query, first we ask a $\text{run}(T[i..j])$ query of Lemma 8 to check if $T[i..j]$ is periodic.

We compute $\text{COUNTDISTINCT}_{\mathcal{B}}(i, j)$ which takes $\mathcal{O}(\log n)$ time due to Lemma 7. If $T[i..j]$ is non-periodic, then it is the final result due to Lemma 27.

Otherwise $T[i..j]$ is periodic. Let F, F_1, F_2 be as in Lemma 29. We answer $\text{RunSquares}(F)$ and $BSq(F, F_1, F_2)$ queries in $\mathcal{O}(1)$ time using the algorithm from [7] and Lemma 28, respectively. Finally, $\text{COUNTDISTINCT}(i, j)$ is computed using (1). ◀

7 Final Remarks

The general framework for dynamic dictionaries, presented in the full version of this paper, essentially consists in rebuilding a static data structure after every k updates. We return correct answers by performing individual queries for the patterns inserted or deleted from the dictionary since the data structure was built. In particular, we show that an application of this framework – with some tweaks – to the data structure of Section 4 yields the following.

► **Theorem 31.** *For any $k \in [1, n]$, we can construct a data structure in $\tilde{\mathcal{O}}(n + d)$ time, which processes each update to the dictionary in $\tilde{\mathcal{O}}(n/k)$ time and answers $\text{COUNTDISTINCT}(i, j)$ queries 2-approximately in $\tilde{\mathcal{O}}(k)$ time.*

We leave open the problem of whether an $\tilde{\mathcal{O}}(n + d)$ -size data structure answering $\text{COUNTDISTINCT}(i, j)$ queries exactly in time $\tilde{\mathcal{O}}(1)$ exists.

References

- 1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
- 2 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.
- 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 Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173. SIAM, 2010. doi:10.1137/1.9781611973075.15.
- 5 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPICs*, pages 22:1–22:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ISAAC.2019.22.
- 6 Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.
- 7 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.
- 8 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.
- 9 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.
- 10 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.
- 11 Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013. doi:10.1016/j.tcs.2013.06.004.
- 12 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. doi:10.1016/j.jcss.2004.03.004.
- 13 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.
- 14 Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008. doi:10.1137/070684483.
- 15 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.
- 16 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPICs*, pages 28:1–28:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.CPM.2016.28.
- 17 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: <https://mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

- 19 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.
- 20 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.
- 21 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, ICALP 2008, Part I*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.
- 22 Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002. doi:10.1016/S0304-3975(01)00121-9.
- 23 Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 649–658. ACM, 2003. doi:10.1145/780542.780636.
- 24 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Dynamic String Alignment

Panagiotis Charalampopoulos 

Department of Informatics, King's College London, UK

Institute of Informatics, University of Warsaw, Poland

panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka 

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

kociumaka@mimuw.edu.pl

Shay Mozes 

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

smozes@idc.ac.il

Abstract

We consider the problem of dynamically maintaining an optimal alignment of two strings, each of length at most n , as they undergo insertions, deletions, and substitutions of letters. The string alignment problem generalizes the longest common subsequence (LCS) problem and the edit distance problem (also with non-unit costs, as long as insertions and deletions cost the same). The conditional lower bound of Backurs and Indyk [J. Comput. 2018] for computing the LCS in the static case implies that strongly sublinear update time for the dynamic string alignment problem would refute the Strong Exponential Time Hypothesis. We essentially match this lower bound when the alignment weights are constants, by showing how to process each update in $\tilde{O}(n)$ time.¹ When the weights are integers bounded in absolute value by some $w = n^{O(1)}$, we can maintain the alignment in $\tilde{O}(n \cdot \min\{\sqrt{n}, w\})$ time per update. For the $\tilde{O}(nw)$ -time algorithm, we heavily rely on Tiskin's work on semi-local LCS, and in particular, in an implicit way, on his algorithm for computing the $(\min, +)$ -product of two simple unit-Monge matrices [Algorithmica 2015]. As for the $\tilde{O}(n\sqrt{n})$ -time algorithm, we employ efficient data structures for computing distances in planar graphs.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string alignment, edit distance, longest common subsequence, (unit-)Monge matrices, $(\min, +)$ -product

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.9

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL (no. 677651) under the EU's Horizon 2020 Research and Innovation Programme.

Tomasz Kociumaka: Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.

Shay Mozes: Partially supported by Israel Science Foundation grant ISF no. 592/17.

1 Introduction

The problems of computing an optimal string alignment, a longest common subsequence (LCS), or the edit distance of two strings have been studied for more than 50 years. In the string alignment problem, we are given weights w_{match} for aligning a pair of matching letters, w_{mis} for aligning a pair of mismatching letters, and w_{gap} for letters that are not aligned, and the goal to compute an alignment with maximum weight. The edit distance $d_E(S, T)$ of two strings S and T is the minimum cost of transforming string S to string T using insertions, deletions, and substitutions of letters, under specified costs c_{ins} , c_{del} , and

¹ The $\tilde{O}(\cdot)$ notation suppresses $\log^{O(1)} n$ factors.



c_{sub} , respectively. When all costs are 1, this is also known as the Levenshtein distance of S and T [24]. Note that if $c_{ins} = c_{del}$, the edit distance problem is a special case of the string alignment problem, with $w_{match} = 0$, $w_{mis} = -c_{sub}$, and $w_{gap} = -c_{ins} = -c_{del}$. In turn, the LCS problem can be seen as a special case of the edit distance problem: Let the length of an LCS of S and T be denoted by $\text{LCS}(S, T)$. Then, for $c_{ins} = c_{del} = 1$ and $c_{sub} = 2$, we have $d_E(S, T) = |S| + |T| - 2 \cdot \text{LCS}(S, T)$. In this work, we consider the dynamic version of the string alignment problem, in which the strings S and T , each of length at most n , are maintained subject to insertions, deletions, and substitutions of letters, and we are to report an optimal alignment after each such update.

The textbook dynamic programming (DP) $\mathcal{O}(n^2)$ -time algorithm for the (static) LCS and edit distance problems has been rediscovered several times, e.g. in [38, 28, 29, 30, 39]. When the desired output is just the edit distance or the length of an LCS, the space required by the DP algorithm is trivially $\mathcal{O}(n)$ as one needs to store just two rows or columns of the DP matrix. Hirschberg showed how to actually retrieve an LCS within $\mathcal{O}(n^2)$ time using only $\mathcal{O}(n)$ space [17]. A line of works has improved the complexity of the classic DP algorithm by factors polylogarithmic with respect to n (see [25, 40, 11, 5, 15]).

While we are the first to consider the dynamic string alignment problem in the general variant where edits are allowed in any position of either of the strings, already the DP algorithm is inherently “dynamic” in the sense that it supports appending a letter and deleting the last letter in either of the strings in linear time. A series of works examined variants of incremental and decremental LCS and edit distance problems [23, 21, 18]. The most general of these variants was considered by Tiskin in [32], where he presented a linear-time algorithm for maintaining an LCS in the case that both strings are subject to the following updates: prepending or appending a letter, and deleting the first or the last letter. Tiskin’s solution not only maintains the LCS, but implicitly also the *semi-local LCS information*: the LCS lengths between all prefixes of S (resp. T) and all suffixes of T (resp. S), as well as the LCS between S (resp. T) and all fragments (substrings) of T (resp. S).

One of the main technical contributions of Tiskin in this area is an efficient algorithm for computing the $(\min, +)$ *product* (also known as *distance product*) of two simple unit-Monge matrices [37].² The algorithm itself and the ideas behind it have found numerous applications to variants of the LCS and string alignment problems. We refer the reader to Tiskin’s monograph [32] as well as [33, 34, 35, 31, 36].

On the lower-bound side, Backurs and Indyk showed that an $\mathcal{O}(n^{2-\epsilon})$ -time algorithm for computing the edit distance of two strings of length at most n would refute the Strong Exponential Time Hypothesis (SETH) [4]. Bringmann and Künnemann generalized this conditional lower bound by showing that it holds even for binary strings under any non-trivial assignment of weights c_{ins} , c_{del} , and c_{sub} [7] – an assignment of weights is trivial if it allows one to infer the edit distance in constant time. Further consequences of subquadratic-time algorithms for the edit distance or LCS problems were shown by Abboud et al. [1]; interestingly, they proved that even shaving arbitrarily large polylogarithmic factors from n^2 would have major consequences. In light of the above results, an $\mathcal{O}(n^{1-\epsilon})$ -time algorithm maintaining an optimal string alignment of two strings of length $\mathcal{O}(n)$ subject to edit operations seems highly unlikely, as it would directly imply an $\mathcal{O}(n^{2-\epsilon})$ -time algorithm for the static version of the problem.

² A matrix M is a Monge matrix if $M[i, j] + M[i', j'] \leq M[i', j] + M[i, j']$ for all $i < i'$ and $j < j'$ [26]. An $n \times n$ Monge matrix is a simple unit-Monge matrix if its leftmost column and bottommost row consist of zeroes, while its rightmost column and topmost row consist of subsequent integers from 0 to $n - 1$ [37].

Our results and approach. We heavily rely on Tiskin’s work on efficient distance multiplication of simple unit-Monge matrices and its applications to the string alignment problem. Specifically for the LCS problem, Tiskin showed that the semi-local LCS information of two strings of length at most n can be retrieved from an $\tilde{O}(n)$ -size representation as a *permutation matrix* $P_{S,T}$. Based on his efficient algorithm for computing the $(\min, +)$ -product of two simple unit-Monge matrices, he showed that given permutation matrices $P_{S,T}$ and $P_{S,T'}$, one can efficiently compute $P_{S,TT'}$. We formalize this in the preliminaries (Section 2).

In Section 3, we first describe our algorithm for maintaining an LCS of two strings S and T in $\tilde{O}(n)$ time per edit operation, and then we extend it to maintaining a string alignment under integer weights. Our algorithm maintains a hierarchical partition of strings S and T to fragments of length roughly 2^s for each scale s , $0 \leq s \leq \log n$, and permutation matrices P_{S_i,T_j} for all pairs of fragments (S_i, T_j) at each scale. Then, upon an update to S or T , we need to update $\Theta(n/2^s)$ permutation matrices at each scale s . This is in contrast with the sequential approach of combining the permutation matrices in Tiskin’s work.

In Section 4, we show that efficient data structures for computing distances in planar graphs outperform the approach outlined above when the alignment weights cannot be expressed as small integers.

2 Preliminaries

Let $T = T[0]T[1] \cdots T[n-1]$ be a *string* of length $|T| = n$ over an alphabet Σ . The elements of Σ are called *letters*. For two positions i and j in T , we denote by $T[i..j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $i < j$). The fragment $T[i..j]$ is an *occurrence* of the underlying *substring* $T[i] \cdots T[j]$. A fragment of T is represented in $\mathcal{O}(1)$ space by specifying the indices i and j . We sometimes denote the fragment $T[i..j]$ as $T[i..j+1)$. A *prefix* of T is a fragment that starts at position 0 ($T[0..j]$) and a *suffix* is a fragment that ends at position $n-1$ ($T[i..n-1]$ or $T[i..n)$).

A *longest common subsequence* (LCS) of two strings S and T is a longest string that is a subsequence of both S and T . We denote the length of an LCS of S and T by $\text{LCS}(S, T)$.

► **Example 1.** An LCS of $S = \text{acbccddaaea}$ and $T = \text{abbccdec}$ is abcde ; $\text{LCS}(S, T) = 5$.

For strings S and T , of length m and n respectively, the alignment graph $G_{S,T}$ of S and T is a directed acyclic graph with vertex set $\{v_{i,j} : 0 \leq i \leq m, 0 \leq j \leq n\}$. For every $0 \leq i \leq m$ and $0 \leq j \leq n$, the graph $G_{S,T}$ has the following edges (defined only if both endpoints exist):

- $v_{i,j}v_{i+1,j}$ and $v_{i,j}v_{i,j+1}$ of length 0,
- $v_{i,j}v_{i+1,j+1}$ of length 1, present if and only if $S[i] = T[j]$.

Intuitively, $G_{S,T}$ is an $(m+1) \times (n+1)$ grid graph (with length-0 edges) augmented with length-1 diagonal edges corresponding to matching letters of S and T . We think of the vertex $v_{0,0}$ as the top left vertex of the grid and the vertex $v_{m,n}$ as the bottom right vertex of the grid. We shall refer to the rows and columns of $G_{S,T}$ in a natural way. It is easy to see that $\text{LCS}(S, T)$ equals the length of the highest scoring path between $v_{0,0}$ and $v_{m,n}$ in $G_{S,T}$.

We index matrices from 0. Let us define some matrices of interest.

► **Definition 2.** The *distribution matrix* $\sigma(M)$ of an $m \times n$ matrix M is the $(m+1) \times (n+1)$ matrix satisfying $\sigma(M)[i, j] = \sum_{r \geq i, c < j} M[r, c]$.

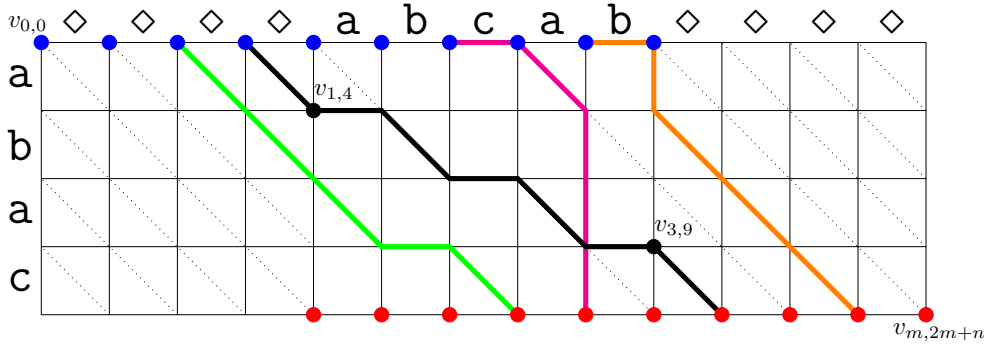
► **Definition 3.** An $n \times n$ binary matrix is a *permutation matrix* if it has exactly one 1 entry in each row and each column. Such a matrix can be represented in $\mathcal{O}(n)$ space.

By constructing a 2D orthogonal range counting data structure over the non-zero entries of a permutation matrix, one obtains the following lemma.

► **Lemma 4** ([32, Theorem 2.15]; [9]). *An $n \times n$ permutation matrix P can be preprocessed $\mathcal{O}(n\sqrt{\log n})$ time so that any entry of $\sigma(P)$ can be retrieved in time $\mathcal{O}(\log n / \log \log n)$.*

Let \diamond be a wildcard letter, i.e., a letter that matches all letters. Tiskin [32] defines an $(m + n + 1) \times (m + n + 1)$ distance matrix $H_{S,T}$ over $G_{S,\diamond^m T \diamond^m}$ so that $H_{S,T}[i, j]$ equals the highest weight of a path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S,\diamond^m T \diamond^m}$. Note that if $j = i - m$, then $H_{S,T}[i, j] = 0$. By convention, if $j < i - m$, then $H_{S,T}[i, j] = j - (i - m) < 0$. The matrix $H_{S,T}$ captures so-called *semi-local LCS* values as follows; see Figure 1 for an illustration.

$$H_{S,T}[i, j] = \begin{cases} \text{LCS}(S[m - i \dots m], T[0 \dots j]) + m - i & \text{if } i \leq m \text{ and } j \leq n, \\ \text{LCS}(S[0 \dots m + n - j], T[i - m \dots n]) + j - n & \text{if } i \geq m \text{ and } j \geq n, \\ \text{LCS}(S[m - i \dots m + n - j], T) + m - i + j - n & \text{if } i \leq m \text{ and } n + i \geq j \geq n, \\ m & \text{if } n + i \leq j, \\ \text{LCS}(S, T[i - m \dots j]) & \text{if } i \geq m \text{ and } i - m \leq j \leq n, \\ j + m - i & \text{if } j \leq i - m. \end{cases}$$



■ **Figure 1** The figure illustrates how $H_{S,T}$ captures semi-local LCS information for $S = abac$ and $T = abcab$. We have $m = 4$ and $n = 5$. The value $H_{S,T}(i, j)$ captures the length of the highest scoring path from the i -th blue node to the j -th red node in the above figure (in the left-to-right order). The underlying idea is that when there are wildcards \diamond involved, one may always choose to use the diagonal edges corresponding to them and then fill in the rest of the path. Let us analyze one of the cases thoroughly, the analysis of the other cases is analogous.

- The highest weight of a path from $v_{0,3}$ to $v_{4,4+6}$ is 4, which corresponds to $H_{S,T}(3, 6) = 4 = \text{LCS}(S[4 - 3, 9 - 6], T) + 4 - 3 + 6 - 5 = \text{LCS}(S[1, 2], T) + 2$ (case 3 of the equation above). The highest scoring path (in black), after trimming diagonal edges corresponding to wildcards, yields a highest scoring path from $v_{1,4}$ to $v_{3,4+5}$. Its weight indeed corresponds to $\text{LCS}(S[1 \dots 2], T) = 2$.
- The $v_{0,2}$ -to- $v_{4,4+3}$ highest scoring path (in green) illustrates case 1 of the equation above: $H_{S,T}(2, 3) = 4 = \text{LCS}(S[4 - 2 \dots 4], T[0 \dots 3]) + 4 - 2 = \text{LCS}(S[2 \dots 3], T[0 \dots 2]) + 2$.
- The $v_{0,8}$ -to- $v_{4,4+8}$ highest scoring path (in orange) illustrates case 2 of the equation above: $H_{S,T}(8, 8) = 3 = \text{LCS}(S[0 \dots 9 - 8], T[8 - 4 \dots 5]) + 8 - 5 = \text{LCS}(S[0], T[4]) + 3$.
- The $v_{0,6}$ -to- $v_{4,4+4}$ highest scoring path (in magenta) illustrates case 5 of the equation above: $H_{S,T}(6, 4) = 1 = \text{LCS}(S, T[6 - 4 \dots 4]) = \text{LCS}(S, T[2 \dots 3])$.

► **Remark 5.** Let us try to provide some extra intuition by considering the *indel distance*, for which we get a more uniform formula. The indel distance of two strings, denoted $\delta(S, T)$, is the minimum number of insertions and deletions that are needed to transform S to T . In other words, $\delta(S, T) = |S| + |T| - 2\text{LCS}(S, T)$. Then $2m + j - i - 2H_{S,T}[i, j]$, which can be interpreted as the number of length-0 edges on the highest scoring path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S, \delta^m T^{\delta^m}}$, admits a more uniform formula:

$$2m + j - i - 2H_{S,T}[i, j] = \begin{cases} \delta(S[m - i \dots m], T[0 \dots j]) & \text{if } i \leq m \text{ and } j \leq n, \\ \delta(S[0 \dots m + n - j], T[i - m \dots n]) & \text{if } i \geq m \text{ and } j \geq n, \\ \delta(S[m - i \dots m + n - j], T) & \text{if } i \leq m \text{ and } n + i \geq j \geq n, \\ j - i & \text{if } n + i \leq j, \\ \delta(S, T[i - m \dots j]) & \text{if } i \geq m \text{ and } i - m \leq j \leq n, \\ i - j & \text{if } j \leq i - m. \end{cases}$$

We now return to the LCS problem. Tiskin shows that the $(n + m) \times (n + m)$ matrix $P_{S,T}$ defined as

$$P_{S,T}[i, j] = H_{S,T}[i, j] + H_{S,T}[i + 1, j + 1] - H_{S,T}[i + 1, j] - H_{S,T}[i, j + 1], \quad (1)$$

is a permutation matrix and satisfies $H_{S,T}[i, j] = j + m - i - \sigma(P_{S,T})[i, j]$. Note that for constant-length strings S and T , the matrix $P_{S,T}$ can be computed naively in constant time from $H_{S,T}$. Conversely, each entry of $H_{S,T}$ can be retrieved in time $\mathcal{O}(\log(n + m) / \log \log(n + m))$ after an $\mathcal{O}((n + m)\sqrt{\log(n + m)})$ -time preprocessing of $P_{S,T}$ by Lemma 4. Crucially for our approach, Tiskin shows the following result.

► **Theorem 6** ([32, Theorem 4.21]).

- (a) Given $P_{S,T}$ and $P_{S,T'}$ for three strings S, T, T' , each of length at most n , one can compute $P_{S,TT'}$ in $\mathcal{O}(n \log n)$ time.
- (b) Given $P_{T,S}$ and $P_{T',S}$ for three strings S, T, T' , each of length at most n , one can compute $P_{TT',S}$ in $\mathcal{O}(n \log n)$ time.

Actually, only part (a) of the above theorem is stated explicitly in [32]. Part (b) can be derived by symmetry as follows. One can check using the characterization of $H_{S,T}$ in terms of the semi-local LCS values that $H_{S,T}[i, j] = H_{T,S}[n + m - i, n + m - j] + m - i + j - n$; see [32, Lemma 4.14]. In particular, this means that $H_{S,T}$ can be obtained from $H_{T,S}$ by first performing a 180-degree rotation and then off-setting the values in every row i by $m - i$ and the values in every column j by $j - n$. This, in turn, means that $P_{T,S}$ can be obtained from $P_{S,T}$ just through a 180-degree rotation, as the offsets are cancelled out in the computation of $P_{S,T}[i, j]$ from $H_{T,S}$; see (1). Thus, we can rotate $P_{T,S}$ and $P_{T',S}$ to obtain $P_{S,T}$ and $P_{S,T'}$, compute $P_{S,TT'}$ using Theorem 6(a), and then rotate $P_{S,TT'}$ to obtain $P_{TT',S}$.

3 Main Algorithm

We show how to maintain the permutation matrix $P_{S,T}$ in $\mathcal{O}((m + n) \log(m + n))$ time per update when the strings S and T undergo substitutions, insertions, and deletions of single letters. Within the stated update time we can recompute the orthogonal range counting data structure that allows us to report, in $\mathcal{O}(\log(m + n) / \log \log(m + n))$ time, any element of the matrix $H_{S,T}$.

The high-level idea is to maintain the permutation matrices $P_{A,B}$ for fragments A of S and B of T , at exponentially growing scales. Local changes to S and T , such as substitutions, insertions, and deletions, only affect a single fragment at each scale. We can therefore use Theorem 6 to recompute the affected matrices efficiently in a bottom-up fashion.

We first describe the maintenance of a data structure that can only support substitutions in order to demonstrate the general approach. We will then describe how to also support insertions and deletions.

3.1 Supporting Only Substitutions

We can assume that both S and T are of length n and that n is a power of two; otherwise, we pad S with $\$$ characters and T with $\#$ characters such that $\$ \neq \#$ and neither $\$$ nor $\#$ is in the alphabet. We define $\log n + 1$ scales, where at scale s , each of S and T is partitioned into non-overlapping fragments of length 2^s . At every scale, and for every pair of fragments S_i and T_j of S and T , respectively, we store the permutation matrix P_{S_i,T_j} corresponding to H_{S_i,T_j} . At scale s , there are $(n/2^s)^2$ matrices, each stored in $\mathcal{O}(2^s)$ space. Thus, the overall space required by the data structure is $\mathcal{O}(n^2)$. Building the data structure in a bottom-up manner requires time $\sum_{s=0}^{\log n} (n/2^s)^2 \cdot 2^s \cdot s = \mathcal{O}(n^2)$ by Theorem 6.

Suppose, without loss of generality, that a letter of S is substituted (the other case is symmetric). We work in order of increasing scales $s = 0, 1, \dots, \log n$. Let S_i be the unique fragment of S in scale s that contains the substituted letter. We recompute the matrices P_{S_i,T_j} for each one of the $n/2^s$ fragments T_j of T at scale s . At scale $s = 0$, both S_i and T_j consist of single letters, and we recompute the constant-size permutation matrices P_{S_i,T_j} from scratch in total $\mathcal{O}(n)$ time. (In fact, there are only two types of matrices, one corresponding to the case that the letter S_i matches the letter T_j , and the other corresponding to a mismatch.) To recompute a matrix P_{S_i,T_j} at scale $s > 0$, let S'_i, S''_i be the two fragments of S at scale $s - 1$ such that $S_i = S'_i S''_i$. Similarly, let T'_j, T''_j be the two fragments of T at scale $s - 1$ such that $T_j = T'_j T''_j$. We repeatedly apply Theorem 6 to $P_{S'_i,T'_j}, P_{S'_i,T''_j}, P_{S''_i,T'_j}, P_{S''_i,T''_j}$ to obtain P_{S_i,T_j} in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to update all affected permutation matrices at all scales (and, in particular, to obtain the matrix $P_{S,T}$) is $\sum_{s=0}^{\log n} \frac{n}{2^s} \cdot s \cdot 2^s = \mathcal{O}(n \log^2 n)$.

3.2 Supporting Insertions and Deletions

To support insertions and deletions we use the same approach. However, as each update increases or decreases the length of the string it is applied to, we can no longer use fixed-length fragments at each scale. At each scale s , we maintain a partition of each string into consecutive fragments, each of length between $\frac{1}{4} \cdot 2^s$ and $2 \cdot 2^s$, such that the partition at scale s is a refinement of the partition at scale $s + 1$. Let us denote by R_s (resp. C_s) the partition of S (resp. T) at level s . We only describe the process for S ; the string T is handled analogously. The *refinement property* for R_s can be stated formally as follows. For any $s' > s$, for each fragment $S[a..b] \in R_s$ there exists a fragment $S[a'..b'] \in R_{s'}$ with $a' \leq a \leq b \leq b'$. We maintain each R_s as a linked list of the fragments, which are represented by their start and end indices, sorted by the start indices in increasing order. Upon an update in S , we update the partitions in a bottom-up manner.

Let us first describe how to insert a letter in S after the letter at position k . We first scan R_s for all s and increment by 1 all the start indices that are greater than k and all the end indices that are at least k . This way, the newly inserted letter is assigned to a unique fragment in each partition. Then, we process the scales in increasing order, starting from scale 0. If the fragment $U_0 \in R_0$ that contains the newly inserted letter has just become of

length greater than $2 \cdot 2^0 = 2$, then we split U_0 into two fragments of length at most 2. Note that this potential split does not violate the refinement property. Then, we proceed to the next scale. Generally, at scale s , if the length of the fragment $U_s \in R_s$ that contains the newly inserted letter does not exceed $2 \cdot 2^s$, we just proceed to the next scale. Otherwise, we need to make adjustments, ensuring that the refinement property is not violated. Note that, if $|U_s| = 2 \cdot 2^s + 1$, then, since fragments at scale $s - 1$ have been already processed and respect the length constraint, the refinement property implies that U_s is the concatenation of at least three (and at most nine) fragments V_1, V_2, \dots, V_t at scale $s - 1$. Let the middle letter of U_s belong to V_i . Then, either $\sum_{g < i} |V_g| \geq 2^s/4$ or $\sum_{g > i} |V_g| \geq 2^s/4$; let us assume without loss of generality that we are in the first case. We replace U at scale s by $V_1 \cdots V_{i-1}$ and $V_i \cdots V_t$. If such a replacement happens at the highest scale s (with $U_s = S$), then we create a new level $s + 1$ with $R_{s+1} = \{U_s\} = \{S\}$. Note that the refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We now treat the complementary case of deleting $S[k]$. Again, we first scan R_s for all scales s and decrement by 1 all the start/end indices that are at least k – ensuring that none of them becomes negative. If some fragment becomes of length 0, then we remove it. We again process levels in increasing order. Suppose that the fragment $U_s \in R_s$ that contained the deleted letter has just become shorter than $\frac{1}{4} \cdot 2^s$. If R_s is the top level of the decomposition, then we simply remove this level. Otherwise, consider the fragment $U_{s+1} \in R_{s+1}$ that contains U_s . Note that, $1 \leq |U_s| = \frac{1}{4} \cdot 2^s - 1$ implies that the length $|U_s| + 1$ of the fragment corresponding to U_s prior to the deletion is smaller than $\frac{1}{4} \cdot 2^{s+1}$, and hence $|U_{s+1}| > |U_s|$. Thus, there exists a fragment V at scale s that is adjacent to U_s and is also a subfragment of U_{s+1} . Let us assume without loss of generality that V lies to the right of U_s – the other case is symmetric. If $|V| < \frac{7}{4} \cdot 2^s$, then we can just replace U_s and V in R_s by their concatenation, $U_s V$. Otherwise, let the first element of the decomposition of V at scale $s - 1$ be X . In this case, we can replace U_s and V in R_s by $U_s X$ and $Y = V[|X|..|V| - 1]$, since $\frac{1}{4} \cdot 2^s \leq |U_s X| < \frac{1}{4} \cdot 2^s + 2 \cdot 2^{s-1} < 2 \cdot 2^s$ and $\frac{1}{4} \cdot 2^s < \frac{7}{4} \cdot 2^s - 2 \cdot 2^{s-1} \leq |V| - |X| = |Y| < |V| \leq 2 \cdot 2^s$. The refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We maintain $P_{A,B}$ for each pair of fragments $(A, B) \in R_s \times C_s$ at scale s . In the case that R_s simply consists of S at scale s , while T is still fragmented, we consider R_j for any $j > s$ to simply consist of S . (Symmetrically for the opposite case.) The number of pairs of fragments that are affected at scale s is $\mathcal{O}((n + m)/2^s)$. We compute $P_{A,B}$, for each such pair (A, B) , using a constant number of applications of Theorem 6 in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to handle scale s is $\mathcal{O}((n + m)s)$ and the total time to handle all scales is $\mathcal{O}((m + n) \log^2(m + n))$.

► **Remark 7.** Deletions of fragments of either of the strings can also be processed within the same time complexity with a straightforward generalisation of the above process.

Obtaining the longest common subsequence. We now describe how one can obtain the longest common subsequence, and not just its length, within $\tilde{\mathcal{O}}(n + m)$ time. Let us consider the following auxiliary problem: given some pair of fragments S_i, T_j at scale $s > 0$, compute the longest common subsequence of either some prefix of S_i (resp. T_j) and some suffix of T_j (resp. S_i), or some fragment of S_i (resp. T_j) and T_j (resp. S_i). Consider the refinement, at scale $s - 1$, of S_i to U_1, \dots, U_k and of T_j to V_1, \dots, V_ℓ . Let $G_{S,T}(S[i_1..i_2], T[j_1..j_2])$ be the subgraph of $G_{S,T}$ induced by the set of vertices $\{v_{i',j'} : i_1 \leq i' \leq i_2 + 1, j_1 \leq j' \leq j_2 + 1\}$. Our aim is to decompose the highest scoring path in scope (say $v_{a,b}$ -to- $v_{c,d}$) into subpaths, each lying entirely on some $G_{S,T}(U_r, V_t)$. We can then apply this procedure recursively.

P_{S_i, T_j} was obtained from the $k \times \ell$ matrices P_{U_t, V_r} through some order of applications of Theorem 6. We can store such intermediate matrices, preprocessed as in Lemma 4, without any extra asymptotic cost in the complexities. We refine the path by considering the reverse order. For clarity of presentation, let us assume that $k = \ell = 2$ and the intermediate matrices were $P_{U_1 U_2, V_1}$ and $P_{U_1 U_2, V_2}$. We can decompose the path to at most two subpaths, one lying entirely on $J_1 = G_{S, T}(U_1 U_2, V_1)$ and one lying entirely on $J_2 = G_{S, T}(U_1 U_2, V_2)$. The case that both $v_{a, b}$ and $v_{c, d}$ lie on one of J_1 or J_2 is trivial. In the other case, we wish to find a node that lies on both J_1 and J_2 and is on the path. To this end, we query $P_{U_1 U_2, V_1}$ (resp. $P_{U_1 U_2, V_2}$) for the length of the highest scoring $v_{a, b}$ -to- u (resp. u -to- $v_{c, d}$) path for all nodes u that belong to both J_1 and J_2 . Using Lemma 4, this can be done in $\mathcal{O}(2^s \cdot s / \log s)$ time (for $s > 0$). Any u for which the sum of these values equals the length of the highest scoring $v_{a, b}$ -to- $v_{c, d}$ path is a valid vertex to decompose the path. We then recurse, further refining the path. Note that the $v_{a, b}$ -to- $v_{c, d}$ path gets decomposed into $\mathcal{O}((n + m)/2^s)$ pieces at scale s , for all s . Hence, by summing over all scales, the total time required for applying this procedure is $\mathcal{O}((n + m) \log^2(n + m))$.

Fragment-to-fragment LCS queries. Our data structure also enables us to answer queries of the type $\text{LCS}(S[i_1 \dots i_2], T[j_1 \dots j_2])$ in time $\tilde{\mathcal{O}}(n + m)$ (in fact, in time $\tilde{\mathcal{O}}(1 + i_2 - i_1 + j_2 - j_1)$). Note that $G_{S, T}(S[i_1 \dots i_2], T[j_1 \dots j_2])$ can be decomposed in $\tilde{\mathcal{O}}(n + m)$ time to multiple pieces $G_{S, T}(U, V)$, overlapping at their boundaries, such that U and V are of the same scale and there are $\mathcal{O}((n + m)/2^s)$ pairs (U, V) of scale s . This can be done, intuitively, using a greedy approach, that each time uses a piece from the largest possible scale. One can also think of this as extending a rectangle using a constant number of layers consisting of pieces corresponding to pairs of strings at scale s , in order of decreasing s . Finally, a repeated application of Theorem 6 yields the claimed result.

3.3 Extension to String Alignment Under Integer Weights

Let us now consider the problem of computing an alignment of two strings S and T , under integer weights w_{match} , w_{mis} and w_{gap} – one may assume that $2w_{\text{match}} > 2w_{\text{mis}} \geq w_{\text{gap}}$ [32]. In this problem, the goal is to compute a highest scoring path from $v_{0, 0}$ to $v_{m, n}$ in the following modification $\hat{G}_{S, T}$ of $G_{S, T}$. Edges of the form $v_{i, j} v_{i+1, j}$ and $v_{i, j} v_{i, j+1}$ have weight w_{gap} , while edges of the form $v_{i, j} v_{i+1, j+1}$ have weight w_{match} if $T[i] = S[i]$ and w_{mis} otherwise.

Tiskin shows in Section 6.1 of his monograph [32] that the alignment problem between strings S and T , can be reduced to the LCS problem between strings S' and T' , obtained as follows. First, replace every letter a in S or in T by the string $\$^\mu a^{\nu - \mu}$, where $\$ \notin \Sigma$ and

$$\frac{\mu}{\nu} = \frac{w_{\text{mis}} - 2w_{\text{gap}}}{w_{\text{match}} - 2w_{\text{gap}}}.$$

Then, if one defines matrix $\hat{H}_{S, T}$ over $\hat{G}_{S, T}$ analogously to the definition of $H_{S, T}$ over $G_{S, T}$, we have that $\hat{H}_{S, T}(i, j) = \frac{1}{\nu} \cdot H_{S', T'}(\nu i, \nu j)$.

We maintain the same information as in the previous subsections, making sure that each fragment of each partition is a multiple of ν . At scale 0, we have only two options about how $P_{A, B}$ can look like, despite it being a $\nu \times \nu$ matrix; its structure only depends on whether $A = B$ or not. We precompute such possible $P_{A, B}$'s. This way, upon an update on S or T , updating scale 0 requires $\mathcal{O}(n\nu)$ time. At every other scale, the total length of the involved strings has just blown up by a ν multiplicative factor and hence the total update time is

$\mathcal{O}(n\nu \log^2(n\nu))$. The same reasoning shows that the preprocessing time is

$$\mathcal{O}\left(\sum_{s=0}^{\log n} \left(\frac{nw}{2^s w}\right)^2 \cdot 2^s w \cdot \log(2^s w)\right) = \mathcal{O}(n^2 w \log n \log w).$$

We summarize the results of this section in the following theorem.

► **Theorem 8.** *Given two strings S and T and integer weights w_{match} , w_{mis} and w_{gap} , bounded by w , the alignment score of S and T as they undergo insertions, deletions and substitutions of letters can be maintained in $\mathcal{O}(nw \log^2(nw))$ time per operation after an $\mathcal{O}(n^2 w \log n \log w)$ -time preprocessing. The actual alignment can be retrieved in time $\mathcal{O}(nw \log^2(nw))$. In addition, the following queries are supported:*

- *the score of any semi-local string alignment can be computed in $\mathcal{O}(\log(nw)/\log \log(nw))$ time,*
- *the score of any fragment-to-fragment alignment can be computed in $\tilde{\mathcal{O}}(nw)$ time.*

4 Handling Large Weights

In this section, we describe an algorithm for string alignment that only relies on the planarity of $\hat{G}_{S,T}$. This algorithm outperforms the one from Theorem 8 when the alignment weights cannot be transformed to integers bounded by (roughly) \sqrt{n} .

Instead of computing a highest scoring path, we can reduce the problem to computing a shortest path in the alignment DAG. Given w_{match} , w_{mis} and w_{gap} , we define $w'_{\text{match}} = 0$, $w'_{\text{mis}} = w_{\text{match}} - w_{\text{mis}}$ and $w'_{\text{gap}} = \frac{1}{2}w_{\text{match}} - w_{\text{gap}}$. Then, a shortest path with respect to the new weights (of length W), corresponds to a highest scoring path with respect to the original weights (of score $\frac{1}{2}(m+n)w_{\text{match}} - W$).

4.1 Data Structures for Planar Graphs

Let us first introduce some data structures for shortest paths in planar graphs.

MSSP. The *multiple-source shortest paths* (MSSP) data structure of Klein [22] represents all shortest path trees rooted at the vertices of a single face f in a planar graph G of size n using a persistent dynamic tree. It can be constructed in $\mathcal{O}(n \log n)$ time, requires $\mathcal{O}(n \log n)$ space, and can report the distance between any vertex of f and any other vertex in G in $\mathcal{O}(\log n)$ time. The actual shortest path p can be retrieved in time $\mathcal{O}(\rho \log \log n)$, where ρ is the number of edges of p .

FR-Dijkstra. Let us consider a subgraph P of a planar graph G , and a face f of P . The *dense distance graph* of P with respect to f , denoted $DDG_{P,f}$ is a complete directed graph on the set of vertices F that lie on f . Each edge (u, v) has weight $d_P(u, v)$, equal to the length of the shortest u -to- v path in P . $DDG_{P,f}$ can be computed in time $\mathcal{O}(|F|^2 + |P| \log |P|)$ using MSSP. In their seminal paper, Fakcharoenphol and Rao [12] designed an efficient implementation of Dijkstra's algorithm on any union of DDG s – it is nicknamed FR-Dijkstra. The algorithm exploits the fact that, due to planarity, certain submatrices of the adjacency matrix of $DDG_{P,f}$ satisfy the Monge property. We next give a – convenient for our purposes – interface for FR-Dijkstra, which was essentially proved in [12], with some additional components and details from [20, 27].

► **Theorem 9** ([12, 20, 27]). *Given a set of DDGs with $\mathcal{O}(M)$ vertices in total (with multiplicities), each having at most m vertices, we can (independently) preprocess each DDG with k vertices in time and extra space $\mathcal{O}(k \log k)$, so that, after this preprocessing, Dijkstra’s algorithm can be run on the union of any subset of these DDGs with $\mathcal{O}(N)$ vertices in total (with multiplicities) in time $\mathcal{O}(N \log N \log m)$.*

► **Remark 10.** For an improvement in the logarithmic factors of Theorem 9 see [13].

4.2 Direct Application to String Alignment

Our approach is essentially the same as the one for dynamic distance oracles in planar graphs due to Klein [22], with extensions in [19, 20, 10]. We want to maintain a data structure that enables us to compute the length of the shortest $v_{0,0}$ -to- $v_{m,n}$ path. However, instead of a single update to the graph, we have a batch of $\mathcal{O}(m+n)$ updates for each update to one of the strings. We rely on the fact that the updates to the graphs are clustered in a constant number of rows/columns of $\hat{G}_{S,T}$ in order to process them more efficiently compared to simply using dynamic distance oracles for planar graphs in a black-box manner.

Let us consider a partition of $\hat{G}_{S,T}$ into $\mathcal{O}((n/r)^2)$ pieces of size $\Theta(r) \times \Theta(r)$ each. We consider the vertices that lie on the infinite face of each piece as its boundary nodes. Then, as each piece has $\mathcal{O}(r)$ boundary vertices, the total number of boundary vertices is $\mathcal{O}(n^2/r)$. We compute the MSSP data structure and the DDG for each piece with respect to its outer face. Note that the shortest path from $v_{0,0}$ to $v_{m,n}$ can be decomposed to subpaths p_1, \dots, p_k such that each p_i lies entirely within some piece P_i and p_i ’s endpoints are boundary nodes of P_i . Thus, we can compute the length of the shortest $v_{0,0}$ -to- $v_{m,n}$ path by running FR-Dijkstra from $v_{0,0}$ in the union of all DDGs in $\tilde{\mathcal{O}}(n^2/r)$ time. In order to retrieve the actual shortest path, we can refine the DDG edges of the shortest $v_{0,0}$ -to- $v_{m,n}$ path to the actual underlying edges using the MSSP data structures for the respective pieces.

Each update to one of the strings affects a constant number of rows or of columns of the original matrix and these are covered by $\mathcal{O}(n/r)$ pieces. The MSSP data structures and DDGs for these pieces can be recomputed using MSSP and preprocessed for efficient shortest path computations in $\tilde{\mathcal{O}}(\frac{n}{r} \cdot r^2) = \tilde{\mathcal{O}}(nr)$ time. The balance is at $n^2/r = nr$, which yields $r = \sqrt{n}$, so the time per operation is $\tilde{\mathcal{O}}(n^{3/2})$. If a piece grows (resp. shrinks) too much, we break it into two pieces (resp. merge it with an adjacent piece and split in the middle) and recompute and preprocess the DDGs for the affected pieces. We obtain the following result.

► **Theorem 11.** *Given two strings S and T and alignment weights w_{match} , w_{mis} , and w_{gap} , the optimal alignment of S and T as they undergo insertions, deletions, and substitutions of letters can be maintained in $\tilde{\mathcal{O}}(n^{3/2})$ time per operation after an $\tilde{\mathcal{O}}(n^2)$ -time preprocessing.*

5 Final Remarks

There has been a recent series of breakthrough papers on approximating the edit distance and length of the LCS, see e.g. [3, 2, 8, 16, 14, 6]. It is natural to ask about the maintenance of an approximation of the edit distance or LCS in the setting of dynamic strings.

References

- 1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016*, pages 375–388. ACM, 2016. doi:10.1145/2897518.2897653.

- 2 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.43.
- 3 Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM Journal on Computing*, 41(6):1635–1648, 2012. doi:10.1137/090767182.
- 4 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.
- 5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. doi:10.1016/j.tcs.2008.08.042.
- 6 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: beyond worst case. In *31st ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1601–1620. SIAM, 2020. doi:10.1137/1.9781611975994.99.
- 7 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15.
- 8 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *59th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00096.
- 9 Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173. SIAM, 2010. doi:10.1137/1.9781611973075.15.
- 10 Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. Exact distance oracles for planar graphs with failing vertices. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 2110–2123. SIAM, 2019. doi:10.1137/1.9781611975482.127.
- 11 Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003. doi:10.1137/S0097539702402007.
- 12 Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006. doi:10.1016/j.jcss.2005.05.007.
- 13 Paweł Gawrychowski and Adam Karczmarz. Improved bounds for shortest paths in dense distance graphs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 61:1–61:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.61.
- 14 Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1101–1120. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00070.
- 15 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016. doi:10.1016/j.dam.2015.10.040.
- 16 MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the \sqrt{n} barrier. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1181–1200. SIAM, 2019. doi:10.1137/1.9781611975482.72.
- 17 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.

- 18 Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In *15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, volume 3623 of *LNCS*, pages 563–574. Springer, 2005. doi:10.1007/11537311_49.
- 19 Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *43rd ACM Symposium on Theory of Computing, STOC 2011*, pages 313–322. ACM, 2011. doi:10.1145/1993636.1993679.
- 20 Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and partial Monge matrices, and their applications. *ACM Transactions on Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.
- 21 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.
- 22 Philip N. Klein. Multiple-source shortest paths in planar graphs. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 146–155. SIAM, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070454>.
- 23 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.
- 24 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics. Doklady*, 10:707–710, 1966.
- 25 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.
- 26 Gaspard Monge. *Mémoire sur la théorie des déblais et des remblais*. De l’Imprimerie Royale, 1781.
- 27 Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *18th Annual European Symposium on Algorithms, ESA 2010, Part II*, volume 6347 of *LNCS*, pages 206–217. Springer, 2010. doi:10.1007/978-3-642-15781-3_18.
- 28 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. doi:10.1016/0022-2836(70)90057-4.
- 29 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972. doi:10.1073/pnas.69.1.4.
- 30 Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974. doi:10.1137/0126070.
- 31 Alexander Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. In *17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *LNCS*, pages 270–281. Springer, 2006. doi:10.1007/11780441_25.
- 32 Alexander Tiskin. Semi-local string comparison: algorithmic techniques and applications, 2007. arXiv:0707.3619.
- 33 Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008. doi:10.1016/j.jda.2008.07.001.
- 34 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.
- 35 Alexander Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009. doi:10.1007/s10958-009-9396-0.
- 36 Alexander Tiskin. Periodic string comparison. In *20th Annual Symposium on Combinatorial Pattern Matching, CPM 2009*, volume 5577 of *LNCS*, pages 193–206. Springer, 2009. doi:10.1007/978-3-642-02441-2_18.
- 37 Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/s00453-013-9830-z.

- 38 Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–57, 1968. doi:10.1007/BF01074755.
- 39 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.
- 40 Sun Wu, Udi Manber, and Eugene W. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996. doi:10.1007/BF01942606.

Unary Words Have the Smallest Levenshtein k -Neighbourhoods

Panagiotis Charalampopoulos 

Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Solon P. Pissis 


CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France
solon.pissis@cwi.nl

Jakub Radoszewski 

Institute of Informatics, University of Warsaw, Poland
Samsung R&D, Warsaw, Poland
jrad@mimuw.edu.pl

Tomasz Waleń 

Institute of Informatics, University of Warsaw, Poland
walen@mimuw.edu.pl

Wiktor Zuba 

Institute of Informatics, University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

The edit distance (a.k.a. the Levenshtein distance) between two words is defined as the minimum number of insertions, deletions or substitutions of letters needed to transform one word into another. The Levenshtein k -neighbourhood of a word w is the set of words that are at edit distance at most k from w . This is perhaps the most important concept underlying BLAST, a widely-used tool for comparing biological sequences. A natural combinatorial question is to ask for upper and lower bounds on the size of this set. The answer to this question has important algorithmic implications as well. Myers notes that "such bounds would give a tighter characterisation of the running time of the algorithm" behind BLAST. We show that the size of the Levenshtein k -neighbourhood of any word of length n over an arbitrary alphabet is not smaller than the size of the Levenshtein k -neighbourhood of a unary word of length n , thus providing a tight lower bound on the size of the Levenshtein k -neighbourhood. We remark that this result was posed as a conjecture by Dufresne at WCTA 2019.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases combinatorics on words, Levenshtein distance, edit distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.10

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 872539.

Panagiotis Charalampopoulos: Supported by ERC grant TOTAL under the European Union's Horizon 2020 Research and Innovation Programme (agreement no. 677651).

Jakub Radoszewski: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

Tomasz Waleń: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

Wiktor Zuba: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.



© Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Tomasz Waleń, and Wiktor Zuba;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 10; pp. 10:1–10:12



Leibniz International Proceedings in Informatics

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



1 Introduction

BLAST (Basic Local Alignment Search Tool) is a widely-used tool for comparing biological sequences such as the amino-acid sequences of proteins or the nucleotides of DNA or RNA sequences. A BLAST search enables to compare a subject sequence, called a *query*, against a database of sequences to identify the ones that resemble the query sequence above a certain threshold. The paper describing BLAST [1] is one of the most highly cited papers in science.

According to Myers [8], the most important algorithmic idea underlying BLAST is that of searching for exact matches to words in the neighbourhood of fixed-length fragments selected from the query sequence. We call these fragments *words*. Let δ be a sequence comparison measure that given two words v and w returns a numeric measure $\delta(v, w)$ of the degree to which the two words differ. Given a word w , the k -neighbourhood of w with respect to δ is the set of all words whose best alignment with w under measure δ is no more than k . The most widely-used case is where δ is the *edit distance* (a.k.a. the *Levenshtein distance*), which is the minimum number of insertions, deletions or substitutions of letters needed to transform one word into another [6]. When δ is the Levenshtein distance, we call this neighbourhood the *Levenshtein k -neighbourhood* of w and we denote it by $N_{k,\Sigma}(w)$, where Σ is the considered alphabet. We provide an example below.

► **Example 1** (Levenshtein k -neighbourhood). Let $w = baab$, $k = 1$ and $\Sigma = \{a, b\}$. Then $N_{1,\Sigma}(baab)$ is:

$$\{bbab, bbaab, babb, bab, babab, baab, baabb, baaba, baa, baaa, baaab, abaab, aab, aaab\}.$$

From an algorithmic point of view, the most natural question is how we can generate the Levenshtein k -neighbourhood in time that is proportional to the size of the neighbourhood. In fact, this is the core computational task underlying BLAST. Myers described an algorithm for generating a condensed version of this neighbourhood efficiently (see [8] for more details). Another natural question is how we can compute the size of the Levenshtein k -neighbourhood. Touzet gave an algorithm for computing $|N_{k,\Sigma}(w)|$ for a word w of length n over an alphabet Σ that works in time linear in n but exponential in k [11]. This algorithm is based on a variant of the so-called *Universal Levenshtein Automaton* [7], which in turn is based on the *Levenshtein automaton* of w : the non-deterministic finite automaton recognising all words which are at Levenshtein distance at most k from w . For other related works, see [2, 3, 9, 10].

From a combinatorial point of view, the most natural question asks for upper and lower bounds on the size of the Levenshtein k -neighbourhood. Myers provided recurrences for counting the number of distinct sequences of k edit operations that one could perform on a given word and notes that “such bounds would give a tighter characterisation of the running time of the algorithm” behind BLAST [8]. A word is called *unary* if it consists of a single element of Σ . The main result of this work can be formally stated as follows.

► **Theorem 2.** *Let $a \in \Sigma$ be an arbitrary element of alphabet Σ . For any positive integers n and k , we have $|N_{k,\Sigma}(a^n)| < |N_{k,\Sigma}(w)|$, for any non-unary word w of length n .*

The course of our proof is to construct, for every word $u \in N_{k,\Sigma}(a^n)$, a distinct word $u' \in N_{k,\Sigma}(w)$ that can be obtained by a similar sequence of edit operations. In particular, we show that, for any n , k , and Σ ,

$$|N_{k,\Sigma}(a^n)| = \sum_{i=0}^k \sum_{j=i-k}^k \binom{n+j}{i} (\sigma-1)^i$$

is the size of the smallest Levenshtein k -neighbourhood of a word of length n , where $a \in \Sigma$ and $\sigma = |\Sigma|$. We remark that our main result was posed as a conjecture by Dufresne in [5].

Organisation of the Paper. The basic definitions and notation used throughout are introduced in Section 2. In Section 3, we present the main result of this work for binary alphabets – apart from the strictness of the inequality. We then generalise this result to arbitrary alphabets in Section 4 and prove the strictness of the inequality directly in this more general case. We conclude this paper in Section 5 with some final remarks.

2 Preliminaries

An *alphabet* Σ is a finite non-empty set of size $\sigma = |\Sigma|$ whose elements are called *letters*. A *word* over Σ is a sequence of letters from Σ . We call a word w *unary* if it consists of a single letter of Σ and *non-unary* if it consists of at least two letters of Σ . Σ^n denotes the set of words of length n over Σ and Σ^* denotes the set of finite words over Σ . For a word w , by $|w|$ we denote its length, and by $w[i]$, for $i = 1, \dots, |w|$, we denote its subsequent letters. The word of length 0 is the *empty word*, which we denote by ε .

We consider the following elementary edit operations: insertion, deletion, and substitution. For two words x and y , we define the *edit distance* (a.k.a. the *Levenshtein distance*) as the minimum number of edit operations that transform x to y , and we denote it by $\text{Lev}(x, y)$. The function Lev is then a metric on Σ^* [4].

Given a word w , an alphabet Σ , and a positive integer k , we define $N_{k, \Sigma}(w)$ as the set of all words in Σ^* that are at Levenshtein distance at most k from w . Formally, we have that

$$N_{k, \Sigma}(w) = \{v \in \Sigma^* : \text{Lev}(v, w) \leq k\}.$$

We call $N_{k, \Sigma}(w)$ the *Levenshtein* (k, Σ) -*neighbourhood* of w .

For any binary alphabet Σ , we define the *complement* of a word w over Σ as the word obtained by substituting $w[i]$ for letter $a \neq w[i]$, with $a \in \Sigma$, for all $i = 1, \dots, |w|$. We denote the complement of w by \bar{w} and we call a single such substitution operation a *flip*.

3 Main Result for Binary Alphabets

In this section we consider $\Sigma = \{a, b\}$, write $N_k(w)$ and refer to Levenshtein k -neighbourhood for simplicity. We present the main result but do not show the strictness of the inequality. We generalise this result to an arbitrary alphabet Σ and show the strictness in Section 4.

Let $N_k^j(w) = \{u \in N_k(w) : |u| = j\}$. Further let $\#_a(u)$ denote the number of a 's in word u . We illustrate the main ideas of our approach on a simple case and first consider the words of the neighbourhood that are of length at most n .

► **Observation 3.** Any $u \in N_k(a^n)$ with $|u| \leq n$ can be obtained from a^n by the following sequence of at most k edit operations: $n - |u|$ deletions of a 's in the beginning of a^n followed by a sequence of $|u| - \#_a(u)$ flips.

► **Example 4.** Let $w = aaaa$ and $k = 2$. Then $u = aba \in N_2(aaaa)$ can be obtained from $aaaa$ by deleting $n - |u| = 1$ letter a to obtain aaa and then by $|u| - \#_a(u) = 1$ flip to obtain aba .

Intuitively, the size of the set $N_k^j(a^n)$ is equal to the number of subsets of $\{1, \dots, j\}$ of size at most $k - (n - j)$; $n - j$ is the number of deletions and $k - (n - j)$ the number of flips.

► **Lemma 5.** If $j \leq n$, then $|N_k^j(a^n)| \leq |N_k^j(w)|$ for all $w \in \Sigma^n$.

10:4 Unary Words Have the Smallest Levenshtein k -Neighbourhoods

Proof. We use the characterisation of Observation 3. Let us argue why $|N_k^j(w)|$, for any word w of length n , is at least as big as $|N_k^j(a^n)|$. Consider the following procedure applied on word w : the deletion of the first $n - j$ letters of w followed by the flipping of at most $k - (n - j)$ letters. Clearly all words obtained by this procedure are distinct. This procedure thus gives us a subset of $N_k^j(w)$ that is of size equal to $|N_k^j(a^n)|$. ◀

Let us now consider the case of the words of the neighbourhood that have length greater than n . In particular, we denote the neighbourhood $\bigcup_{j>n} N_k^j(w)$ of these words by $N_k^{>n}(w)$ and thus $N_k^{\leq n}(w) = N_k(w) \setminus N_k^{>n}(w)$. For a word $u \in N_k^{>n}(a^n)$, we distinguish between two cases depending on the number of a 's in u :

- Case 1: $\#_a(u) \geq n$,
- Case 2: $\#_a(u) < n$.

The following observation states that in each case the word $u \in N_k^{>n}(a^n)$ can be obtained by a restricted sequence of edit operations.

► **Observation 6.**

- (1) Any $u \in N_k^{>n}(a^n)$ with $\#_a(u) \geq n$ can be obtained from a^n by the following sequence of at most k edit operations: $\#_a(u) - n$ insertions of a 's in the beginning of a^n followed by a sequence of $|u| - \#_a(u)$ insertions of b 's.
- (2) Any $u \in N_k^{>n}(a^n)$ with $\#_a(u) < n$ can be obtained from a^n by the following sequence of at most k edit operations: $n - \#_a(u)$ flips followed by a sequence of $|u| - n$ insertions of b 's. The insertions can be restricted to the part of the word after the rightmost flip.

► **Example 7.** Let $w = aaaa$ and $k = 2$. For Case 1, $u = aaaaba \in N_2^{>n}(aaaa)$ with $\#_a(u) = 5 \geq n = 4$ can be obtained by $\#_a(u) - n = 1$ insertion of a in the beginning of $aaaa$ to obtain $aaaaa$ and then by $|u| - \#_a(u) = 1$ insertion of b to obtain $aaaaba$. For Case 2, $u = aabab \in N_2^{>n}(aaaa)$ with $\#_a(u) = 3 < n = 4$ can be obtained by $n - \#_a(u) = 1$ flip to obtain $aaba$ and then by $|u| - n = 1$ insertion of b to the right of the flip to obtain $aabab$.

Intuitively, in Case 1, we insert the relevant number of a 's to reach $\#_a(u)$ because we have fewer a 's than needed, and then insert the relevant number of b 's. In Case 2, we flip the relevant number of a 's to go down to $\#_a(u)$ because we have more a 's than what is needed, and then insert the remaining b 's to the right of the rightmost flip.

Proof Strategy. Let u be an arbitrary element of $N_k(a^n)$, for some positive integers n and k . We define a function $f_u : \Sigma^n \rightarrow \Sigma^*$, such that:

1. $f_u(w) \in N_k(w)$, for all $w \in \Sigma^n$; and
2. Given w and $f_u(w)$ we can retrieve u .

Such an f_u directly yields the desired bound (apart from the strictness) since it implies that for any word w we cannot have $f_u(w) = f_{u'}(w)$ for $u, u' \in N_k(a^n)$, $u \neq u'$. In particular, we have that $|N_k(a^n)| \leq |N_k(w)|$ for any $w \in \Sigma^n$; see Table 1 for a complete example.

Note that for $|u| \leq n$ we already used the same idea to lower bound $|N_k^{\leq n}(w)|$ by $|N_k^{\leq n}(a^n)|$. Indeed, we implicitly defined $f_u(w)$ for $u \in N_k^{\leq n}(a^n)$ that consists in removing the first $n - |u|$ letters of w , resulting in a word w' , and then flipping the letters of w' at positions j where $u[j] = b$ (see Lemma 5).

■ **Table 1** Let $n = 3$, $w = aab$ and $k = 2$. The table presents an assignment f_u from every word in $N_2(a^3)$ to a different word in $N_2(aab)$ that is used in the proof of the main result. Note, however, that as per Theorem 2 there is at least one more word in $N_2(aab)$, namely, the word a .

(a) f_u for $u \in N_2^{\leq 3}(a^3)$.

$u \in N_2^{\leq 3}(a^3)$	$f_u(aab)$
a	b
aa	ab
ab	aa
ba	bb
aaa	aab
aab	aaa
aba	abb
abb	aba
baa	bab
bab	baa
bba	bbb

(c) f_u for $\#_a(u) \geq 3$ (Case 1).

$u \in N_2^{>3}(a^3)$	$f_u(aab)$
$aaaa$	$aaab$
$aaab$	$aaba$
$aaba$	$aabb$
$abaa$	$abab$
$baaa$	$baab$
$aaaaa$	$aaaaab$
$aaaab$	$aaaba$
$aaaba$	$aaabb$
$aaabb$	$aabaa$
$aabaa$	$aabab$
$aabab$	$aabba$
$aabba$	$aabbb$
$abaaa$	$abaab$
$abaab$	$ababa$
$ababa$	$ababb$
$abbaa$	$abbab$
$baaaa$	$baaab$
$baaab$	$baaba$
$baaba$	$baabb$
$babaa$	$babab$
$bbaaa$	$bbaab$

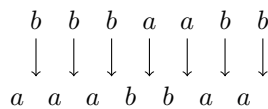
(b) f_u for $\#_a(u) < 3$ (Case 2).

$u \in N_2^{>3}(a^3)$	$f_u(aab)$
$aabb$	$aaaa$
$abab$	$abba$
$abba$	$abbb$
$baab$	$baba$
$baba$	$babb$
$bbaa$	$bbab$

Definition of f_u . Let us start by introducing the following edit operation on a non-empty word w . It takes as input parameters an integer $j \in [1, |w|]$ and a positive integer t .

$\text{ins-diff}(w, j, t)$: inserts a block of length t of letters equal to $\overline{w[j]}$ after the letter $w[j]$ (1)

See Figure 1 for an illustration of operation $\text{ins-diff}(w, j, t)$.



■ **Figure 1** For every position j of $w = aaabbaa$ (bottom), the letter (top) of which a block inserted by $\text{ins-diff}(w, j, 1)$ after position j would comprise.

In what follows, we assume that all insertions are with respect to the original indices of w . This can be achieved, for example, by performing insertions in a right-to-left manner when they are given as an ordered batch. Before providing the definition of f_u , we define two auxiliary operators g_x and h_x , for a word x .

Let us start with g_x . For any word x starting with a , we define an operator g_x that can be applied to any word y such that $|y| = \#_a(x)$. Intuitively, to construct word $g_x(y)$, the letters

10:6 Unary Words Have the Smallest Levenshtein k -Neighbourhoods

of y get in $g_x(y)$ the positions that letters a possess in x , and for every *maximal* block of b 's in between in x , i.e. a block consisting of only b 's that is neither preceded nor succeeded by a b , we apply an *ins-diff* operation on y . Specifically, we define $g_x(y) = v$ as follows: starting with $v = y$, for each maximal block $x[r..r+t-1]$ of b 's in x , with $\#_a(x[1..r-1]) = m$, perform *ins-diff*(v, m, t). Note that $|g_x(y)| = |x|$; see Example 8.

► **Example 8.** Let $x = aababbababaab$ and $y = aaabbaa$; note that $\#_a(x) = 7 = |y|$. We have $g_x(y) = v = aababbaa$; see also the figure below and recall that we perform this procedure from right to left. Starting from $v = y$, for the first maximal block $x[r..r+t-1] = x[13..13] = b$ with $\#_a(x[1..r-1]) = \#_a(x[1..12]) = m = 7$, we perform *ins-diff*($v, 7, 1$), which constructs $aaabbaab$. For the next maximal block $x[r..r+t-1] = x[10..10] = b$ with $\#_a(x[1..r-1]) = \#_a(x[1..9]) = m = 5$, we perform *ins-diff*($v, 5, 1$), which constructs $aaabbaaab$. For the next maximal block $x[r..r+t-1] = x[8..8] = b$ with $\#_a(x[1..r-1]) = \#_a(x[1..7]) = m = 4$, we perform *ins-diff*($v, 4, 1$), which constructs $aaababbaaab$, and so on.

$$\begin{array}{rcccccccccccc} x : & \boxed{a} & \boxed{a} & b & \boxed{a} & b & b & \boxed{a} & b & \boxed{a} & b & \boxed{a} & \boxed{a} & b \\ y : & a & a & & a & & & b & b & & a & a & & \\ g_x(y) : & a & a & b & a & b & b & b & a & b & a & a & a & b \end{array}$$

For any word x , we also define an operator h_x that takes as input a word y of length $|x|$ and flips its letters on positions in which x has b 's.

We are now in a position to define $f_u(w)$, for all $w \in \Sigma^n$. Recall that $u \in N_k^{>n}(a^n)$. We have the following two cases for f_u .

Case 1: $\#_a(u) \geq n$. Let us split u in its shortest suffix s that contains n a 's and the remaining (possibly empty) prefix p . We then define $f_u(w)$ for words u and w in this case as follows (see Example 9):

$$f_u(w) = p \cdot g_s(w). \quad (2)$$

► **Example 9.** Let $w = aaabbaa$ and $k = 4$; note that $n = 7$. If $u = abaaaaaaba$, then $\#_a(u) = 9 \geq n$, so we are in Case 1. We have $p = aba$ and $s = aaaaaaba$ is the shortest suffix that contains $n = 7$ occurrences of the letter a . Then $f_u(w)$ is constructed by concatenating p with the word $g_s(w)$ as shown in the figure below.

$$\begin{array}{rcccccccc} u : & a & b & a & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & b & \boxed{a} & \boxed{a} \\ w : & & & & a & a & a & b & b & & a & a \\ f_u(w) : & a & b & a & a & a & a & b & b & a & a & a \end{array}$$

Case 2: $\#_a(u) < n$. In this case we split u in its shortest suffix s' that contains $|u| - n$ b 's and the remaining prefix p' . Note that p' is always non-empty. We then define $f_u(w)$ for words u and w in this case as follows (see Example 10):

$$f_u(w) = h_{p'}(w') \cdot g_x(w)[|p'| + 1..|u|], \text{ where } x = a^{|p'|}s' \text{ and } w' = w[1..|p'|]. \quad (3)$$

In particular, $\#_a(x) = |x| - \#_b(x) = |u| - (|u| - n) = n$, and so applying g_x is well-defined.

► **Example 10.** Let $w = aaabbaa$ and $k = 4$; note that $n = 7$. If $u = aababbaab$, then $\#_a(u) = 5 < n$, so we are in Case 2. We have $p' = aabab$ and $s' = baab$ is the shortest suffix

that contains $|u| - n = 2$ occurrences of letter b . Then $f_u(w)$ is constructed by concatenating two words: the first one is $h_{p'}(w')$, where $w' = w[1..|p'|]$; and the second one is composed of the final $|s'|$ letters of $g_x(w)$, where x is the word obtained from u by changing the first $n - \#_a(u) = 2$ occurrences of b to a as shown in the figure below.

$u :$	a	a	b	a	b	b	a	a	b
$x :$	a	a	a	a	a	b	a	a	b
$w :$	a	a	a	b	b		a	a	
$g_x(w) :$	a	a	a	b	b	a	a	a	b
$h_{p'}(w') :$	a	a	b	b	a				
$f_u(w) :$	a	a	b	b	a	a	a	a	b

In Table 1 we provide a complete example of applying f_u , for each $u \in N_k(a^3)$, to $w = aab$. Let us now show the following fact.

► **Fact 11.** $|f_u(w)| = |u|$ and $Lev(w, f_u(w)) \leq Lev(a^n, u)$.

Proof. The first part can be readily verified. As for the second part, one can obtain $f_u(w)$ from w by the same sequence of edit operations (types and positions) that yield u from a^n , according to Observation 6. ◀

We next prove the main lemma on which our main result relies. A pseudocode implementing the algorithm used in the proof of this lemma can be found as Algorithm 1. Note that by considering a 's as 0's and b 's as 1's, we have that $h_x(y) \otimes y = x$, where \otimes denotes the XOR operation. Consider, for instance, $x = aabab$, $y = aaabb$ and $h_x(y) = aabba$.

■ **Algorithm 1** RETRIEVE($w, f_u(w)$) for $\Sigma = \{a, b\}$.

Input: Two words w and $f_u(w)$.

Output: A word u .

```

1:  $u \leftarrow \varepsilon$ 
2:  $k_1 \leftarrow |w|$ 
3:  $k_2 \leftarrow |f_u(w)|$ 
4: while  $k_2 > k_1$  and  $k_1 > 0$  do
5:   if  $w[k_1] = f_u(w)[k_2]$  then
6:      $k_1 \leftarrow k_1 - 1$ 
7:     prepend( $a, u$ )
8:   else
9:     prepend( $b, u$ )
10:   $k_2 \leftarrow k_2 - 1$ 
11: if  $k_1 = 0$  then                                     \\  $k_2 > 0$ ; Case 1 with  $p \neq \varepsilon$ 
12:   prepend( $f_u(w)[1..k_2], u$ )
13: else                                                 \\  $k_1 = k_2$ ; Case 2 (or Case 1 with  $p = \varepsilon$ )
14:   prepend( $w[1..k_1] \otimes f_u(w)[1..k_2], u$ )
15: return  $u$ 

```

► **Lemma 12.** Let u be an arbitrary element of $N_k^{>n}(a^n)$, for some positive integers n and k . Given w and $f_u(w)$, for any $w \in \Sigma^n$, we can retrieve u .

Proof. Let us note that, as we only had insertions and flips of letters, conceptually for each position in w there is a corresponding position in $f_u(w)$. The correspondence is given by ignoring the letters of $f_u(w)$ that were inserted by operation *ins-diff*. Our aim is to find all such pairs of corresponding positions in order to retrieve u .

To this end, we will swipe both w and $f_u(w)$ from right to left and prepend letters to an initially empty word u , which in the end will be equal to $u \in N_k^{>n}(a^n)$. We will maintain a position in each of the words, k_1 initiated as $|w|$ and k_2 initiated as $|f_u(w)|$.

Intuitively, we are first processing the part of $f_u(w)$ that comes from an application of operator g to w or to a suffix of w , depending on which case we are in. While processing this part, we maintain the invariant that the letter of $f_u(w)$ corresponding to $w[k_1]$ is the rightmost occurrence of $w[k_1]$ in $f_u(w)[1..k_2]$, relying on the definition of g . Then we can apply the following procedure repeatedly; see Lines 4-10 in Algorithm 1. We compute the rightmost occurrence of $w[k_1]$ in $f_u(w)[1..k_2]$; let it be at position j . We have that $u[j..k_2] = ab^{k_2-j}$. We prepend ab^{k_2-j} to u , decrement k_1 and set k_2 to $j - 1$; see Example 13.

Let us now focus on the stopping condition of this procedure, i.e. the point where the remaining prefix of $f_u(w)$ does not originate from an application of g . If we are in Case 1, while $k_1 > 0$ we must have that $k_2 \geq k_1 + |p|$. If we are in Case 2, while $k_2 > k_1$ we must have that $k_1 \geq |p'|$. Overall, while $0 < k_1 < k_2$, we must have that $k_2 > |p|$ if we are in Case 1 or $k_2 > |p'|$ if we are in Case 2.

If at some point k_1 reaches 0, i.e. we have consumed all of w , then we are in Case 1. Thus, $f_u(w)[1..k_2] = p$ and we prepend this prefix to u ; see Lines 11-12 in Algorithm 1.

Else, if at some point $k_1 = k_2$, i.e. we are left with equal-length prefixes of w and $f_u(w)$, then we are either in Case 2 or in Case 1 with $p = \varepsilon$. By using the XOR operation $w[1..k_1] \otimes f_u(w)[1..k_2]$ in the former case we retrieve p' and in the latter case we get a^{k_1} which is the missing prefix of s . In either case we prepend the result to u ; see Lines 13-14 in Algorithm 1. ◀

► **Example 13.** Let $u = abba \in N_2(a^3)$, $w = aab$, and $f_u(w) = abbb$. We have $k_1 = 3$ and $k_2 = 4$. At the first iteration of the while loop in Algorithm 1 we have $w[3] = f_u(w)[4] = b$ and so we set $k_1 = 2$, $u = a$ and $k_2 = 3$. At the second iteration we have $w[2] = a \neq f_u(w)[3] = b$ and so we get $u = ba$ and $k_2 = 2$. At this point we exit the while loop (because $k_1 = k_2$), and since we are at Case 2 we prepend $w[1..2] \otimes f_u(w)[1..2] = aa \otimes ab = ab$ to $u = ba$, which gives us $u = abba$. At this point we have retrieved $u = abba \in N_2(a^3)$.

By combining Lemmas 5 and 12 and Fact 11 we get $|N_k^j(a^n)| \leq |N_k^j(w)|$ for every j . This implies our main result for binary alphabets, apart from the strictness of the inequality. We leave the latter for the next section.

4 Generalisation to Arbitrary Alphabets and Strictness

For an arbitrary alphabet $\Sigma = \{0, \dots, \sigma - 1\}$ we only need to make minor adjustments in the definition of function f_u and in the algorithm for retrieving u from w and $f_u(w)$. Specifically, we replace the XOR operation by addition/subtraction modulo σ . Intuitively, one can think of 0's as a 's in the binary case, and of non-0's as b 's in the binary case.

Definition of f_u . Let u and v be two words of equal length. Let us denote by $u \oplus v$ the position-wise sum of words u and v modulo σ , e.g. for $\sigma = 4$ we have $1312 \oplus 1112 = 2020$. We analogously denote by $u \ominus v$ the position-wise subtraction of words u and v modulo σ .

We adapt operation *ins-diff* as follows, with z being a word containing only positive letters:

$$\text{ins-diff}(w, j, z) = \text{insert word } z \oplus (w[j])^{|z|} \text{ after the letter at position } j \text{ in word } w. \quad (4)$$

Operator g_x can now be applied to any word y such that $|y| = \#_0(x)$. g_x considers maximal blocks of letters in x not containing 0's instead of maximal blocks of b 's (in the binary case). For such a block z , it performs an *ins-diff*(y, j, z) operation on a word y .

The definition of f_u becomes as follows.

Case 1: $\#_0(u) \geq n$. We split the word u into p and s exactly as in the binary case and define $f_u(w) = p \cdot g_s(w)$; see Example 14.

► **Example 14.** Let $\Sigma = \{0, 1, 2\}$, $w = 201120$ and $k = 7$; note that $n = 6$. If $u = 0210001200210$, then $\#_0(u) = 7 \geq n$, so we are in Case 1. We have $p = 021$ and $s = 0001200210$ is the shortest suffix that contains $n = 6$ occurrences of the letter 0. Then $f_u(w)$ is constructed by concatenating p with the word $g_s(w)$ as shown in the figure below.

$$\begin{array}{rcccccccccccc} u : & 0 & 2 & 1 & \boxed{0} & \boxed{0} & \boxed{0} & 1 & 2 & \boxed{0} & \boxed{0} & 2 & 1 & \boxed{0} \\ w : & & & & 2 & 0 & 1 & & 1 & 2 & & & & 0 \\ f_u(w) : & 0 & 2 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 1 & 0 & 0 \end{array}$$

Case 2: $\#_0(u) < n$. The split of u into p' and s' is the same as in the binary case, but instead of s' containing $n - \#_a(u)$ b 's it now contains $n - \#_0(u)$ non-0's. $f_u(w) = (p' \oplus w[1..|p'|]) \cdot g_x(w)[|p'| + 1..|u|]$, where $x = 0^{|p'|}s'$; see Example 15.

► **Example 15.** Let $\Sigma = \{0, 1, 2\}$, $w = 201120$ and $k = 5$; note that $n = 6$. If $u = 21021020$, then $\#_0(u) = 3 < n$, so we are in Case 2. We have $p' = 2102$ and $s' = 1020$ is the shortest suffix that contains $|u| - n = 2$ occurrences of letters different than 0. Then $f_u(w)$ is constructed by concatenating two words: the first one is $h_{p'}(w')$, where $w' = w[1..|p'|]$; and the second one is composed of the final $|s'|$ letters of $g_x(w)$, where x is the word obtained from u by changing the first $n - \#_0(u) = 3$ occurrences of non-0 letters to 0 as shown in the figure below.

$$\begin{array}{rcccccccc} u : & 2 & 1 & 0 & 2 & | & 1 & 0 & 2 & 0 \\ x : & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & | & 1 & \boxed{0} & 2 & \boxed{0} \\ w : & 2 & 0 & 1 & 1 & | & 2 & & 0 & \\ g_x(w) : & 2 & 0 & 1 & 1 & | & 2 & 2 & 1 & 0 \\ h_{p'}(w') : & 1 & 1 & 1 & 0 & | & & & & \\ f_u(w) : & 1 & 1 & 1 & 0 & | & 2 & 2 & 1 & 0 \end{array}$$

Algorithm 2 is an adaptation of Algorithm 1 for $\Sigma = \{0, \dots, \sigma - 1\}$. Note that the two constructions are identical for $|\Sigma| = 2$, $a = 0$ and $b = 1$.

The proof of Lemma 5 that considers words of length at most n in $N_k(w)$ can be directly generalised for arbitrary alphabets, by allowing substitutions of letters instead of flips. This concludes the description of the generalisation.

The following theorem summarises all the results and introduces strictness in the inequality.

► **Theorem 2.** *Let $a \in \Sigma$ be an arbitrary element of alphabet Σ . For any positive integers n and k , we have $|N_{k,\Sigma}(a^n)| < |N_{k,\Sigma}(w)|$, for any non-unary word w of length n .*

10:10 Unary Words Have the Smallest Levenshtein k -Neighbourhoods

■ **Algorithm 2** RETRIEVE($w, f_u(w)$) for $\Sigma = \{0, \dots, \sigma - 1\}$.

Input: Two words w and $f_u(w)$.

Output: A word u .

```

1:  $u \leftarrow \varepsilon$ 
2:  $k_1 \leftarrow |w|$ 
3:  $k_2 \leftarrow |f_u(w)|$ 
4: while  $k_2 > k_1$  and  $k_1 \geq 1$  do
5:   prepend( $f_u(w)[k_2] \ominus w[k_1], u$ )
6:   if  $w[k_1] = f_u(w)[k_2]$  then
7:      $k_1 \leftarrow k_1 - 1$ 
8:    $k_2 \leftarrow k_2 - 1$ 
9: if  $k_1 = 0$  then                                \\  $k_2 > 0$ ; Case 1 with  $p \neq \varepsilon$ 
10:  prepend( $f_u(w)[1..k_2], u$ )
11: else                                             \\  $k_1 = k_2$ ; Case 2 (or Case 1 with  $p = \varepsilon$ )
12:  prepend( $f_u(w)[1..k_2] \ominus w[1..k_1], u$ )
13: return  $u$ 

```

Proof. We have $|N_{k,\Sigma}^j(a^n)| \leq |N_{k,\Sigma}^j(w)|$ for every j by combining the counterparts of Lemmas 5 and 12 and Fact 11 for an arbitrary alphabet. It thus suffices to find some value of j for which this inequality is strict.

Let us first consider the case that $k < n$, in which we claim that

$$|N_{k,\Sigma}^{n-k}(w)| > |N_{k,\Sigma}^{n-k}(a^n)| = 1.$$

Note that in this case words of length $n - k$ can be obtained only by performing k deletions, i.e. no insertions or substitutions are allowed. Hence $|N_{k,\Sigma}^{n-k}(a^n)| = 1$. For a non-unary w , we can, for instance, delete letters in lexicographic or reverse lexicographic order, breaking ties arbitrarily, obtaining words with different multiplicities for some letter.

Let us now proceed to the complementary case that $k \geq n$.

Then, each word $u \in N_{k,\Sigma}^{k+1}(a^n)$ can be obtained by exactly $k+1-n$ insertions and at most $n-1$ substitutions. Let us restrict ourselves to determining the size of $N'(w) \subseteq N_{k,\Sigma}^{k+1}(w)$, defined as the set of elements of $N_{k,\Sigma}^{k+1}(w)$ that can be obtained from w using exactly $k+1-n$ insertions and at most $n-1$ substitutions. In particular, one letter from w remains unchanged and gets shifted to the right by at most $k+1-n$ positions – possibly not shifted at all. Thus, each word $u \in N'(w)$ can be obtained as follows. We first choose the position i in u where the shifted letter has landed. For such a position i , it is a letter c occurring in $w[\max(1, i - (k+1-n)).. \min(i, n)]$ – any of those letters can be chosen by picking a right layout of insertions. We then put c at position i and fill the remaining k positions arbitrarily; see Example 16.

Let us do the above process once for each position i , with a fixed letter $\lambda(i)$, arbitrarily chosen from the possible ones. In total, we obtain all words from Σ^{k+1} apart from the ones which differ from $\lambda(i)$ on every position i . In particular, the total number of words that we get for this specific choice of $\lambda(i)$'s is $\sigma^{k+1} - (\sigma - 1)^{k+1}$ and this is equal to $|N'(a^n)| = |N_{k,\Sigma}^{k+1}(a^n)|$. Then, at some position j , since w is non-unary we can actually choose a letter $c \neq \lambda(j)$ instead; for instance any position j such that $w[j-1] \neq w[j]$ will work. Let us now pick this letter c and fill each other position i with a letter different from $\lambda(i)$. This way we obtain a word that was not obtained with the previous choice of $\lambda(i)$'s and hence $|N_{k,\Sigma}^{k+1}(w)| \geq |N'(w)| > |N_{k,\Sigma}^{k+1}(a^n)|$. ◀

► **Example 16.** Let us consider word $w = abc$ and $k = 5$. Every word $u \in N'(w)$ is obtained by 3 insertions and up to 2 substitutions. If the letter a from w is not substituted for, it can land at any of the positions from 1 to 4 in u ; similarly, b and c can land at positions from 2 to 5 and from 3 to 6, respectively. This is shown schematically in the following table.

position in u	1	2	3	4	5	6
landing positions of a	a	a	a	a		
landing positions of b		b	b	b	b	
landing positions of c			c	c	c	c

Then the i -th column specifies the possible choices for $\lambda(i)$, e.g., $\lambda(2) \in \{a, b\}$. Note that if w was unary, then all those sets would be singletons.

One possible choice of $\lambda(1), \dots, \lambda(6)$ is a, a, c, a, b, c . For it, we generate all the words but the 2^6 words that have no positions in common with $aacabc$. For a different choice of $\lambda(i)$'s, say, a, b, c, a, b, c , we obtain a word that was not generated before, e.g., $bbabca$ that has exactly one position in common with $abcabc$.

Let us now complete the picture by showing a closed formula for obtaining the tight lower bound implied by Theorem 2 and thus an efficient way to compute this bound.

► **Fact 17.**

$$|N_{k,\Sigma}(a^n)| = \sum_{i=0}^k \sum_{j=i-k}^k \binom{n+j}{i} (\sigma - 1)^i.$$

Proof. We first choose the number i of letters that are different from a in some $u \in N_{k,\Sigma}(a^n)$ and then the length $n + j$ of the word. Note that $j \geq i - k$ since we can have at most $k - i$ deletions as we need at least i insertions or substitutions to have i letters different from a . We then have $\binom{n+j}{i}$ options to choose the positions where the letter is not a and $(\sigma - 1)$ letters to choose from for each such position. ◀

► **Remark 18.** $|N_{k,\Sigma}(a^n)|$ can be computed with $\mathcal{O}(k^2)$ arithmetic operations.

5 Final Remarks

We showed a tight lower bound on the size of the Levenshtein k -neighbourhood. In particular, we defined a function f_u for each word $u \in N_{k,\Sigma}(a^n)$, such that, for any given $w \in \Sigma^n$, we have that $f_u(w) \in N_{k,\Sigma}(w)$ and $f_u(w) \neq f_{u'}(w)$ for $u \neq u'$. Our construction is not the only one possible. For example, in Case 1 of our construction, one could take $f_u(w) = q \cdot g_s(w)$, where $q = p \oplus 1^{|p|}$ (for the binary case, this corresponds to the negation of p). However, our construction has a neat property that $f_u(a^n) = u$, for any $u \in N_{k,\Sigma}(a^n)$.

The following two questions remain unanswered:

1. Can a similar approach be employed for showing a tight upper bound on $|N_{k,\Sigma}(w)|$?
2. Touzet gave an algorithm for computing $|N_{k,\Sigma}(w)|$ for a word w of length n over an alphabet Σ that works in time linear in n but exponential in k [11]. Can this computation be done in polynomial time or is this problem $\#P$ -hard?

References

- 1 Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. doi:10.1016/S0022-2836(05)80360-2.
- 2 Leonor Becerra-Bonache, Colin de la Higuera, Jean-Christophe Janodet, and Frédéric Tantini. Learning balls of strings from edit corrections. *The Journal of Machine Learning Research*, 9:1841–1870, 2008. URL: <https://dl.acm.org/citation.cfm?id=1442793>.
- 3 Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001. doi:10.1145/502807.502808.
- 4 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 5 Yoann Dufresne. An exploration of Levenshtein neighborhood densities. 14th Workshop on Compression, Text and Algorithms (WCTA), 2019. Talk.
- 6 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.
- 7 Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, December 2004. doi:10.1162/0891201042544938.
- 8 Gene Myers. What’s behind BLAST. In Cédric Chauve, Nadia El-Mabrouk, and Eric Tannier, editors, *Models and Algorithms for Genome Evolution*, pages 3–15. Springer, 2013. doi:10.1007/978-1-4471-5298-9_1.
- 9 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 10 Marie-France Sagot and Yoshiko Wakabayashi. Pattern inference under many guises. In Bruce A. Reed and Cláudia L. Sales, editors, *Recent Advances in Algorithms and Combinatorics*, pages 245–287. Springer New York, 2003. doi:10.1007/0-387-22444-0_8.
- 11 Hélène Touzet. On the Levenshtein automaton and the size of the neighbourhood of a word. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Proceedings*, pages 207–218, 2016. doi:10.1007/978-3-319-30000-9_16.

Summarizing Diverging String Sequences, with Applications to Chain-Letter Petitions

Patty Commins

Department of Computer Science, Carleton College, Northfield, MN, USA
Department of Mathematics, University of Minnesota, Minneapolis, MN, USA
commins.patty@gmail.com

David Liben-Nowell

Department of Computer Science, Carleton College, Northfield, MN, USA
dln@carleton.edu

Tina Liu

Department of Computer Science, Carleton College, Northfield, MN, USA
Surescripts, Minneapolis, MN, USA
tina.jxy.liu@gmail.com

Kiran Tomlinson

Department of Computer Science, Carleton College, Northfield, MN, USA
Department of Computer Science, Cornell University, Ithaca, NY, USA
kt@cs.cornell.edu

Abstract

Algorithms to find optimal alignments among strings, or to find a parsimonious summary of a collection of strings, are well studied in a variety of contexts, addressing a wide range of interesting applications. In this paper, we consider *chain letters*, which contain a growing sequence of signatories added as the letter propagates. The unusual constellation of features exhibited by chain letters (one-ended growth, divergence, and mutation) make their propagation, and thus the corresponding reconstruction problem, both distinctive and rich. Here, inspired by these chain letters, we formally define the problem of computing an optimal summary of a set of diverging string sequences. From a collection of these sequences of names, with each sequence noisily corresponding to a branch of the unknown tree T representing the letter's true dissemination, can we efficiently and accurately reconstruct a tree $T' \approx T$? In this paper, we give efficient exact algorithms for this summarization problem when the number of sequences is small; for larger sets of sequences, we prove hardness and provide an efficient heuristic algorithm. We evaluate this heuristic on synthetic data sets chosen to emulate real chain letters, showing that our algorithm is competitive with or better than previous approaches, and that it also comes close to finding the true trees in these synthetic datasets.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Applied computing → Law, social and behavioral sciences

Keywords and phrases edit distance, tree reconstruction, information propagation, chain letters

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.11

Related Version A full version of the paper including omitted proofs is available at <https://arxiv.org/abs/2004.08993>.

Supplementary Material Related research data and source code hosted at <https://github.com/tomlinsonk/diverging-string-seqs>.

Acknowledgements We thank Jon Kleinberg for extensive discussions, and Anna Johnson, Hailey Jones, Dave Musicant, Layla Oesper, Anna Rafferty, and Ethan Somes for helpful discussions during preliminary or late stages of this project.



© Patty Commins, David Liben-Nowell, Tina Liu, and Kiran Tomlinson;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 11; pp. 11:1–11:15

Leibniz International Proceedings in Informatics



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

1 Introduction

In a range of computational settings, we are given a collection of strings and asked to construct some kind of parsimonious representation of the given set. The task becomes more interesting if the strings result from a generative process, especially if new strings arise from a mechanism involving both replication and mutation of old strings. Now the parsimonious representation might be a *tree* describing the generative history of this population, with nodes corresponding to the strings and the branching structure representing the evolutionary events that produced that population. At this level of description, a host of applications fall under this rubric: reconstructing a phylogeny from a set of genes, tracing the spread of a textual meme in a social network, inferring the version history of a document from its many copies. These domains differ in the way that replication and mutation occur – sometimes randomly (“by nature”) and sometimes intentionally by humans embedded in a social structure – and, perhaps, whether some kind of selective pressure affects which strings survive or replicate.

Here, we consider a specific – and surprisingly rich – social setting in which a population of strings is generated: *chain letters*. Chain letters often feature an outlandish claim (“send a copy of this letter to ten friends, or you will have bad luck forever!”), but, more crucially, recipients are instructed to add their names to the document’s end, make a copy, and send those copies to multiple friends. Importantly, every subsequent recipient may modify any part of the document, or copy it imprecisely; thus each document contains a list of signatures, representing an ordered (if noisy) trace of its particular path through the social network. In the present work, we study the problem of accurately reconstructing the underlying propagation tree from a set of signature lists. If the lists of signatures contained no errors, the problem would be trivial, but errors abound, including both point mutations and structural variants. (In real email-based chain-letter data, some signatories retyped names, often incorrectly, and both block deletions and duplications appear [29]. Worse, some copies of the emails are only available as low-quality scanned images, introducing further errors.)

The propagation of chain letters. There are three crucial properties in chain-letter-like contexts that, together, make this data intriguingly different from other settings:

- (i) *chain letters grow (at one end)*. A document has an “active end,” and a document typically changes via the deposition of additional text (another name) at its active end.
- (ii) *chain letters diverge*. A document can *split* to create multiple “children” documents, which share a prefix up to the split but have differing suffixes below. The split is at the active end; two documents that diverge grow independently after the branching point.
- (iii) *chain letters mutate*. Actors introduce noise: an individual sending a chain letter to a friend makes a (potentially imperfect) *copy* of that document, possibly introducing errors – and those errors are “inherited” by subsequent copies of the letter.

Given a collection of many copies of “the same” chain letter, each with its own sequence of names, one can seek to reconstruct the underlying true record of the propagation – both the structure of the propagation tree *and* the strings representing the true names of the signatories. Together, the above properties make this chain-letter reconstruction problem a tantalizing domain for parsimonious reconstruction: as the rate of noise in document copying increases, naturally the reconstruction problem becomes difficult, but there is a great deal of repetition in the input data, particularly near the root of the propagation tree.

The present work. We formally introduce the *Diverging String Sequence Summarization Problem (DSSSP)*: given a collection X of sequences of strings (strings correspond to names, and each sequence is a noisy list of names in an instantiation of a chain letter), we seek a tree T that optimally summarizes X . (Rather than using the language of chain letters, we will abstract away the particular application and discuss diverging string sequences in general.)¹ What counts as an “optimal” summary depends on a tradeoff between two competing goods: the *accuracy* of T in representing the strings in the given sequences, and the *efficiency* of T in representing the given string sequences without too much redundancy. Our formal definition of the problem is parameterized to reflect the tradeoff between these two competing goods.

Our main theoretical results on DSSSP are (1) an efficient optimal algorithm for the case of $m = 2$ sequences, based on an approach we call *edit distance with give-up* (Theorem 4.1); (2) a proof of hardness for large m (Theorem 5.1); and (3) an exact polynomial-time algorithm for any fixed value of m (Theorem 5.2). We also give a much more efficient heuristic algorithm for large m – using a combination of divergence-aware pairwise alignment and iterative merging, inspired by progressive alignment algorithms [13] – and show empirically that it does a good job of reconstructing synthetically generated trees.

2 Related Work

Chain-letter data. In joint work with Jon Kleinberg, the second author studied the propagation of a widespread email-based anti-war petition [29]. This work focused on the topological structure of the underlying propagation tree; subsequent research sought to explain the shape of the tree through stochastic branching processes [17] or the rarity of sampled email copies [8]. The present work differs in that here we study the problem of accurately reconstructing the propagation tree from signature lists, rather than seeking to understand the structure of that tree. Still, examining the structure of the propagation tree presupposes a reconstructed tree, which in [29] was done using a hard edit distance cutoff to decide whether two signatures belong to the same signatory. (See Section 7.2.) This specific aspect of our problem – do multiple signatures belong to the same signatory? – has been considered in other forms in the past, including error-tolerant recognition of strings with various error models [4, 34], error correction of strings of regular languages [41], and block edit models for approximate string matching [30], all of which use various versions of edit distance.

Chain letters in paper form have also been investigated in the context of constructing a phylogeny based on variations in the text of the document itself (rather than a list of signatories) [3], or the propagation of stories as a network [21].

Other forms of propagation. In rare cases, a situation matching all three key features of chain letters has been studied – including a (controversial) model of the origin of life, based on layered clay accreting over time and even diverging and mutating [5, 6]. More common settings share two of the three features. For example, absent any errors, our reconstruction task is solved by a trie [10, 15] summarizing a set of diverging strings. Online conversations [24] (e.g., comment threads or especially email threads) have an active end at which new contributions appear, and threads can diverge, but there is no obvious notion of mutation.

¹ There is another layer of complication, literally: rather than viewing a document as a sequence of *characters*, we instead view it as a sequence of *signatures* (each of which is a string that consists of a sequence of characters). Thus there is a “two-level” view of edits, in which either an individual character can be corrupted (a single character-level edit within a particular signature) or an individual signature can be corrupted (an entire signature is deleted, inserted, or replaced by a different signature).

11:4 Summarizing Diverging String Sequences

There are also applications in which the objects of interest are strings that grow at one end, with noise but without meaningful divergence. In dendrochronology (the science of dating wood), approaches based on edit distance can be used to study sequences of growth rings in trees, which accumulate on one end, adjacent to the bark [44].

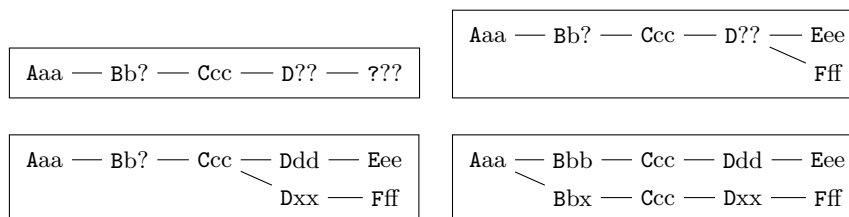
By far, though, the best-studied settings that match two of our three features have strings that mutate and replicate, but have no “active end” at which growth occurs. This is the classical setting of phylogenetic reconstruction, but it also appears in many other contexts. Most prominent is the spread of news, memes, and rumors that evolve as versions are created and shared (e.g., [1, 16, 20, 25, 39]). Mutations in these cases differ from ours, though, in that the content of the information being spread can affect the type of mutations that occur, thereby affecting the likelihood of further propagation (and therefore the structure of the tree). Much of this work seeks to understand various types of dissemination and what factors may impact the propagation structure – different from the goal of reconstructing the underlying tree. Reconstruction of the evolutionary history of a collection of divergent objects is also well studied in a bafflingly wide variety of contexts, from version histories of code snippets in Stack Overflow [2], to variations of the story “Little Red Riding Hood” [40], to diverging cultural histories using textile data [31].

Reconstruction algorithms. In addition to the algorithmic approaches to these various other forms of data, there is a voluminous literature on string alignment in the computational biology literature. The multiple sequence alignment problem is closely related to DSSSP, and many algorithms target a variety of challenges related to it (see [7, 22, 27, 36, 37, 42], among many others). There is also work involving the alignment of amino acid sequences to reconstruct the history of proteins, including mutations and divergence events [12].

3 Summarizing Diverging String Sequences

Before we formally define our abstract problem, we begin with some intuition, with terminology drawn from chain letters. Informally, a *name* is a string over a finite alphabet, and a *petition* is a sequence of names. We are given a *set* of petitions $X = \{x_1, \dots, x_m\}$, and we seek the tree T that best summarizes the set X . But the “best” tree depends on a tradeoff between two competing goods: (i) the efficiency of T (its number of nodes), and (ii) the accuracy of T in representing the petitions in X .

Consider petitions $x_1 = \text{Aaa Bbb Ccc Ddd Eee}$ and $x_2 = \text{Aaa Bbx Ccc Dxx Fff}$, with spaces separating names, as an example. Depending on the relative importance of efficiency and accuracy, there are *four* distinct “best” trees (see Figure 1): a trivial tree that never diverges (if efficiency matters much more than accuracy); a tree that diverges upon any textual discrepancy (if accuracy matters much more); or two intermediate trees that diverge after the Cccs or the D??s (depending on the cost–benefit of adding one node vs. paying for two textual errors).



■ **Figure 1** Four “best” trees for $x_1 = \text{Aaa Bbb Ccc Ddd Eee}$ and $x_2 = \text{Aaa Bbx Ccc Dxx Fff}$.

3.1 Distance between a Summary Tree and a String Sequence

To begin, we need to quantify how accurately a set X of string sequences is represented by a summary structure – and, more fundamentally, what it means to summarize X .

► **Definition 3.1** (Labeled Summary Tree). *Let X be a set of string sequences. A labeled summary tree of X is a pair $\langle T, f \rangle$, where T is a tree with each node labeled with a string, and f is a function mapping each $x \in X$ to a node v_x in T .*

Let $\text{labelseq}_T(v_x)$ denote the sequence of node labels on the path from the root of T to v_x .

That is, a summary of X consists of a labeled tree T , with a node of T designated to correspond to each sequence in X . To assess how accurately a string sequence $x \in X$ is represented, we will compare x with $\text{labelseq}_T(v_x)$. Our metric will be a variation on the classical Levenshtein edit distance [26], with two adjustments:

1. Edit distance is usually defined between two strings, but we wish to compare two *sequences* of strings. We cannot simply concatenate the strings within each sequence and then use standard edit distance, as the edits would no longer respect string boundaries. As such, we need to define an edit distance with two levels of granularity.
2. We insist that every string in each sequence $x \in X$ be represented (perhaps with some error) in the tree. Thus, when we align x to its path in the tree, we do not allow deletion of strings in the sequence. (We forbid deletions from $x \in X$ to preserve the intuition that the optimal summary tree for a singleton sequence $X = \{x\}$ is a nonbranching path successively labeled by the strings in x *even when the cost of nodes is very high*.)

Let x be a string sequence, and let $y = \text{labelseq}_T(v_x)$. Our metric, then, is an asymmetric two-level variant of the edit distance between x and y , allowing deletions from y but not x :

► **Definition 3.2** (Asymmetric Edit Distance). *Let x and y be string sequences. The asymmetric edit distance of x with respect to y , denoted $\text{AED}(x, y)$, is the cost of the cheapest sequence of operations transforming x into y , where the allowable operations are inserting a string into x and substituting a string. (No string can be deleted from x .)*

These operations' costs are given by (classical) edit distance ED : substituting w' for w costs $\text{ED}(w, w')$; inserting a string w into x costs $\text{ED}(w, \varepsilon)$, where ε is the empty string. (Unless otherwise specified, all ED edits have unit cost, but we allow arbitrary cost matrices.)

We compute $\text{AED}(x, y)$ with a variation on the classical dynamic program for edit distance, forbidding deletions and using ED to compute the cost of inserting or substituting a string.

The *distance* between a string sequence x and summary tree $\langle T, f \rangle$, then, is given by $\text{AED}(x, \text{labelseq}_T(f(x)))$ – i.e., the asymmetric edit distance between x and the label sequence on the path from root to the node corresponding to x in T .

3.2 The Problem: Summary Trees for a Set of String Sequences

We can now formally define our problem, where our objective function is – in the style of regularization in machine learning [18] – a weighted sum of the accuracy of the summary tree (as measured by AED) and the simplicity of the tree (as measured by its number of nodes):

► **Definition 3.3** (Diverging String Sequence Summarization Problem [DSSSP]).

Input: A set $X = \{x_1, x_2, \dots, x_m\}$ of string sequences and a nonnegative node cost λ .

Output: A labeled summary tree $\langle T, f \rangle$ (i.e., a tree T and a function f mapping each x_i to a node v_i in T) minimizing the following, where $|T|$ denotes the number of nodes in T :

$$\text{err}_\lambda(T) := \left[\sum_{i=1}^m \text{AED}(x_i, \text{labelseq}_T(v_i)) \right] + \lambda \cdot |T|. \quad (1)$$

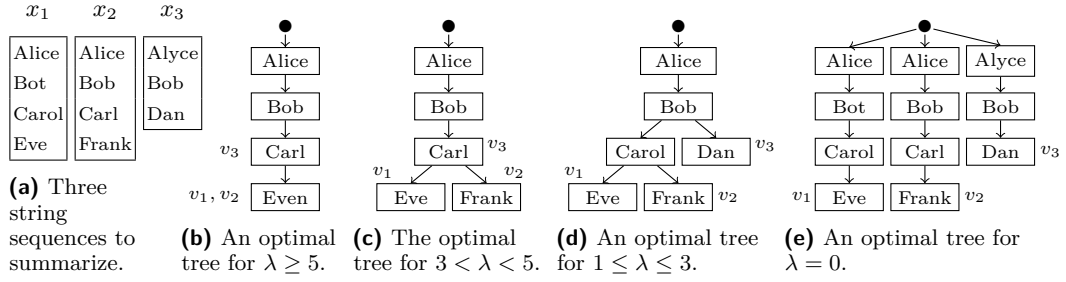


Figure 2 A set of string sequences and their optimal summary trees, for different ranges of λ . The \bullet root node denotes the sentinel string starting every sequence; nodes marked by $\{v_1, v_2, v_3\}$ correspond to the sequences $\{x_1, x_2, x_3\}$. The choice about whether to split Eve and Frank into two nodes (Figure 2b vs. 2c) is a function of their edit distance, $\text{ED}(\text{Eve}, \text{Frank}) = 5$. When $\lambda > 5$, then it is cheaper to accept the cost of aligning both to a single label than to pay for an extra node.

To ensure that T is a tree with a single root, we place sentinel values at the start of each sequence x_i . (Denote by $|T|$ the number of *non-sentinel* nodes in T .) Note that $\text{AED}(x_i, \text{labelseq}_T(v_i))$ is defined only if $|x_i| \leq |\text{labelseq}_T(v_i)|$ – that is, the depth of the node v_i in T is at least the number of strings in the sequence x_i – as it would otherwise be impossible to convert x_i into $\text{labelseq}_T(v_i)$ without deleting strings.

The parameter λ controls the tradeoff between trees that represent the input sequences accurately and trees that provide more concise summaries of the input set. When $\lambda = 0$, a trivial branch-immediately tree T is optimal; when $\lambda = \infty$, a trivial never-branching tree of depth $\max_i |x_i|$ is. Intermediate values of λ give more interesting structures. See Figure 2.

4 Solving DSSSP for Two Sequences: Edit Distance with Give-up

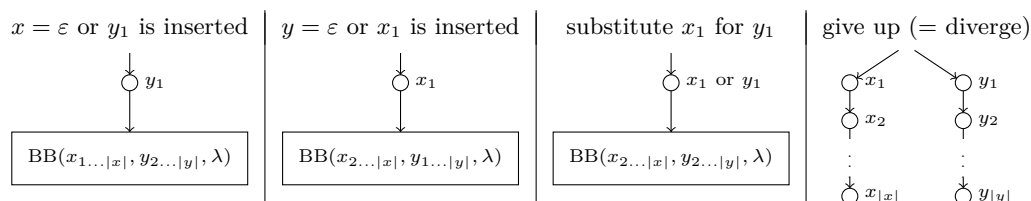
Consider first the case of just two input sequences, $m = |X| = 2$. (Even with $m = 2$, the problem has interesting subtleties.) We can compute an optimal tree through an alignment algorithm we call *edit distance with give-up*. The resulting tree has exactly one leaf or two leaves; in the latter case, we call the tree a *bifurcation*.

As with AED, the idea is similar to the classical dynamic program for edit distance, but with one additional operation permitted: *give up* entirely on aligning the remaining portions of the sequences, and declare a split at this point. We also modify the costs in the edit distance dynamic program to reflect the λ per-node cost of each operation, corresponding to the node-cost term in $\text{err}_\lambda(T)$. Writing $\text{EDG}(i, j, \lambda)$ to denote the cost of the best alignment of $x_i, \dots, |x|$ and $y_j, \dots, |y|$ under node cost λ , and writing $\text{EDG}(x, y, \lambda) = \text{EDG}(1, 1, \lambda)$, we have

$$\begin{aligned} \text{EDG}(|x| + 1, |y| + 1, \lambda) &= 0 & (2) \\ \text{EDG}(i, |y| + 1, \lambda) &= \lambda(|x| - i + 1) & \text{for any } 0 \leq i \leq |x| \\ \text{EDG}(|x| + 1, j, \lambda) &= \lambda(|y| - j + 1) & \text{for any } 0 \leq j \leq |y| \end{aligned}$$

and, for any $0 \leq i \leq |x|$ and any $0 \leq j \leq |y|$,

$$\text{EDG}(i, j, \lambda) = \min \begin{cases} \text{EDG}(i + 1, j + 1, \lambda) + \lambda + \text{ED}(x_i, y_j) & \text{(substitution)} \\ \text{EDG}(i, j + 1, \lambda) + \lambda + \text{ED}(\varepsilon, y_j) & \text{(insertion)} \\ \text{EDG}(i + 1, j, \lambda) + \lambda + \text{ED}(x_i, \varepsilon) & \text{(deletion)} \\ \lambda(|x| - i + 1) + \lambda(|y| - j + 1) & \text{(give up)} \end{cases}$$



■ **Figure 3** Constructing the tree in $\text{BUILD BIFURCATION}(x, y, \lambda)$. Whenever a node corresponding to $x_{|x|}$ or $y_{|y|}$ is placed, we map the corresponding input sequence to that node of the bifurcation. We write “BB” to abbreviate BUILD BIFURCATION , and we output an empty tree when $x = y = \varepsilon$.

For example, consider the insertion case of the minimum. Here we match the string y_j with no corresponding entry in x , and recursively align $y_{j+1, \dots, |y|}$ with $x_{i, \dots, |x|}$, with a total cost of

$$\underbrace{\text{EDG}(i, j + 1, \lambda)}_{\text{cost of alignment of remaining strings}} + \underbrace{\lambda}_{\text{cost of creating the root node}} + \underbrace{\text{ED}(\varepsilon, y_j)}_{\text{cost of inserting } y_j \text{ into } \text{labelseq}_T(x)}$$

The cost of “giving up” – i.e., declaring a split in the alignment – is large, requiring $(|x| - i + 1)$ nodes on the x branch and $(|y| - j + 1)$ on the y branch, each of which incurs cost λ .

Denote by BUILD BIFURCATION the natural dynamic programming algorithm that computes EDG using (2). (We abuse notation: $\text{BUILD BIFURCATION}(x, y, \lambda)$ denotes either the resulting bifurcation or its cost. We can construct this bifurcation simultaneously with the construction of the alignment; see Figure 3.)

BUILD BIFURCATION optimally solves DSSSP for $m = 2$ sequences – i.e., the tree T built by $\text{BUILD BIFURCATION}(x, y, \lambda)$ minimizes $\text{err}_\lambda(T)$. (Due to space constraints, the proof is omitted here, but is available in the full version of the paper linked on the title page.)

► **Theorem 4.1.** *The tree $T^* := \text{BUILD BIFURCATION}(x, y, \lambda)$ is an optimal summary tree for the string sequences $\{x, y\}$ with node cost λ . Specifically, $\text{err}_\lambda(T^*) = \text{EDG}(x, y, \lambda)$.*

Writing $n = \max(|x|, |y|)$ to denote the length of the longer of the two sequences, and $k = \max(\max_i |x_i|, \max_j |y_j|)$ to denote the length of the longest string in either sequence, then the running time of $\text{BUILD BIFURCATION}(x, y, \lambda)$ is $O(n^2 k^2)$.

5 Optimal Summaries of Larger Sets of String Sequences

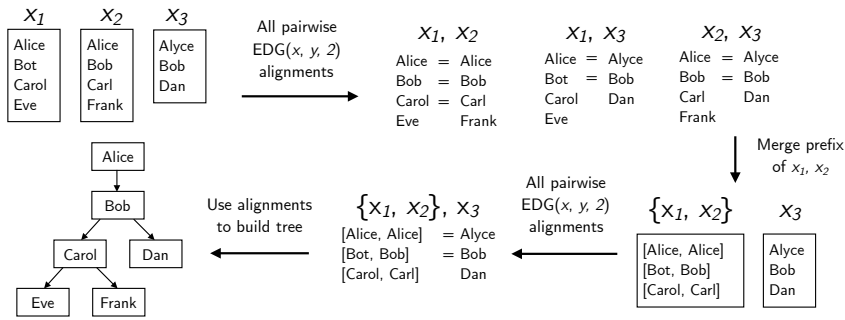
BUILD BIFURCATION efficiently finds the optimal summary tree for $m = 2$ sequences, but DSSSP with large m is computationally intractable.

► **Theorem 5.1.** *DSSSP (for an arbitrary number m of string sequences) is NP-hard.*

Proof (idea). For large λ , hardness follows from a reduction from String Median, which we will encounter shortly.

While the reduction from Median String is simpler, we can also give an alternative reduction from Shortest Common Supersequence (SCS) [35], which applies for smaller λ as well. (Note that SCS is hard in general, but for a small number of strings it is efficiently solvable [14].) The details of the latter proof are available in the full version. ◀

On the other hand, if we are willing to tolerate running times that are exponential in m (but polynomial in the other measures of input size), we can solve DSSSP in polynomial time:



■ **Figure 4** An example run of BUILD_TREE(X, λ) for $\lambda = 2$ and the sequences from Figure 2.

► **Theorem 5.2.** *Let X be a set of string sequences, where $m = |X|$ is the number of sequences, $n = \max_i |x_i|$ is the length of the longest sequence, and $k = \max_j \max_i |x_{i,j}|$ is the length of the longest string in any of the sequences. Then there is an algorithm solving DSSSP on X (for any λ) that runs in time $O(n^{m^2} \cdot 2^m \cdot k^m \cdot \text{poly}(k, m, n))$.*

Proof. The approach is brute force: we look at every possible tree topology τ , which specifies, for any $x, x' \in X$, the indices i and i' into x and x' at which they diverge. Every τ defines a set of nonbranching segments between divergences; the summary tree problem is then a collection of summary “path” problems, one per segment.

The path problem can be seen as multiple sequence alignment (MSA) [37], solvable via dynamic programming [7] (see also [22, 36, 42]). To implement the MSA dynamic program, we must compute the cost of assigning a set of strings $S = \{s_1, s_2, \dots, s_\ell\}$ to a single node u . If we label u with the string z , then the alignment cost of this node is $\lambda + \sum_i \text{ED}(s_i, z)$; thus the best label is the *string median* of S – that is, the string z minimizing the summed edit distance to the strings in S . While string median is NP-hard [11], a dynamic programming algorithm solves string median for a fixed number of strings [37] (see also [23, 33, 38]).

There are $O(n^{m \cdot (m-1)})$ tree topologies. Each defines a tree with $\leq m$ leaves and thus $\leq 2m$ nonbranching segments. Each segment contains $\leq m$ subsequences, each of length $\leq n$; thus each multiple sequence alignment requires $O(n^m)$ time [7]. Whenever we compute the string median of a candidate node, we have $\leq m$ strings each of length $\leq k$; computing these medians takes $O((2k)^m)$ time [37]. Finally, it takes $\text{poly}(k, m, n)$ time to compute $\text{err}_\lambda(T)$ for each tree. Thus the overall running time is $O(n^{m \cdot (m-1)} \cdot n^m \cdot (2k)^m \cdot \text{poly}(n, m, k))$. ◀

6 An Efficient Heuristic for Larger Sets of Sequences

Given DSSSP’s hardness (Theorem 5.1) and the abominable running time of our exact algorithm (Theorem 5.2), we turn here to an efficient heuristic for DSSSP with larger m . Our algorithm is greedy, and seeks to repeatedly identify the pair of sequences in X with the longest shared prefix, and then merge that shared prefix into a single sequence (as in BUILD_BIFURCATION). See Figure 4. There are several issues that we must resolve:

Measuring and merging the shared prefix of x_i and x_j . To calculate how well x_i and x_j match, we compute the $\text{EDG}(x_i, x_j, \lambda)$ alignment. Define the *number of substitutions* (ignoring insertions and deletions) *in the pre-divergence section* $p_{i,j}$ of this alignment as their overlap. Then, for the pair $\{x_i, x_j\}$ with the largest overlap, replace $\{x_i, x_j\}$ with $p_{i,j}$ in X . Note that $p_{i,j}$ is a sequence of *lists of strings*, not a sequence of *strings*; thus we need to generalize EDG to sequences of lists of strings, not just individual strings.

Reconciling labels in the final resulting tree. Repeating this merging process will define a tree, except that each node is labeled by a *list* of strings, not just one. To produce the final labels, we use the *medoid* string. For a list of strings A , the medoid of A is the string in A whose sum of edit distances to strings in A is minimized.²

Generalizing EDG to lists of strings. Define the edit distance between a string x and a set of strings X as $ED(x, X) := ED(x, \text{medoid}(X))$ – using the medoid of X as its representative string. Then, letting $\mathcal{C}(A, B)$ denote the cost of merging lists A and B , we define

$$\mathcal{C}(A, B) := \sum_{x \in A \cup B} ED(x, A \cup B) - \sum_{x \in A} ED(x, A) - \sum_{y \in B} ED(y, B). \tag{3}$$

This cost quantifies the amount of additional disagreement incurred by merging the lists, relative to leaving them separate. Insertion and deletion costs are found using (3) with a list of empty strings of appropriate length in place of x_i or y_j . We thus define the EDG recurrence (cf. Equation (2)) for sequences of lists of strings x and y as follows:

$$EDG(i, j, \lambda) = \min \begin{cases} EDG(i + 1, j + 1, \lambda) + \lambda + \mathcal{C}(x_i, y_j) & \text{(substitution)} \\ EDG(i, j + 1, \lambda) + \lambda + \mathcal{C}(\{|x_i| \text{ copies of } \varepsilon\}, y_j) & \text{(insertion)} \\ EDG(i + 1, j, \lambda) + \lambda + \mathcal{C}(x_i, \{|y_j| \text{ copies of } \varepsilon\}) & \text{(deletion)} \\ \lambda(|x| - i + 1) + \lambda(|y| - j + 1) & \text{(give up)} \end{cases}$$

Define $BUILDTREE(X, \lambda)$ as the greedy iterative algorithm suggested above: until there is only one sequence left in X , find the pair of sequences $x_i, x_j \in X$ with the largest number of substitutions in $EDG(x_i, x_j, \lambda)$, and replace $\{x_i, x_j\}$ by their merged prefix $p_{i,j}$. (Save the post-divergence branches of the bifurcation; we will reattach those branches at the bottom of $p_{i,j}$ in the final tree.) When there is only one sequence left, reattach all of the saved branches, and replace each node’s list-of-strings label by the medoid of that label list. (See Figure 4.)

7 Evaluation and Parameter Selection

$BUILDTREE$ is suboptimal both because greedy merging can yield a poor topology and because medoids can be poor node labels; see Examples 7.1 and 7.2. Still, we will show that it nonetheless performs well on simulated data, suggesting that it is a good heuristic.

► **Example 7.1** (A bad example for greedy merging). Consider the instance

$$x_1 = \text{a b c} \quad x_2 = \text{a b d} \quad x_3 = \text{a e d} \quad x_4 = \text{a e f}$$

with $0.5 < \lambda < 1$. The optimal tree T^* is shown below, with $\text{err}_\lambda(T^*) = 7\lambda$. However, $BUILDTREE$ will choose to merge sequences with the most closely aligned pair of sequences according to the EDG alignment. In this case, it would choose to merge x_2 and x_3 first. The tree T returned by $BUILDTREE$ has $\text{err}_\lambda(T) = 5\lambda + 2$, making it suboptimal.



² When we say “the” medoid of A , we mean the lexicographically first medoid of A . (Which medoid we choose never affects the sum of the distances at hand – e.g., in the sums in (3) – but we need to identify one in particular for the summands to be well-defined.) Note that the medoid, unlike the median, must be an element of A ; we use it because it is efficiently computable (unlike median) and is a $(2 - o(1))$ -approximation to the median (an implication of the triangle inequality).

► **Example 7.2** (A bad example for medoids). Consider the set $S = \{\text{XABC}, \text{AXBC}, \text{ABXC}, \text{ABCX}\}$, where each string sequence contains just one string. For large λ , the optimal tree is just a single node. Here the optimal label is the median ABC , which has total edit distance 4 to the set S ; the medoid label ABCX has total edit distance 6 to S .

7.1 Generating Synthetic Data

We will generate synthetic data based on several parameters: the number m of string sequences, the string length k , a string substitution probability σ_s , a string deletion probability δ_s , a character substitution probability σ_c , and a character deletion probability δ_c .

We generate trees using branching processes (i.e., *Galton–Watson trees* [43]): each node chooses to have exactly i children with probability p_i ; we fix $p_0 = 0.03$, $p_1 = 0.94$, and $p_2 = 0.03$ to approximate real email petition data [17,29]. We generate a Galton–Watson tree T until it has m leaves, restarting if the branching process terminates early. We label each node $u \in T$ with a random alphabetic string $\ell(u)$ of length k . Let x_i denote $\text{labelseq}_T(u_i)$ for the i th leaf u_i . Call T the *true tree*, ℓ the *true labels*, and x_i the *true sequences*.

Now, we simulate noisy propagation. To mirror petition data derived from low-quality scans of printed emails, we introduce string- and character-level errors in separate phases:

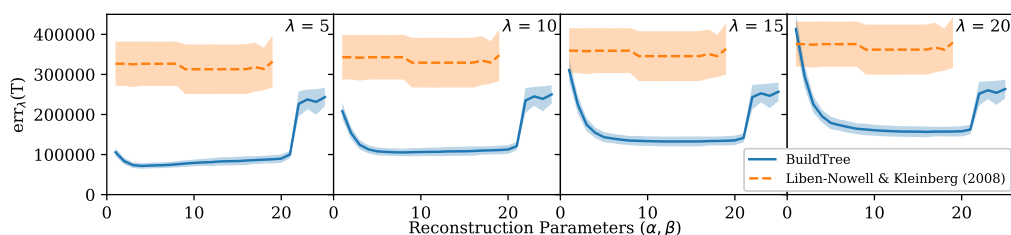
- (1) *string-level errors (which are inherited)*. Each node u inherits from its parent p the noisy history h_p of its ancestral labels. (The root “inherits” an empty sequence.) The node u (further) corrupts h_p : for each string in h_p , substitute it with a random alphabetic string of length k with probability σ_s , and delete it with probability δ_s . Finally, node u appends its true label $\ell(u)$ to h_p ; call the resulting sequence h_u . Now each leaf u stores h_u , a noisy version of $\text{labelseq}_T(u)$. Let $X' = \{x'_1, \dots, x'_m\}$ be the set of histories at the leaves.
- (2) *character-level errors (which appear independently)*. For each $x'_i \in X'$, substitute each character in each string in x'_i with a random character with probability σ_c , and delete the character with probability δ_c . Let $X'' = \{x''_1, \dots, x''_m\}$ be the resulting sequences.

Our experiments use string length $k = 25$, string error rates $\sigma_s = \delta_s = 0.001$, character error rates $\sigma_c = \delta_c = 0.1$, and $m \in \{15, 100\}$. Because string-level errors compound, 0.001 is a nontrivial error rate (roughly comparable to the 10% character-level error rate).

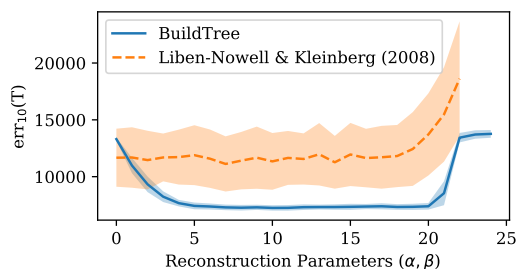
7.2 Comparing Reconstruction to Synthetic Ground Truth

We generate a true tree T , with true sequences $X = \{x_1, \dots, x_m\}$ and corrupted sequences $X'' = \{x''_1, \dots, x''_m\}$. We then use our heuristic algorithm to build a reconstructed tree $T' = \text{BUILD TREE}(X'', \alpha)$, for some choice of a node-cost parameter α to use in the reconstruction algorithm. (See Section 7.3 regarding how to choose α .) Note the distinction between the reconstruction parameter α and the evaluation parameter λ : BUILD TREE seeks to optimize $\text{err}_\alpha(T')$ for some value of α , but we can assess the quality of T' using $\text{err}_\lambda(T')$ whether or not $\alpha = \lambda$, to evaluate sensitivity to α .

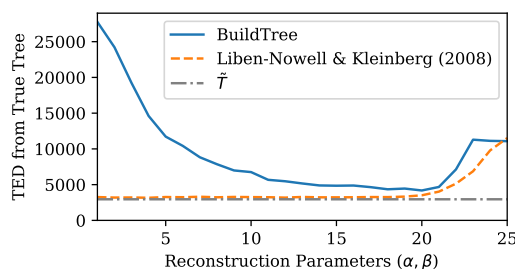
We will compare the quality of BUILD TREE to the threshold-based reconstruction algorithm from [29], which we briefly describe here. (1) Construct a weighted directed graph G with nodes labeled by all strings in all $x \in X$, and with an $a \rightarrow b$ edge if string a immediately precedes string b in any sequence $x \in X$. The weight of this edge is the *number* of sequences $x \in X$ containing a and b successively. (2) To handle minor signature errors, treat two signatures as equivalent if they follow equivalent signatories and have an edit distance below a fixed threshold β . (3) Compute the tree as a max-weight spanning arborescence of G , with extraneous nodes pruned away. (Note that β is on the same scale as α ; it defines a cutoff of $\text{ED}(a, a')$ indicating when a and a' should be assigned to two nodes versus one.)



(a) Error under $\text{err}_\lambda(T)$ with $|X| = 100$, for several λ values, averaged across 8 trials.



(b) Error under $\text{err}_{10}(T)$ with $|X| = 15$, averaged across 500 trials. Smaller X makes the error smaller than in (a), but the trend is similar.



(c) Error under ordered tree edit distance (TED) on a $|X| = 15$ dataset. The dash-dotted line shows TED between \tilde{T} and the real tree.

■ **Figure 5** Comparing $\text{BUILD TREE}(X, \alpha)$ to the algorithm from [29] with edit-distance threshold β . Shaded regions in (a) and (b) show standard deviations across the stated number of trials. Observe the flat-bottomed basin shape of the error curve for $\text{BUILD TREE}(X, \alpha)$ as α varies: high error rates when $\alpha < 5$ and when $\alpha > 20$, and roughly constant low error for all α in between.

Measures of performance. We assess the quality of our reconstructed trees T' in two ways.

First, we use the $\text{err}_\lambda(T')$ measure from (1), the sum of $\lambda \cdot |T'|$ and all label distances. Second, we use *tree edit distance* (TED) to compare the structure of T' to the true tree T . Edit distance between unordered trees is NP-hard [46], so we estimate the distance by ordering our trees and computing *ordered* TED using Zhang–Shasha [45], as implemented by Henderson [19]. (Specifically, we recursively order sibling nodes in T' to minimize the number of leaf order inversions with respect to T . As the number of siblings is typically small, this order is not expensive to compute.) To better interpret our results, we also compute TED to a tree with the correct structure but with label corruptions, denoted \tilde{T} ; this represents the best reconstruction we could hope for (without computing medians). We construct \tilde{T} by labeling the ancestors of each leaf u_i using the noisy history h_{u_i} (picking the medoid for nodes assigned multiple labels), deleting nodes with empty labels.

Evaluation results. We generated a tree T with $m = 100$ sequences, and corrupted sequences X'' (with 8 independent trials generating different X'' from T). Figure 5a compares $\text{err}_\lambda(\cdot)$ of our reconstruction with the method from [29], showing that BUILD TREE performs significantly better over a range of λ values. These trees were too big to compute tree edit distance, which is computationally prohibitive.

In addition, we generated a tree with $m = 15$ sequences and introduced error in 500 trials. Figure 5 shows both $\text{err}_{10}(T)$ (Figure 5b) and TED (Figure 5c) between the reconstructed tree \hat{T} and the true tree T . For many values of α , $\text{BUILD TREE}(X'', \alpha)$ produces significantly better solutions than the method from [29], as shown by the $\text{err}_{10}(T)$ measure. It is also competitive at $\alpha = 20$ according to (ordered) tree edit distance, a metric BUILD TREE was not designed to optimize.

7.3 How Should the Node Cost be Selected?

To reconstruct a tree from real sequence data, we must select a value for the reconstruction parameter α . If α is too low, nodes are too cheap and trees diverge too early; if α is too high, nodes are too costly and trees branch too rarely. But what value should we choose?

Intuitively, we wish to map two (corrupted) strings to the same node if they are (corruptions of) the same true string. Imagine two strings u and $v \neq u$, and let u' , u'' , and v' be the result of independently introducing errors to u , u , and v , respectively. We desire a value of α so that u' and u'' would probably be mapped to the same node, but u' and v' probably diverge. That is, the cost of creating a new node should be greater than the cost of aligning two corrupted versions of the same string, but the cost of diverging should be less than the cost of deleting/inserting all remaining strings. Thus we want $E[\text{ED}(u', u'')] < \alpha < E[\text{ED}(u', v')]$.

$E[\text{ED}(u', u'')]$ depends on the error rate of the corruption process and must be estimated from data. But estimates of $E[\text{ED}(u', v')]$ appear in the literature if the true strings u and v are uniformly random and have equal length. If the cost of substitution is twice that of insertion/deletion, then we can calculate $E[\text{ED}(u', v')]$ using Chvátal–Sankoff numbers [9] (the expected longest common subsequence length for two equal-length random strings); for unit edit costs, it is conjectured that $E[\text{ED}(u', v')] \approx |u'| \left(1 - \frac{1}{|\Sigma|}\right)$ for random equal-length strings over Σ [32]. If strings are not uniformly random or have different lengths, then we can estimate $E[\text{ED}(u', v')]$ from data and subsequently select an α between the upper and lower bounds. (In fact, there is some evidence that many values of α in this range perform well; see the flat-bottomed basin shape of Figure 5b.)

8 Discussion and Future Work

We described an efficient, practical heuristic to reconstruct a propagation tree from a noisy set of diverging string sequences – and in Section 7 we showed that BUILDTREE performs well on synthetic data. But, after all, the motivation for introducing this particular theoretical problem was for its application to real data, particularly chain-letter petitions. Rigorously testing BUILDTREE on real data, then, is perhaps the most natural direction for future work. (Testing on more realistic synthetic data is also an interesting future direction. Our data-generation process in Section 7.1 is unrealistic in a number of ways, perhaps most strikingly in its assumption that a name is a length- k alphabetic string chosen uniformly at random. More realistic randomized name-generation processes would make the synthetic task more similar to the real one.)

That said, there are several potentially interesting theoretical avenues for further exploration of DSSSP, too. The problem is (at least theoretically) tractable for any fixed number m of string sequences – but the dependence on m in our brute-force algorithm is brutal. Is there a more efficient algorithm for small m ? Or are there efficient algorithms with provable approximation guarantees for general m ? We can approximate the best labels for a fixed tree topology using medoids or, even better, using a PTAS for the string median problem [28]. Identifying the best tree topology seems more challenging, but perhaps this topological source of error in BUILDTREE (or some other heuristic) can be bounded.

There is another set of interesting open questions related to efficient algorithms for DSSSP when λ is small. (At the other extreme, the problem remains intractable when λ is very large: even if the optimal tree’s topology is the trivial non-branching one, as in Figure 2b, choosing the labels requires repeatedly solving instances of the NP-hard string median problem.) DSSSP is trivial when $\lambda = 0$; the diverge-at-the-root tree (as in Figure 2e) is optimal, with total cost 0. Even for strictly positive but small values of λ , there is an easy solution: if

$\lambda \leq \frac{1}{nm}$, it optimal to diverge upon encountering even a one-character difference between strings (i.e., the optimal summary tree is precisely the trie representation of the set X). Do related approaches make the problem tractable for bigger values of λ – e.g., if λ is small enough that we can only afford a bounded budget of edits in node labels? For how large a value of λ are there exact polynomial-time algorithms?

References

- 1 Lada Adamic, Thomas Lento, Eytan Adar, and Pauline Ng. Information evolution in social networks. In *International Conference on Web Search and Data Mining (WSDM'16)*, 2016.
- 2 Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of Stack Overflow code snippets. In *International Conference on Mining Software Repositories (MSR'19)*, pages 191–194, 2019.
- 3 Charles Bennett, Ming Li, and Bin Ma. Chain letters and evolutionary histories. *Scientific American*, 288(6):76–81, 2003.
- 4 Eric Brill and Robert Moore. An improved error model for noisy channel spelling correction. In *Proc. Association for Computational Linguistics (ACL'00)*, pages 286–293, 2000.
- 5 Theresa Bullard, John Freudenthal, Serine Avagyan, and Bart Kahr. Test of Cairns-Smith's 'crystals-as-genes' hypothesis. *Faraday Discussions*, 136:231–245, 2007.
- 6 Alexander Graham Cairns-Smith. *Seven clues to the origin of life: a scientific detective story*. Cambridge University Press, 1990.
- 7 Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, pages 1073–1082, 1988.
- 8 Flavio Chierichetti, David Liben-Nowell, and Jon Kleinberg. Reconstructing patterns of information diffusion from incomplete observations. In *Advances in Neural Information Processing Systems (NeurIPS'11)*, pages 792–800, 2011.
- 9 Vacláv Chvátal and David Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12(2):306–315, 1975.
- 10 Rene De La Briandais. File searching using variable length keys. In *Western Joint Computer Conference*, pages 295–298, 1959.
- 11 Colin de la Higuera and Francisco Casacuberta. Topology of strings: Median string is NP-complete. *Theoretical Computer Science*, 230(1-2):39–48, 2000.
- 12 Russell Doolittle. Reconstructing history with amino acid sequences. *Protein Science*, 1(2):191–200, 1992.
- 13 Da-Fei Feng and Russell Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.
- 14 Campbell Bryce Fraser. *Subsequences and supersequences of strings*. PhD thesis, University of Glasgow, 1995.
- 15 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.
- 16 Adrien Friggeri, Lada Adamic, Dean Eckles, and Justin Cheng. Rumor cascades. In *Eighth International AAAI Conference on Weblogs and Social Media (ICWSM'14)*, 2014.
- 17 Benjamin Golub and Matthew Jackson. Using selection bias to explain the observed structure of Internet diffusions. *Proceedings of the National Academy of Sciences*, 107(23):10833–10836, 2010.
- 18 Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.
- 19 Tim Henderson. Zhang-Shasha: Tree edit distance in Python. <https://github.com/timtadh/zhang-shasha>, 2019.
- 20 Manoel Horta Ribeiro, Kristina Gligoric, and Robert West. Message distortion in information cascades. In *The World Wide Web Conference (WWW'19)*, pages 681–692, 2019.
- 21 Folgert Karsdorp and Antal Van den Bosch. The structure and evolution of story networks. *Royal Society Open Science*, 3(6):160071, 2016.

11:14 Summarizing Diverging String Sequences

- 22 John Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Symposium on Combinatorial Pattern Matching (CPM'93)*, pages 106–119, 1993.
- 23 Joseph Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.
- 24 Ravi Kumar, Mohammad Mahdian, and Mary McGlohon. Dynamics of conversations. In *Intl. Conference on Knowledge Discovery and Data Mining (KDD'10)*, pages 553–562, 2010.
- 25 Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Intl. Conference on Knowledge Discovery and Data Mining (KDD'09)*, pages 497–506, 2009.
- 26 Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
- 27 Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.
- 28 Ming Li, Bin Ma, and Lusheng Wang. Finding similar regions in many sequences. *Journal of Computer and System Sciences*, 65(1):73–96, 2002.
- 29 David Liben-Nowell and Jon Kleinberg. Tracing information flow on a global scale using Internet chain-letter data. *Proceedings of the National Academy of Sciences*, 105(12):4633–4638, 2008.
- 30 Daniel Lopresti. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.
- 31 Luke Matthews, Jamie Tehrani, Fiona Jordan, Mark Collard, and Charles Nunn. Testing for divergent transmission histories among cultural characters: A study using Bayesian phylogenetic methods and Iranian tribal textile data. *PLoS ONE*, 6(4), 2011.
- 32 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
- 33 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(2-4):390–415, 2005. Previously in CPM'03.
- 34 Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, 1996.
- 35 Kari-Jouko R  ih   and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981.
- 36 Benjamin Raphael, Degui Zhi, Haixu Tang, and Pavel Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Research*, 14(11):2336–2346, 2004.
- 37 David Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, 1975.
- 38 David Sankoff, Robert Cedergren, and Guy Lapalme. Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. *Journal of Molecular Evolution*, 7(2):133–149, 1976.
- 39 Matthew Simmons, Lada Adamic, and Eytan Adar. Memes online: Extracted, subtracted, injected, and recollected. In *International Conference on Web and Social Media (ICWSM'11)*, 2011.
- 40 Jamshid Tehrani. The phylogeny of Little Red Riding Hood. *PLoS One*, 8(11):e78871, 2013.
- 41 Robert Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, May 1974.
- 42 Michael Waterman, Temple Smith, and William Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.
- 43 Henry William Watson and Francis Galton. On the probability of the extinction of families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4:138–144, 1875.

- 44 Carola Wenk. Applying an edit distance to the matching of tree ring sequences in dendrochronology. In *Symposium on Combinatorial Pattern Matching (CPM'99)*, pages 223–242, 1999.
- 45 Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.
- 46 Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.

Detecting k -(Sub-)Cadences and Equidistant Subsequence Occurrences

Mitsuru Funakoshi 

Department of Informatics, Kyushu University, Fukuoka, Japan
mitsuru.funakoshi@inf.kyushu-u.ac.jp

Yuto Nakashima 


Department of Informatics, Kyushu University, Fukuoka, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Shunsuke Inenaga 

Department of Informatics, Kyushu University, Fukuoka, Japan
PRESTO, Japan Science and Technology Agency, Kawaguchi, Japan
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai 

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

Masayuki Takeda 

Department of Informatics, Kyushu University, Fukuoka, Japan
takeda@inf.kyushu-u.ac.jp

Ayumi Shinohara 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

Abstract

The equidistant subsequence pattern matching problem is considered. Given a pattern string P and a text string T , we say that P is an *equidistant subsequence* of T if P is a subsequence of the text such that consecutive symbols of P in the occurrence are equally spaced. We can consider the problem of equidistant subsequences as generalizations of (sub-)cadences. We give bit-parallel algorithms that yield $o(n^2)$ time algorithms for finding k -(sub-)cadences and equidistant subsequences. Furthermore, $O(n \log^2 n)$ and $O(n \log n)$ time algorithms, respectively for equidistant and Abelian equidistant matching for the case $|P| = 3$, are shown. The algorithms make use of a technique that was recently introduced which can efficiently compute convolutions with linear constraints.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases string algorithms, pattern matching, bit parallelism, subsequences, cadences

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.12

Funding *Mitsuru Funakoshi*: JSPS KAKENHI Grant Number JP20J21147.

Yuto Nakashima: JSPS KAKENHI Grant Number JP18K18002.

Shunsuke Inenaga: JSPS KAKENHI Grant Number JP17H01697, JST PRESTO Grant Number JPMJPR1922.

Hideo Bannai: JSPS KAKENHI Grant Numbers JP16H02783, JP20H04141.

Masayuki Takeda: JSPS KAKENHI Grant Number JP18H04098.

Ayumi Shinohara: JSPS KAKENHI Grant Number JP15H05706.



© Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, and Ayumi Shinohara;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 12; pp. 12:1–12:11



Leibniz International Proceedings in Informatics

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

1 Introduction

Pattern matching on strings is a very important topic in string processing. Usually, strings are regarded and stored as one dimensional sequences and many pattern matching algorithms have been proposed to efficiently find particular substrings occurring in them [9, 2, 4, 8, 6, 3]. However, when one is to view the string/text data on paper or on a screen, it is usually shown in two dimensions: the single dimensional sequence is displayed in several lines folded by some length. It is known that the two dimensional arrangement can be used to embed hidden messages, and/or cause occurrences of unexpected or unintentional messages in the text. A common form for such an embedding is to consider the occurrence of a pattern in a linear layout: vertically or possibly diagonally along the two dimensional display.

For example, there was a (rather controversial) paper [12] on the so called Bible Code, claiming that the Bible contains statistically significant occurrences of various related words, occurring vertically and/or diagonally, in close proximity. Furthermore, there was an incident with a veto letter by the California State Governor [11]; Although it was considered a “weird coincidence”, the first character on each line of the letter could be connected and interpreted as a very provocative message. In Japanese internet forums, there was a culture of actively using these techniques, referred to as “tate-yomi”(vertical reading) and “naname-yomi”(diagonal reading), where the author of a message purposely embeds a hidden message in his/her post. Most commonly, the author will write a message that praises some object or opinion in question, but embed a message with a completely opposite meaning bearing the author’s true intention. The hidden message can be recovered by reading the text message vertically or diagonally from some position, and is used as form of sarcasm, as well as a clever method to mock those who were unable to *get* it.

Assuming that the text is folded into lines of equal length, vertical or diagonal occurrences of the pattern in two dimensions can be regarded as a subsequence of the original text, where the distance between each character is equal. We call the problem of detecting such occurrences of the pattern as the *equidistant subsequence matching* problem. To the best of the authors’ knowledge, there exist only publications concerning the statistical properties of the occurrence of equidistant subsequence patterns, mainly with the so called Bible Code.

Recently, a notion of regularities in strings called *(sub)-cadences*, defined by equidistant occurrences of the same character, was considered by Amir et al. [1]. A k -sub-cadence of a string can be viewed as an occurrence of an equidistant subsequence of length k that consists of the same character. A k -sub-cadence is a k -cadence, if the starting position is less than or equal to d and the ending position is greater than $n - d$, where d is the distance between each consecutive character occurrence and n is the length of the string. To date, algorithms for detecting anchored cadences (cadences whose starting position is equal to d), 3-(sub)-cadences, and (π_1, π_2, π_3) -partial-3-cadences (an occurrence of an equidistant subsequence that can become a cadence by changing a character at most all but three positions $i + \pi_1 d$, $i + \pi_2 d$, and $i + \pi_3 d$, where i is the starting position of the equidistant subsequence.) have been proposed [1, 5]. However, no efficient algorithm for detecting k -(sub)-cadences for arbitrary k ($1 \leq k \leq n$) is known so far.

In this paper, we present counting algorithms for k -sub-cadences, k -cadences, equidistant subsequence patterns of length m and length 3, and equidistant Abelian subsequence patterns of length 3. Table 1 shows a summary of the results. All algorithms run in $O(n)$ space. Furthermore, we present locating algorithms for k -sub-cadences, k -cadences, and equidistant subsequence patterns of length m . The time complexities of these algorithms can be obtained by adding *occ* to the second term inside the minimum function of each time complexity of

the counting algorithm. To the best of the authors' knowledge, these are the first $o(n^2)$ time algorithms for k -(sub)-cadences and equidistant subsequence patterns. In this paper, we assume a word RAM model with word size $\Theta(\log n)$. Also, unless otherwise noted, we assume that strings over a general ordered alphabet.

■ **Table 1** Summary of results. Note that an equidistant Abelian subsequence pattern is an equidistant subsequence of any permutation of a given pattern.

Counting time	For a constant size alphabet	For a general ordered alphabet
k -sub-cadences	$O\left(\min\left\{\frac{n^2}{k}, \frac{n^2}{\log n}\right\}\right)$	$O\left(\min\left\{\frac{n^2}{k}, \frac{n^2\sqrt{k}}{\sqrt{\log n}}\right\}\right)$
k -cadences	$O\left(\frac{n^2}{k^2 \log n}\right)$	$O\left(\min\left\{\frac{n^2}{k^2}, \frac{n^2}{\sqrt{k}\sqrt{\log n}}\right\}\right)$

Counting time	For a general ordered alphabet
Equidistant subsequence pattern	$O\left(\min\left\{\frac{n^2}{m}, \frac{n^2}{\log n}\right\}\right)$
Equidistant subsequence pattern of length three	$O(n \log^2 n)$
Equidistant Abelian subsequence pattern of length three	$O(n \log n)$

2 Preliminaries

Let Σ be the *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. String $s \in \Sigma^*$ is said to be a *subsequence* of string $T \in \Sigma^*$ if s can be obtained by removing zero or more characters from T .

For a string T and an integer $1 \leq i \leq |T|$, $T[i]$ denotes the i -th character of T . For two integers $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T that begins at position i and ends at position j . For convenience, let $T[i..j] = \varepsilon$ when $i > j$.

2.1 k -(Sub-)Cadences

The term ‘‘cadence’’ has been used in slightly different ways in the literature (e.g., see [7, 10, 1]). In this paper, we use the definitions of cadences and sub-cadences which are used in [1] and [5].

For integers i and d , the pair (i, d) is called a k -*sub-cadence* of $T \in \Sigma^n$ if $T[i] = T[i + d] = T[i + 2d] = \dots = T[i + (k - 1)d]$, where $1 \leq i \leq n$ and $1 \leq d \leq \lfloor \frac{n-i}{k-1} \rfloor$. The set of k -sub-cadences of T can be defined as follows:

► **Definition 1.** For $T \in \Sigma^n$, $n \in \mathcal{N}$, and $k \in [1..n]$,

$$KSC(T, k) = \left\{ (i, d) \mid \begin{array}{l} T[i] = T[i + d] = T[i + 2d] = \dots = T[i + (k - 1)d] \\ 1 \leq i \leq n, 1 \leq d \leq \lfloor \frac{n-i}{k-1} \rfloor \end{array} \right\}.$$

For integers i and d , the pair (i, d) is called a k -*cadence* of $T \in \Sigma^n$ if (i, d) is a k -sub-cadence and satisfies the inequalities $i - d \leq 0$ and $n < i + kd$. The set of k -cadences of T can be defined as follows:

► **Definition 2.** For $T \in \Sigma^n$, $n \in \mathcal{N}$, and $k \in [1..n]$,

$$KC(T, k) = \left\{ (i, d) \mid \begin{array}{l} T[i] = T[i + d] = T[i + 2d] = \dots = T[i + (k - 1)d] \\ 1 \leq i \leq d, \frac{n-i}{k} < d \leq \lfloor \frac{n-i}{k-1} \rfloor \end{array} \right\}.$$

2.2 Equidistant Subsequence Occurrences

For integers i and d , we say that pair (i, d) is an *equidistant subsequence occurrence* of $P \in \Sigma^m$ in $T \in \Sigma^n$ if $P = T[i] \cdot T[i + d] \cdot T[i + 2d] \cdots T[i + (m - 1)d]$, where $1 \leq i \leq n$ and $1 \leq d \leq \lfloor \frac{n-i}{m-1} \rfloor$. The set of equidistant subsequence occurrences of P in T can be defined as follows:

► **Definition 3.** For $T \in \Sigma^n, P \in \Sigma^m$ and $n, m \in \mathcal{N}$,

$$ESP(T, P) = \left\{ (i, d) \mid \begin{array}{l} P = T[i] \cdot T[i + d] \cdot T[i + 2d] \cdots T[i + (m - 1)d] \\ 1 \leq i \leq n, 1 \leq d \leq \lfloor \frac{n-i}{m-1} \rfloor \end{array} \right\}.$$

2.3 Equidistant Abelian Subsequence Occurrences

Two strings S_1 and S_2 are said to be *Abelian equivalent* if S_1 is a permutation of S_2 , or vice versa. Now for integers i and d , we say that pair (i, d) is an *equidistant Abelian subsequence occurrence* of $P \in \Sigma^m$ in $T \in \Sigma^n$ if $T[i] \cdot T[i + d] \cdot T[i + 2d] \cdots T[i + (m - 1)d]$ and P are Abelian equivalent, where $1 \leq i \leq n$ and $1 \leq d \leq \lfloor \frac{n-i}{m-1} \rfloor$. The set of equidistant Abelian subsequence occurrences of P in T can be defined as follows:

► **Definition 4.** For $T \in \Sigma^n, P \in \Sigma^m$ and $n, m \in \mathcal{N}$,

$$EASP(T, P) = \left\{ (i, d) \mid \begin{array}{l} T[i] \cdot T[i + d] \cdots T[i + (m - 1)d] \text{ and } P \text{ are Abelian equivalent} \\ 1 \leq i \leq n, 1 \leq d \leq \lfloor \frac{n-i}{m-1} \rfloor \end{array} \right\}.$$

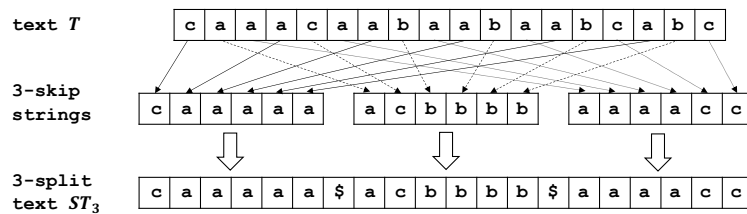
When it is clear from the context, we denote $KSC(T, k)$ as KSC , $KC(T, k)$ as KC , and $ESP(T, P)$ as ESP .

3 Detecting k -Sub-Cadences

In this section, we consider algorithms for detecting k -sub-cadences.

3.1 Algorithm 1

One of the simplest methods is as follows: For each distance d with $1 \leq d \leq \lfloor \frac{n-1}{k-1} \rfloor$, we construct text $ST_d = T[1] \cdot T[1 + d] \cdots T[1 + d \lfloor \frac{n-1}{d} \rfloor] \cdot \$ \cdot T[2] \cdot T[2 + d] \cdots T[2 + d \lfloor \frac{n-2}{d} \rfloor] \cdot \$ \cdots T[d] \cdot T[2d] \cdots T[d \lfloor \frac{n}{d} \rfloor]$ of length $d \lfloor \frac{n}{d} \rfloor + d - 1$. Then, the strings $T[1] \cdot T[1 + d] \cdots T[1 + d \lfloor \frac{n-1}{d} \rfloor]$, $T[2] \cdot T[2 + d] \cdots T[2 + d \lfloor \frac{n-2}{d} \rfloor]$, ..., $T[d] \cdot T[2d] \cdots T[d \lfloor \frac{n}{d} \rfloor]$ are called *d -skip-strings*, and the ST_d is called *d -split text*. If we would like to find k -sub-cadences with distance d in text T , we find concatenations of the same character of length k as substrings in ST_d .



■ **Figure 1** Preprocessing for Algorithm 1.

Fig. 1 is an example of the 3-split text ST_3 . In ST_d , we use a symbol $\$ \notin \Sigma$ in order to prevent detecting false occurrences of concatenation of same character of the length k across the ends of d -skip strings as a k -sub-cadence. The text obtained by concatenating all ST_d for

all $1 \leq d \leq \lfloor \frac{n-1}{k-1} \rfloor$ and $\$$ is called the *split text*. If we prepare the split text, we can compute KSC simply by checking that the same character is repeated k times.

The length of ST_d is at most $n + d$ including $\$$. The maximum value of d is $\lfloor \frac{n-1}{k-1} \rfloor$, and therefore, the number of ST_d is at most $\lfloor \frac{n-1}{k-1} \rfloor$. Hence, the length of the split text of T is $O(\frac{n^2}{k})$. We can check that the same character is repeated k times in the split text in $O(\frac{n^2}{k})$ time. Although we have presented the split text to ease the description, it does not have to be constructed explicitly.

From the above, we can get the following result.

► **Theorem 5.** *There is an algorithm for locating all k -sub-cadences for given k ($1 \leq k \leq n$) which uses $O(\frac{n^2}{k})$ time and $O(n)$ space.*

As can be seen from the example of $T = \mathbf{a}^n$, $|KSC|$ can be $\Omega(\frac{n^2}{k})$. Therefore, when we locate all $(i, d) \in KSC$, this algorithm is optimal in the worst case. In the next subsection, we show a counting algorithm that is efficient when the value of k is small. Moreover, we show a locating algorithm that is efficient when both the value of k and $|KSC|$ is small.

3.2 Algorithm 2

In this subsection, we will show the following result:

► **Theorem 6.** *For a constant size alphabet, there is an $O(\frac{n^2}{\log n})$ time algorithm for counting all k -sub-cadences for given k . We can also locate these occurrences in $O(\frac{n^2}{\log n} + occ)$ time, where occ is the number of the outputs. For a general ordered alphabet, there is an $O(\frac{n^2\sqrt{k}}{\sqrt{\log n}})$ time algorithm for counting all k -sub-cadences for given k . We can also locate these occurrences in $O(\frac{n^2\sqrt{k}}{\sqrt{\log n}} + occ)$ time. These algorithms run in $O(n)$ space.*

Note that for counting all k -sub-cadences, for a constant size alphabet (resp. for a general ordered alphabet), this algorithm is faster than Algorithm 1 if k is $o(\log n)$ (resp. $o(\sqrt[3]{\log n})$). For locating all k -sub-cadences, for a constant size alphabet (resp. for a general ordered alphabet), if $|KSC|$ is $o(\frac{n^2}{k})$ and k is $o(\log n)$ (resp. $o(\sqrt[3]{\log n})$), then this algorithm is faster.

Now we will show how to count all k -sub-cadences of character $c \in \Sigma$. Let $\delta_c[1..n]$ be a binary sequence given by the indicator function for the character c :

$$\delta_c[i] := \begin{cases} 1 & \text{if } T[i] = c, \\ 0 & \text{if } T[i] \neq c. \end{cases}$$

If (i, d) is a k -sub-cadence with character c , $\delta_c[i] = \delta_c[i + d] = \dots = \delta_c[i + (k-1)d] = 1$. Therefore, we can check whether (i, d) is a k -sub-cadence or not by computing $\delta_c[i] \cdot \delta_c[i + d] \cdot \dots \cdot \delta_c[i + (k-1)d]$. To compute this, we use bit-parallelism, i.e, the bit-wise operations AND and SHIFT_LEFT, denoted by $\&$ and \ll , respectively, as in the C language. For each d with $1 \leq d \leq \lfloor \frac{n-1}{k-1} \rfloor$, let $Q_d = \delta_c \& (\delta_c \ll d) \& (\delta_c \ll 2d) \& \dots \& (\delta_c \ll (k-1)d)$. If $Q_d[i] = 1$, then (i, d) is a k -sub-cadence. See Figure 2 for a concrete example.

If we want to count all k -sub-cadences with d , we only have to count the number of 1's in Q_d . If we want to locate all k -sub-cadences with d , we have to locate all 1's in Q_d .

In the word RAM model, SHIFT_LEFT and AND operations can be done in constant time per operation on bit sequences of length $O(\log n)$. Since δ_c is a binary sequence of length n , one SHIFT_LEFT or AND operation can be done in $O(\frac{n}{\log n})$ time. Therefore, Q_d

$$\begin{array}{rcl}
 T & = & \text{c a a a c a a b a a b a a b c a b c} \\
 \delta_a & = & 0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 \\
 \\
 \delta_a & = & 0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 \\
 (\delta_a \ll 3) & = & 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 0 0 0 \\
 (\delta_a \ll 6) & = & 1 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 \\
 (\delta_a \ll 9) & = & 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 \\
 \hline
 Q_3 & = & 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
 \end{array}$$

■ **Figure 2** Let $T = \text{caaacaabaabaabcabc}$. $(3, 3)$, $(4, 3)$, and $(7, 3)$ are 4-sub-cadences of character ‘a’ with $d = 3$.

can be obtained in $O(k \frac{n}{\log n})$ time. Since it is known that the number of 1’s in a bit sequence of length $O(\log n)$ can be obtained in $O(1)$ time by using the “popcnt” operation which is a standard operation on the word RAM model, the number of 1’s in Q_d can be counted in $O(\frac{n}{\log n})$ time. Hence, for all $1 \leq d \leq \lfloor \frac{n-1}{k-1} \rfloor$, we can count all k -sub-cadences of character c in $O\left(k \frac{n}{\log n} \lfloor \frac{n-1}{k-1} \rfloor + \frac{n}{\log n} \lfloor \frac{n-1}{k-1} \rfloor\right) \subseteq O(\frac{n^2}{\log n})$ time. Also, it is known that the position of the rightmost 1 (the least significant set bit) in a bit sequence of length $O(\log n)$ can be answered in constant time by using bit-wise operations. We split Q_d into $O(\frac{n}{\log n})$ blocks of length $O(\log n)$. For each block, the least significant set bit can be found in $O(1)$ time if the block contains at least one 1. After finding the least significant set bit, we mask this bit to 0 and do the above operation again. Bit mask operation can be done in $O(1)$ time. Hence, we can answer all the positions of 1’s in Q_d in $O(\frac{n}{\log n} + occ)$ time. Therefore, we can locate all k -sub-cadences of character c in $O(\frac{n^2}{\log n} + occ)$ time.

We showed how to detect all k -sub-cadences of character c , so we can detect all k -sub-cadences by doing the above operations for each character in Σ . For a constant size alphabet, since we only do the above operations a constant number of times, we can count all k -sub-cadences in $O(\frac{n^2}{\log n})$ time. We can also locate these occurrences in $O(\frac{n^2}{\log n} + occ)$ time. However, for a general ordered alphabet, we have to do the above operations $|\Sigma|$ times.

For a general ordered alphabet, if the number of occurrences of the character is small, we use another algorithm that generalizes Amir et al.’s algorithm [1] for detecting 3-cadences to k -sub-cadences: Let N_c be the set of positions which are occurrences of a character c . If we pick two positions in N_c and regard the smaller one as the starting position i of k -sub-cadences and the larger one as the second position $i + d$ of a k -sub-cadence, then the distance d is uniquely determined. We can check whether the pair (i, d) is a k -sub-cadence or not in $O(k)$ time. Since the number of pairs is at most $|N_c|^2$, we can count or locate k -sub-cadences of character c in $O(k|N_c|^2)$ time.

Thus, for a general ordered alphabet, all k -sub-cadences can be counted in $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n}\})$ time. Since $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n}\})$ is maximized when $k|N_c|^2 = \frac{n^2}{\log n}$, then $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n}\}) \subseteq O((\sum_{c \in \Sigma} |N_c|) \frac{n\sqrt{k}}{\sqrt{\log n}}) \subseteq O(\frac{n^2\sqrt{k}}{\sqrt{\log n}})$. Therefore we can count in $O(\frac{n^2\sqrt{k}}{\sqrt{\log n}})$ time by using Algorithm 2 and the above algorithm that generalizes Amir et al.’s algorithm. Also, all k -sub-cadences can be located in $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n} + occ_c\})$ time where occ_c is the number of k -sub-cadences of character c . Since $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n} + occ_c\}) \subseteq O((\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{\log n}\}) + occ) \subseteq O(\frac{n^2\sqrt{k}}{\sqrt{\log n}} + occ)$, we can locate all k -sub-cadences in $O(\frac{n^2\sqrt{k}}{\sqrt{\log n}} + occ)$ time.

From the above, we obtain the following result:

► **Theorem 7.** *For a constant size alphabet (resp. for a general ordered alphabet), all k -sub-cadences with given k can be counted in $O\left(\min\left\{\frac{n^2}{k}, \frac{n^2}{\log n}\right\}\right)$ time (resp. $O\left(\min\left\{\frac{n^2}{k}, \frac{n^2\sqrt{k}}{\sqrt{\log n}}\right\}\right)$ time) and $O(n)$ space, and can be located in $O\left(\min\left\{\frac{n^2}{k}, \frac{n^2}{\log n} + occ\right\}\right)$ time (resp. $O\left(\min\left\{\frac{n^2}{k}, \frac{n^2\sqrt{k}}{\sqrt{\log n}} + occ\right\}\right)$ time) and $O(n)$ space.*

4 Detecting k -Cadences

In this section, we consider algorithms for detecting k -cadences.

4.1 Algorithm 3

Again, each (i, d) has to satisfy the following formulas: $1 \leq i \leq d$ and $\frac{n-i}{k} < d \leq \lfloor \frac{n-i}{k-1} \rfloor$. Then, each distance d satisfies $\frac{n}{k+1} < d < \frac{n}{k-1}$. We use same techniques of Algorithm 1 for each d with $\frac{n}{k+1} < d < \frac{n}{k-1}$. Since the number of possible values for d is $O(\frac{n}{k^2})$, we can check that the same character is repeated k times in the split text in $O(\frac{n^2}{k^2})$ time. Therefore, we can obtain the following result:

► **Theorem 8.** *There is an algorithm for locating all k -cadences for given k which uses $O\left(\frac{n^2}{k^2}\right)$ time and $O(n)$ space.*

4.2 Algorithm 4

Now, we will show the following result:

► **Theorem 9.** *For a constant size alphabet, there is an $O\left(\frac{n^2}{k^2 \log n}\right)$ time algorithm for counting all k -cadences for given k . We can also locate these occurrences in $O\left(\frac{n^2}{k^2 \log n} + occ\right)$ time. For a general ordered alphabet, there is an $O\left(\frac{n^2}{\sqrt{k} \sqrt{\log n}}\right)$ time algorithm for counting all k -sub-cadences for given k . We can also locate these occurrences in $O\left(\frac{n^2}{\sqrt{k} \sqrt{\log n}} + occ\right)$ time. These algorithms run in $O(n)$ space.*

Note that when we count all k -sub-cadences, for a constant size alphabet, this algorithm is faster than Algorithm 3. Also, for a general ordered alphabet, this algorithm is faster if k is $o(\sqrt[3]{\log n})$. (This is because if $\frac{n^2}{\sqrt{k} \sqrt{\log n}}$ is less than $\frac{n^2}{k^2}$, then $k\sqrt{k} \leq \sqrt{\log n}$.) When we locate all k -sub-cadences, for a constant size alphabet (resp. for a general ordered alphabet), if $|KC|$ is $o(\frac{n^2}{k^2})$ (resp. $|KC|$ is $o(\frac{n^2}{k^2})$ and k is $o(\sqrt[3]{\log n})$) then this algorithm is faster.

Now we will show how to count all k -cadences of character $c \in \Sigma$. If (i, d) is a k -cadence with character c , then $\delta_c[i] = \delta_c[i+d] = \dots = \delta_c[i+(k-1)d] = 1$, $1 \leq i \leq d$, and $\frac{n-i}{k} < d \leq \lfloor \frac{n-i}{k-1} \rfloor$. Therefore, to calculate k -cadences, we need only the range $[1..d]$ of i for d with $\frac{n-i}{k} < d \leq \lfloor \frac{n-i}{k-1} \rfloor$. For each d with $\frac{n-i}{k} < d \leq \lfloor \frac{n-i}{k-1} \rfloor$, let $Q'_d = \delta_c[1..d] \& \delta_c[d+1..2d] \& \delta_c[3d+1..4d] \& \dots \& \delta_c[(k-1)d+1..kd]$. If $Q'_d[i] = 1$, (i, d) is a k -cadence. Q'_d can be obtained by the following operation: $Q'_d = \delta_c[1..d] \& (\delta_c \ll d)[1..d] \& (\delta_c \ll 2d)[1..d] \& \dots \& (\delta_c \ll (k-1)d)[1..d]$. By using the same techniques of Algorithm 2, we can compute Q'_d in $O(k \frac{d}{\log n})$ time. Hence, for all $\frac{n}{k+1} < d < \frac{n}{k-1}$, we can count all k -cadences of a character in $O(k \frac{d}{\log n} \frac{n}{k^2}) \subseteq O(\log n \frac{n^2}{k^2})$ time.

12:8 Detecting k -(Sub-)Cadences and Equidistant Subsequence Occurrences

For a locating algorithm and for a general ordered alphabet, we can use same techniques of the above section. Then we can locate all k -cadences of a character in $O(\frac{n^2}{k^2 \log n} + occ)$ time. For a general ordered alphabet, all k -cadences can be counted in $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{k^2 \log n}\})$ time. Since $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{k^2 \log n}\})$ is maximized when $k|N_c|^2 = \frac{n^2}{k^2 \log n}$, then $O(\sum_{c \in \Sigma} \min\{k|N_c|^2, \frac{n^2}{k^2 \log n}\}) \subseteq O((\sum_{c \in \Sigma} |N_c|) \frac{n}{\sqrt{k} \sqrt{\log n}}) \subseteq O(\frac{n^2}{\sqrt{k} \sqrt{\log n}})$. Therefore we can count in $O(\frac{n^2}{\sqrt{k} \sqrt{\log n}})$ time. Also, all k -sub-cadences can be located in $O(\frac{n^2}{\sqrt{k} \sqrt{\log n}} + occ)$ time. From the above, we obtain the following result:

► **Theorem 10.** *For a constant size alphabet (resp. for a general ordered alphabet), all k -cadences with given k can be counted in $O(\frac{n^2}{k^2 \log n})$ time (resp. $O(\min\{\frac{n^2}{k^2}, \frac{n^2}{\sqrt{k} \sqrt{\log n}}\})$ time) and $O(n)$ space, and can be located in $O(\frac{n^2}{k^2 \log n} + occ)$ time (resp. $O(\min\{\frac{n^2}{k^2}, \frac{n^2}{\sqrt{k} \sqrt{\log n}} + occ\})$ time) and $O(n)$ space.*

5 Detecting Equidistant Subsequence Pattern

In this section, we consider algorithms for detecting equidistant subsequence pattern.

5.1 Algorithm 5

We use similar techniques as of Algorithm 1. For each distance d with $1 \leq d \leq \lfloor \frac{n-1}{m-1} \rfloor$, we construct text ST_d . After preparing the split text, we can compute ESP using existing substring pattern matching algorithms. Since Knuth-Morris-Pratt algorithm [9] runs in $O(n)$ time for a text of length n , we obtain the following result:

► **Theorem 11.** *There is an algorithm for locating all equidistant subsequence occurrences for given pattern P of length m which uses $O(\frac{n^2}{m})$ time and $O(n)$ space.*

Like KSC , for text $T = a^n$ and pattern $P = a^m$, $|ESP|$ can be $\Omega(\frac{n^2}{m})$. Therefore, when we locate all $(i, d) \in ESP$, this algorithm is optimal in the worst case. In the next subsection, we show a counting algorithm that is efficient when the value of m is small. And we show a locating algorithm that is efficient when the value of m and $|ESP|$ is small.

5.2 Algorithm 6

Now we will show the following results:

► **Theorem 12.** *There is an algorithm for counting all equidistant subsequence occurrences which uses $O(\frac{n^2}{\log n})$ time and $O(\frac{|\Sigma_P|n}{\log n})$ space, where Σ_P is the set of distinct characters in the given pattern P . We can also locate these occurrences in $O(\frac{n^2}{\log n} + occ)$ time and $O(\frac{|\Sigma_P|n}{\log n})$ space.*

First, we construct δ_c for all $c \in \Sigma_P$. For each d with $1 \leq d \leq \lfloor \frac{n-1}{m-1} \rfloor$, let $Q_d'' = \delta_{P[1]} \& (\delta_{P[2]} \ll d) \& (\delta_{P[3]} \ll 2d) \& \dots \& (\delta_{P[m]} \ll (m-1)d)$. If $Q_d''[i] = 1$, (i, d) is an occurrence of equidistant subsequence pattern P . See Figure 3 for a concrete example.

All of the elements of ESP can be counted / located by using a method similar to Algorithm 2 for Q_d'' . After constructing δ_c for all $c \in \Sigma$, all occurrences of equidistant

$$\begin{array}{l}
 T = \text{c a a a c a a b a a b a a b c a b c} \\
 P = \text{a a c c} \\
 \delta_a = 0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 \\
 \delta_c = 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 \\
 \\
 \delta_a = 0 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 \\
 (\delta_a \ll 3) = 1 0 1 1 0 1 1 0 1 1 0 0 1 0 0 0 0 0 \\
 (\delta_c \ll 6) = 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 \\
 (\delta_c \ll 9) = 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 \\
 \hline
 Q''_3 = 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 \end{array}$$

Figure 3 Let $T = \text{caaacaabaabaabcabc}$ and $P = \text{aac}$. $(9, 3)$ is an occurrence of equidistant subsequence pattern with $d = 3$.

subsequence pattern can be counted in $O(\frac{n^2}{\log n})$ time and $O(n)$ space and can be located in $O(\frac{n^2}{\log n} + occ)$ time and $O(n)$ space. Constructing δ_c for all $c \in \Sigma_P$ needs $O(\frac{|\Sigma_P|n}{\log n})$ time and space. Since $\frac{|\Sigma_P|n}{\log n}$ is at most $O(\frac{n^2}{\log n})$, we get Theorem 12.

If m is $o(\log n)$, Algorithm 6 is faster than Algorithm 5 and $O(\frac{|\Sigma_P|n}{\log n}) \subseteq O(n)$. From the above, we obtain the following result:

► **Theorem 13.** *All occurrences of equidistant subsequence pattern can be counted in $O\left(\min\left\{\frac{n^2}{m}, \frac{n^2}{\log n}\right\}\right)$ time and $O(n)$ space and can be located in $O\left(\min\left\{\frac{n^2}{m}, \frac{n^2}{\log n} + occ\right\}\right)$ time and $O(n)$ space.*

6 Detecting Equidistant Subsequence Pattern of Length Three

In this section, we show more efficient algorithms that count all occurrences of an equidistant subsequence pattern for the case where the length of the pattern is three. In addition, we show an algorithm for counting all occurrences of equidistant Abelian subsequence patterns of length three. Since we heavily use the techniques of [5] for 3-sub-cadences, we first show their algorithm for 3-sub-cadences and then generalize it for solving the equidistant subsequence pattern matching problem.

6.1 Counting 3-sub-cadences [5]

Let $a[0..n]$ and $b[0..n]$ be two sequences. The sequence $h[1..2n]$ can be computed by the discrete acyclic convolution $h[z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2}^{x+y=z} a[x]b[y]$. The discrete acyclic convolution can be computed in $O(n \log n)$ time by using the fast Fourier transform. This convolution can be interpreted geometrically as follows: $h[z] = \sum_{(x,y) \in G \cap \mathbb{Z}^2}^{x+y=z} a[x]b[y]$, where G is the square given by $\{(x, y) : 0 \leq x, y \leq n\}$.

Funakoshi and Pape-Lange [5] showed that 3-sub-cadences can be counted by using the discrete acyclic convolution. If (i, d) is a 3-sub-cadence with a character c , then $\delta_c[i] \cdot \delta_c[i+2d] = 1$ and $T[i+d] = c$. Let $h[2z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2}^{x+y=2z} \delta_c[x] \delta_c[y]$, then $h[2z]$ counts how many pairs x and y there are that satisfies $x + y = 2z$ and $T[x] = T[y] = c$ for the index z . Since

12:10 Detecting k -(Sub-)Cadences and Equidistant Subsequence Occurrences

$z + z = 2z$ and $\delta_c[z] \cdot \delta_c[z] = 1$ if $T[z] = 1$, then $h[2z]$ counts one false positive. In addition, $x + y = 2z$ and $\delta_c[x] \cdot \delta_c[y] = 1$ if $x \neq y$, then $h[2z]$ counts twice for same x and y . Let $f[z]$ be the number of all 3-sub-cadences with a character c such that the middle index of 3-sub-cadences is z . $f[z]$ can be computed in $O(n \log n)$ time as follows:

$$f[z] := \begin{cases} \frac{h[2z]-1}{2} & \text{if } T[z] = c, \\ 0 & \text{if } T[z] \neq c. \end{cases}$$

Furthermore, they extended the geometric interpretation of convolution and showed that if G is a triangle with perimeter p , the sequence c can be computed in $O(p \log^2 p)$ time.

6.2 Counting Equidistant Subsequence Patterns of Length Three

Now we show an algorithm for counting all occurrences of equidistant subsequence patterns whose length is three. Let $g[z]$ be the number of all occurrences of the equidistant subsequence pattern such that the middle index of P is z . If $P = \alpha\alpha\alpha$, this problem is equal to the counting all 3-sub-cadences problem. Therefore, $g[z]$ can be computed in $O(n \log n)$ time as follows:

$$g[z] := \begin{cases} \frac{h[2z]-1}{2} & \text{if } T[z] = \alpha \\ 0 & \text{if } T[z] \neq \alpha \end{cases}$$

where $h[2z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2} \delta_\alpha[x] \delta_\alpha[y]$.

If $P = \alpha\beta\alpha$, since the pattern is symmetrical, $g[z]$ can be computed in $O(n \log n)$ time as follows, by using almost the same technique as above:

$$g[z] := \begin{cases} \frac{h[2z]}{2} & \text{if } T[z] = \beta \\ 0 & \text{if } T[z] \neq \beta \end{cases}$$

where $h[2z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2} \delta_\alpha[x] \delta_\alpha[y]$.

However, if $P = \alpha\beta\gamma$, then $h[2z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2} \delta_\alpha[x] \delta_\gamma[y]$ would also include occurrences of the equidistant subsequence pattern $\gamma\beta\alpha$. Thus, in order to compute $g[z]$, we further add the condition $x < y$. By using triangle convolution of [5], $g[z]$ can be computed in $O(n \log^2 n)$ time as follows:

$$g[z] := \begin{cases} h[2z] & \text{if } T[z] = \beta \\ 0 & \text{if } T[z] \neq \beta \end{cases}$$

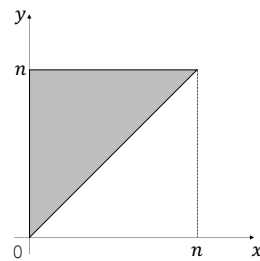
where $h[2z] = \sum_{(x,y) \in G \cap \mathbb{Z}^2} \delta_\alpha[x] \delta_\gamma[y]$, where G is the triangle as in the following Figure 4.

If $P = \alpha\alpha\gamma$ or $P = \alpha\gamma\gamma$, we can compute $g[z]$ by using the same technique as for the case of $P = \alpha\beta\gamma$. Therefore, we get the following result:

► **Theorem 14.** *All occurrences of equidistant subsequence pattern of length three can be counted in $O(n \log^2 n)$ time and $O(n)$ space.*

6.3 Counting Equidistant Abelian Subsequence Patterns of Length Three

Now we show the algorithm for counting all occurrences of equidistant Abelian subsequence pattern whose length is three. In this subsection we consider the case where all of the three characters are distinct, namely, $P = \alpha\beta\gamma$. The other cases can be computed similarly.



■ **Figure 4** The triangle G .

In the previous subsection, we showed that if $P = \alpha\beta\gamma$, then $h[2z] = \sum_{(x,y) \in [0,1,2,\dots,n]^2} \delta_\alpha[x]\delta_\gamma[y]$ includes the occurrences of equidistant subsequence pattern $\gamma\beta\alpha$. Therefore, we can compute all occurrences of equidistant subsequence pattern $\alpha\beta\gamma$, $\gamma\beta\alpha$, $\beta\gamma\alpha$, $\alpha\gamma\beta$, $\gamma\alpha\beta$, and $\beta\alpha\gamma$ by using discrete acyclic convolution for $P = \alpha\beta\gamma$, $P = \beta\gamma\alpha$, and $P = \gamma\alpha\beta$. Hence, we can get following result:

► **Theorem 15.** *All occurrences of equidistant Abelian subsequence pattern of length three can be counted in $O(n \log n)$ time and $O(n)$ space.*

References

- 1 Amihood Amir, Alberto Apostolico, Travis Gagie, and Gad M. Landau. String cadences. *Theoretical Computer Science*, 698:4–8, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo).
- 2 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
- 3 Maxime Crochemore and Dominique Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, 1991.
- 4 Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016.
- 5 Mitsuru Funakoshi and Julian Pape-Lange. Non-rectangular convolutions and (sub-)cadences with three elements. In *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPICs*, pages 30:1–30:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 6 Zvi Galil and Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.
- 7 J. Gardelle. Cadences. *Mathématiques et Sciences humaines*, 9:31–38, 1964.
- 8 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.
- 9 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
- 10 M. Lothaire. *Combinatorics on Words*. Cambridge Mathematical Library. Cambridge University Press, 1997.
- 11 Phillip Matier and Andrew Ross. Did Schwarzenegger drop 4-letter bomb in veto? San Francisco Chronicle, 2009. URL: <http://www.sfgate.com/cgi-bin/article.cgi?f=/c/a/2009/10/28/MNBN1ABKB8.DTL>.
- 12 Doron Witztum, Eliyahu Rips, and Yoav Rosenberg. Equidistant letter sequences in the book of genesis. *Statistical Science*, 9(3):429–438, 1994.

FM-Index Reveals the Reverse Suffix Array

Arnab Ganguly

Department of Computer Science, University of Wisconsin - Whitewater, WI, USA
gangulya@uww.edu

Daniel Gibney

Department of Computer Science, University of Central Florida, Orlando, FL, USA
Daniel.Gibney@ucf.edu

Sahar Hooshmand

Department of Computer Science, University of Central Florida, Orlando, FL, USA
sahar@cs.ucf.edu

M. Oğuzhan Külekci

Informatics Institute, Istanbul Technical University, Turkey
kulekci@itu.edu.tr

Sharma V. Thankachan

Department of Computer Science, University of Central Florida, Orlando, FL, USA
sharma.thankachan@ucf.edu

Abstract

Given a text $T[1, n]$ over an alphabet Σ of size σ , the suffix array of T stores the lexicographic order of the suffixes of T . The suffix array needs $\Theta(n \log n)$ bits of space compared to the $n \log \sigma$ bits needed to store T itself. A major breakthrough [FM-Index, FOCS'00] in the last two decades has been encoding the suffix array in near-optimal number of bits ($\approx \log \sigma$ bits per character). One can decode a suffix array value using the FM-Index in $\log^{O(1)} n$ time.

We study an extension of the problem in which we have to also decode the suffix array values of the reverse text. This problem has numerous applications such as in approximate pattern matching [Lam et al., BIBM' 09]. Known approaches maintain the FM-Index of both the forward and the reverse text which drives up the space occupancy to $2n \log \sigma$ bits (plus lower order terms). This brings in the natural question of whether we can decode the suffix array values of both the forward and the reverse text, but by using $n \log \sigma$ bits (plus lower order terms). We answer this question positively, and show that given the FM-Index of the forward text, we can decode the suffix array value of the reverse text in near logarithmic average time. Additionally, our experimental results are competitive when compared to the standard approach of maintaining the FM-Index for both the forward and the reverse text. We believe that applications that require both the forward and reverse text will benefit from our approach.

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

Keywords and phrases Data Structures, Suffix Trees, String Algorithms, Compression, Burrows-Wheeler transform, FM-Index

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.13

Supplementary Material <https://github.com/oguzhankulekci/reverseSA>

Funding This research is supported in part by the U.S. National Science Foundation under CCF-1703489 and by the MGA-2019-42224 project of the Research Fund of Istanbul Technical University, Turkey.



© Arnab Ganguly, Daniel Gibney, Sahar Hooshmand, M. Oğuzhan Külekci, and Sharma V. Thankachan;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 13; pp. 13:1–13:14

Leibniz International Proceedings in Informatics



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

1 Introduction

The suffix tree is arguably the central data structure in Stringology. Briefly speaking, the suffix tree (ST) of a text $T[1, n]$ over an alphabet $\Sigma = [\sigma] \cup \{\$\}$ is a compact trie over all suffixes, where $\$$ is the unique terminal symbol. Its linear time construction [10, 25, 32, 33] and efficient tree-navigational features make it a versatile tool in the design of various string matching algorithms. As a practical alternative, suffix arrays were introduced later. Probably the greatest beneficiary of these data structures is bioinformatics; in fact, it is safe to say that the field would not have been the same without them [1, 31]. We refer to Gusfield’s book [18] for an exhaustive list of algorithms aided by suffix trees and suffix arrays.

In the era of data deluge, a negative aspect of suffix trees and suffix arrays is their memory footprint of $\Theta(n)$ words or $\Theta(n \log n)$ bits. In comparison, the text can be encoded in just $n \log \sigma$ bits, or even lower space using compression techniques. To put this into perspective, the suffix tree takes around 15 bytes per character and the suffix array takes around 4 bytes per character for human genome, where σ is 4. Bridging the complexity gap between data-space and index-space has been a challenging task. The advent of succinct data structures [19] and compressed text indexing, where the goal is to have a data structure in space close to the information theoretical minimum, presented us with new indexes like the FM-Index by Ferragina and Manzini [12] and the Compressed Suffix Array (CSA) by Grossi and Vitter [17]; these indexes encapsulate the functionalities of suffix array in near-optimal number of bits (w.r.t. statistical entropy). While the CSA achieved this goal via the structural properties of suffix trees/arrays, FM-Index relied on the Burrows-Wheeler Transformation (BWT) of the text [7]. Moreover, the FM-index is a self-index, i.e., any portion of the original text can be extracted from the index. These remarkable breakthroughs saved orders of magnitude of space in practice and eventually became the foundations of more advanced indexes [6, 11, 26, 27, 29, 30]. They are the backbone of many widely used bioinformatics tools like the BWA [22], SOAP2 [24], Bowtie [21], etc.

Motivated by the fact that two human genomes differ in hardly 0.1% of their positions, Belazzougui et al. [5] introduced the concept of Relative Compressed Indexes or Reusable-Indexes, where the objective is to leverage the fact that a full text index (say an FM-index) of a string T is already available, while indexing a “closely similar” string T' . They showed that the FM-index of T' can be encoded in $O(\delta)$ extra space (in words), assuming that the FM-index of T is accessible. Here, δ denotes the edit distance between the $\$$ ’s of T and T' . We study a special, but useful instance of this problem, in which T' is the reverse of T .

1.1 Relative Compression of the Reverse Suffix Array

Let $T[1, n] = t_1 t_2 \dots t_{n-1} \$$ be a string over the alphabet $\Sigma = [\sigma] \cup \{\$\}$, where the character $\$$ appears exactly once. The reverse of T is the string $\overleftarrow{T} = t_{n-1} t_{n-2} \dots t_1 \$$. We use the following lexicographic order: $\$ < 1 < 2 < \dots < \sigma$. We use $T[i, j]$ (resp., $\overleftarrow{T}[i, j]$) to denote the substring of T (resp., \overleftarrow{T}) from position i to j .

The suffix array $\text{SA}[1, n]$ stores the starting positions of the lexicographically arranged suffixes, i.e., $\text{SA}[i] = j$ if the i^{th} lexicographically smallest suffix is $T[j, n]$. The inverse suffix array $\text{ISA}[1, n]$ is defined as: $\text{ISA}[j] = i$ if and only if $\text{SA}[i] = j$. Thus, the suffix array and its inverse can be stored in $\Theta(n)$ words, i.e., $\Theta(n \log n)$ bits. The BWT of T is an array $\text{BWT}[1, n]$ such that $\text{BWT}[i] = T[\text{SA}[i] - 1]$, where $T[0] = T[n]$. An FM-Index is essentially a combination of the BWT (with rank-select functionality support via a wavelet tree [16]) and a sampled (inverse) suffix array. Likewise, we can define the suffix array and the inverse suffix array for the reverse text \overleftarrow{T} , denoted by $\overleftarrow{\text{SA}}[1, n]$ and $\overleftarrow{\text{ISA}}[1, n]$, respectively.

► **Problem 1.** *Can we decode $\overleftarrow{SA}[\cdot]$ and $\overleftarrow{ISA}[\cdot]$ values efficiently using the FM-index of T ?*

1.2 Motivation and Related Work

We observe that when the application mandates performing search both in forward and reverse directions and we already have an index on the forward text, it is possible to calculate the SA (or ISA) values of the reversed text on the fly efficiently by using the forward index, which eliminates the overhead of reverse suffix array. Some text processing applications, particularly in computational biology, are a good example of this case. For instance, the *read mapping* problem [23] in bioinformatics aims to match a given read onto a reference genome. Due to the DNA sequencing technology used, a read may originate from a forward strand as well as the reverse strand of the DNA helix, and the direction is unknown at the time of mapping. Thus, while the read can be aligned to the reference in its original form, its reverse complement should also be considered as it could be sampled from the reverse strand. One way to cope with this problem is to create two indexes [22], one for the forward and the other for the reverse strand mapping, which obviously doubles the space. However, if the forward index can be used to search in the reverse text, the space can be reduced significantly.

The practical applicability of our study addresses this case by showing that we can compute the $\overleftarrow{SA}[i]$ and $\overleftarrow{ISA}[i]$ elements of the reverse text for any possible i , by solely using the FM-Index constructed over the forward text. In a wider sense, any bioinformatics application that makes use of a FM-Index while performing a pattern search on a target sequence, can benefit from our solution to search on the reverse strand of the target without any need of extra space. For example, Lam et al. [20] use both the forward and backward BWT to find matches with k -mismatches allowed; our results eliminate the requirement of the latter, thereby roughly halving the space. It is noteworthy that other relevant elements of the reverse text, such as computing the longest common prefix of two suffixes and \overleftarrow{BWT} -entry can be generated from the $\overleftarrow{SA}[i]$ and $\overleftarrow{ISA}[i]$, becomes *efficiently* computable on the fly.

From a theoretical perspective, one can argue that pattern matching on the reverse text is equivalent to matching the reverse of the pattern in the forward text. However, there are applications, where one needs to find the range of suffixes in the suffix tree/array of the reverse text that are prefixed by a pattern. A typical example is the classic solution for approximate pattern matching with one error, which uses the suffix tree/array of the text as well as that of the reverse of the text, along with an orthogonal range searching data structure [2]. A similar approach is followed in most of the compressed indexes based on LZ-compression, although the forward/reverse suffixes arrays/trees are sparse [3]. Another use is in the (relative) compressed indexing of a collection of sequences that are highly similar. Here, two full text indexes corresponding to the reference sequence and its reverse are maintained. Other sequences are indexed in relative LZ compressed space w.r.t. the reference sequence [9]. On a related note, Ohlebusch et al. [28] provided a procedure to compute the BWT of the reverse text considering the strong correlation between T and \overleftarrow{T} . They compute the reverse BWT from the forward BWT, but in their process to compute the k^{th} entry of the BWT, one has to decode all entries from 1 to $k - 1$. Their technique can also partially fill the reverse suffix array during this computation, where additional effort is required to calculate the missing elements of \overleftarrow{SA} . Our approach on the other hand can directly compute any BWT entry for the reverse text. In another work, Belazzougui et al. [4] showed how to represent the bi-directional BWT (i.e., forward and reverse BWT) so that one can perform efficient navigation of the suffix tree in the forward and backward direction; however, to search in the forward direction, their representation again needs space roughly twice that of the FM-Index.

1.3 Our Results

The following is our main contribution in this paper.

► **Theorem 2.** *Assuming the availability of the FM-Index of $T[1, n]$ (where BWT is stored in the form of a wavelet tree), we can compute the suffix array value $r = \overleftarrow{\text{SA}}[i]$ for any given i (resp. inverse suffix array $i = \overleftarrow{\text{ISA}}[r]$ for any given r) of the reversed text \overleftarrow{T} in time $O(h \cdot \tau_{\text{WT}} + \tau_{\text{SA}})$. Here,*

- h is the length of the shortest unique substring that starts at position r in \overleftarrow{T} ,
- τ_{WT} is the time to support standard wavelet tree operations on the BWT, and
- τ_{SA} is the time to decode a suffix array (or inverse suffix array) value using FM-Index.

In the most common implementations of the FM-Index, $\tau_{\text{WT}} = O(\log \sigma)$ and $\tau_{\text{SA}} = O(\log^{1+\epsilon} n)$, where $\epsilon > 0$ is arbitrarily small. On average h can be expected to be $O(\log_{\sigma} n)$ when the text is assumed to be independently and identically distributed over the alphabet Σ [8]. Thus, we get the following corollary.

► **Corollary 3.** *Given the FM-Index of T (where BWT is stored in the form of a wavelet tree), we can decode a suffix array value or inverse suffix array value of the reversed text in $O(\log^{1+\epsilon} n)$ expected time, where $\epsilon > 0$ is arbitrarily small.*

We complement the above results with experiments. Since Corollary 3 may not hold on sequences with skewed symbol distributions such as natural language texts, we also include such cases in the experiments to analyze the performance. The experiments show that our results are competitive when compared to the standard approach of maintaining the FM-Index for both the forward text and the reverse text.

2 Burrows-Wheeler Transform and FM-Index

Given an array $A[1, m]$ over an alphabet Σ of size σ , by using the wavelet tree data structure of size $m \log \sigma + o(m)$ bits, the following queries can be answered in $O(\log \sigma)$ time [12, 14, 16]:

- $A[i]$,
- $\text{rank}_A(i, j, x)$ = the number of occurrences of x in $A[i, j]$,
- $\text{select}_A(i, j, k, x)$ = the k^{th} occurrence of x in $A[i, j]$,
- $\text{quantile}_A(i, j, k)$ = the k^{th} smallest character in $A[i, j]$,
- $\text{rangeCount}_A(i, j, x, y)$ = the number of positions $k \in [i, j]$ satisfying $x \leq A[k] \leq y$

Burrows and Wheeler [7] introduced a reversible transformation of the text, known as the Burrows-Wheeler Transform (BWT). Let T_x be the circular suffix starting at position x , i.e., $T_1 = T$ and $T_x = T[x, n] \circ T[1, x - 1]$, where $x > 1$ and \circ denotes concatenation. Then, the BWT of T is obtained as follows: first create a conceptual matrix M , such that each row of M corresponds to a unique circular suffix, and then lexicographically sort all rows. Thus the i th row in M is given by $T_{\text{SA}[i]}$. The BWT is the last column L of M , i.e., $\text{BWT}[i] = T_{\text{SA}[i]}[n] = T[\text{SA}[i] - 1]$, where $T[0] = T[n] = \$$. The main component of the FM-Index is the last-to-first column mapping (in short, LF mapping). For any $i \in [1, n]$, $\text{LF}(i)$ is the row j in the matrix M where $\text{BWT}[i]$ appears as the first character in $T_{\text{SA}[j]}$. Specifically, $\text{LF}(i) = \text{ISA}[\text{SA}[i] - 1]$, where $\text{SA}[0] = \text{SA}[n]$.

To compute $\text{LF}(i)$, we store a wavelet tree over $\text{BWT}[1, n]$ in $n \log \sigma + o(n \log \sigma)$ bits. Let the number of occurrences of symbol $i \in \Sigma \setminus \{\$\}$ in T be f_i . We store another array $C[1, \sigma]$ such that $C[i]$ is the number of characters in T that are lexicographically smaller than i . Specifically, $C[1] = 1$, and $C[i] = 1 + \sum_{j < i} f_j$ when $i > 1$. As a convention, we denote

$C[\$] = 0$ and $C[\sigma + 1] = n$. Using these, we can compute $\text{LF}(i)$ mapping in $O(\log \sigma)$ time as $\text{LF}(i) = C[\text{BWT}[i]] + \text{rank}(1, i, \text{BWT}[i])$. We can decode $\text{SA}[i]$ in $O(\log^{1+\epsilon} n)$ time by using LF mapping and by maintaining a sampled-suffix array, which occupies $o(n \log \sigma)$ bits in total. The idea is to explicitly store $(i, \text{SA}[i])$ pairs for all $\text{SA}[i] \in \{1, 1 + \Delta, 1 + 2\Delta, \dots\}$, where $\Delta = \lceil \log_\sigma n \log^\epsilon n \rceil$. The space needed is $O(\frac{n}{\Delta} \log n) = o(n \log \sigma)$ bits. Then, $\text{SA}[i]$ can be obtained directly if the value has been explicitly stored; otherwise, it can be computed via at most Δ number of LF mapping operations in time $O(\Delta \cdot \log \sigma) = O(\log^{1+\epsilon} n)$. We can also decode $\text{ISA}[\cdot]$ using the sampled array in $O(\log^{1+\epsilon} n)$ time.

3 The Method

A substring $T[a, b]$ of T is unique if a is the only occurrence of $T[a, b]$ in T . Note that the unique substring starting at a position a is always defined (since T ends in $\$$). Moreover the reverse of $T[a, b]$ is also unique in \overleftarrow{T} , and it ends at the position $(n - a + 1)$ in \overleftarrow{T} .

3.1 Decoding $\overleftarrow{\text{SA}}[i]$ for a given i

Our algorithm hinges on the following main lemma.

► **Lemma 4.** *Given the FM-Index of T (where the BWT is equipped with a wavelet tree), we can compute the shortest unique substring $\overleftarrow{\text{SUS}}$ in \overleftarrow{T} starting at $\overleftarrow{\text{SA}}[i]$ in $O(h \cdot \tau_{\text{WT}})$ time, where $h = |\overleftarrow{\text{SUS}}|$. We can then compute $r = \overleftarrow{\text{SA}}[i]$ in $O(\tau_{\text{SA}})$ time.*

Proof of Lemma 4. Our task is to compute $r = \overleftarrow{\text{SA}}[i]$ for some i , where h is the length of the shortest unique substring $\overleftarrow{\text{SUS}} = \overleftarrow{T}[r] \circ \overleftarrow{T}[r + 1] \circ \dots \circ \overleftarrow{T}[r + h - 1]$ of \overleftarrow{T} starting at position r . Let the range $[\alpha_k, \beta_k]$ be such that for any $j \in [\alpha_k, \beta_k]$ the suffix $T[\text{SA}[j], n]$ starts with the string $\overleftarrow{T}[r + k - 1] \circ \overleftarrow{T}[r + k - 2] \circ \dots \circ \overleftarrow{T}[r]$. Moreover, let q_k be such that $\overleftarrow{T}[r + k]$ is the q_k th smallest character in $\text{BWT}[\alpha_k, \beta_k]$.

Our idea is to successively compute the ranges $[\alpha_1, \beta_1], [\alpha_2, \beta_2], \dots$ and q_1, q_2, \dots until we get a range $[\alpha_h, \beta_h]$ that contains exactly one suffix, i.e., $\alpha_h = \beta_h$. At each step, we are going to decode the characters $\overleftarrow{T}[r], \overleftarrow{T}[r + 1], \dots, \overleftarrow{T}[r + h - 1]$. Clearly, $\overleftarrow{\text{SUS}} = \overleftarrow{T}[r] \circ \overleftarrow{T}[r + 1] \circ \dots \circ \overleftarrow{T}[r + h - 1]$, and the starting position of SUS in T is $\text{SA}[\alpha_h]$. Therefore,

$$r = n - (\text{SA}[\alpha_h] + h - 1) = n - \text{SA}[\alpha_h] - h + 1$$

We now present the details, starting with the following simple observation. The i^{th} lexicographic suffix of \overleftarrow{T} starts with the same character as the i^{th} lexicographic suffix of T . Therefore, $\overleftarrow{T}[r] = T[\text{SA}[i]]$, which is essentially the i^{th} smallest character in $\text{BWT}[1, n]$, and is given by $\text{quantile}(1, n, i)$. Now, we find the range $[\alpha_1, \beta_1]$ in constant time using the array \mathbf{C} and $\overleftarrow{T}[r]$. The next step is to decode the character $\overleftarrow{T}[r + 1]$, and compute the range $[\alpha_2, \beta_2]$. Note that $\overleftarrow{T}[r, n]$ is the $(i - \alpha_r + 1)^{\text{th}}$ lexicographically smallest suffix that starts with $\overleftarrow{T}[r]$. In other words, $\overleftarrow{T}[r + 1]$ is exactly the $(i - \alpha_r + 1)^{\text{th}}$ smallest character in $\text{BWT}[\alpha_1, \beta_1]$. Therefore $q_1 = (i - \alpha_r + 1)$.

The next steps are to decode the character $\overleftarrow{T}[r + 1]$ and compute $[\alpha_2, \beta_2]$, then decode $\overleftarrow{T}[r + 2]$ and compute $[\alpha_3, \beta_3]$, and so on. To do so, we rely on the following recursions. From the definition, $\overleftarrow{T}[r + k] = \text{quantile}(\alpha_k, \beta_k, q_k)$ for any $k \geq 1$. We now show how to compute $[\alpha_{k+1}, \beta_{k+1}]$. Let a be the smallest index $\geq \alpha_k$ and let b be the largest index $\leq \beta_k$, such that $\text{BWT}[a] = \text{BWT}[b] = \overleftarrow{T}[r + k]$.

$$\alpha_{k+1} = \text{LF}(a) = C[\overleftarrow{T}[r + k]] + \text{rank}(1, \alpha_k - 1, \overleftarrow{T}[r + k]) + 1$$

$$\beta_{k+1} = \text{LF}(b) = C[\overleftarrow{T}[r + k]] + \text{rank}(1, \beta_k, \overleftarrow{T}[r + k])$$

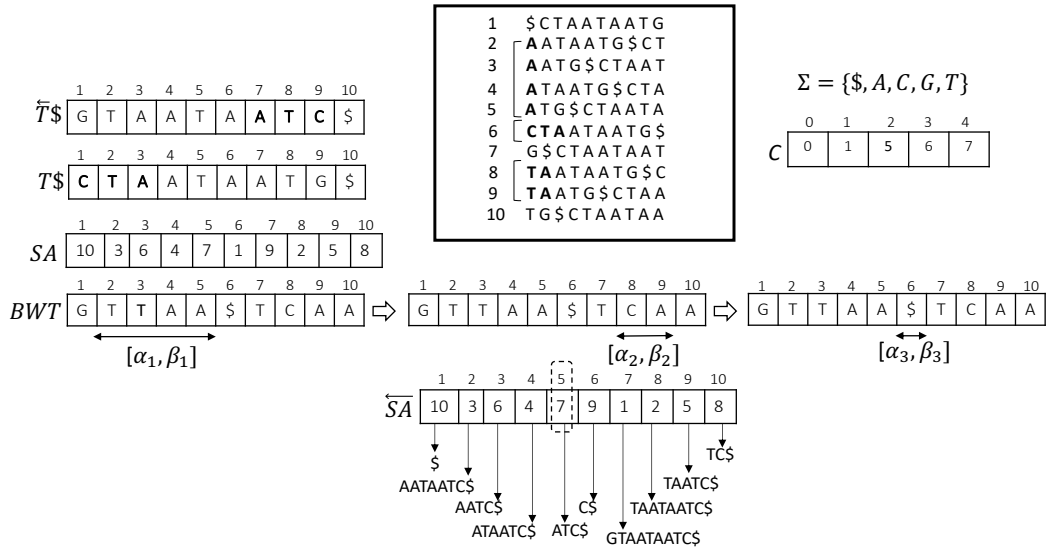


Figure 1 Computing $\overleftarrow{SA}[5] = 7$ via Lemma 4. Here $\overleftarrow{SUS} = ATC$, the suffix range $[\alpha_1, \beta_1]$ of A is $[2, 5]$, the suffix range $[\alpha_2, \beta_2]$ of TA is $[8, 9]$, and the suffix range $[\alpha_3, \beta_3]$ of CTA is $[6, 6]$.

Finally, $q_{k+1} = (q_k - d)$, where d is the number of characters in $BWT[\alpha_k, \beta_k]$ that are lexicographically smaller than $\overleftarrow{T}[r+k]$, which can be computed via a `rangeCount` query.

We repeat this process until we reach $[\alpha_h, \beta_h]$, where $\alpha_h = \beta_h$. This takes $O(h \cdot \tau_{WT})$ time. Then r is decoded in additional $O(\tau_{SA})$ time. This completes the proof of Lemma 4. ◀

Algorithm 1 for computing $\overleftarrow{SA}[i]$.

```

1: procedure COMPUTE  $\overleftarrow{SA}[i]$ 
2:   if ( $i = 1$ ) then return  $n$ 
3:    $\alpha \leftarrow 1, \beta \leftarrow n, q \leftarrow i, h \leftarrow 0$ 
4:   while ( $\alpha < \beta$ ) do
5:      $c \leftarrow \text{quantile}(\alpha, \beta, q)$ 
6:     if ( $q \neq \$$ ) then  $q \leftarrow q - \text{rangeCount}(\alpha, \beta, 1, c - 1)$ 
7:     if ( $\alpha > 1$ ) then  $\alpha \leftarrow C[c] + \text{rank}(\alpha - 1, c) + 1$ 
8:     else  $\alpha \leftarrow C[c] + 1$ 
9:      $\beta \leftarrow C[c] + \text{rank}(\beta, c)$ 
10:     $h \leftarrow h + 1$ 
11:  return  $n - SA[\alpha] - h + 1$ 

```

3.2 Decoding $\overleftarrow{ISA}[r]$ for a given r

To compute $\overleftarrow{ISA}[r]$ for some position r , the main intuition is as follows. Let γ_1 be the number of entries in $BWT[1, n]$ that are lexicographically smaller than $\overleftarrow{T}[r]$. Then, $\overleftarrow{ISA}[r] \geq \gamma_1 = C[\overleftarrow{T}[r]]$. Now consider the range $[\alpha_1, \beta_1]$ such that for any $j \in [\alpha_1, \beta_1]$, the suffix $T[SA[j], n]$ starts with $\overleftarrow{T}[r]$. Let γ_2 be the number of entries in $BWT[\alpha_1, \beta_1]$ that are lexicographically smaller than $\overleftarrow{T}[r+1]$. Then, $\overleftarrow{ISA}[r] \geq \gamma_1 + \gamma_2$. Now, consider the range $[\alpha_2, \beta_2]$ such that for

any $j \in [\alpha_2, \beta_2]$, the suffix $T[\text{SA}[j], n]$ starts with $\overleftarrow{\text{T}}[r+1] \circ \overleftarrow{\text{T}}[r]$. Compute γ_3 . We repeat the process until we reach the range $[\alpha_h, \beta_h]$ such that $\alpha_h = \beta_h$. Clearly, the unique suffix $T[\text{SA}[\alpha_h], n]$ starts with $\text{SUS} = \overleftarrow{\text{T}}[r+h-1] \circ \overleftarrow{\text{T}}[r+h-2] \cdots \circ \overleftarrow{\text{T}}[r]$. Since SUS is the smallest unique prefix of $\overleftarrow{\text{T}}[r, n]$, $\sum_{s \leq h} \gamma_s$ is the number of suffixes in $\overleftarrow{\text{T}}$ that are lexicographically smaller than $\overleftarrow{\text{T}}[r, n]$. Thus, $\overleftarrow{\text{ISA}}[r] = 1 + \sum_{s \leq h} \gamma_s$. To compute the $\gamma_1, \gamma_2, \dots, \gamma_h$, we use `rangeCount` operation. Computing the range $[\alpha_k, \beta_k]$ from $[\alpha_{k-1}, \beta_{k-1}]$ is achieved using the array C and rank operation, as in proof of Lemma 4.

The algorithm has $h = |\text{SUS}|$ rounds. Each round comprises of a constant number of wavelet tree operations, and accesses to the C array. Additionally, in the k^{th} round, we have to decode the character $\overleftarrow{\text{T}}[r+k-1]$. To do this we use the following technique. If $r = n$, then $\overleftarrow{\text{T}}[r] = \$$; so, assume otherwise. Note that $\overleftarrow{\text{T}}[r] = T[n-r+1]$; thus, $\overleftarrow{\text{T}}[r] = \text{BWT}[\text{ISA}[n-r+2]]$ is found in $O(\tau_{\text{SA}} + \tau_{\text{WT}})$ time. Now, $\overleftarrow{\text{T}}[r+1], \overleftarrow{\text{T}}[r+2], \dots$ are given by $\text{BWT}[\text{LF}(\text{ISA}[n-r+2])], \text{BWT}[\text{LF}(\text{LF}(\text{ISA}[n-r+2]))], \dots$, in $O(\tau_{\text{WT}})$ time for each k . Hence, the time taken to compute $\overleftarrow{\text{ISA}}[r]$ is $O(\tau_{\text{SA}} + h \cdot \tau_{\text{WT}})$. We have the following lemma.

► **Lemma 5.** *Given the FM-Index of T (where the BWT is equipped with a wavelet tree), we can compute $i = \overleftarrow{\text{ISA}}[r]$ in $O(\tau_{\text{SA}} + h \cdot \tau_{\text{WT}})$ time, where h is the length of the shortest unique substring of $\overleftarrow{\text{T}}$ that starts at r .*

From Lemma 4 and Lemma 5, Theorem 2 is immediate. We outline Lemmas 4 and 5 formally in Algorithms 1 and 2 respectively. ◀

■ **Algorithm 2** for computing $\overleftarrow{\text{ISA}}[r]$.

```

1: procedure COMPUTE  $\overleftarrow{\text{ISA}}[r]$ 
2:   if ( $r = n$ ) then return 1
3:    $i \leftarrow \text{ISA}[n-r+2], c \leftarrow \text{BWT}[i]$ 
4:    $\alpha \leftarrow 1, \beta \leftarrow n, \gamma \leftarrow \text{C}[c]$ 
5:   while ( $\alpha < \beta$ ) do
6:     if ( $\alpha > 1$ ) then  $\alpha \leftarrow \text{C}[c] + \text{rank}(\alpha-1, c) + 1$ 
7:     else  $\alpha \leftarrow \text{C}[c] + 1$ 
8:      $\beta \leftarrow \text{C}[c] + \text{rank}(\beta, c)$ 
9:      $i \leftarrow \text{LF}(i), c \leftarrow \text{BWT}[i]$ 
10:    if ( $c \neq \$$ ) then
11:       $\gamma \leftarrow \gamma + \text{rangeCount}(\alpha, \beta, 1, c-1)$ 
12:    return  $(1 + \gamma)$ 

```

4 Experimental Results

The proposed algorithms eliminate the necessity to separately maintain the SA and ISA of the reverse text $\overleftarrow{\text{T}}$ by computing $\overleftarrow{\text{SA}}[i]$ and $\overleftarrow{\text{ISA}}[i]$ directly from the FM-index of T . The natural question is how the performance of the introduced method is compared with the regular access via the FM-index that could be built on $\overleftarrow{\text{T}}$. We have implemented the proposed

algorithms¹ by using the `sdsl-lite` framework² [15] and performed some tests on 50MB `dna`, `english`, `proteins`, `sources`, and `dblp.xml` files from `Pizza&Chili`³ corpus to analyze the practical performance of the introduced methods.

For each file, we have created the FM-index and measured the elapsed time of our algorithm to retrieve $\overleftarrow{SA}[i]$ / $\overleftarrow{ISA}[i]$ for 100K randomly selected distinct i positions. We benchmark that elapsed time against a regular SA / ISA access on the FM-index created over \overleftarrow{T} , assuming that both forward and reverse FM-indices apply the same sampling strategy with the same sampling frequency,

All operations in Algorithms 1 and 2, namely the `quantile`, `rangeCount`, `rank` queries, `backwards_search`, and `LF – mappings`, are achieved in logarithmic time. The execution times of the introduced algorithms are directly proportional to the number of times they are repeated, which is determined by the length of the matching `SUS`. Hence, on positions where the `SUS` is extremely long, the execution time will increase. It makes sense to define a threshold such that the proposed methods stay compatible in practice. Therefore, for those positions that have a `SUS` longer than this threshold, it may be preferred to pre-compute and maintain their $\overleftarrow{SA}/\overleftarrow{ISA}$ values offline. We suggest to set this threshold to the `SA/ISA` sampling frequency used in the FM-index construction. While selecting the random positions in the experiments, those with `SUS` lengths longer than the threshold are excluded. Table 1 lists the average `SUS` length of the 100K randomly selected positions with this restriction on each file for each `SA` sampling frequency. The percentage of all positions that has a shorter `SUS` than the corresponding sampling frequencies, are also presented.

■ **Table 1** The average `SUS` lengths of the selected positions on each file per each sampling frequency, and the percentage of all positions in that file, which has `SUS` length less than or equal to that sampling frequency.

Sampling Frequency:	Average SUS Length			Positions with SUS length \leq Sampling Frequency		
	32	64	128	32	64	128
<code>dblp.xml</code>	19.04	29.34	41.37	58.57%	81.18%	96.81%
<code>dna</code>	15.22	16.46	17.11	96.91%	99.45%	99.89%
<code>english</code>	12.89	14.17	17.48	97.81%	99.05%	99.64%
<code>protein</code>	8.48	11.04	17.23	84.58%	87.80%	91.11%
<code>sources</code>	16.13	21.63	28.10	86.11%	93.36%	96.37%

The experiments were run on an iMac using MacOS 10.13.16 and equipped with 16GB memory and 3.23 GHz Intel Core i5 processor. The software was compiled with the clang LLVM compiler with full optimization (`-O3`). During the experiments, we considered the sampling factors of 32, 64 and 128, along with both `text-ordered` and `suffix-ordered` sampling strategies [13].

The shape of the wavelet tree (WT) representing the BWT may also be an important factor in practical performance to achieve the queries we use in our algorithms. The `lex_ordered(i, j, c)` function of the `sdsl-lite` platform, which returns the number of symbols lexicographically smaller/greater than `c` in the (i, j) interval of a wavelet tree WT, is used in

¹ The implementation is available online at <https://github.com/oguzhankulekci/reverseSA>.

² The `sdsl-lite` framework is available online at <https://github.com/simongog/sdsl-lite>.

³ <http://pizzachili.dcc.uchile.cl/index.html>.

the implementation. This function requires the WT to support lexicographical ordering⁴, where Hu-Tucker and balanced WTs are the only options, and thus, included in the experiments. The Huffman-shaped WT is not used as it does not support the lexicographical ordering.

Figure 2 represents the comparison of the average elapsed time to retrieve the $\overleftarrow{SA}[i]$ with Algorithm 1 versus the regular access via FM-index of \overleftarrow{T} on `english`, `dna`, and `protein` files⁵, whose alphabet sizes are respectively 239, 16, and 27. Total time to retrieve the $\overleftarrow{SA}[i]$ via the Algorithm 1 is equal to the sum of the SUS extraction and SA access on forward FM-index. It is observed that Hu-Tucker shaped WT provides better running time than the balanced shaped and, `text-order` based sampling is superior to the `suffix-order` based. The average SA access time on both forward and reverse directions are approximately equal. Therefore, the expected latency in the proposed technique depends on the SUS-detection phase. As shown in Table 1, due to the limitation applied in the selection process, the average SUS length is increasing as the sampling frequency gets larger. This reflects on the SUS extraction cost in Figure 2, where the SUS extraction time expands proportionally by the increment of the average SUS length in each data type.

■ **Table 2** The ratios representing the overall execution time of the Algorithms 1 and 2 divided by the regular SA and ISA access on different sampling ratios and strategies.

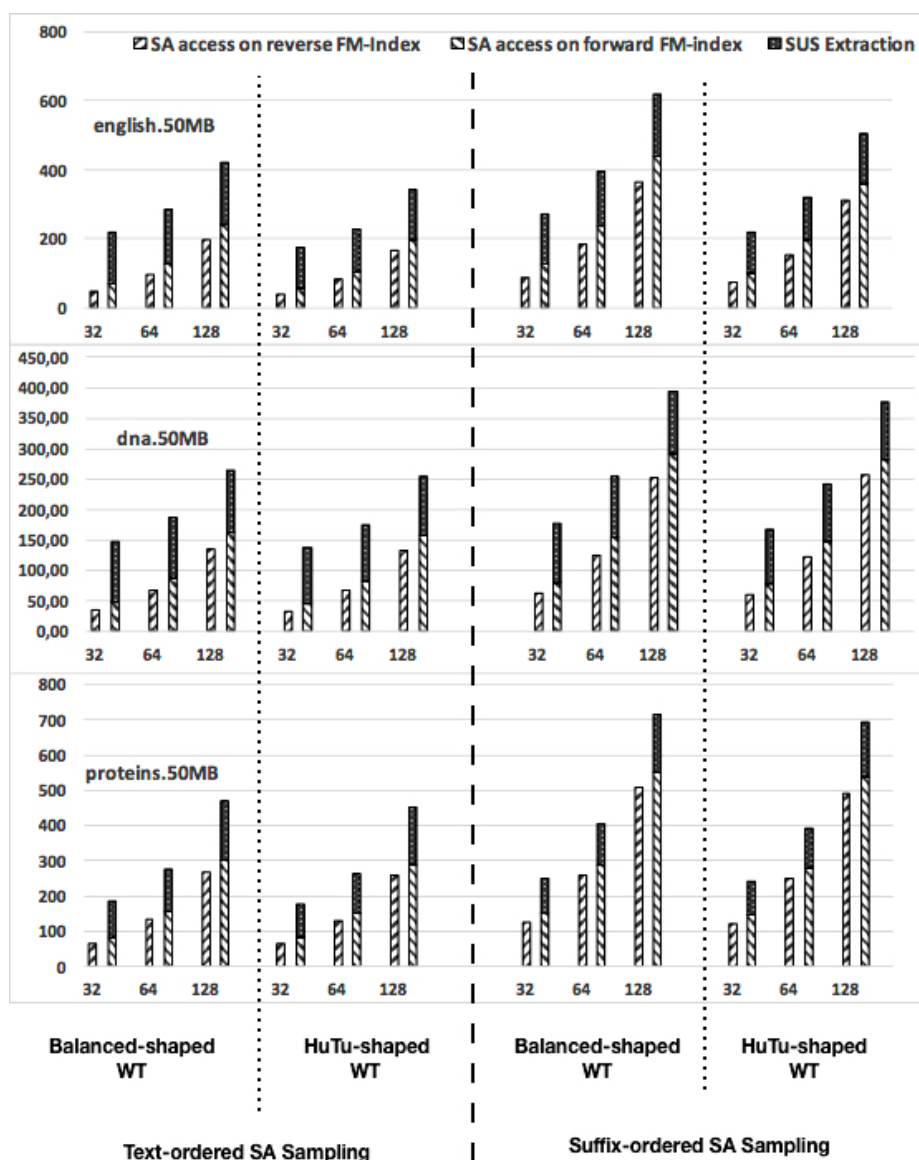
Sampling Strategy: Sampling Frequency:	\overleftarrow{SA} Benchmark (Algorithm 1)						\overleftarrow{ISA} Benchmark (Algorithm 2)					
	Text-ordered			Suffix-ordered			Text-ordered			Suffix-ordered		
	32	64	128	32	64	128	32	64	128	32	64	128
<code>dblp.xml</code>	7.0	5.2	3.5	5.0	3.4	2.5	9.5	6.2	3.9	9.4	6.0	3.8
dna	4.2	2.7	1.9	2.7	2.0	1.5	5.1	3.5	2.1	5.5	3.2	2.2
<code>english</code>	4.3	3.0	2.0	2.7	2.0	1.6	5.3	3.3	2.4	5.3	3.2	2.4
protein	2.7	2.1	1.7	1.9	1.6	1.4	3.4	2.5	2.0	3.4	2.5	2.1
<code>sources</code>	4.8	3.4	2.5	3.2	2.4	1.8	6.0	4.1	2.9	6.0	4.1	2.9

With the aim of having a better understanding about the running time of Algorithms 1 and 2, the elapsed time to access a random $\overleftarrow{SA}[i]$ (and $\overleftarrow{ISA}[i]$), is divided by the time required to achieve these queries with a regular FM-index constructed over \overleftarrow{T} . Table 2 lists these ratios. Since the proposed methods can retrieve the $\overleftarrow{SA}[i]$ and $\overleftarrow{ISA}[i]$ values without the FM-index on \overleftarrow{T} , a slow-down is actually expected in the general paradigm of time-memory trade-off. On `dna` sequences, Algorithm 1 is only 2.7, 2.0, and 1.5 times slower for corresponding sampling frequencies, while using Suffix-ordered method. On `protein` sequences, the ratios are even better to be 1.9, 1.6, and 1.4 respectively. We observed the worst results on `dblp.xml` file, which is highly repetitive, and thus, the SUS extraction times have been observed to be significantly longer. It's reasonable to particularly underline the performance of our proposed algorithm on biological sequences. Since, text operations on reverse direction are expected to be a more common demand in terms of computational biology applications. The proposed solution, especially the \overleftarrow{SA} calculation, competes better in suffix-based SA sampling strategy. This favours the practical applicability of our theoretical results since suffix-based approach is the default choice in practice due to its space conservation. The \overleftarrow{ISA} computations with Algorithm 2 are generally observed to be ≈ 1.5 times worse than the \overleftarrow{SA} computations, which is due to the fact that retrieving ISA is nearly two times faster than accessing SA on an FM-index with equal SA and ISA sampling ratios⁶.

⁴ Indicated as `lex_ordered` in <http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf>

⁵ The `sources` and `dblp` results are not shown in Figure 2 to save space.

⁶ As also considered in `sdsl-lite` framework by setting the default ISA sampling frequency to two times of the SA sampling frequency.



■ **Figure 2** Experimental analysis to compare the speed of the proposed method and the regular SA access on the FM-index constructed over the reversed text. Y-axis represents the elapsed time in microseconds and X-axis indicate the sampling frequencies. For the representation of the BWT, both “balanced” and “Hu-Tucker” shaped wavelet trees that supports lexicographical ordering (which is required by the methods we use in the algorithms), are considered.

The practical performance of the Algorithms 1 and 2 depends heavily on the length of the corresponding SUS. Short SUSs are expected to be more common, where the method will execute fast. On the other hand, even much less frequently observed longer SUS cases degrade the overall average timings. Thus, for a deeper investigation, the diffraction in Table 3 lists the percentages of the positions on which the elapsed time with the proposed algorithm is “X” times of the regular access on the FM-index constructed over \overleftarrow{T} .

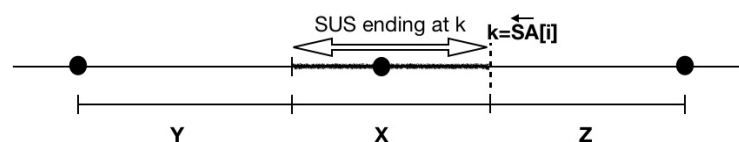
■ **Table 3** Percentage of the positions on which the elapsed time with the proposed algorithms are " X " times of the regular suffix array access on the FM-index constructed over \overleftarrow{T} . For instance, on **protein** sequence, Algorithm 1 answered faster than the regular FM-index over \overleftarrow{T} on 21.88% of the queries when the sampling factor is 32 with a suffix-ordered strategy. On 40.76% of the cases, the speed is slower than the regular access, but not more than two times. Similarly, the speed is between two and three times of the SA access with FM-index constructed over \overleftarrow{T} on 21.17% of cases.

	dna			english			protein		
	32	64	128	32	64	128	32	64	128
<1x	0.00	0.03	13.93	0.00	0.30	13.75	0.06	12.92	23.89
1x-2x	0.02	24.69	41.86	0.50	19.42	40.97	25.23	36.87	40.36
2x-3x	12.30	38.71	36.84	10.52	34.57	34.24	35.94	38.83	27.69
3x-4x	31.36	30.13	5.59	27.27	32.14	6.66	30.81	7.04	4.50
>4x	56.32	6.43	1.77	61.70	13.56	4.38	7.96	4.33	3.55
\overleftarrow{SA} results with text-ordered sampling strategy									
<1x	0.03	20.03	41.50	1.37	19.28	38.31	21.88	37.95	45.77
1x-2x	36.46	42.96	34.16	35.97	41.83	33.26	40.76	34.19	31.24
2x-3x	33.49	20.70	14.27	31.76	21.04	15.83	21.17	16.29	14.00
3x-4x	16.22	9.13	5.84	16.10	9.87	7.22	9.28	6.90	5.46
>4x	13.81	7.18	4.24	14.80	7.98	5.38	6.91	4.67	3.53
\overleftarrow{SA} results with suffix-ordered sampling strategy									
<1x	0.00	0.00	5.95	0.00	0.03	5.59	0.00	0.96	10.93
1x-2x	0.00	3.03	42.67	0.02	10.90	36.45	2.14	20.43	36.48
2x-3x	0.11	31.02	41.83	1.57	33.41	38.12	17.93	34.89	36.45
3x-4x	12.10	36.81	6.80	12.63	33.14	12.71	28.05	27.85	9.40
>4x	87.78	29.14	2.75	85.79	22.53	7.14	51.88	15.86	6.75
\overleftarrow{ISA} results with text-ordered sampling strategy									
<1x	0.00	0.00	3.23	0.00	0.03	5.39	0.00	2.26	17.35
1x-2x	0.00	6.99	38.88	0.03	9.46	35.95	4.42	33.26	37.49
2x-3x	0.10	36.14	42.38	2.28	32.59	38.52	32.67	37.99	33.09
3x-4x	10.87	38.00	12.69	14.63	35.07	12.46	36.96	18.20	5.12
>4x	89.03	18.88	2.81	83.06	22.86	7.67	25.95	8.30	6.96
\overleftarrow{ISA} results with suffix-ordered sampling strategy									

Actually, Algorithm 1 already includes a regular SA access in itself (line 11). So, it's quite surprising to observe that there are cases where Algorithm 1 executes faster than the regular access. Such cases appear since the access time to suffix arrays on FM-indices differs in forward and reverse directions. Figure 3, depicts this by sketching the number of accessed symbols with Algorithm 1 and with a regular SA access on reverse FM-index. Algorithm 1 starts with extracting the SUS which ends at k in forward direction (or starts at k in reverse direction). Once X symbols long SUS is extracted, the algorithm calls the regular SA access on the FM-index of T , which tells the location of this SUS on T . This access requires backwards traversal of Y symbols on T via the FM-index, where Y is the distance to the closest sampled point on the left of the SUS. The result of this access is then used to compute the exact value of k on \overleftarrow{T} . On the other hand, when an FM-index of \overleftarrow{T}

13:12 FM-Index Reveals the Reverse Suffix Array

(reverse FM-index) is available, Z symbols are subject to backwards traversal (as backwards on \overleftarrow{T} means left-to-right movement on the Figure 3). When the summation of “SA call on forward FM-index” and the “SUS extraction cost” is smaller than the SA access on the reverse FM-index ($cost(X) + cost(Y) < cost(Z)$), such interesting cases may occur.



■ **Figure 3** Sketching the number of backwards traversal steps with Algorithm 1 ($X + Y$) versus FM-index of \overleftarrow{T} (Z). Dark circles represent the sampled positions in both directions.

5 Conclusion

We have presented two algorithms to compute the $\overleftarrow{SA}[i]$ and $\overrightarrow{SA}[i]$ values by using the FM-index of the forward text T . Experiencing slowdown in such space preserving approaches is expected, and hence, we conducted experiments to observe this effect in practice. The benchmark results stated in Table 2 reveals that the \overleftarrow{SA} and \overrightarrow{SA} calculations are respectively 2-3 times and 3-4 times slower on the average when compared to a regular FM-index constructed over \overleftarrow{T} with suffix-ordered sampling strategy. Particularly on biological sequences, such as the `dna` and `protein` files, the ratios even improve better supporting their usage in practice. Although the execution time of the introduced algorithms increase on sections of long repeats of the input data (as the SUS extraction is the key of the proposed methods), the methods respond quite fast in most cases as shown in Table 3 since the majority of the SUS lengths are centric around shorter lengths. Another interesting application of the proposed methods might be in fully-parallel constructing the BWT of the reverse text from the forward BWT, which has been mentioned in the previous study of Ohlebusch et al. [28] with a solution by computing $\overleftarrow{BWT}[k]$ under assumption that $\overleftarrow{BWT}[1], \overleftarrow{BWT}[2], \dots, \overleftarrow{BWT}[k-1]$ are already available, and k iterates from 1 to n . They observed that some positions are independent of the previous ones, which provide an opportunity in parallelizing the execution. However, the level of parallelization here is bounded by the number of such independent start points. Contrary to that, our solution in computing $\overleftarrow{SA}[i]$ does not introduce any prerequisites for any i , and thus, is fully parallelizable that is scalable up to n processors.

References

- 1 Srinivas Aluru. *Handbook of Computational Molecular Biology*. Chapman & Hall/CRC, 2005.
- 2 Amihood Amir, Dmitry Kesselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text indexing and dictionary matching with one error. *Journal of Algorithms*, 37(2):309–325, 2000. doi:10.1006/jagm.2000.1104.
- 3 Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Stronger Lempel–Ziv based compressed text indexing. *Algorithmica*, 62(1-2):54–101, 2012. doi:10.1007/s00453-010-9443-8.
- 4 Djamel Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Algorithms - ESA 2013 - 21st Annual European Symposium*, pages 133–144, 2013. doi:10.1007/978-3-642-40450-4_12.
- 5 Djamel Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. Relative FM-indexes. In *String Processing and Information Retrieval - 21st International Symposium*,

- SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, pages 52–64, 2014. doi:10.1007/978-3-319-11918-2_6.
- 6 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 225–235, 2012. doi:10.1007/978-3-642-33122-0_18.
 - 7 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (now part of Hewlett-Packard, Palo Alto, CA), 1994.
 - 8 Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992. doi:10.1137/0221005.
 - 9 Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theor. Comput. Sci.*, 532:14–30, 2014. doi:10.1016/j.tcs.2013.07.024.
 - 10 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, 1997. doi:10.1109/SFCS.1997.646102.
 - 11 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009. An extended abstract appeared in *FOCS 2005* under the title “Structuring labeled trees for optimal succinctness, and beyond”. doi:10.1145/1613676.1613680.
 - 12 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. An extended abstract appeared in *FOCS 2000* under the title “Opportunistic Data Structures with Applications”. doi:10.1145/1082036.1082039.
 - 13 Paolo Ferragina, Jouni Sirén, and Rossano Venturini. Distribution-aware compressed full-text indexes. *Algorithmica*, 67(4):529–546, 2013.
 - 14 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval, SPIRE '09*, pages 1–6, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03784-9_1.
 - 15 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.
 - 16 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual Symposium on Discrete Algorithms ACM-SIAM, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003.
 - 17 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. An extended abstract appeared in *STOC 2000*. doi:10.1137/S0097539702402354.
 - 18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
 - 19 Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988. AAI8918056.
 - 20 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36, 2009. doi:10.1109/BIBM.2009.42.
 - 21 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.
 - 22 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

- 23 Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.
- 24 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
- 25 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
- 26 Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. doi:10.1093/bioinformatics/btx067.
- 27 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007. doi:10.1145/1216370.1216372.
- 28 Enno Ohlebusch, Timo Beller, and Mohamed I Abouelhoda. Computing the Burrows–Wheeler transform of a string and its reverse in parallel. *Journal of Discrete Algorithms*, 25:21–33, 2014.
- 29 Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 95–106, 2011. doi:10.1145/1989284.1989300.
- 30 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. doi:10.1016/j.ic.2011.03.007.
- 31 Wing-Kin Sung. *Algorithms in Bioinformatics: A Practical Introduction*. Chapman & Hall/CRC, 1st edition, 2009.
- 32 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 33 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.

On Indeterminate Strings Matching

Paweł Gawrychowski

Institute of Computer Science, University of Wrocław, Poland

Samah Ghazawi

Department of Computer Science, University of Haifa, Israel

Gad M. Landau

Department of Computer Science, University of Haifa, Israel

Department of Computer Science and Engineering,

NYU Tandon School of Engineering, Brooklyn, NY, USA

Abstract

Given two indeterminate equal-length strings p and t with a set of characters per position in both strings, we obtain a determinate string p_w from p and a determinate string t_w from t by choosing one character per position. Then, we say that p and t match when p_w and t_w match for some choice of the characters. While the most standard notion of a match for determinate strings is that they are simply identical, in certain applications it is more appropriate to use other definitions, with the prime examples being parameterized matching, order-preserving matching, and the recently introduced Cartesian tree matching. We provide a systematic study of the complexity of string matching for indeterminate equal-length strings, for different notions of matching. We use n to denote the length of both strings, and r to be an upper-bound on the number of uncertain characters per position. First, we provide the first polynomial time algorithm for the Cartesian tree version that runs in deterministic $\mathcal{O}(n \log^2 n)$ and expected $\mathcal{O}(n \log n \log \log n)$ time using $\mathcal{O}(n \log n)$ space, for constant r . Second, we establish NP-hardness of the order-preserving version for $r = 2$, thus solving a question explicitly stated by Henriques et al. [CPM 2018], who showed hardness for $r = 3$. Third, we establish NP-hardness of the parameterized version for $r = 2$. As both parameterized and order-preserving indeterminate matching reduce to the standard determinate matching for $r = 1$, this provides a complete classification for these three variants.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string matching, indeterminate strings, Cartesian trees, order-preserving matching, parameterized matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.14

Funding *Samah Ghazawi*: Partially supported by the Israel Science Foundation (ISF) grant 1475/18.

Gad M. Landau: Partially supported by the Israel Science Foundation (ISF) grant 1475/18, and the United States-Israel Binational Science Foundation (BSF) grant No. 2018141.

1 Introduction

String matching, in the sense of comparing two equal-length strings, is one of the fundamental problems in computer science with multiple practical applications. While exact matching is trivial to solve in optimal linear time by comparing the strings character-by-character, for many of the applications it seems more appropriate to work with some kind of approximate matching. Prime examples include string matching with swaps [2], parameterized string matching [6], string matching with gaps [9], jumbled string matching [10], string matching with don't cares [29], and edit distance [32]. In all of such problems, one needs to first precisely define when do two strings match.

Parameterized matching is a classical notion motivated by finding identical sections of code [3, 4, 5, 6, 19, 34]. Formally, two strings p and t of length n are a parameterized match when for every $i, j \in \{1, \dots, n\}$, $p[i] = p[j]$ iff $t[i] = t[j]$. This is denoted by $p \sim_{=} t$.



© Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 14; pp. 14:1–14:14

Leibniz International Proceedings in Informatics



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

14:2 On Indeterminate Strings Matching

Order-preserving matching is a more recent but already well-studied notion motivated by stock price analysis and musical melody matching [11, 16, 17, 25, 26]. Formally, two strings p and t of length n are a order-preserving match when for every $i, j \in \{1, \dots, n\}$, $p[i] \leq p[j]$ iff $t[i] \leq t[j]$. This is denoted by $p \sim_{\leq} t$.

Very recently, a different notion called Cartesian tree matching has been proposed [28]. The Cartesian tree of a given string p ($CT(p)$), first defined in [31], is constructed according to the following rules:

- If p is an empty string, $CT(p)$ is an empty tree.
- If $p[1..n]$ is not empty and $p[i]$ is the leftmost minimum value in p , $CT(p)$ is the tree with $p[i]$ being the root, $CT(p[1..i-1])$ the left subtree, and $CT(p[i+1..n])$ the right subtree.

Even though the most well-known applications of Cartesian trees are probably in designing space-efficient structures for finding the minimum in a range, they can be also used to compare strings. Similarly to order-preserving matching, this notion is motivated by applications concerned with time-series data such as stock price analysis, and has gained considerable attention during the last year [7, 18, 30]. Formally, two strings p and t of equal length n are a Cartesian tree match when their Cartesian trees $CT(p)$ and $CT(t)$ are identical, i.e. $CT(p)$ and $CT(t)$ have the same shape while the labels on the nodes may differ. This is denoted by $p \sim_C t$.

We consider the complexity of string matching for indeterminate strings defined as follows.

► **Definition 1.** *An indeterminate string is a sequence of sets of characters $p[1]p[2]\dots p[n]$, where $p[i] \subseteq \mathbb{N}$. Each position is specified by writing $p[i] = a_1| \dots | a_r$, such that $a_\ell \in \mathbb{N}$, which means that we can choose $p[i]$ to be any a_ℓ .*

Indeterminate strings were studied earlier, among others, covering problems for indeterminate strings [1, 14] and indeterminate strings in graph theory [20, 12, 27]. Indeterminate string matching was investigated lately from different angles [8, 13, 15, 23, 22, 24]. It provides a convenient formalism for compactly capturing situations in which there are some uncertainties concerning characters at some positions. Indeed, an indeterminate string p of length n describes r^n determinate strings. We write \tilde{p} to denote the set of all such strings, and p_w when referring to a single determinate string described by p .

First, we consider the complexity of Cartesian tree matching for indeterminate strings defined as follows.

Problem: CARTESIAN TREE MATCHING OF INDETERMINATE STRINGS (CTMIS)
Input: Two indeterminate strings p and t of length n with up to r of uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w Cartesian tree matches t_w ?

A naive solution to the CTMIS would be to apply the solution of [28] to each $t_w \in \tilde{t}$ and $p_w \in \tilde{p}$ in $\mathcal{O}(n^2 r^n)$ time. In Section 2 we provide the first polynomial algorithm for this problem that works in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space, assuming that r is constant. Additionally, in the Word RAM model of computation we further improve the time complexity to expected $\mathcal{O}(n \log n \log \log n)$.

► **Example 2.** Consider the following indeterminate strings:

$$p = (2|4|7, 2|5|6, 1|4|8, 4|7|8, 3|10|16)$$

$$t = (2|7|10, 5|20|31, 10|17|25, 0|9|11, 1|8|18).$$



■ **Figure 1** The Cartesian trees of $p_w = (7, 2, 8, 4, 16)$ and $t_w = (10, 5, 17, 9, 18)$.

$p_w = (7, 2, 8, 4, 16)$ and $t_w = (10, 5, 17, 9, 18)$ define the same Cartesian tree, see Figure 1. Therefore, we say that $p \sim_C t$. Note that p and t define other matching or non-matching Cartesian trees.

Second, we consider the complexity of order-preserving matching for indeterminate strings defined as follows.

Problem: ORDER-PRESERVING MATCHING OF INDETERMINATE STRINGS (OPMIS)
Input: Two indeterminate strings p and t of length n with up to r uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w order-preserving matches t_w ?

Henriques et al. [21] proved that OPMIS is NP-hard for $r = 3$. As for $r = 1$ there is a simple linear-time algorithm, this left $r = 2$ as the only open case (CPM version of the paper [21] claims a polynomial time algorithm for this case, but this has been clarified in the arXiv version [13]). In Section 4 we provide a different reduction that establishes NP-hardness of OPMIS already for $r = 2$, thus fully resolving the complexity of this problem and answering an open question explicitly stated by Costa et al [13]. In contrast with the previous work, our reduction exploits the order between elements instead of just their equality, and is more involved.

Third, we consider the complexity of parameterized matching for indeterminate strings defined as follows.

Problem: PARAMETERIZED MATCHING OF INDETERMINATE STRINGS (PMIS)
Input: Two indeterminate strings p and t of length n with up to r uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w parameterized matches t_w ?

NP-hardness proof by Henriques et al. [21] implicitly shows hardness of PMIS for $r = 3$. This, again, leaves $r = 2$ as the only open case. In Section 5 we provide a reduction that establishes NP-hardness of PMIS for $r = 2$.

2 CTMIS in $\mathcal{O}(n^3)$ Time and $\mathcal{O}(n^2)$ Space

In this section, we describe a warm-up solution for the CTMIS problem. The input is two equal-length indeterminate strings p and t with two uncertain characters per position, and the output is whether $p \sim_C t$ or not. The solution can be generalized to any constant value of r in a straightforward manner. We will assume that both p and t consists of distinct values, which can be always ensured by an appropriate perturbation.

14:4 On Indeterminate Strings Matching

First, note that for each index i , we have $p[i] = a_i|a'_i$ and $t[i] = b_i|b'_i$, hence each i defines a set consisting 4 pairs $\{(a_i, b_i), (a_i, b'_i), (a'_i, b_i), (a'_i, b'_i)\}$ (called thresholds) denoted by $\text{THRESHOLDS}(i)$. The main idea of the algorithm is to determine for each index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$:

1. for which indices k we have $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i , respectively.
2. for which indices j we have $p[i, j] \sim_C t[i, j]$ with the roots x_i and y_i , respectively.

Consider an interval $[k, i]$, the reasoning for an interval $[i, j]$ is similar. We have $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i iff there exists an index ℓ and a threshold $(x_\ell, y_\ell) \in \text{THRESHOLDS}(\ell)$ where $k \leq \ell \leq i-1$, $x_i < x_\ell$ and $y_i < y_\ell$ such that $p[k, \ell] \sim_C t[k, \ell]$ and $p[\ell, i-1] \sim_C t[\ell, i-1]$.

We process all possible intervals $[k, i]$ and $[i, j]$ in an increasing order of their lengths using dynamic programming. For each index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$ we compute the answer for all left intervals $[k, i-1]$ and all right intervals $[i+1, j]$, see Figure 2. We define two types of states and associate a boolean value with each of them as follows:

Left states $L_{k,i}(x_i, y_i) = \text{true}$ iff $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i , respectively.

Right states $R_{i,j}(x_i, y_i) = \text{true}$ iff $p[i, j] \sim_C t[i, j]$ with the roots x_i and y_i , respectively.

$$\begin{array}{ccc}
 \text{left Cartesian subtree } CT(p[k, i-1]) & & \text{right Cartesian subtree } CT(p[i+1, j]) \\
 \hline
 p = (a_1|a'_1, \dots, a_k|a'_k, \dots, a_{i-1}|a'_{i-1}, a_i|a'_i, a_{i+1}|a'_{i+1}, \dots, a_j|a'_j, \dots, a_n|a'_n) \\
 \hline
 \text{left Cartesian subtree } CT(t[k, i-1]) & & \text{right Cartesian subtree } CT(t[i+1, j]) \\
 \hline
 t = (b_1|b'_1, \dots, b_k|b'_k, \dots, b_{i-1}|b'_{i-1}, b_i|b'_i, b_{i+1}|b'_{i+1}, \dots, b_j|b'_j, \dots, b_n|b'_n)
 \end{array}$$

■ **Figure 2** An interval $[k, j]$ of the strings p and t with the root at index i , defining left and right Cartesian subtrees.

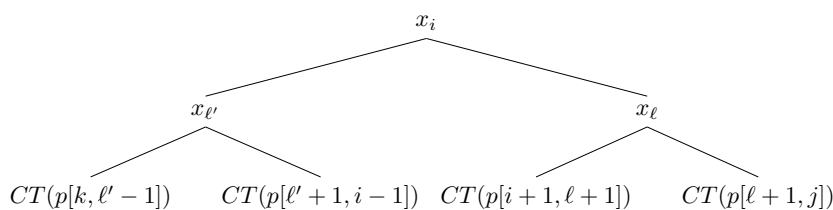
► **Example 3.** Let $p = (4|7, 2|6, 1|8, 3|20, 10|16)$ and $t = (2|10, 20|31, 10|17, 0|11, 8|18)$. Considering index 3 as a possible root for the Cartesian tree in both strings, the thresholds defined by index 3 are $(x_3, y_3) \in \{(1, 10), (8, 10), (1, 17), (8, 17)\}$. The right states are $R_{3,4}(x_3, y_3)$ and $R_{3,5}(x_3, y_3)$. The left states are $L_{2,3}(x_3, y_3)$ and $L_{1,3}(x_3, y_3)$. Some of their corresponding boolean values are as follows:

1. $R_{3,4}(1, 10) = \text{true}$ for $p[3, 4] = (1, 3)$ and $t[3, 4] = (10, 11)$.
2. $R_{3,4}(8, 10) = \text{true}$ for $p[3, 4] = (8, 20)$ and $t[3, 4] = (10, 11)$.
3. $L_{2,3}(1, 10) = \text{true}$ for $p[2, 3] = (6, 1)$ and $t[2, 3] = (31, 10)$.
4. $L_{1,3}(1, 10) = \text{true}$ for $p[1, 3] = (4, 6, 1)$ and $t[2, 3] = (2, 31, 10)$.

From the definition of a Cartesian tree we directly obtain the following proposition illustrated in Figure 3.

► **Proposition 4.**

- (a) $R_{i,j}(x_i, y_i) = \text{true}$ iff $\exists \ell \in [i+1, j]$ such that $R_{\ell,j}(x_\ell, y_\ell) = \text{true}$ and $L_{i+1,\ell}(x_\ell, y_\ell) = \text{true}$ where $x_\ell > x_i$ and $y_\ell > y_i$.
- (b) $L_{k,i}(x_i, y_i) = \text{true}$ iff $\exists \ell' \in [k, i-1]$ such that $R_{\ell',i-1}(x_{\ell'}, y_{\ell'}) = \text{true}$ and $L_{k,\ell'}(x_{\ell'}, y_{\ell'}) = \text{true}$ where $x_{\ell'} > x_i$ and $y_{\ell'} > y_i$.



■ **Figure 3** The Cartesian tree $CT(p[k, j])$ with the root at index i . Note that, the Cartesian tree $CT(t[k, j])$ is identical to the Cartesian tree above with the proper values $y_i, y_{l'}$ and y_l on the nodes, and with the proper subtrees $CT(t[k, l' - 1]), CT(t[l' + 1, i - 1]), CT(t[i + 1, l + 1])$ and $CT(t[l + 1, j])$. Moreover, in both Cartesian trees, the left Cartesian subtrees correspond to the left state $L_{k,i}(x_i, y_i)$, while the right Cartesian subtrees correspond to the right state $R_{i,j}(x_i, y_i)$.

Recall that we apply dynamic programming in an increasing order of the lengths of the intervals. Therefore, the states $R_{\ell,j}(x_\ell, y_\ell)$ and $L_{i+1,\ell}(x_\ell, y_\ell)$ from Proposition 4(a) are computed before the state $R_{i,j}(x_i, y_i)$. Similarly, the states $R_{\ell',i-1}(x_{\ell'}, y_{\ell'})$ and $L_{k,\ell'}(x_{\ell'}, y_{\ell'})$ are computed before the state $L_{k,i}(x_i, y_i)$. Therefore, for every interval we can simply consider all relevant ℓ and ℓ' , access their corresponding states, and update the answer. Finally, after having processed all the intervals, we conclude that $p \sim_C t$ iff there exists an index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$ such that $L_{1,i}(x_i, y_i) = \text{true}$ and $R_{i,n}(x_i, y_i) = \text{true}$.

► **Example 5.** Let $p = (4|7, 2|6, 1|8, 3|20, 10|16)$ and $t = (2|10, 20|31, 10|17, 0|11, 8|18)$ as in the previous example above. We have $L_{1,3}(1, 10) = \text{true}$ for $p[1, 3] = (4, 6, 1)$ and $t[2, 3] = (2, 31, 10)$, and $R_{3,5}(1, 10) = \text{true}$ for $p[3, 5] = (1, 3, 16)$ and $t[3, 5] = (10, 11, 18)$. Hence, $p \sim_C t$ with the roots 1 and 10 respectively.

Time complexity. For each state $L_{k,i}(x_i, y_i)$ and $R_{i,j}(x_i, y_i)$ we consider $\mathcal{O}(n)$ relevant indices ℓ and ℓ' , respectively. Each such index is processed in constant time, thus the overall time complexity is $\mathcal{O}(n^3)$. The space complexity is bounded by the number of states processed in the dynamic programming, which is $\mathcal{O}(n^2)$.

3 CTMIS in $\mathcal{O}(n \log^2 n)$ Time and $\mathcal{O}(n \log n)$ Space

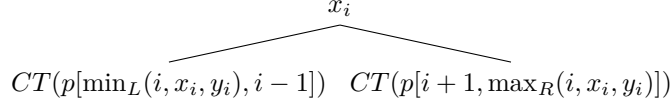
In this section we present an efficient solution for the CTMIS problem that builds on the slower algorithm presented in the previous section.

The input is two equal-length indeterminate strings p and t with 2 uncertain characters per position, and the output is whether $p \sim_C t$, or not. The solution can be generalized to any constant value of r in a straightforward manner. The main idea of the algorithm is to find, for each index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$, the largest matching Cartesian trees with the root in both trees being x_i and y_i at index i , respectively. As in the previous algorithm, we consider each index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$ separately. However, now instead of computing the answer for all intervals $[k, i]$ and $[i, j]$ we use the following definition.

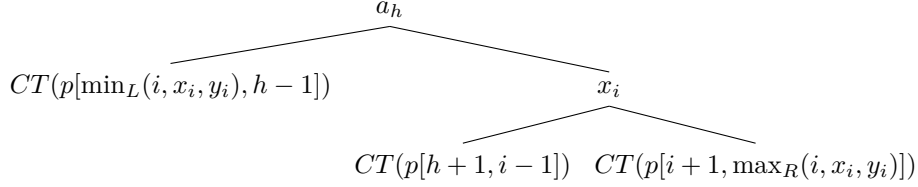
► **Definition 6.** For an index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$:

- $\min_L(i, x_i, y_i)$ denotes the smallest index such that $p[\min_L(i, x_i, y_i), i] \sim_C t[\min_L(i, x_i, y_i), i]$,
- $\max_R(i, x_i, y_i)$ denotes the largest index such that $p[i, \max_R(i, x_i, y_i)] \sim_C t[i, \max_R(i, x_i, y_i)]$, with the root in both trees being x_i and y_i at index i , respectively.

(I)



(II)



■ **Figure 4** Consider the strings:

$$\begin{aligned}
 p &= (a_1 | a'_1, \dots, a_{\min_L(i, x_i, y_i)} | a'_{\min_L(i, x_i, y_i)}, \dots, a_h | a'_h, \dots, a_i | a'_i, \dots, a_{\max_R(i, x_i, y_i)} | a'_{\max_R(i, x_i, y_i)}, \dots, a_n | a'_n) \\
 t &= (b_1 | b'_1, \dots, b_{\min_L(i, x_i, y_i)} | b'_{\min_L(i, x_i, y_i)}, \dots, b_h | b'_h, \dots, b_i | b'_i, \dots, b_{\max_R(i, x_i, y_i)} | b'_{\max_R(i, x_i, y_i)}, \dots, b_n | b'_n)
 \end{aligned}$$

assuming $a_h < x_i < a'_h$, the figure illustrates (I) the Cartesian tree of the substring $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ with x_i as a root when choosing a'_h at index h , and (II) the Cartesian tree of the substring $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ with a_h as the root after changing a'_h to a_h at index h . Note that, assuming $b_h < y_i < b'_h$, the Cartesian trees of $t[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ are identical to the Cartesian trees in (I) and (II) above with the proper roots and the proper Cartesian subtrees.

Computing $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ fully describes the situation, as the above definition together with the definition of a Cartesian tree matching directly imply the following:

- $p[\ell, i] \sim_C t[\ell, i]$ iff $\min_L(i, x_i, y_i) \leq \ell \leq i$.
- $p[i, r] \sim_C t[i, r]$ iff $i \leq r \leq \max_R(i, x_i, y_i)$.

We also note that $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)] \sim_C t[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ due to a Cartesian tree with the root in both trees being x_i and y_i at index i , respectively. Consequently, $p \sim_C t$ iff there exists an index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$ such that $\min_L(i, x_i, y_i) = 1$ and $\max_R(i, x_i, y_i) = n$. Thus, in the remaining part of this section we focus on efficiently computing the values of \min_L and \max_R .

Our algorithm is based on the following observation. Consider an index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$, and assume that $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ have been already computed. Then, the following holds:

1. for any index $h \in [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\max_R(i, x_i, y_i)$ is a potential candidate for $\max_R(h, x_h, y_h)$.
2. for any index $h \in [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\min_L(i, x_i, y_i)$ is a potential candidate for $\min_L(h, x_h, y_h)$.

Each index h and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ might be considered for several indices i and thresholds $(x_i, y_i) \in \text{THRESHOLDS}(i)$ in the above statement, hence we might have several potential candidates for $\min_L(h, x_h, y_h)$ and $\max_R(h, x_h, y_h)$. By the definition of a Cartesian tree, one of these potential candidates corresponds to the sought $\min_L(h, x_h, y_h)$ and $\max_R(h, x_h, y_h)$ as defined above if they are not equal to h . See Figure 4.

The high-level description of the algorithm is as follows. Please see Algorithm 1 for the pseudocode. We iterate over all indices i and thresholds $(x_i, y_i) \in \text{THRESHOLDS}(i)$ in a specific order that will be precisely defined later. For each index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$ we aim to:

Step 1 Compute efficiently the indices $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ (See Definition 6 above).

Step 2 Add for all indices $h \in [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\max_R(i, x_i, y_i)$ as a potential candidate $\max_R(h, x_h, y_h)$. Add for all indices $h \in [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\min_L(i, x_i, y_i)$ as a potential candidate $\min_L(h, x_h, y_h)$.

We need to ensure that, for any index i and a threshold $(x_i, y_i) \in \text{THRESHOLDS}(i)$, and any index h and a threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_h < x_i$ and $y_h < y_i$, $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ are already computed when we are considering threshold $(x_h, y_h) \in \text{THRESHOLDS}(h)$. This will be guaranteed by the algorithm as explained below.

The algorithm considers all indices i and thresholds $(x_i, y_i) \in \text{THRESHOLDS}(i)$ in the reverse lexicographical order, that is, the decreasing order of x_i and, if there is a tie, the decreasing order of y_i . Before we explain how to implement **Step 1** and **Step 2** efficiently, we define the necessary data structures. We maintain a balanced binary search tree T_y on the values of y_i , and identify y_i with its corresponding node of T_y . In each node u of T_y we have its associated secondary trees $T_{\min}(u)$ and $T_{\max}(u)$. Each $T_{\min}(u)$ and $T_{\max}(u)$ stores a collection of intervals $[\ell, r]$. The update adds a new interval $[\ell, r]$ to the collection. The query in $T_{\min}(u)$ for i finds the smallest ℓ such that $[\ell, r]$ containing i belongs to the collection, while the query in $T_{\max}(u)$ finds the largest r . By symmetry, it is enough to explain how to implement $T_{\min}(u)$. We maintain the following invariant: there are no two intervals $[\ell, r]$ and $[\ell', r']$ such that $[\ell, r] \subseteq [\ell', r']$. Clearly, such $[\ell, r]$ is not an answer to any query. Note that this implies that if we sort all the remaining intervals $[\ell_1, r_1], [\ell_2, r_2], \dots, [\ell_s, r_s]$ so that $\ell_1 < \ell_2 < \dots < \ell_s$ then we also have $r_1 < r_2 < \dots < r_s$. This gives us a linear order on the intervals, and so we can maintain them in any balanced binary search tree. After adding the new interval $[\ell, r]$ to the collection, we can check if it is not contained in any of the already existing intervals, and if so find the already existing intervals that should be removed, with standard operations on the balanced binary search tree.

Now we explain how to implement **Step 1** and **Step 2** efficiently using T_y and the secondary structures associated with its nodes. Let i and $(x_i, y_i) \in \text{THRESHOLDS}(i)$ be the index and the threshold we are currently considering. We begin our discussion with **Step 2** and therefore assume that we already computed $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ for this threshold. Note that all thresholds $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $x_i < x_h$ and $y_i < y_h$ have been already processed. Moreover, all thresholds $(x_c, y_c) \in \text{THRESHOLDS}(c)$ such that $x_i < x_c$ have been already processed and will not be considered in the future, so we don't need to be concerned with updating their answer. Hence, in **Step 2** we update all thresholds $(x_h, y_h) \in \text{THRESHOLDS}(h)$ such that $y_h < y_i$, regardless of the value of x_h . To this end, we consider every ancestor y_h of y_i such that $y_h < y_i$, plus the node y_i itself, and add the interval $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ or $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ to their corresponding T_{\max} and T_{\min} , respectively. To implement **Step 1**, we consider every ancestor y_c of y_i such that $y_c > y_i$, plus the node y_i itself, and we query their corresponding T_{\min} and T_{\max} . It can be readily verified that by the choice of which ancestors are updated, this is enough to implicitly consider every $y_h > y_i$, as such y_h must have updated one of the ancestors y_c .

■ **Algorithm 1** CTMIS in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space.

Data: indeterminate length- n strings p and t with 2 uncertain characters per position.

Output: Does $p \sim_C t$ hold?

- 1 THRESHOLDS $\leftarrow \{(x_i, y_i) \mid x_i \in p[i] \text{ and } y_i \in t[i] \text{ for some } i = 1, 2, \dots, n\}$
- 2 Build a balanced binary search tree T_y on the values of y_i
- 3 **foreach** node $u \in T_y$ **do**
- 4 Create secondary balanced search trees $T_{\min}(u)$ and $T_{\max}(u)$ of intervals $[\ell, r]$
 - ▷ $T_{\min}(u)$ is ordered by ℓ while $T_{\max}(u)$ is ordered by r
- 5 **foreach** $(x_i, y_i) \in \text{THRESHOLDS}$ by decreasing order of x_i **do**
- 6 Let $u \in T_y$ be the node satisfying $\text{VALUE}(u) = y_i$
 - ▷ $\text{VALUE}(u)$ returns the corresponding y_i of a node $u \in T_y$.
- 7 $\min_L(i, x_i, y_i) \leftarrow i$
- 8 $\max_R(i, x_i, y_i) \leftarrow i$
 - ▷ **Step 1:** Query the potential candidates structures.
- 9 **foreach** v an ancestor of u in T_y **do** ▷ including u itself
- 10 **if** $\text{VALUE}(v) > y_i$ **then**
- 11 **if** $\min\{\ell \mid [\ell, r] \in T_{\min}(v) \text{ and } i \in [\ell, r]\} < \min_L(i, x_i, y_i)$ **then**
- 12 $\min_L(i, x_i, y_i) \leftarrow \min\{\ell \mid [\ell, r] \in T_{\min}(v) \text{ and } i \in [\ell, r]\}$
- 13 **if** $\max\{r \mid [\ell, r] \in T_{\max}(v) \text{ and } i \in [\ell, r]\} > \max_R(i, x_i, y_i)$ **then**
- 14 $\max_R(i, x_i, y_i) \leftarrow \max\{r \mid [\ell, r] \in T_{\max}(v) \text{ and } i \in [\ell, r]\}$
- 15 **if** $\min_L(i, x_i, y_i) = 1$ **and** $\max_R(i, x_i, y_i) = n$ **then**
- 16 **return true**
- 17 ▷ **Step 2:** Update the potential candidates structures.
- 18 **foreach** v an ancestor of u in T_y **do** ▷ including u itself
- 19 **if** $\text{VALUE}(v) < y_i$ **then**
- 20 **if** $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1] \not\subseteq [\ell, r]$ for all $[\ell, r] \in T_{\min}(v)$ **then**
- 21 Add $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ to $T_{\min}(v)$
- 22 Remove from $T_{\min}(v)$ every $[\ell', r'] \subseteq [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$
- 23 **if** $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)] \not\subseteq [\ell, r]$ for all $[\ell, r] \in T_{\max}(v)$ **then**
- 24 Add $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ to $T_{\max}(v)$
- Remove from $T_{\max}(v)$ every $[\ell', r'] \subseteq [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$
- 25 **return false**

Time complexity. The time complexity of the algorithm is $\mathcal{O}(n \log^2 n)$. First, we need to sort the $4n$ thresholds in $\mathcal{O}(n \log n)$ time. Each of these thresholds is processed by considering $\mathcal{O}(\log n)$ nodes of T_y . At each of these nodes u we spend $\mathcal{O}(\log n)$ amortized time to update and query $T_{\min}(u)$ and $T_{\max}(u)$. Furthermore, the space complexity is $\mathcal{O}(n \log n)$, because each interval appears in $\mathcal{O}(\log n)$ secondary structures. Instead of using balanced binary search trees with $\mathcal{O}(\log n)$ query and update time for the secondary structures, we can plug in any predecessor structure that stores a collection of s integers from $\{1, 2, \dots, n\}$ in $\mathcal{O}(s)$ space with expected $\mathcal{O}(\log \log n)$ query and update time [33].

4 Order-Preserving Matching of Indeterminate Strings

Given two indeterminate strings p and t of equal-length n with at most 2 uncertain characters per position, we want to check if there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{\leq} t_w$. The goal of this section is to prove that this is NP-hard by reducing checking satisfiability of a 3-CNF formula.

We start with rephrasing the question in a graph-theoretical language. Let Σ_p and Σ_t be the sets of characters that occur in p and t , respectively. We consider a complete undirected bipartite graph G with Σ_p corresponding to the nodes on the one side and Σ_t corresponding to the nodes on the other side. We claim that there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{\leq} t_w$ iff there exists a non-crossing matching M in G , where non-crossing means that we cannot have two edges $(x, y), (x', y')$ such that $x < x'$ but $y' < y$, such that the following holds for every position $i = 1, 2, \dots, n$:

$$p[i] = x \text{ and } t[i] = y : (x, y) \in M,$$

$$p[i] = x_1|x_2 \text{ and } t[i] = y : (x_1, y) \in M \text{ or } (x_2, y) \in M$$

$$p[i] = x \text{ and } t[i] = y_1|y_2 : (x, y_1) \in M \text{ or } (x, y_2) \in M,$$

$$p[i] = x_1|x_2 \text{ and } t[i] = y_1|y_2 : (x_1, y_1) \in M \text{ or } (x_1, y_2) \in M \text{ or } (x_2, y_1) \in M \text{ or } (x_2, y_2) \in M.$$

The proof is straightforward.

We consider a 3-CNF formula ϕ on n variables $1, 2, \dots, n$ and m clauses. We reduce checking satisfiability of ϕ to finding a non-crossing matching M with some additional constraints in a complete undirected bipartite graph G . Each constraint is of the form $M \cap X \times Y \neq \emptyset$, for some subsets of the nodes X and Y such that $|X|, |Y| \leq 2$, or $X \times Y$ for short. As long as the size of G and the number of constraints is polynomial, this will establish NP-hardness of our problem, as we can create two strings p and t and encode each constraint by setting up some $p[i]$ and $t[i]$ appropriately.

We start with creating nodes $1, 2, \dots, n$ on the left side and $1, 2, 3, 4, \dots, 2n$ on the right side of G . We add a constraint $\{i\} \times \{2i-1, 2i\}$, for every $i = 1, 2, \dots, n$. We add a constraint $\{2n+1\} \times \{2n+1\}$. For every $k = 1, 2, \dots, m$, we consider the k -th clause $(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3})$, where $\ell_{k,1}, \ell_{k,2}, \ell_{k,3}$ are literals. Let $s = 2n + 2 + 5(k-1)$. We add the following constraints: $\{s\} \times \{s, s+1\}$, $\{s+2, s+3\} \times \{s+3\}$, $\{s, s+2\} \times \{s+1, s+3\}$. This is illustrated in Figure 5. Then we add a constraint for every literal:

1. If $\ell_{k,1} = x$ then we add $\{x, s\} \times \{2x, s\}$, and if $\ell_{k,1} = \neg x$ then we add $\{x, s\} \times \{2x-1, s\}$.
2. If $\ell_{k,2} = y$ then we add $\{y, s+1\} \times \{2y, s+2\}$, and if $\ell_{k,2} = \neg y$ then we add $\{x, s+1\} \times \{2y-1, s+2\}$.
3. If $\ell_{k,3} = z$ then we add $\{z, s+3\} \times \{2z, s+3\}$, and if $\ell_{k,3} = \neg z$ then we add $\{z, s+3\} \times \{2z-1, s+3\}$.

Due to the constraint $\{2n+1\} \times \{2n+1\}$, a variable constraint $\{v, a\} \times \{2v-1, b\}$ translates into $(v, 2v-1) \in M$ or $(a, b) \in M$. Similarly, $\{v, a\} \times \{2v, b\}$ translates into $(v, 2v) \in M$ or $(a, b) \in M$.

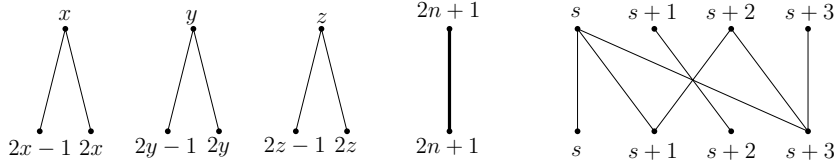
We need to prove that ϕ is satisfiable iff there exists a non-crossing matching M in G that respects all the constraints.

First, assume that ϕ is satisfiable and fix a satisfying valuation of all the variables. We obtain M by first adding $(v, 2v-1)$ or $(v, 2v)$ to M depending on whether v is set to false or true, respectively. We also add $(2n+1, 2n+1)$ to M . Then, we proceed as follows for the k -th clause. For concreteness assume that the clause is $(x \vee y \vee z)$, the argument is symmetric for the other cases. If x is set to false then we add (s, s) to M . If y is set to false then we add $(s+1, s+2)$ to M . Finally, if z is set to false then we add $(s+3, s+3)$ to M .

14:10 On Indeterminate Strings Matching

Because at least one of x, y, z is set to true, at least one of these three edges is not in M . If $(s+1, s+2) \notin M$ then we add $(s+2, s+1)$ to M . If $(s, s) \notin M$ then we add $(s, s+1)$ to M . Finally, if $(s+3, s+3) \notin M$ then we add $(s+2, s+3)$ to M . In all cases, the constraints corresponding to the k -clause are fulfilled. Due to how we compose the gadgets, M being a non-crossing matching in every gadget implies that M is a non-crossing matching in the whole G .

Second, assume that we have a non-crossing matching M in G . For every $v = 1, 2, \dots, n$, M contains exactly one of the edges $(v, 2v-1), (v, 2v)$. We set v to false if $(v, 2v-1) \in M$ and to true if $(v, 2v) \in M$. We must have $(2n+1, 2n+1) \in M$. We need to verify that every clause is satisfied by the obtain valuation of the variables. Again, for concreteness assume that the clause is $(x \vee y \vee z)$. We cannot have all edges $(s, s), (s+1, s+2), (s+3, s+3)$ in M , as in such case the constraint $\{s, s+2\} \times \{s+1, s+3\}$ cannot be fulfilled. If $(s, s) \notin M$ then due to the constraint $\{x, s\} \times \{2x, s\}$ we must have $(x, 2x) \in M$, so x is set to true. If $(s+1, s+2) \notin M$ then due to the constraint $\{y, s+1\} \times \{2y, s+2\}$ we must have $(y, 2y) \in M$, so y is set to true. Finally, if $(s+3, s+3) \notin M$ then due to the constraint $\{z, s+3\} \times \{2z, s+3\}$ we must have $(z, 2z) \in M$, so z is set to true. So, one of the variable x, y, z is set to true, making the clause satisfied.



■ **Figure 5** Gadget created for the k -th clause concerning variables x, y, z .

5 Parameterized Matching of Indeterminate Strings

Given two indeterminate strings p and t of equal-length n with at most 2 uncertain characters per position, we want to check if there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{=} t_w$. The goal of this section is to prove that this is NP-hard by reducing checking if a given undirected graph has a vertex cover consisting of at most k vertices.

As in the previous section, we start with rephrasing the question in a graph-theoretical language. Let Σ_p and Σ_t be the sets of characters that occur in p and t , respectively. We consider a complete undirected bipartite graph G with Σ_p corresponding to the nodes on the one side and Σ_t corresponding to the nodes on the other side. We claim that there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{=} t_w$ iff there exists a matching M in G , such that the following holds for every position $i = 1, 2, \dots, n$:

$$\begin{aligned}
 p[i] = x \text{ and } t[i] = y & : (x, y) \in M, \\
 p[i] = x_1|x_2 \text{ and } t[i] = y & : (x_1, y) \in M \text{ or } (x_2, y) \in M \\
 p[i] = x \text{ and } t[i] = y_1|y_2 & : (x, y_1) \in M \text{ or } (x, y_2) \in M, \\
 p[i] = x_1|x_2 \text{ and } t[i] = y_1|y_2 & : (x_1, y_1) \in M \text{ or } (x_1, y_2) \in M \text{ or } (x_2, y_1) \in M \text{ or } \\
 & (x_2, y_2) \in M.
 \end{aligned}$$

The proof is straightforward.

We consider an undirected graph H on n vertices $V = \{1, 2, \dots, n\}$ and m edges E together with a parameter $k \leq n$. We reduce checking if there is a subset S of k vertices of H such that for every edge $(u, v) \in E$ we have $u \in S$ or $v \in S$ to finding a matching M in a complete undirected bipartite graph G that respects a number of constraints of the

form $M \cap X \times Y \neq \emptyset$, for $|X|, |Y| \leq 2$, or $X \times Y$ for short. As long as the size of G and the number of constraints is polynomial, this will establish NP-hardness of our problem, as we can create two strings p and t and encode each constraint by setting up some $p[i]$ and $t[i]$ appropriately.

We start with creating nodes $1, 2, \dots, n$ on the left side and $1, 2, \dots, n$ on the right side of G . We add a constraint $\{u, v\} \times \{u, v\}$ for every $(u, v) \in E$. For every $i = 1, 2, \dots, n$, $(i, j) \in M$ for some $j \in \{1, 2, \dots, n\}$ corresponds to including i in the sought vertex cover. The remaining part of H is constructed as to guarantee that there are at least k nodes $i \in \{1, 2, \dots, n\}$ such that $(i, j) \in M$ for some $j \neq \{1, 2, \dots, n\}$. To this end, we design a gadget G_{2s} with the following property:

1. there are distinguished $2s$ nodes v_1, v_2, \dots, v_{2s} on the left side, each v_i is incident to a unique edge e_i ,
2. there are also some additional internal nodes on the left and on the right and some constraints that concern both the internal and the distinguished nodes,
3. if none of the edges e_i belongs to M then it is not possible to satisfy the constraints of G_{2s} ,
4. for any nonempty subset S of distinguished nodes, it is possible to select some of the edges with both endpoints being internal nodes in such a way that, together with the edges e_i for $i \in S$, they satisfy all constraints of G_{2s} .

We will first show that G_4 exists, and then explain how to obtain $G_{2(s+1)}$ from G_{2s} .

► **Lemma 7.** G_4 with the sought properties exists.

Proof. G_4 consists of nodes v_1, v_2, v_3, v_4 and internal nodes v'_1, v'_2, v'_3, v'_4 and x, y, z . We set $e_i = (v_i, v'_i)$ for $i = 1, 2, 3, 4$ and create the following constraints: $\{v_1, x\} \times \{v'_1, y\}$, $\{v_2, x\} \times \{v'_2, y\}$, $\{v_3, z\} \times \{v'_3, y\}$, $\{v_4, z\} \times \{v'_4, y\}$ and $\{y\} \times \{x, z\}$. See Figure 6.

Assume that none of the edges e_i belongs to M . By symmetry, we can assume that $(x, y) \in M$. But then we must have $(v'_3, z), (v'_4, z) \in M$, which is impossible.

Let S be a nonempty set of distinguished nodes. By symmetry, we can assume that $v_1 \in S$. Then, we include $(y, z) \in M$ and if $e_2 \notin S$ we also include $(v'_2, x) \in M$. ◀

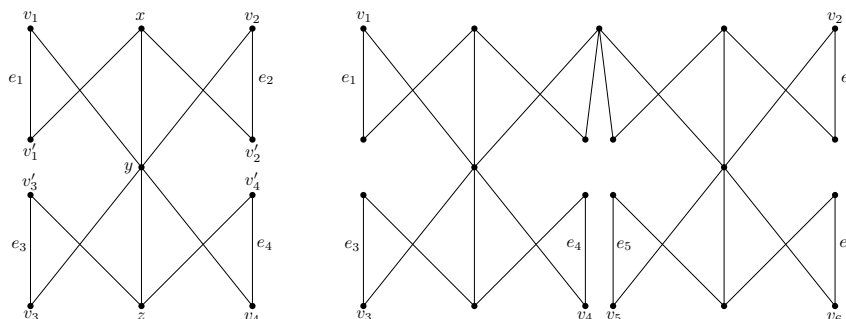
► **Lemma 8.** $G_{2(s+1)}$ with the sought properties can be obtained in polynomial time from G_{2s} with the sought properties.

Proof. We take a copy of G_{2s} , let its distinguished nodes and their corresponding edges be v_1, v_2, \dots, v_{2s} and e_1, e_2, \dots, e_{2s} . We also take a copy of G_4 , let its distinguished nodes and their corresponding edges be u_1, u_2, u_3, u_4 and f_1, f_2, f_3, f_4 . To obtain $G_{2(s+1)}$ we identify v_{2s} with u_1 and add a constraint that enforces including e_{2s} or f_1 in M . The distinguished nodes and their corresponding edges of $G_{2(s+1)}$ are $v_1, v_2, \dots, v_{2s-1}, u_2, u_3, u_4$ and $e_1, e_2, \dots, e_{2s-1}, f_2, f_3, f_4$.

Assume that none of the edges $e_1, e_2, \dots, e_{2s-1}, f_2, f_3, f_4$ belongs to M . Either $e_{2s} \notin M$ and we obtain that none of the edges e_1, e_2, \dots, e_{2s} belongs to M , or $f_1 \notin M$ and none of the edges f_1, f_2, f_3, f_4 belongs to M . In either case we obtain a contradiction by the construction of G_{2s} or G_4 .

Let S be a nonempty set of distinguished nodes and assume that $v_1 \in S$ (other cases are essentially the same). We set $S' = S \cap \{v_1, v_2, \dots, v_{2s-1}\}$ and $S'' = (S \cap \{u_2, u_3, u_4\}) \cup \{u_1\}$. Then $S', S'' \neq \emptyset$, and by assumption we can select some of the edges with both endpoints being internal nodes of G_{2s} or G_4 in such a way that, together with the edges e_i for $i \in S'$ and f_j for $j \in S''$, they satisfy all constraints of G_{2s} and G_4 . Additionally, the constraint that enforces including e_{2s} or f_1 is satisfied by taking f_1 . So, by selecting the edges with

both endpoints being internal nodes of G_{2s} or G_4 together with f_1 we obtain a set of edges with both endpoints being internal nodes of $G_{2(s+1)}$ that, together with the edges associated with the nodes in S , satisfy all constraints of $G_{2(s+1)}$ as required. ◀



■ **Figure 6** Gadgets G_4 (left) and G_8 (right).

With the gadget G_{2s} in hand, we are ready to complete the reduction. By duplicating the graph H we can assume that $n = 2s$. We add $n - k$ copies of the gadget G_{2s} to G . Let v_1, v_2, \dots, v_{2s} be the distinguished nodes of one such copy. We identify v_i with the node i on the left side of G . This guarantees that for each gadget we must have a unique node i such that $(i, 1), (i, 2), \dots, (i, n) \notin M$. We claim that the resulting graph G has a matching that satisfies all the constraints if and only if H admits a vertex cover of cardinality at most k . In one direction, consider the set C consisting of all nodes $i \in \{1, 2, \dots, n\}$ such that $(i, 1), (i, 2), \dots, (i, n) \notin M$. By the properties of G_{2s} , $|C| \leq k$. We need to argue that C is a vertex cover. Consider any $(u, v) \in E$. Due to the constraint $\{u, v\} \times \{u, v\}$, one of the edges $(u, u), (u, v), (v, u), (v, v)$ must belong to M . But then either u or v cannot be matched to any node not belonging to $\{1, 2, \dots, n\}$, so $u \in C$ or $v \in C$ as required. In other direction, let C be a vertex cover of H of cardinality at most k . For every $i \in C$, we include the edge (i, i) in M . This clearly satisfies every constraint $\{u, v\} \times \{u, v\}$ by C being a vertex cover. Then, for every copy of G_{2s} we choose a unique node $i \notin C$ (that is not matched to any other node yet) and use the properties of G_{2s} to add its internal edges to M in such a way that, together with the edge associated to i , they satisfy all the constraints.

References

- 1 A. Alatabbi, A. S. M. Sohidull Islam, M. S. Rahman, R. J. Simpson, and W. F. Smyth. Enhanced covers of regular & indeterminate strings using prefix tables. *Automata, Languages and Combinatorics*, 21(3):131–147, 2016.
- 2 A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.
- 3 A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.
- 4 A. Apostolico, Péter L. Erdős, and M. Lewenstein. Parameterized matching with mismatches. *Discrete Algorithms*, 5(1):135–140, 2007.
- 5 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *25th STOC*, pages 71–80, 1993.
- 6 B. S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.
- 7 M. Bataa, S. G. Park, A. Amir, G. M. Landau, and K. Park. Finding periods in cartesian tree matching. In *30th IWOCA*, volume 11638, pages 70–84, 2019.

- 8 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *46th ICALP*, pages 21:1–21:15, 2019.
- 9 P. Bille, I. Li Gørtz, H. W. Vildhøj, and D. K. Wind. String matching with variable length gaps. *Theoretical Computer Science*, 443:385–394, October 2010.
- 10 P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták. Algorithms for jumbled pattern matching in strings. *International Journal of Foundations of Computer Science*, 23(02):357–374, 2012.
- 11 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.
- 12 M. Christodoulakis, P. J. Ryan, W. F. Smyth, and S. Wang. Indeterminate strings, prefix arrays & undirected graphs. *Theoretical Computer Science*, 600:34–48, 2015.
- 13 D. Costa, L. M. S. Russo, R. Henriques, H. Bannai, and A. P. Francisco. Order-preserving pattern matching indeterminate strings. In *30th CPM*, 2019. [arXiv:1905.02589](https://arxiv.org/abs/1905.02589).
- 14 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Covering problems for partial words and for indeterminate strings. In *25th ISAAC*, pages 220–232, 2014.
- 15 J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. Watson. Efficient pattern matching in degenerate strings with the burrows-wheeler transform. *Information Processing Letters*, 147, 2017.
- 16 P. Gawrychowski and P. Uznański. Order-preserving pattern matching with k mismatches. *Theoretical Computer Science*, 638:136–144, 2016.
- 17 G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. Shur, and T. Waleń. String periods in the order-preserving model. In *35th STACS*, volume 96, pages 1–16, 2018.
- 18 G. Gu, S. Song, S. Faro, T. Lecroq, and K. Park. Fast multiple pattern cartesian tree matching. In *14th WALCOM*, pages 107–119, 2020.
- 19 C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms (TALG)*, 3(3):29–44, 2007.
- 20 J. Helling, P. J. Ryan, W. F. Smyth, and M. Soltys. Constructing an indeterminate string from its associated graph. *Theoretical Computer Science*, 710, March 2017.
- 21 Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai. Order-preserving pattern matching indeterminate strings. In *29th CPM*, volume 105, pages 2:1–2:15, 2018.
- 22 J. Holub and W. F. Smyth. Algorithms on indeterminate strings. In *14th AWOCA*, pages 36–45, 2003.
- 23 J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *Discrete Algorithms*, 6(1):37–50, 2008.
- 24 C. Iliopoulos, R. Kundu, and S. Pissis. Efficient pattern matching in elastic-degenerate strings. In *11th LATA*, pages 131–142, 2017.
- 25 J. Kim, P. Eades, R. Fleischer, S. H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.
- 26 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
- 27 R. McIntyre and M. Soltys. An improved upper bound and algorithm for clique covers. *J. Discrete Algorithms*, 48:42–56, 2018.
- 28 S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian tree matching and indexing. In *30th CPM*, pages 16:1–16:14, 2019.
- 29 M. S. Rahman and C. S. Iliopoulos. Pattern matching algorithms with don’t cares. In *33th SOFSEM*, pages 116–126, 2007.
- 30 S. Song, C. Ryu, S. Faro, T. Lecroq, and K. Park. Fast cartesian tree matching. In *26th SPIRE*, pages 124–137, 2019.
- 31 J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

14:14 On Indeterminate Strings Matching

- 32 R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
- 33 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.
- 34 B. Zeidman. Software v. software. *IEEE Spectrum*, 47:32–53, 2010.

The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time

Shay Golan 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
golansh1@cs.biu.ac.il

Tomasz Kociumaka 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Tsvi Kopelowitz 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kopelot@gmail.com

Ely Porat 

Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
porately@cs.biu.ac.il

Abstract

We revisit the k -mismatch problem in the streaming model on a pattern of length m and a streaming text of length n , both over a size- σ alphabet. The current state-of-the-art algorithm for the streaming k -mismatch problem, by Clifford et al. [SODA 2019], uses $\tilde{O}(k)$ space and $\tilde{O}(\sqrt{k})$ worst-case time per character. The space complexity is known to be (unconditionally) optimal, and the worst-case time per character matches a conditional lower bound. However, there is a gap between the total time cost of the algorithm, which is $\tilde{O}(n\sqrt{k})$, and the fastest known offline algorithm, which costs $\tilde{O}(n + \min(\frac{nk}{\sqrt{m}}, \sigma n))$ time. Moreover, it is not known whether improvements over the $\tilde{O}(n\sqrt{k})$ total time are possible when using more than $O(k)$ space.

We address these gaps by designing a randomized streaming algorithm for the k -mismatch problem that, given an integer parameter $k \leq s \leq m$, uses $\tilde{O}(s)$ space and costs $\tilde{O}(n + \min(\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}))$ total time. For $s = m$, the total runtime becomes $\tilde{O}(n + \min(\frac{nk}{\sqrt{m}}, \sigma n))$, which matches the time cost of the fastest offline algorithm. Moreover, the worst-case time cost per character is still $\tilde{O}(\sqrt{k})$.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases Streaming pattern matching, Hamming distance, k -mismatch

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.15

Related Version A full version of the paper is available at <https://arxiv.org/abs/2004.12881>.

Funding This work was supported in part by ISF grants no. 1278/16 and 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

1 Introduction

In the fundamental Hamming distance problem, given two same-length strings X and Y , the goal is to compute $\text{Ham}(X, Y)$, which is the number of aligned mismatches between X and Y . In the pattern matching version of the Hamming distance problem, the input is a pattern P of length m and a text T of length n , both over a size- σ alphabet, and the goal is to compute the Hamming distance between P and every length- m substring of T . In this paper, we focus on a well studied generalization known as the k -mismatch problem [1, 2, 4, 7, 10, 11, 12, 15, 18, 19, 22], which is the “fixed-threshold” version of the



© Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 15; pp. 15:1–15:15

Leibniz International Proceedings in Informatics



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

pattern matching Hamming distance problem: for a given parameter k , for each length- m substring S of T , if $\text{Ham}(P, S) \leq k$, then compute $\text{Ham}(P, S)$, and otherwise, report that $\text{Ham}(P, S) > k$. Currently, the state-of-the-art (offline) algorithms for the k -mismatch problem are: (1) the algorithm of Fischer and Paterson [11], whose runtime is $\tilde{O}(\sigma n)$, and (2) the algorithm of Gawrychowski and Uznański [15], whose runtime is $\tilde{O}(n + \frac{nk}{\sqrt{m}})$; see also [4].

The online and streaming models. The growing size of strings to be processed, often exceeding the available memory limits, motivated the study of pattern matching in the streaming model, where the characters of T arrive in a stream one at a time, and every occurrence of P needs to be identified as soon as the last character of the occurrence arrives [3, 4, 6, 7, 8, 13, 16, 17, 21, 20, 23]. In the *streaming k -mismatch problem*, the goal is to compute the Hamming distance between P and the current length- m suffix of T after each new character arrives, unless the Hamming distance is larger than k (which the algorithm reports in this case). Algorithms in the streaming model are typically required to use space of size sublinear in m . A closely related model is the *online model*, where the space usage of the algorithm is no longer explicitly limited.

Porat and Porat [20] introduced the first streaming k -mismatch algorithm using $\tilde{O}(k^2)$ time per character and $\tilde{O}(k^3)$ space. Subsequent improvements [7, 16] culminated in an algorithm by Clifford et al. [8], which solves the streaming k -mismatch problem in $\tilde{O}(\sqrt{k})$ time per character using $\tilde{O}(k)$ space. The total time cost of $\tilde{O}(n\sqrt{k})$ matches the time cost of the offline algorithm of Amir et al. [2], and the worst-case per-character running time matches a recent lower bound (valid for $\sigma = \Omega(\sqrt{k})$ even with unlimited space usage) by Gawrychowski and Uznański [14], conditioned on the combinatorial Boolean matrix multiplication conjecture. However, the $\tilde{O}(n + \frac{nk}{\sqrt{m}})$ total time cost of the offline algorithm of Gawrychowski and Uznański [15] is smaller, and the $\tilde{O}(\sigma n)$ total time cost of the offline algorithm of Fischer and Paterson [11] is smaller for small σ .

In the online model, where $O(m)$ space usage is allowed, the fastest algorithms follow from a generic reduction by Clifford et al. [5], which shows that if the offline k -mismatch problem can be solved in $O(n \cdot t(m, k))$ time, then the online k -mismatch problem can be solved in $O(n \sum_{i=0}^{\lceil \log m \rceil} t(2^i, k))$ time. In particular, this yields online algorithms with a total runtime of $\tilde{O}(n\sqrt{k})$ and $\tilde{O}(n\sigma)$. Nevertheless, this approach cannot benefit from the state-of-the-art the offline algorithm of Gawrychowski and Uznański [15] since the running time of this algorithm degrades as m decreases. Thus, a natural question arises:

► **Question 1.** Is there an online/streaming algorithm for the k -mismatch problem whose total time cost is $\tilde{O}(n + \min(\frac{nk}{\sqrt{m}}, \sigma n))$?

Space usage. It is straightforward to show that any streaming algorithm for the k -mismatch problem must use $\Omega(k)$ space [8]. Thus, the space usage of the algorithm of Clifford et al. [8] is optimal. Remarkably, we are unaware of any other tradeoffs between (sublinear) space usage and runtime for the k -mismatch problem. This leads to the following natural question.

► **Question 2.** Is there a time-space tradeoff algorithm for the k -mismatch problem, using $s \geq \Omega(k)$ space?

Our results. We address both Question 1 and Question 2 by proving the following theorem.

► **Theorem 3.** *There exists a randomized streaming algorithm for the k -mismatch problem that, given an integer parameter $k \leq s \leq m$, costs $\tilde{O}(n + \min(\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}))$ total time and uses $\tilde{O}(s)$ space. Moreover, the worst-case time cost per character is $\tilde{O}(\sqrt{k})$. The algorithm is correct with high probability¹.*

Theorem 3 answers Question 2 directly. However, for Question 1, Theorem 3 only addresses the online setting, where $s = m$ can be set: since $k < \sqrt{m}$ yields $n > \frac{nk}{\sqrt{m}} > \frac{nk^2}{m}$, the total time cost is $\tilde{O}(n + \min(\frac{nk^2}{m}, \frac{nk}{\sqrt{m}}, \frac{\sigma nm}{m})) = \tilde{O}(n + \min(\frac{nk}{\sqrt{m}}, \sigma n))$. However, Question 1 remains open for the streaming model.

Another natural research direction is to extend Theorem 3 so that the pattern P could also be processed in a streaming fashion using $\tilde{O}(s)$ space, $\tilde{O}(\sqrt{k})$ time per character, and $\tilde{O}(m + \min(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}))$ time in total. To the best of our knowledge, among the existing streaming k -mismatch algorithms, only that of Clifford et al. [8] is accompanied with an efficient streaming procedure for preprocessing the pattern.

2 Algorithmic Overview and Organization

A string S of length $|S| = n$ is a sequence of characters $S[0]S[1] \cdots S[n-1]$ over an alphabet Σ . A *substring* of S is denoted by $S[i..j] = S[i]S[i+1] \cdots S[j]$ for $0 \leq i \leq j < n$. If $i = 0$, the substring is called a *prefix* of S , and if $j = n-1$, the substring is called a *suffix* of S . For two strings S and S' of the same length $|S| = n = |S'|$, we denote by $\text{Ham}(S, S')$ the Hamming distance of S and S' , that is, $\text{Ham}(S, S') = |\{0 \leq i \leq n-1 : S[i] \neq S'[i]\}|$. An integer ρ is a d -period of a string S if $\text{Ham}(S[0..n-\rho-1], S[\rho..n-1]) \leq d$.

2.1 Overview

To prove Theorem 3, we consider two cases, depending on whether or not there exists an integer $\rho \leq k$ that is a d -period of P for some $d = O(k)$. If such a ρ exists, then we say that P is *periodic*², and otherwise P is said to be *aperiodic*.

Tail partitioning. In both cases of whether P is periodic or not, our algorithms use the well-known *tail partitioning* technique [6, 7, 8, 9, 17], which decomposes P into two substrings: a suffix P_{tail} and the complementary prefix P_{head} of length $m - |P_{\text{tail}}|$. Accordingly, the algorithm has two components. The first component computes the Hamming distance of P_{head} and every length- $|P_{\text{head}}|$ substring of T with some delay: the reporting of $\text{Ham}(P_{\text{head}}, T[i - |P| + 1..i - |P_{\text{tail}}|])$ is required to be completed before the arrival of $T[i]$. The second component computes the Hamming distance of P_{tail} and carefully selected length- $|P_{\text{tail}}|$ substrings of T . The decision mechanism for selecting substrings for the second component is required to guarantee that whenever $\text{Ham}(P_{\text{head}}, T[i - |P| + 1..i - |P_{\text{tail}}|]) \leq k$: if $\text{Ham}(P_{\text{tail}}, T[i - |P_{\text{tail}}| + 1..i]) \leq k$ then the second component computes $\text{Ham}(P_{\text{tail}}, T[i - |P_{\text{tail}}| + 1..i])$; otherwise, the second component reports $\text{Ham}(P_{\text{tail}}, T[i - |P_{\text{tail}}| + 1..i]) > k$. The second component has no delay.

¹ An event \mathcal{E} happens with high probability if $\Pr[\mathcal{E}] \geq 1 - n^{-c}$ for a constant parameter $c \geq 1$.

² The classic notion of periodicity is usually much simpler than the one we define here. However, since in this paper we do not use the classic notion of periodicity, we slightly abuse the terminology.

Notice that if either $\text{Ham}(P_{head}, T[i - |P| \dots i - |P_{tail}|]) > k$, which is detected by the first component, or $\text{Ham}(P_{tail}, T[i - |P_{tail}| + 1 \dots i]) > k$, which is detected by the second component, then it must be that $\text{Ham}(P, T[i - |P| \dots i]) > k$. Otherwise, $\text{Ham}(P, T[i - |P| \dots i])$ is computed by summing $\text{Ham}(P_{head}, T[i - |P| \dots i - |P_{tail}|])$ and $\text{Ham}(P_{tail}, T[i - |P_{tail}| + 1 \dots i])$. In either case, the information is available for the algorithm right after $T[i]$ arrives.

Thus, our algorithm has four main components, depending on whether P is periodic or not, and depending on the head or tail case of the tail partitioning technique.

The aperiodic case. The algorithms for the aperiodic case are a combination of straightforward modifications of previous work together with the naïve algorithm; the details are given in Section 7. Nevertheless, we provide an overview below. In this case, $|P_{tail}| = 2k$.

The algorithm for P_{head} in the aperiodic case is a slight modification of an algorithm designed by Golan et al. [16], which reduces the streaming k -mismatch problem to the problem of finding occurrences of multiple patterns in multiple text-streams.

The algorithm for P_{tail} in the aperiodic case is the naïve algorithm of comparing all aligned pairs of characters. While in general the naïve algorithm could cost $O(|P_{tail}|)$ time per character, in our setting the algorithm uses the output of the algorithm on P_{head} as a filter, and so the algorithm computes $\text{Ham}(P_{tail}, T[i - |P_{tail}| + 1 \dots i])$ only if $\text{Ham}(P_{head}, T[i - |P| + 1 \dots i - |P_{tail}|]) \leq k$. Since P is aperiodic, we are able to show that occurrences of P_{head} are distant enough so that the naïve algorithm for P_{tail} costs $\tilde{O}(1)$ worst-case time per character. In order for the filter to be effective, instead of guaranteeing that the algorithm for computing $\text{Ham}(P_{head}, T[i - |P| + 1 \dots i - |P_{tail}|])$ is completed before $T[i]$ arrives, we refine the tail partitioning technique so that the computation of $\text{Ham}(P_{head}, T[i - |P| + 1 \dots i - |P_{tail}|])$ completes before $T[i - \frac{1}{2}|P_{tail}|]$ arrives, and if the naïve algorithm should be used, the execution takes place through the arrivals of the subsequent $\frac{1}{2}|P_{tail}|$ characters $T[i - \frac{1}{2}|P_{tail}| + 1 \dots i]$. The effects of this refinement on the runtime is only by constant multiplicative factors.

The periodic case. We begin by first assuming that P and T have a common $O(k)$ -period $\rho \leq k$, and that $n \leq \frac{3}{2}m$. In this case, we represent the strings as characteristic functions (one function for each character in Σ). Since both P and T are assumed to be periodic, each characteristic function, when treated as a string, is also periodic. Next, we use the notion of *backward differences*: for any function $f : \mathbb{Z} \rightarrow \mathbb{Z}$, the *backward difference* of f due to ρ is $\Delta_\rho[f](i) = f(i) - f(i - \rho)$. Clifford et al. [8] showed that the Hamming distance of two strings can be derived from a summation of convolutions of backward differences due to ρ of characteristic functions; see Section 3.

In the case of P_{head} , a delay of up to $2s$ characters is allowed. To solve this case, we define the problem of computing the convolutions of the backward differences in batches; the details for this case are given in Section 4. Our solution uses an offline algorithm for computing the convolutions described in Section 3.1. In the case of P_{tail} , we use a solution for the online version of computing the convolutions of the backward differences, which is adapted from Clifford et al. [5]; the details are given in Section 5. In both cases, since we assume that P and T are periodic, our algorithms leverage the fact that the backward differences of the characteristic functions have a small number of non-zero entries. This lets the algorithms compute the Hamming distance of P_{tail} and every substring of T which has length $|P_{tail}|$.

In Section 6, we remove the periodicity assumption on T by applying a technique by Clifford et al. [7] which identifies at most one periodic region of T that contains all the k -mismatch occurrences of P . Moreover, we drop the $n \leq \frac{3}{2}m$ assumption using a standard trick of partitioning T into overlapping fragments of length $\frac{3}{2}m$.

3 Hamming Distance and the Convolution Summation Problem

Recall that the *support* of a function f is $\text{supp}(f) := \{x \mid f(x) \neq 0\}$. Let $|f| = |\text{supp}(f)|$. Throughout, we only consider functions with finite support mapping \mathbb{Z} to \mathbb{Z} . The *convolution* of two functions $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ 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 and a character $c \in \Sigma$, 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 be 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

$$X \otimes Y = \sum_{c \in \Sigma} X_c * Y_c^R.$$

► **Lemma 4** ([8, Fact 7.1]). *Let P, T be strings. For $|P| - 1 \leq i < |T|$, we have $[T \otimes P](i) = |P| - \text{Ham}(P, T[i - |P| + 1 .. i])$. For $i < 0$ and for $i \geq |P| + |T|$, we have $[T \otimes P](i) = 0$.*

By Lemma 4, in order to compute $\text{Ham}(P, T[i - |P| + 1 .. i])$, it suffices to compute $[T \otimes P](i)$.

The *backward difference* of a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ due to ρ is $\Delta_\rho[f](i) = f(i) - f(i - \rho)$.

► **Observation 5** ([8, Obs. 7.2]). *If a string X has a d -period ρ , then $\sum_{c \in \Sigma} |\Delta_\rho[X_c]| \leq 2(d + \rho)$.*

Our computation of $T \otimes P$ in a streaming fashion is based on the following lemma:

► **Lemma 6** (Based on [8, Fact 7.4 and Corollary 7.5]). *For every $i \in \mathbb{Z}$ and $\rho \in \mathbb{Z}_+$, we have $[T \otimes P](i) = [\sum_{c \in \Sigma} \Delta_\rho[T_c] * \Delta_\rho[P_c^R]](i) - [T \otimes P](i - 2\rho) + 2[T \otimes P](i - \rho)$.*

When computing $[T \otimes P](i)$, if the algorithm maintains a buffer of the last 2ρ values of $T \otimes P$, then the algorithm already has the values of $[T \otimes P](i - \rho)$ and $[T \otimes P](i - 2\rho)$. Thus, in order to construct $T \otimes P$, the focus is on constructing $\sum_{c \in \Sigma} \Delta_\rho[T_c] * \Delta_\rho[P_c^R]$.

3.1 Convolution Summation Problem

We express the task of constructing $\sum_{c \in \Sigma} \Delta_\rho[T_c] * \Delta_\rho[P_c^R]$ in terms of a more abstract *convolution summation* problem stated as follows. The input is two sequences of functions $\mathcal{F} = (f_1, f_2, \dots, f_t)$ and $\mathcal{G} = (g_1, g_2, \dots, g_t)$ such that for every $1 \leq i \leq t$ we have $f_i : \mathbb{Z} \rightarrow \mathbb{Z}$ and $g_i : \mathbb{Z} \rightarrow \mathbb{Z}$, and the goal is to construct the function $\mathcal{F} \otimes \mathcal{G} = \sum_{j=1}^t (f_j * g_j)$.

Let \mathcal{H} be a sequence of functions. We define the *support* of \mathcal{H} as $\text{supp}(\mathcal{H}) = \bigcup_{h \in \mathcal{H}} \text{supp}(h)$. The total number of non-zero entries in all of the functions of \mathcal{H} is denoted by $\|\mathcal{H}\| = \sum_{h \in \mathcal{H}} |h|$. The *diameter* of a function f is $\text{diam}(f) = \max(\text{supp}(f)) - \min(\text{supp}(f)) + 1$ if $\text{supp}(f) \neq \emptyset$, and $\text{diam}(f) = 0$ otherwise. We define the *diameter* of a sequence of functions \mathcal{H} as $\text{diam}(\mathcal{H}) = \max \{\text{diam}(h) \mid h \in \mathcal{H}\}$.

In our setting, the input for the convolution summation problem is two sequences of *sparse functions*, which are functions that have a small support. Thus, we assume that the input functions are given in an efficient *sparse representation*, e.g., a linked list (of the non-zero functions) of linked lists (of non-zero entries).

Algorithm for the offline convolution summation problem. The following lemma, proved in the full version of the paper, provides an algorithm that efficiently computes $\mathcal{F} \otimes \mathcal{G}$ for two sequences of functions \mathcal{F} and \mathcal{G} which are given in a sparse representation. Notice that the output of the algorithm is also restricted to the non-zero values of $\mathcal{F} \otimes \mathcal{G}$ only.

► **Lemma 7** (Based on [4, Lemma 7.5]). *Let $\mathcal{F} = (f_1, \dots, f_t)$ and $\mathcal{G} = (g_1, \dots, g_t)$ be two sequences of functions, such that $\text{diam}(\mathcal{F}), \text{diam}(\mathcal{G}) \in [1..n]$, and also $\text{diam}(\mathcal{F} \otimes \mathcal{G}) = O(n)$. Then there exists an (offline) algorithm that computes the non-zero entries of $\mathcal{F} \otimes \mathcal{G}$ using $O(n)$ space, whose time cost is*

$$\Psi(\mathcal{F}, \mathcal{G}) = \tilde{O}\left(\|\mathcal{F}\| + \|\mathcal{G}\| + \sum_{j=1}^t \min(|f_j| |g_j|, n)\right) = \tilde{O}\left(\min\left(tn, \|\mathcal{F}\| \cdot \|\mathcal{G}\|, (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{n}\right)\right).$$

4 Periodic Pattern and Text – with Delay

Our approach is based on the reduction to the convolution summation problem of Section 3. The text arrives online, so we consider a similar setting for convolution summation.

4.1 The Incremental Batched Convolution Summation Problem

In the incremental batched version of the convolution summation problem, the algorithm is given two sequences of t functions \mathcal{F} and \mathcal{G} , where both $\text{supp}(\mathcal{F}), \text{supp}(\mathcal{G}) \subseteq [0..n-1]$. The sparse representation of \mathcal{G} is available for preprocessing, whereas \mathcal{F} is revealed online in batches of diameter s : the i th batch consists of all of the non-zero entries of the functions of \mathcal{F} in the range $[(i-1) \cdot s..i \cdot s]$, also in a sparse representation. After each update, the goal is to compute the values of $\mathcal{F} \otimes \mathcal{G}$ in the same range as the input, $[(i-1) \cdot s..i \cdot s]$. In the rest of this section, we prove the following lemma.

► **Lemma 8.** *There exists a deterministic algorithm that solves the incremental batched convolution summation problem for $s = \Omega(\|\mathcal{F}\| + \|\mathcal{G}\|)$, using $O(s)$ space, $\tilde{O}((\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s})$ time per batch arrival and $\tilde{O}\left(n + \min\left(\|\mathcal{F}\| \cdot \|\mathcal{G}\|, \frac{n(\|\mathcal{F}\| + \|\mathcal{G}\|)}{\sqrt{s}}, \frac{tn^2}{s}\right)\right)$ total time.*

A natural approach for proving Lemma 8 is to utilize the algorithm of Lemma 7, whose runtime depends on the diameters of a pair of sequences of functions. Thus, in order to use this approach, we design a mechanism for reducing the diameters of \mathcal{F} and \mathcal{G} while still being able to properly compute the values of $\mathcal{F} \otimes \mathcal{G}$.

Reducing the diameter. For a function $h : \mathbb{Z} \rightarrow \mathbb{Z}$ and a domain $D \subseteq \mathbb{Z}$, let $h|_D : \mathbb{Z} \rightarrow \mathbb{Z}$ be a function where $h|_D(i) = h(i)$ for $i \in D$ and $h|_D(i) = 0$ for $i \notin D$. For a sequence of functions $\mathcal{H} = (h_1, h_2, \dots, h_t)$, denote the sequence of functions restricted to domain D as $\mathcal{H}|_D = (h_1|_D, h_2|_D, \dots, h_t|_D)$. For $a, b \in \mathbb{Z}$, let us define integer intervals $\Gamma_a = [s \cdot (a-1) .. s \cdot (a+1)]$ and $\Phi_b = [s \cdot (b-1) .. s \cdot b]^3$.

► **Observation 9.** *Each integer is in exactly one range Φ_b and in exactly two ranges Γ_a .*

Notice that the i th batch consists of $\mathcal{F}|_{\Phi_i}$. In response to the i th batch, the algorithm needs to compute $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i}$. For this, we express $\mathcal{F}_i = \mathcal{F}|_{[0..si]} = \mathcal{F}|_{\Phi_1 \cup \Phi_2 \cup \dots \cup \Phi_i}$ (which is the aggregate of all the batches received so far) and \mathcal{G} in terms of two sequences of functions \mathcal{F}_i^* and \mathcal{G}^* , so that $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i} = [\mathcal{F}_i^* \otimes \mathcal{G}^*]|_{\Phi_i}$. The motivation for using \mathcal{F}_i^* and \mathcal{G}^* is to reduce the diameter, which comes at the price of increasing the number of functions.

Let $q = \lceil \frac{n}{s} \rceil$, and define

$$\begin{aligned} \mathcal{F}_i^* &= (f_1|_{\Phi_{i-0}}, \dots, f_t|_{\Phi_{i-0}}, f_1|_{\Phi_{i-1}}, \dots, f_t|_{\Phi_{i-1}}, \dots, f_1|_{\Phi_{i-(q-1)}}, \dots, f_t|_{\Phi_{i-(q-1)}}), \\ \mathcal{G}^* &= (g_1|_{\Gamma_0}, \dots, g_t|_{\Gamma_0}, g_1|_{\Gamma_1}, \dots, g_t|_{\Gamma_1}, \dots, g_1|_{\Gamma_{q-1}}, \dots, g_t|_{\Gamma_{q-1}}). \end{aligned}$$

³ Note that the Γ intervals are used for partitioning \mathcal{G} while Φ intervals are used for partitioning \mathcal{F} .

► **Lemma 10.** *For any $i \in [1..q]$ we have $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i} = [\mathcal{F}_i^* \otimes \mathcal{G}^*]|_{\Phi_i}$.*

Proof. For two sets $X, Y \subseteq \mathbb{Z}$, we denote $X - Y = \{x - y \mid x \in X \text{ and } y \in Y\}$. For any pair of functions $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ and any $j \in \Phi_i$, since the ranges Φ_b for $b \in \mathbb{Z}$ form a partition of \mathbb{Z} , we have that $(f * g)(j) = \sum_{a \in \mathbb{Z}} (f|_{\Phi_{i-a}} * g)(j)$. Moreover, by the definition of the convolution operator and since $\Phi_i - \Phi_{i-a} \subseteq \Gamma_a$, we have $\sum_{a \in \mathbb{Z}} (f|_{\Phi_{i-a}} * g)(j) = \sum_{a \in \mathbb{Z}} (f|_{\Phi_{i-a}} * g|_{\Gamma_a})(j)$. Hence, $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i} = \sum_{a \in \mathbb{Z}} [\mathcal{F}|_{\Phi_{i-a}} \otimes \mathcal{G}|_{\Gamma_a}]|_{\Phi_i}$. However, since $\text{supp}(\mathcal{F}) \subseteq [0..qs]$, we have that for every $b \leq 0$ and every $f \in \mathcal{F}$, it must be that $f|_{\Phi_b} = 0$. Similarly, since $\text{supp}(\mathcal{G}) \subseteq [0..qs]$, we have that for every $a < 0$ and every $g \in \mathcal{G}$, it must be that $g|_{\Gamma_a} = 0$. Thus, for $a \notin [0..q-1]$, we have that $\mathcal{F}|_{\Phi_{i-a}} \otimes \mathcal{G}|_{\Gamma_a} = 0$, and so $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i} = \sum_{a=0}^{q-1} [\mathcal{F}|_{\Phi_{i-a}} \otimes \mathcal{G}|_{\Gamma_a}]|_{\Phi_i}$. Finally, since \mathcal{F}_i^* is the concatenation of $\mathcal{F}|_{\Phi_{i-a}}$ for $a \in [0..q-1]$, and \mathcal{G}^* is the concatenation of $\mathcal{G}|_{\Gamma_a}$ for $a \in [0..q-1]$, the lemma follows. ◀

The following is a consequence of the definitions of \mathcal{F}^* and \mathcal{G}^* , and Observation 9. Recall that the diameter of a sequence of functions \mathcal{H} is $\text{diam}(\mathcal{H}) = \max \{\text{diam}(h) \mid h \in \mathcal{H}\}$.

► **Observation 11.** *The sequences \mathcal{F}_i^* and \mathcal{G}^* consist of $t \cdot q$ functions each. Moreover, $\text{diam}(\mathcal{F}_i^*) \leq s$, $\text{diam}(\mathcal{G}^*) \leq 2s$, $\|\mathcal{F}_i^*\| \leq \|\mathcal{F}\|$, and $\|\mathcal{G}^*\| \leq 2 \cdot \|\mathcal{G}\|$.*

The algorithm. During the preprocessing phase, the algorithm transforms \mathcal{G} into \mathcal{G}^* , and constructs \mathcal{F}_0^* , which is an empty linked list.

Upon receiving the i th batch, which is $\mathcal{F}|_{\Phi_i}$, the algorithm computes \mathcal{F}_i^* as follows: First, the algorithm concatenates $\mathcal{F}|_{\Phi_i}$ with \mathcal{F}_{i-1}^* and truncates the last t functions from the resulting sequence of $(q+1)t$ functions. It is easy to implement the concatenation (including updating indices in the sparse representation) with a time cost which is linear in the number of non-zero functions in $\mathcal{F}|_{\Phi_i}$ and \mathcal{F}_{i-1}^* , and is at most $O(\|\mathcal{F}\|)$. The implementation of the truncation is void since only the first $(i-1) \cdot t \leq (q-1) \cdot t$ functions of \mathcal{F}_{i-1}^* are non-zero⁴.

Next, the algorithm applies the procedure of Lemma 7 in order to compute $\mathcal{F}_i^* \otimes \mathcal{G}^*$. Finally, the algorithm returns $[\mathcal{F}_i^* \otimes \mathcal{G}^*]|_{\Phi_i}$ which, by Lemma 10, is $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_i}$.

Complexities. The preprocessing phase is done in $O(\|\mathcal{G}\|)$ time using $O(s)$ space.

For the i th batch, the computation of \mathcal{F}_i^* costs $O(\|\mathcal{F}\|)$ time, which is $O(q\|\mathcal{F}\|) = O(\frac{n}{s}\|\mathcal{F}\|) = O(n)$ time in total because $s = \Omega(\|\mathcal{F}\| + \|\mathcal{G}\|)$. Then, the computation of $\mathcal{F}_i^* \otimes \mathcal{G}^*$ with the procedure of Lemma 7 costs $O(\Psi(\mathcal{F}_i^*, \mathcal{G}^*)) = \tilde{O}((\|\mathcal{F}_i^*\| + \|\mathcal{G}^*\|)\sqrt{s}) = \tilde{O}((\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s})$ time. In the following, we derive several upper bounds on $\sum_{i=1}^q \Psi(\mathcal{F}_i^*, \mathcal{G}^*)$, which together upper bound the total time cost.

Total time $\tilde{O}\left(\frac{n(\|\mathcal{F}\| + \|\mathcal{G}\|)}{\sqrt{s}}\right)$. Recall that $q = O(\frac{n}{s})$, and, by Observation 11, for any $i \in [1..q]$ we have $\|\mathcal{F}_i^*\| \leq \|\mathcal{F}\|$ and $\|\mathcal{G}^*\| \leq 2\|\mathcal{G}\|$. Thus,

$$\begin{aligned} \sum_{i=1}^q \Psi(\mathcal{F}_i^*, \mathcal{G}^*) &= \sum_{i=1}^q \tilde{O}((\|\mathcal{F}_i^*\| + \|\mathcal{G}^*\|)\sqrt{s}) = \tilde{O}\left(\sum_{i=1}^q (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s}\right) = \\ &\tilde{O}(q \cdot (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s}) = \tilde{O}\left(\frac{n}{s} \cdot (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s}\right) = \tilde{O}\left(\frac{n(\|\mathcal{F}\| + \|\mathcal{G}\|)}{\sqrt{s}}\right). \end{aligned}$$

⁴ The reason for mentioning the truncation, even though it is void, is in order to guarantee that \mathcal{F}_i^* matches the mathematical definition introduced above.

15:8 The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time

Total time $\tilde{O}\left(\frac{tn^2}{s}\right)$. Recall that for any $i \in [1..q]$ the sequences \mathcal{G}^* and \mathcal{F}_i^* consist of $q \cdot t$ functions of diameter $O(s)$, and $q = O\left(\frac{n}{s}\right)$. Thus,

$$\sum_{i=1}^q \Psi(\mathcal{F}_i^*, \mathcal{G}^*) = \sum_{i=1}^q \tilde{O}(q \cdot t \cdot s) = \tilde{O}(q^2 \cdot t \cdot s) = \tilde{O}\left(\frac{n^2}{s^2} \cdot t \cdot s\right) = \tilde{O}\left(\frac{tn^2}{s}\right)$$

Total time $\tilde{O}(n + \|\mathcal{F}\| \cdot \|\mathcal{G}\|)$. Let $\mathcal{X} = \{x_1, x_2, \dots, x_\tau\}$ and $\mathcal{Y} = \{y_1, y_2, \dots, y_\tau\}$ be two sequences of τ functions from \mathbb{Z} to \mathbb{Z} , and let $\nu = \max(\text{diam}(\mathcal{X}), \text{diam}(\mathcal{Y}))$. Recall that

$$\Psi(\mathcal{X}, \mathcal{Y}) = \tilde{O}\left(\|\mathcal{X}\| + \|\mathcal{Y}\| + \sum_{j=1}^{\tau} \min(|x_j| |y_j|, \nu)\right) = \tilde{O}\left(\|\mathcal{X}\| + \|\mathcal{Y}\| + \sum_{j=1}^{\tau} |x_j| |y_j|\right).$$

In our case, the run time is $\sum_{i=1}^q \Psi(\mathcal{F}_i^*, \mathcal{G}^*)$. Recall that the functions in \mathcal{F}_i^* are $f_j|_{\Phi_{i-a}}$, and the functions in \mathcal{G}^* are $g_j|_{\Gamma_a}$, both for $a \in [0..q]$ and $j \in [1..t]$. Thus,

$$\begin{aligned} \sum_{i=1}^q \Psi(\mathcal{F}_i^*, \mathcal{G}^*) &= \sum_{i=1}^q \tilde{O}\left(\|\mathcal{F}_i^*\| + \|\mathcal{G}^*\| + \sum_{a=0}^{q-1} \sum_{j=1}^t |f_j|_{\Phi_{i-a}}|g_j|_{\Gamma_a}\right) \\ &= \sum_{i=1}^q \tilde{O}(\|\mathcal{F}\| + \|\mathcal{G}\|) + \tilde{O}\left(\sum_{j=1}^t \sum_{a=0}^{q-1} |g_j|_{\Gamma_a} \sum_{i=1}^q |f_j|_{\Phi_{i-a}}\right) \quad \triangleright \text{by Obs. 11} \\ &= \tilde{O}\left(\frac{n}{s}s\right) + \tilde{O}\left(\sum_{j=1}^t \sum_{a=0}^{q-1} |g_j|_{\Gamma_a} |f_j|\right) \quad \triangleright \text{by Obs. 9} \\ &= \tilde{O}(n) + \tilde{O}\left(\sum_{j=1}^t |g_j| |f_j|\right) \quad \triangleright \text{by Obs. 9} \\ &= \tilde{O}(n) + \tilde{O}\left(\|\mathcal{G}\| \sum_{j=1}^t |f_j|\right) = \tilde{O}(n) + \tilde{O}(\|\mathcal{G}\| \cdot \|\mathcal{F}\|) = \tilde{O}(n + \|\mathcal{F}\| \cdot \|\mathcal{G}\|) \end{aligned}$$

This completes the proof of Lemma 8. ◀

4.2 Reduction from Hamming Distance to Incremental Batch Sparse Convolution Summation Problem

Now we show how to compute the Hamming distance between P and substrings of T with delay of $2s$, based on the algorithm of Lemma 8. Our result is stated in the following lemma.

► **Lemma 12.** *Suppose that there exists $\rho \leq k$ which is a d -period of both P and T for some $d = O(k)$. Then, there exists a deterministic streaming algorithm for the k -mismatch problem with $n = \frac{3}{2}m$ that, given an integer parameter $k \leq s \leq m$, uses $\tilde{O}(s)$ space and costs $\tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$ total time. Moreover, the worst-case time cost per character is $\tilde{O}(\sqrt{k})$. The algorithm has delay of $2s$ characters.*

Proof. The algorithm receives the characters of T one by one and splits them into blocks of length s so that the indices in the b th block form Φ_b , as defined in Section 4.1. The last $O(s)$ characters of T and the last $O(s)$ values of $T \otimes P$ are buffered in $O(s)$ space.

For $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$, define sequences of functions $\mathcal{G} = (\Delta_\rho[P_{c_1}^R], \Delta_\rho[P_{c_2}^R], \dots, \Delta_\rho[P_{c_\sigma}^R])$ and $\mathcal{F} = (\Delta_\rho[T_{c_1}], \Delta_\rho[T_{c_2}], \dots, \Delta_\rho[T_{c_\sigma}])$. The algorithm initializes an instance of the procedure of Lemma 8 with \mathcal{G} during the arrival of the first block. During the arrival of the b th block, the algorithm creates the batch $\mathcal{F}|_{\Phi_b}$ as follows: after the arrival of $T[i]$, if $T[i] \neq T[i - \rho]$, then the algorithm sets $\Delta_\rho[T_{T[i]}](i)$ to be 1 and $\Delta_\rho[T_{T[i-\rho]}](i)$ to be -1 . During the arrival of the $(b+1)$ th block, the batch $\mathcal{F}|_{\Phi_b}$ is processed using Lemma 8 in order to compute $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_b}$. Finally, for all $i \in \Phi_b$, the algorithm uses the buffer of $T \otimes P$ together with the values of $[\mathcal{F} \otimes \mathcal{G}]|_{\Phi_b}$ in order to report the values $[T \otimes P](i) = [\mathcal{F} \otimes \mathcal{G}](i) - [T \otimes P](i - 2\rho) + 2[T \otimes P](i - \rho)$ (see Lemma 6).

Complexities. For each incoming character of T , updating the text buffer and the functions $\Delta_\rho[T_c]$ for all $c \in \Sigma$ costs $O(1)$ time since changes are needed in $\Delta_\rho[T_c]$ for at most two characters c . Since $\rho \leq k$ is a d -period of both P and T , Observation 5 yields $\|\mathcal{G}\| \leq 2(k + d) = O(k)$ and $\|\mathcal{F}\| \leq 2(k + d) = O(k)$. Thus, the initialization of the procedure of Lemma 8 costs $O(\|\mathcal{G}\|) = O(k)$ time, and executing the procedure of Lemma 8 on the b th batch $\mathcal{F}|_{\Phi_b}$ costs $\tilde{O}((\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{s}) = \tilde{O}(k\sqrt{s})$ time. Hence, applying the algorithm of Lemma 8 during the arrival of *any* block costs $\tilde{O}(k + k\sqrt{s}) = \tilde{O}(k\sqrt{s})$ time, which, by a standard de-amortization, is $\tilde{O}(\frac{1}{s} \cdot k\sqrt{s}) = \tilde{O}(\frac{k}{\sqrt{s}}) = \tilde{O}(\sqrt{k})$ time per character.

Note that $\text{supp}(\mathcal{G}) \subseteq [0..m + \rho] = [0..O(m)]$ and $\text{supp}(\mathcal{F}) \subseteq [0..\frac{3}{2}m + \rho] = [0..O(m)]$. Moreover, both \mathcal{F} and \mathcal{G} are sequences of σ functions. Hence, the total time cost of applying the algorithm of Lemma 8, updating the buffers, and computing the functions $\Delta_\rho[T_c]$ is $\tilde{O}\left(m + \min\left(\|\mathcal{F}\| \cdot \|\mathcal{G}\|, \frac{m(\|\mathcal{F}\| + \|\mathcal{G}\|)}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right) = \tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$.

Delay. For any index $i \in \Phi_b$, after the arrival of $T[i]$, at most s character arrivals take place until the call to the procedure of Lemma 8 involving $\mathcal{F}|_{\Phi_b}$. Moreover, due to the de-amortization, the computation of all the results $[T \otimes P]|_{\Phi_b}$ takes place during the arrivals of another s characters. Hence, the delay of the algorithm is at most $2s$ character arrivals. ◀

5 Periodic Pattern and Text – without Delay

In this section, we show how to compute the distances between P and substrings of T without any delay, assuming that $\rho \leq k$ is a d -period of both P and T for some $d = O(k)$. Our approach is to use the tail partition technique, described in Section 2. To do so, we set $|P_{tail}| = 2s$ and use the algorithm of Lemma 12 on P_{head} (notice that ρ is a d -period of both P_{head} and P_{tail}). The remaining task is to describe how to compute $\text{Ham}(P_{tail}, T[i - 2s + 1..i])$.

In the online version of the convolution summation problem, the algorithm is given two sequences of functions \mathcal{F} and \mathcal{G} , where $\text{supp}(\mathcal{F}) \subseteq [0..n]$ and $\text{supp}(\mathcal{G}) \subseteq [0..m]$. The sparse representation of \mathcal{G} is available for preprocessing, whereas \mathcal{F} is revealed online index by index: At the i th update, the algorithm receives $\mathcal{F}|_{\{i\}}$, i.e., all the non-zero entries $f(i)$ for $f \in \mathcal{F}$, and the task is to compute $[\mathcal{F} \otimes \mathcal{G}](i)$. The procedure we use for this problem is based on the algorithm of Clifford et al. [5]. Nevertheless, since we state the procedure in terms of the convolution summation problem, whereas [5] states the algorithm in terms of pattern matching problems, we describe the details in the full version.

► **Lemma 13** (Based on [5, Theorem 1]). *Let \mathcal{F} and \mathcal{G} be two sequences of t functions each such that $\text{supp}(\mathcal{F}) \subseteq [0..n]$, $\text{supp}(\mathcal{G}) \subseteq [0..m]$, and $\delta = \max(\max_i \|\mathcal{F}|_{\{i\}}\|, \max_i \|\mathcal{G}|_{\{i\}}\|)$ is the maximum number of non-zero entries at a single index. There exists an online algorithm that upon receiving $\mathcal{F}|_{\{i\}}$ for subsequent indices i computes $[\mathcal{F} \otimes \mathcal{G}](i)$ using $O(\delta m)$ space in $\tilde{O}(\sqrt{\delta}(\|\mathcal{F}\| + \|\mathcal{G}\|))$ time per index. Moreover, the total running time of the algorithm is $\tilde{O}(\min(n\delta + (\|\mathcal{F}\| + \|\mathcal{G}\|)\sqrt{n}, nt))$.*

15:10 The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time

Using the algorithm of Lemma 13 and the reduction from computing Hamming distance to the convolution summation problem defined in Section 3, we achieve the following lemma.

► **Lemma 14.** *Suppose that there exists $\rho \leq k$ which is a d -period of both P and T for some $d = O(k)$. Then, there exists a deterministic online algorithm for the k -mismatch problem that uses $\tilde{O}(m)$ space and costs $\tilde{O}(n + \min(k\sqrt{n}, n\sigma))$ total time. Moreover, the worst-case time cost per character is $\tilde{O}(\sqrt{k})$.*

Proof. The algorithm maintains a buffer of the last 2ρ values of $T \otimes P$ and a buffer of the last ρ characters of T . Define sequences of functions $\mathcal{G} = (\Delta_\rho[P_{c_1}^R], \Delta_\rho[P_{c_2}^R], \dots, \Delta_\rho[P_{c_\sigma}^R])$ and $\mathcal{F} = (\Delta_\rho[T_{c_1}], \Delta_\rho[T_{c_2}], \dots, \Delta_\rho[T_{c_\sigma}])$, where $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$. The algorithm initializes an instance of the procedure of Lemma 13. After the arrival of $T[i]$, if $T[i] \neq T[i - \rho]$, then the algorithm sets $\Delta_\rho[T_{T[i]}](i)$ to be 1 and $\Delta_\rho[T_{T[i-\rho]}](i)$ to be -1 . The algorithm transfers these values to the procedure of Lemma 13, which responds with the value of $[\mathcal{F} \otimes \mathcal{G}](i)$. Then, the algorithm reports $[T \otimes P](i)$, which equals $[\mathcal{F} \otimes \mathcal{G}](i) - [T \otimes P](i - 2\rho) + 2[T \otimes P](i - \rho)$ by Lemma 6. In this expression, the first term is returned by the procedure of Lemma 13 and the other two terms are retrieved from the buffer.

The update of the text buffer and $\Delta_\rho[T_c]$ for at most two characters c after the arrival of any text character costs $O(1)$ time per character and $O(n)$ time in total. Note that by Observation 5, since $\rho \leq k$ is a d -period of both P and T , we have that $\|\mathcal{G}\| \leq 2(d + k) = O(k)$ and $\|\mathcal{F}\| \leq 2(d + k) = O(k)$. Moreover, $\delta = \max(\max_i \|\mathcal{F}|_{\{i\}}\|, \max_i \|\mathcal{G}|_{\{i\}}\|) \leq 2$. Consequently, the algorithm of Lemma 13 uses $\tilde{O}(m)$ space and costs $\tilde{O}(\sqrt{k})$ time per character and $\tilde{O}(\min(n + k\sqrt{n}, n\sigma))$ time in total. All the other parts of the algorithm cost $O(1)$ time per character and $O(n)$ time in total. ◀

Thus, by combining Lemma 12 and Lemma 14, we obtain an algorithm that, given ρ which is a d -period of both P and T , computes the Hamming distances up to k for every substring of the text without delay.

► **Lemma 15.** *Suppose that there exists $\rho \leq k$ which is a d -period of both P and T for some $d = O(k)$. Then, there exists a deterministic streaming algorithm for the k -mismatch problem with $n = \frac{3}{2}m$ that, given an integer parameter $k \leq s \leq m$, uses $\tilde{O}(s)$ space, and costs $\tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$ total time and $\tilde{O}(\sqrt{k})$ time per character in the worst case.*

Proof. Let P_{tail} be the suffix of P of length $2s$, and let P_{head} be the complementary prefix of P . Note that ρ is a d -period of both P_{head} and P_{tail} . We run the algorithm of Lemma 12 with P_{head} as a pattern. The results of the algorithm of Lemma 12 are added into a buffer of length $2s$. Thus, right after the arrival of $T[i]$, it is guaranteed that $\text{Ham}(P_{head}, T[i - |P| + 1 \dots i - |P_{tail}|])$ is already computed and available for the algorithm. In addition, the algorithm of Lemma 14 is executed with P_{tail} as pattern, and this algorithm computes $\text{Ham}(P_{tail}, T[i - |P_{tail}| + 1 \dots i])$ right after the arrival of $T[i]$, if this value is at most k . After the arrival of $T[i]$, the algorithm reports $\text{Ham}(P, T[i - |P| + 1 \dots i]) = \text{Ham}(P_{head}, T[i - |P| + 1 \dots i - |P_{tail}|]) + \text{Ham}(P_{tail}, T[i - |P_{tail}| + 1 \dots i])$.

The time per character of both the algorithm of Lemma 12 and the algorithm of Lemma 14 is $\tilde{O}(\sqrt{k})$. The total time cost of the algorithm of Lemma 12 is $\tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$. Furthermore, the total time cost of the algorithm of Lemma 14 is $\tilde{O}(\min(m + k\sqrt{m}, m\sigma)) = \tilde{O}\left(m + \min\left(k^2, \frac{km}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$ due to $k \leq s \leq m$ and $m + k\sqrt{m} = O(m + k^2)$. Hence, the lemma follows. ◀

6 Periodic Pattern and Arbitrary Text – without Delay

In this section, we generalize Lemma 15 to the case where ρ is not necessarily a d -period of T , but ρ is still a d -period of P and $n = \frac{3}{2}m$. We use ideas building upon Clifford et al. [7, Lemma 6.2] to show that there exists a substring of T , denoted T^* , such that ρ is a $(2d + 4k + \rho)$ -period of T^* , and T^* contains all of the k -mismatch occurrences of P in T . Our construction, specified below, exploits the following property of approximate periods.

► **Observation 16.** *Let X and Y be two equal length strings. If ρ is a d -period of X and $\text{Ham}(X, Y) \leq x$ then ρ is a $(d + 2x)$ -period of Y .*

Let T_L be the longest suffix of $T[0.. \frac{1}{2}m - 1]$ such that ρ is a $(d + 2k)$ -period of T_L , and let T_R be the longest prefix of $T[\frac{1}{2}m.. n - 1]$ such that ρ is a $(d + 2k)$ -period of T_R . Finally, let T^* be the concatenation $T^* = T_L \cdot T_R$.

► **Lemma 17.** *All the k -mismatch occurrences of P in T are contained within T^* . Moreover, ρ is a $(2d + 4k + \rho)$ -period of T^* .*

Proof. The second claim follows directly from the fact that $T^* = T_L \cdot T_R$ is a concatenation of two strings with $(d + 2k)$ -period ρ (the extra ρ mismatches might occur at the boundary between T_L and T_R). Henceforth, we focus on the first claim.

We assume that P has at least one k -mismatch occurrence in T ; otherwise, the claim holds trivially. Let $T[\ell.. r]$ be the smallest fragment of T containing all the k -mismatch occurrences of P in T (so that the leftmost and the rightmost occurrences starts at positions ℓ and $r - m + 1$, respectively). Our goal is to prove that $T[\ell.. r]$ is contained within T^* .

By Observation 16, since $T[\ell.. \ell + m - 1]$ is a k -mismatch occurrence of P and ρ is a d -period of P , it must be that ρ is a $(d + 2k)$ -period of $T[\ell.. \ell + m - 1]$. In particular, since $\ell + m \geq \frac{1}{2}m$, we have that ρ is a $(d + 2k)$ -period of $T[\ell.. \frac{1}{2}m - 1]$. Hence, by its maximality, T_L must start at position ℓ or to the left of ℓ . Similarly, by Observation 16, since $T[r - m + 1.. r]$ is a k -mismatch occurrence of P and ρ is a d -period of P , it must be that ρ is a $(d + 2k)$ -period of $T[r - m + 1.. r]$. In particular, since $r - m + 1 \leq n - m \leq \frac{1}{2}m$, we have that ρ is a $(d + 2k)$ -period of $T[\frac{1}{2}m.. r]$. Hence, by its maximality, T_R must end at position r or to the right of r . ◀

The algorithm works in two high-level phases. In the first phase, the algorithm receives $T[0.. \frac{1}{2}m - 1]$, and the goal is to compute T_L . In the second phase, the algorithm receives $T[\frac{1}{2}m.. n - 1]$ and transfers $T^* = T_L \cdot T_R$ to the subroutine of Lemma 12. The transfer starts with a delay of $|T_L|$ characters and a standard de-amortization speedup is applied to reduce the delay to 0 by the time $2|T_L| \leq m$ characters are transferred, which is before the subroutine of Lemma 12 may start producing output. The algorithm terminates as soon as it reaches the end of T_R , i.e., when it encounters more than $d + 2k$ mismatches in $T[\frac{1}{2}m.. n - 1]$.

The following *periodic representation* (similar to one by Clifford et al. [8]) is used for storing substrings of T .

► **Fact 18.** *For every positive integer ρ , there exists an algorithm that maintains a representation of $T[\ell.. r]$ and supports the following operations in $O(1)$ time each:*

- (a) *Change the representation to represent $T[\ell + 1.. r]$ and return $T[\ell]$.*
- (b) *Given $T[r + 1]$ and $T[r + 1 - \rho]$, change the representation to represent $T[\ell.. r + 1]$.*
- (c) *Given $\ell' \geq \ell$ such that $T[i] = T[i + \rho]$ for $\ell \leq i < \ell'$, change the representation to represent $T[\ell'.. r]$.*

If ρ is a d -period of $T[\ell.. r]$ then the space usage is $O(d + \rho)$.

15:12 The Streaming k -Mismatch Problem: Tradeoffs Between Space and Total Time

Proof. $T[\ell..r]$ is represented by a string S of length ρ such that $T[i] = S[i \bmod \rho]$ for $\ell \leq i \leq \min(\ell + \rho - 1, r)$, and a list $\mathbf{L} = \{(i, T[i]) : \ell + \rho \leq i \leq r, T[i - \rho] \neq T[i]\}$. Notice that if ρ is a d -period of $T[\ell..r]$, then this representation uses $O(d + \rho)$ space.

To implement operation (a), the algorithm first retrieves $T[\ell] = S[\ell \bmod \rho]$. The algorithm then checks if the leading element of \mathbf{L} is $(\ell + \rho, T[\ell + \rho])$. If so, the algorithm removes this pair from \mathbf{L} and sets $S[\ell \bmod \rho] = T[\ell + \rho]$. To implement operation (b), the algorithm compares $T[r + 1]$ with $T[r + 1 - \rho]$. If these values are different, then $(r + 1, T[r + 1])$ is appended to \mathbf{L} . The implementation of operation (c) is trivial (neither S nor \mathbf{L} is changed). ◀

► **Lemma 19.** *Suppose that there exists $\rho \leq k$ which is a d -period of P for some $d = O(k)$. Then, there exists a deterministic streaming algorithm for the k -mismatch problem with $n = \frac{3}{2}m$ that, given an integer parameter $k \leq s \leq m$, uses $\tilde{O}(s)$ space, and costs $\tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$ total time and $\tilde{O}(\sqrt{k})$ time per character in the worst case.*

Proof. Based on the pattern P and the period ρ , the algorithm initializes an instance **ALG** of the algorithm of Lemma 12. Then, the algorithm processes T in two phases, while maintaining a buffer of ρ text characters and the representation of Fact 18 of a suffix T' of the already processed prefix of T .

First Phase. During the first phase, when the algorithm receives $T[0.. \frac{1}{2}m - 1]$, the suffix T' is defined as the longest suffix for which ρ is a $(d + 2k)$ -period. Suppose T' is $T[\ell..i - 1]$ after processing $T[0..i - 1]$. The algorithm first appends $T[i]$ to T' , extending it to $T[\ell..i]$ using Fact 18(b). If ρ is still a $(d + 2k)$ -period of T' , i.e., the list \mathbf{L} has at most $d + 2k$ elements, then the algorithm proceeds to the next character. Otherwise, T' is first trimmed to $T[\ell'..i]$, where $(\ell' + \rho, T[\ell' + \rho])$ is the first element of \mathbf{L} , using Fact 18(c), and then to $T[\ell' + 1..i]$ using Fact 18(a). The latter operation decrements the size of \mathbf{L} to $d + 2k$.

At the end of the first phase, T' is by definition equal to T_L . The running time of the algorithm in the first phase is $O(1)$ per character, and the space complexity is $O(d + k) = O(k)$.

Second Phase. At the second phase, the algorithm receives $T[\frac{m}{2}..n - 1]$, while counting the number of mismatches with respect to ρ . As soon as this number exceeds $d + 2k$, which happens immediately after receiving the entire string T_R , the algorithm stops. As long as T' is non-empty, each input character is appended to T' using Fact 18(b), and the two leading characters of T' are popped using Fact 18(a) and transferred to **ALG**. Once T' becomes empty (which is after **ALG** receives $2|T_L| \leq m$ characters), the input characters are transferred directly to **ALG**. This process guarantees that the input to **ALG** is T^* and that **ALG** is executed with no delay by the time the first m characters of T^* are passed. By Lemma 17, all k -mismatch occurrences of P in T are contained in T^* , so all these occurrences are reported in a timely manner.

Since ρ is a $(2d + 4k + \rho)$ -period of T^* (by Lemma 17) and T' is contained in T , then ρ is also a $(2d + 4k + \rho)$ -period of T' at all times. Consequently, the space complexity is $O(k)$ on top of the space usage of **ALG**, which is $\tilde{O}(s)$. Thus, in total, the algorithm uses $\tilde{O}(s)$ space.

The per-character running time is dominated by the time cost of **ALG**, which is $\tilde{O}(\sqrt{k})$. The total running time of the algorithm is also dominated by the total running time of **ALG**, which, by Lemma 12, is $\tilde{O}\left(m + \min\left(k^2, \frac{mk}{\sqrt{s}}, \frac{\sigma m^2}{s}\right)\right)$. ◀

The following corollary is obtained from Lemma 19 by the standard trick of splitting the text into $O(\frac{n}{m})$ substrings of length $\frac{3}{2}m$ with overlaps of length m .

► **Corollary 20.** *Suppose that there exists $\rho \leq k$ which is a d -period of P . Then, there exists a deterministic streaming algorithm for the k -mismatch problem that, given an integer parameter $k \leq s \leq m$, uses $\tilde{O}(s)$ space and costs $\tilde{O}\left(n + \min\left(\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}\right)\right)$ total time. Moreover, the worst-case time cost per character is $\tilde{O}(\sqrt{k})$.*

7 Aperiodic Pattern and Arbitrary Text

The following lemma, proved in the full version, appears in [16] with small modifications.

► **Lemma 21** (Based on [16, Theorem 5]). *Suppose that the smallest $4k$ -period of the pattern P is $\Omega(k)$. Then, there exists a randomized streaming algorithm for the k -mismatch problem that uses $\tilde{O}(k)$ space and costs $\tilde{O}(1)$ time per character. The algorithm has delay of k characters and is correct with high probability.*

In order to improve the algorithm to report results without any delay, we use ideas similar to those introduced in Section 5. The following fact appeared in [7].

► **Fact 22** (Based on [7, Fact 3.1]). *If ρ is the smallest d -period of a pattern P , then the $\frac{1}{2}d$ -mismatch occurrences of P in any text T start at least ρ positions apart.*

► **Lemma 23.** *Suppose that the smallest $6k$ -period of the pattern P is $\Omega(k)$. Then, there exists a randomized streaming algorithm for the k -mismatch problem that uses $\tilde{O}(k)$ space and costs $\tilde{O}(1)$ time per character. The algorithm is correct with high probability.*

Proof. Let P_{tail} be the suffix of P of length $2k$ and let P_{head} be the complementary prefix of P . Since the smallest $6k$ -period of P is $\Omega(k)$, $|P_{tail}| = 2k$, and $6k - 2k = 4k$, the smallest $4k$ -period of P_{head} is also $\Omega(k)$. Thus, we execute the procedure of Lemma 21 with P_{head} . Then, whenever the procedure reports $\text{Ham}(P_{head}, T[i - |P| + 1 .. i - 2k])$ to be at most k , the algorithm starts a process that computes $\text{Ham}(P_{tail}, T[i - 2k + 1 .. i])$. The procedure of Lemma 21 reports $\text{Ham}(P_{head}, T[i - |P| + 1 .. i - 2k])$ before $T[i - k + 1]$ arrives. Hence, there are still at least k character arrivals until $\text{Ham}(P, T[i - |P| + 1 .. i])$ has to be reported. During these character arrivals, the computation of $\text{Ham}(P_{tail}, T[i - 2k + 1 .. i])$ is done simply by comparing pairs of characters. The total time of this computation is $O(k)$, and by standard de-amortization, this is $O(1)$ time per character during the arrival of the k characters.

Since the smallest $4k$ -period of P is $\Omega(k)$, Fact 22 implies that any two k -mismatch occurrences of P in T are at distance $\Omega(k)$. Therefore, the maximum number of processes computing distances to P_{tail} at any time is $O(1)$. Thus, the time cost per character of the algorithm is dominated by the procedure of Lemma 21. ◀

8 Proof of Main Theorem

We conclude the paper with a proof of Theorem 3, which is our main result. In the preprocessing, the shortest $6k$ -period ρ of the pattern P is determined. If $\rho \leq k$, then the text is processed using Corollary 20. This procedure uses $\tilde{O}(s)$ space and costs $\tilde{O}(\sqrt{k})$ time per character and $\tilde{O}\left(n + \min\left(\frac{nk^2}{m}, \frac{nk}{\sqrt{s}}, \frac{\sigma nm}{s}\right)\right)$ time in total. Otherwise, the text is processed based on the Lemma 23. The space complexity in this case is $\tilde{O}(k) = \tilde{O}(s)$, whereas the running time is $\tilde{O}(1) = \tilde{O}(\sqrt{k})$ per character and $\tilde{O}(n)$ in total.

References

- 1 Karl R. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987. doi:10.1137/0216067.
- 2 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.
- 3 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.
- 4 Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern Hamming distances. In *52nd Annual ACM Symposium on Theory of Computing, STOC 2020*, 2020. arXiv:2001.00211.
- 5 Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. *Information and Computation*, 209(4):731–736, 2011. doi:10.1016/j.ic.2010.12.007.
- 6 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In Nikhil Bansal and Irene Finocchi, editors, *23rd Annual European Symposium on Algorithms, ESA 2015*, volume 9294 of *LNCS*, pages 361–372. Springer, 2015. doi:10.1007/978-3-662-48350-3_31.
- 7 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k -mismatch problem revisited. In Robert Krauthgamer, editor, *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2039–2052. SIAM, 2016. doi:10.1137/1.9781611974331.ch142.
- 8 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k -mismatch problem. In Timothy M. Chan, editor, *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1106–1125. SIAM, 2019. doi:10.1137/1.9781611975482.68.
- 9 Raphaël Clifford and Benjamin Sach. Pseudo-realtime pattern matching: Closing the gap. In Amihood Amir and Laxmi Parida, editors, *21st Annual Symposium on Combinatorial Pattern Matching, CPM 2010*, volume 6129 of *LNCS*, pages 101–111. Springer, 2010. doi:10.1007/978-3-642-13509-5_10.
- 10 Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002. doi:10.1137/S0097539700370527.
- 11 Michael J. Fischer and Michael S. Paterson. String matching and other products. In Richard M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125. Providence, RI, 1974. AMS.
- 12 Zvi Galil and Raffaele Giancarlo. Parallel string matching with k mismatches. *Theoretical Computer Science*, 51:341–348, 1987. doi:10.1016/0304-3975(87)90042-9.
- 13 Paweł Gawrychowski and Tatiana Starikovskaya. Streaming dictionary matching with mismatches. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM*, volume 128 of *LIPICs*, pages 21:1–21:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.21.
- 14 Paweł Gawrychowski and Przemysław Uznański. Personal communication, November 2018.
- 15 Paweł Gawrychowski and Przemysław Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPICs*, pages 62:1–62:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ICALP.2018.62.
- 16 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPICs*, pages 65:1–65:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ICALP.2018.65.

- 17 Shay Golan and Ely Porat. Real-time streaming multi-pattern search for constant alphabet. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017*, volume 87 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.41.
- 18 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.
- 19 Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989. doi:10.1016/0196-6774(89)90010-2.
- 20 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 315–323. IEEE Computer Society, 2009. doi:10.1109/FOCS.2009.11.
- 21 Jakub Radoszewski and Tatiana Starikovskaya. Streaming k -mismatch with error correcting and applications. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *2017 Data Compression Conference, DCC 2017*, pages 290–299. IEEE, 2017. doi:10.1109/DCC.2017.14.
- 22 Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual Symposium on Foundations of Computer Science, FOCS*, pages 320–328. IEEE Computer Society, 1996. doi:10.1109/SFCS.1996.548491.
- 23 Tatiana Starikovskaya, Michal Svagerka, and Przemysław Uznański. L_p pattern matching in a stream, 2019. arXiv:1907.04405.

Approximating Longest Common Substring with k mismatches: Theory and Practice

Garance Gourdel

ENS Paris Saclay, France
garance.gourdel@ens-paris-saclay.fr

Tomasz Kociumaka 

Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Jakub Radoszewski 

Institute of Informatics, University of Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
jrad@mimuw.edu.pl

Tatiana Starikovskaya

DIENS, École normale supérieure, PSL Research University, France
tat.starikovskaya@gmail.com

Abstract

In the problem of the longest common substring with k mismatches we are given two strings X, Y and must find the maximal length ℓ such that there is a length- ℓ substring of X and a length- ℓ substring of Y that differ in at most k positions. The length ℓ can be used as a robust measure of similarity between X, Y . In this work, we develop new approximation algorithms for computing ℓ that are significantly more efficient than previously known solutions from the theoretical point of view. Our approach is simple and practical, which we confirm via an experimental evaluation, and is probably close to optimal as we demonstrate via a conditional lower bound.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases approximation algorithms, string similarity, LSH, conditional lower bounds

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.16

Related Version A full version of the paper is available at <https://arxiv.org/abs/2004.13389>.

Supplementary Material To ensure the reproducibility of our results, our complete experimental setup, including data files, is available at https://github.com/fnareoh/LCS_Approx_k_mis.

Funding *Tomasz Kociumaka*: Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.

Jakub Radoszewski: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

1 Introduction

For decades, the edit distance and its variants remained the most relevant measure of similarity between biological sequences. However, there is strong evidence that the edit distance cannot be computed in strongly subquadratic time [7]. One possible approach to overcoming the quadratic time barrier is computing the edit distance approximately, and last year in the breakthrough paper Chakraborty et al. [8] showed a constant-factor approximation algorithm that computes the edit distance between two strings of length n in time $\tilde{O}(n^{2-2/7})$. Nevertheless, the algorithm is highly non-trivial and because of that is likely to be impractical.



© Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya; licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 16; pp. 16:1–16:15

Leibniz International Proceedings in Informatics



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

A different approach is to consider alignment-free measures of similarities. Ideally, we want the measure to be robust and simple enough so that we could compute it efficiently. One candidate for such a measure is the length of the longest common substring with k mismatches. Formally, given two strings X, Y of lengths at most n and an integer k , we want to find the maximal length $\text{LCS}_k(X, Y)$ of a substring of X that occurs in Y with at most k mismatches. Computing this value constitutes the LCS with k Mismatches problem.

The LCS with k Mismatches problem was first considered for $k = 1$ [6, 13], with current best algorithm taking $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. The first algorithm for the general value of k was shown by Flouri et al. [13]. Their simple approach used quadratic time and linear space. Grabowski [15] focused on a data-dependent approach, namely, he showed two linear-space algorithms with running times $\mathcal{O}(n((k+1)(\text{LCS}+1))^k)$ and $\mathcal{O}(n^2 k / \text{LCS}_k)$, where LCS is the length of the longest common substring of X and Y and LCS_k , similarly to above, is the length of the longest common substring with k mismatches of X and Y . Abboud et al. [1] showed a $k^{1.5} n^2 / 2^{\Omega(\sqrt{(\log n)/k})}$ -time randomised solution to the problem via the polynomial method. Thankachan et al. [24] presented an $\mathcal{O}(n \log^k n)$ -time, $\mathcal{O}(n)$ -space solution for constant k . This approach was recently extended by Charalampopoulos et al. [10] to develop an $\mathcal{O}(n)$ -time and $\mathcal{O}(n)$ -space algorithm for the case of $\text{LCS}_k = \Omega(\log^{2k+2} n)$.

On the other hand, Kociumaka, Radoszewski, and Starikovskaya [19] showed that there is $k = \Theta(\log n)$ such that the LCS with k Mismatches problem cannot be solved in strongly subquadratic time, even for the binary alphabet, unless the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi, and Zane [16] is false. This conditional lower bound implies that there is little hope to improve existing solutions to LCS with k Mismatches. To overcome this barrier, they introduced an approximation approach to LCS with k Mismatches, inspired by the work of Andoni and Indyk [4].

► **Problem 1** (LCS with Approximately k Mismatches). *Two strings X, Y of length at most n , an integer k , and a constant $\varepsilon > 0$ are given. Return a substring of X of length at least $\text{LCS}_k(X, Y)$ that occurs in Y with at most $(1 + \varepsilon) \cdot k$ mismatches.*

Kociumaka, Radoszewski, and Starikovskaya [19] also showed that for any $\varepsilon \in (0, 2)$ the LCS with Approximately k Mismatches problem can be solved in $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time and $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space. Besides for superlinear space, their solution uses a very complex class of hash functions which requires $n^{4/3+o(1)}$ -time preprocessing, and that is the underlying reason for the bounds on ε . In this work, we significantly improve the complexity of the LCS with Approximately k Mismatches problem and show the following results.

► **Theorem 2.** *Let $\varepsilon > 0$ be an arbitrary constant. The LCS with Approximately k Mismatches problem can be solved correctly with high probability:*

- 1) *In $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$ time and $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$ space assuming a constant-size alphabet;*
- 2) *In $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$ time and $\mathcal{O}(n)$ space for alphabets of arbitrary size.*

Our first solution uses the Approximate Nearest Neighbour data structure [5] as a black box. The definition of this data structure is extremely involved, and we view this result as more of a theoretical interest. On the other hand, our second solution is simple and practical, which we confirm by experimental evaluation (see Section 4 for details).

As a final remark, we note that a construction similar to the one used to show a lower bound for the LCS with k Mismatches problem [19] gives a lower bound for LCS with Approximately k Mismatches. A proof of the following fact can be found in Section 5.

► **Fact 3.** Assuming *SETH*, for every constant $\delta > 0$, there exists a constant $\varepsilon = \varepsilon(\delta)^1$ such that any randomised algorithm that solves the *LCS with Approximately k Mismatches* problem for given X and Y of length at most n correctly with constant probability uses $\Omega(n^{2-\delta})$ time.

Related work. In 2014, Leimester and Morgenstern [20] introduced a related similarity measure, the *k -macs distance*. Let $\text{LCP}_k(X_i, Y_j) = \max\{\ell : d_H(X[i, i+\ell-1], Y[j, j+\ell-1]) \leq k\}$, where d_H stands for Hamming distance, i.e. the number of mismatches between two strings. We have $\text{LCS}_k = \max_{i,j} \text{LCP}_k(X_i, Y_j)$. The *k -macs distance*, on the other hand, is defined as a normalised average of these values. Leimester and Morgenstern [20] showed a heuristic algorithm for computing the *k -macs distance*, with no theoretical guarantees for the precision of the approximation; other heuristic approaches for computing the *k -macs distance* include [25, 26]. The only algorithm with provable theoretical guarantees is [24] and it computes the *k -macs distance* in $\mathcal{O}(n \log^k n)$ time and $\mathcal{O}(n)$ space.

2 Preliminaries

We assume that the alphabet of the strings X, Y is $\Sigma = \{1, \dots, \sigma\}$, where $\sigma = n^{\mathcal{O}(1)}$.

Karp–Rabin fingerprints. The Karp–Rabin fingerprint [18] of a string $S = s_1 s_2 \dots s_\ell$ is defined as

$$\varphi(S) = \left(\sum_{i=1}^{\ell} r^{i-1} s_i \right) \bmod q,$$

where $q = \Omega(\max\{n^5, \sigma\})$ is a prime number, and $r \in \mathbb{F}_q$ is chosen uniformly at random. Obviously, if $S_1 = S_2$, then $\varphi(S_1) = \varphi(S_2)$. Furthermore, for any $\ell \leq n$, if the fingerprints of two ℓ -length strings S_1, S_2 are equal, then S_1, S_2 are equal with probability at least $1 - 1/n^4$ (for a proof, see e.g. [21]).

Dimension reduction. We will exploit a computationally efficient variant of the Johnson–Lindenstrauss lemma [17] which describes a low-distortion embedding from a high-dimensional Euclidean space into a low-dimensional one. Let $\|\cdot\|$ be the Euclidean (L_2) norm of a vector. We will exploit the following claim which follows immediately from [2, Theorem 1.1]:

► **Lemma 4.** Let P be a set of n vectors in \mathbb{R}^ℓ , where $\ell \leq n$. Given $\alpha = \alpha(n) > 0$ and a constant $\beta > 0$, there is $d = \Theta(\alpha^{-2} \log n)$ and a scalar $c > 0$ such that the following holds. Let M be a $d \times \ell$ matrix filled with i.u.d. ± 1 random variables. For all $U \in P$, define $\text{sk}_\alpha(U) = c \cdot MU$. Then for all $U, V \in P$ there is $\|U - V\|^2 \leq \|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)\|U - V\|^2$ with probability at least $1 - n^{-\beta}$.

Since the Hamming distance between binary strings U, V is equal to $\|U - V\|^2$, the matrix M defines a low-distortion embedding from an ℓ -dimensional into a d -dimensional Hamming space as well. For non-binary strings, an extra step is required. Let the alphabet be $\Sigma = \{1, 2, \dots, \sigma\}$ and consider a morphism $\mu : \Sigma \rightarrow \{0, 1\}^\sigma$, where $\mu(a) = 0^{a-1} 1 0^{\sigma-a}$ for all $a \in \Sigma$. We extend μ to strings in a natural way. Note that for two strings U, V over the alphabet Σ the Hamming distance between $\mu(U), \mu(V)$ is exactly twice the Hamming distance between U, V . We therefore obtain:

¹ Here δ is a function of ε for which the explicit form is not known (a condition inherited from [22]).

16:4 Approximating Longest Common Substring with k Mismatches

► **Corollary 5.** *Let P be a set of n strings in Σ^ℓ , where $\ell \leq n$. Given $\alpha = \alpha(n) > 0$ and a constant $\beta > 0$, there is $d = \Theta(\alpha^{-2} \log n)$ and a scalar $c > 0$ such that the following holds. Let M be a $d \times (\sigma \cdot \ell)$ matrix filled with i.u.d. ± 1 random variables. For all $U \in P$, define $\text{sk}_\alpha(U) = c \cdot M\mu(U)$. Then for all $U, V \in P$ there is $d_H(U, V) \leq \|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)d_H(U, V)$ with probability at least $1 - n^{-\beta}$.*

We will use the corollary for dimension reduction, and also to design a simple test that checks whether the Hamming distance between two strings is at most k .

► **Corollary 6.** *Let P be a set of n strings in Σ^ℓ , where $\ell \leq n$. With probability at least $1 - n^{-\beta}$, for all $U, V \in P$:*

- 1) if $\|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)k$, then $d_H(U, V) \leq (1 + \alpha) \cdot k$;
- 2) if $\|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 > (1 + \alpha)k$, then $d_H(U, V) \geq k$.

2.1 The Twenty Questions game

Consider the following version of the classic game “Twenty Questions”. There are two players: Paul and Carole; Carole thinks of two numbers A, B between 0 and N , and Paul must return some number in $[A, B]$. He is allowed to ask questions of form “Is $x \leq A$?”, for any $x \in [0, N]$. If $x \leq A$, Carole must return YES; If $A < x \leq B$, she can return anything; and if $B < x$, she must return NO. Paul must return the answer after having asked at most Q questions where Carole can tell at most $\lceil \rho Q \rceil$ lies, and only in the case when $x \leq A$.

We show that Paul has a winning strategy for $Q = \Theta(\log n)$ and any $\rho < 1/3$ by a black-box reduction to the result of Dhagat, Gács, and Winkler [11] who showed a winning strategy for $A = B$.

► **Theorem 7** ([11]). *For $A = B$, Paul has a winning strategy for all $\rho < \frac{1}{3}$ asking $Q = \lceil \frac{8 \log N}{(1-3\rho)^2} \rceil$ questions.*

This result is obtained by maintaining a stack of trusted intervals. Once Paul knows that A is between ℓ and r , where $\ell \leq r$, he checks whether A is in the left or the right half of the interval $[\ell, r]$. If no inconsistencies appear (like $A < \ell$ or $r < A$), he pushes the new interval to the stack, else he removes the interval $[\ell, r]$ from the stack of trusted intervals. After Q rounds, Paul returns the only number in the top interval in the stack, which is guaranteed to have length 1 and to contain A . We give the pseudocode of Paul’s strategy in Algorithm 1. By $\text{Carole}(x)$, we denote the answer of Carole for a question “Is $x \leq A$?”.

■ **Algorithm 1** The Twenty Questions game.

```

1:  $Q \leftarrow \lceil \frac{8 \log N}{(1-3\rho)^2} \rceil$ 
2:  $S \leftarrow \{[0, N]\}$ 
3: for  $i = 1, 2, \dots, Q/2$  do
4:    $I = [\ell, r] \leftarrow S.\text{top}()$ 
5:    $\text{mid} \leftarrow \lceil \frac{\ell+r}{2} \rceil$ 
6:   if  $\text{Carole}(\text{mid})$  then
7:     if  $\text{Carole}(r)$  then  $S.\text{pop}()$            ▷ The answer is inconsistent with  $I$ ; remove  $I$ .
8:     else  $S.\text{push}([\text{mid}, r])$ 
9:   else
10:    if  $\text{Carole}(\ell)$  then  $S.\text{push}([\ell, \text{mid} - 1])$ 
11:    else  $S.\text{pop}()$                              ▷ The answer is inconsistent with  $I$ ; remove  $I$ .

```

We now show a winning strategy for our variant of the game.

► **Corollary 8.** For $A \leq B$, Paul has a winning strategy for all $\rho < \frac{1}{3}$ asking $Q = \frac{8 \log N}{(1-3\rho)^2}$ questions.

Proof. We introduce just one change to Algorithm 1, namely, we return the argument of the largest YES obtained in the course of the algorithm. From the problem statement it follows that the answer is at most B . We shall now prove that the answer is at least A . If Carole ever returned YES for $A < x \leq B$, then it is obviously the case. Otherwise, Carole actually behaved as if she had $A = B$ in mind: apart from the small fraction of erroneous answers, she returned YES for $x \leq A$, and NO for $x > A$. Thus, the strategy of Dhagat, Gács, and Winkler ends up with A as the answer (and this must be due to a YES for $x = A$). ◀

3 LCS with Approximately k Mismatches

In this section, we prove Theorem 2. Let us first introduce a decision variant of the LCS with Approximately k Mismatches problem.

► **Problem 9.** Two strings X, Y of length at most n , integers k, ℓ , and a constant $\varepsilon > 0$ are given. We must return:

1. YES if $\ell \leq \text{LCS}_k(X, Y)$;
2. Anything if $\text{LCS}_k(X, Y) < \ell \leq \text{LCS}_{(1+\varepsilon)k}(X, Y)$;
3. NO if $\text{LCS}_{(1+\varepsilon)k}(X, Y) < \ell$.

If we return YES, we must also give a witness pair of length- ℓ substrings S_1 and S_2 of X and Y , respectively, such that $d_H(S_1, S_2) \leq (1 + \varepsilon)k$.

The decision variant of the LCS with Approximately k Mismatches problem can be reduced to the following (c, r) -Approximate Near Neighbour problem.

► **Problem 10.** In the (c, r) -Approximate Near Neighbour problem with failure probability f , the aim is, given a set P of n points in \mathbb{R}^d , to construct a data structure supporting the following queries: given any point $q \in \mathbb{R}^d$, if there exists $p \in P$ such that $\|p - q\| \leq r$, then return some point $p' \in P$ such that $\|p' - q\| \leq cr$ with probability at least $1 - f$.

Using the reduction, we will show our first solution to the LCS with Approximately k Mismatches decision problem based on the result of Andoni and Razenshteyn [5], who showed that for any constant f , there is a data structure for the (c, r) -Approximate Near Neighbour problem that has $\mathcal{O}(n^{1+\rho+o(1)} + d \cdot n)$ size, $\mathcal{O}(d \cdot n^{\rho+o(1)})$ query time, and $\mathcal{O}(d \cdot n^{1+\rho+o(1)})$ preprocessing time, where $\rho = 1/(2c^2 - 1)$.

► **Lemma 11.** Assume an alphabet of constant size σ . The decision variant of the LCS with Approximately k Mismatches problem can be solved in space $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$ and time $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$. The answer is correct with constant probability.

Proof. Let P be the set of all length- ℓ substrings of X and Q be the set of all length- ℓ substrings of Y , all encoded in binary using the morphism μ (see Section 2). We start by applying the dimension reduction procedure of Corollary 5 to P and Q with $\alpha = 1/(\log \log n)^{\Theta(1)}$ and $\beta = 2$ to obtain sets P' and Q' . We can implement the procedure in $\mathcal{O}(\sigma n \log^2 n (\log \log n)^{\Theta(1)}) = \mathcal{O}(n \log^{2+o(1)} n)$ time by encoding X, Y using μ and running the FFT algorithm [12] for each of the $\mathcal{O}(\log^{1+o(1)} n)$ rows of the matrix and $\mu(X), \mu(Y)$.

To solve the decision variant of LCS with Approximately k Mismatches, we build the data structure of Andoni and Razenshteyn [5] for $(\sqrt{(1+\varepsilon)(1-\alpha)}, \sqrt{(1+\alpha)k})$ -Approximate Near Neighbour over Q' . We make a query for each string in P' . If, queried for $\text{sk}_\alpha(S_1) \in P'$, where S_1 is a length- ℓ substring of X , the data structure outputs $\text{sk}_\alpha(S_2) \in Q'$, where S_2 is

a length- ℓ substring of Y , then we compute $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S_2)\|^2$. If it is at most $(1 + \varepsilon)k$, we output YES and the witness pair (S_1, S_2) of substrings. As the length of vectors in P' , Q' is $d = \mathcal{O}(\log^{1+o(1)} n)$, we obtain the desired complexity.

To show that the algorithm is correct, suppose that there are length- ℓ substrings S_1 and S_2 of X and Y , respectively, with $d_H(S_1, S_2) \leq k$. By Corollary 5, $\|\text{sk}_\alpha(S_1), \text{sk}_\alpha(S_2)\| \leq \sqrt{(1 + \alpha)k}$ holds with probability at least $1 - 1/n$. Then, when querying for $\text{sk}_\alpha(S_1)$, with constant probability the data structure will output a string $\text{sk}_\alpha(S'_2)$ such that $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S'_2)\|^2 \leq (1 + \varepsilon)(1 - \alpha^2)k \leq (1 + \varepsilon)k$. Then, our algorithm will return YES.

On the other hand, if we output YES with a witness pair (S_1, S_2) , then $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S_2)\|^2 \leq (1 + \varepsilon)k$ implies $d_H(S_1, S_2) \leq (1 + \varepsilon)k$ with high probability by Corollary 5. ◀

While this solution is very fast, it uses quite a lot of space. Furthermore, the data structure of [5] that we use as a black box applies highly non-trivial techniques. To overcome these two disadvantages, we will show a different solution based on a careful implementation of ideas first introduced in [4] that showed a data structure for approximate text indexing with mismatches. In [19], the authors developed these ideas further to show an algorithm that solves the LCS with Approximately k Mismatches problem in $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space and $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time for $\varepsilon \in (0, 2)$ with constant error probability. In this work, we significantly improve and simplify the approach to show the following result:

► **Theorem 12.** *Assume an alphabet of arbitrary size $\sigma = n^{\mathcal{O}(1)}$. The decision variant of LCS with Approximately k Mismatches can be solved in $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time and $\mathcal{O}(n)$ space. The answer is correct with constant probability.*

Let us defer the proof of the theorem until Section 3.1 and start by explaining how we use Lemma 11 and Theorem 12 and the Twenty Questions game to show Theorem 2.

Proof of Theorem 2. We will rely on the modified version of the Twenty Questions game that we described in Section 2.1. In our case, $A = \text{LCS}_k(X, Y)$ and $B = \text{LCS}_{(1+\varepsilon)k}(X, Y)$. For Carole, we use either the algorithm of Lemma 11, or the algorithm of Theorem 12, with an additional procedure verifying the witness pair (S_1, S_2) character by character to check that it indeed satisfies $d_H(S_1, S_2) \leq (1 + \varepsilon)k$. We output the longest pair of (honest) witness substrings found across all iterations. We will return a correct answer assuming that the fraction of errors is $\rho < \frac{1}{3}$. Recall that the algorithm solves the decision variant of the LCS with Approximately k Mismatches problem incorrectly with probability not exceeding a constant δ , and we can ensure $\delta < \frac{1}{3}$ by repeating it a constant number of times. It means that Carole can answer an individual question erroneously with probability less than $\frac{1}{3}$. Therefore, for a sufficiently large constant in the number of queries $Q = \Theta(\log n)$, the fraction of erroneous answers is $\rho < \frac{1}{3}$ with high probability by Chernoff–Hoeffding bounds. The claim of the theorem follows immediately from Lemma 11 and Theorem 12. ◀

3.1 Proof of Theorem 12

We first give an algorithm for the decision version of the LCS with Approximately k Mismatches problem that uses $\mathcal{O}(n \log n)$ space and $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n + \sigma n \log^2 n)$ time, and then we improve the space and time complexity.

We assume to have fixed a Karp–Rabin fingerprinting function φ for a prime $q = \Omega(\max\{n^5, \sigma\})$ and an integer $r \in \mathbb{Z}_q$. With error probability inverse polynomial in n , we can find such q in $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time; see [23, 3].

Let Π be the set of all projections of strings of length ℓ onto a single position, i.e., the value $\pi_i(S)$ of the i -th projection on a string S of length ℓ is simply its i -th character $S[i]$. More generally, for a length- ℓ string S and a function $h = (\pi_{a_1}, \dots, \pi_{a_m}) \in \Pi^m$, we define $h(S)$ as $S[a_1]S[a_2] \cdots S[a_m]$.

Let $p_1 = 1 - k/\ell$ and $p_2 = 1 - (1 + \varepsilon)k/\ell$. We assume that $(1 + \varepsilon)k < \ell$ in order to guarantee $p_1 > p_2 > 0$; the problem is trivial if $(1 + \varepsilon)k \geq \ell$. Further, let $m = \lceil \log_{p_2} \frac{1}{n} \rceil$.

We choose a set \mathcal{H} of $L = \Theta(n^{1/(1+\varepsilon)})$ hash functions in Π^m uniformly at random. Let $C_\ell^{\mathcal{H}}$ be the multiset of all collisions of length- ℓ substrings of X and Y under the functions from \mathcal{H} , i.e. $C_\ell^{\mathcal{H}} = \{(X[i, i + \ell - 1], Y[j, j + \ell - 1], h) : \varphi(h(X[i, i + \ell - 1])) = \varphi(h(Y[j, j + \ell - 1])), 1 \leq i \leq |X| - \ell, 1 \leq j \leq |Y| - \ell\}$.

We will perform two tests. The first test chooses an arbitrary subset $C' \subseteq C_\ell^{\mathcal{H}}$ of size $|C'| = \min\{4nL, |C_\ell^{\mathcal{H}}|\}$ and, for each collision $(S_1, S_2, h) \in C'$, computes $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2$. If this value is at most $(1 + \varepsilon)k$, then the algorithm returns YES and the pair (S_1, S_2) as a witness. The second test chooses a collision $(S_1, S_2, h) \in C_\ell^{\mathcal{H}}$ uniformly at random and computes the Hamming distance between S_1 and S_2 character by character in $\mathcal{O}(\ell) = \mathcal{O}(n)$ time. If the Hamming distance is at most $(1 + \varepsilon)k$, the algorithm returns YES and the witness pair (S_1, S_2) . Otherwise, the algorithm returns NO. See Algorithm 2.

■ **Algorithm 2** LCS with Approximately k Mismatches (decision variant).

-
- 1: Choose a set \mathcal{H} of L functions from Π^m uniformly at random
 - 2: $C_\ell^{\mathcal{H}} = \{(S_1, S_2, h) : S_1, S_2 \text{ -- length-}\ell \text{ substrings of } X, Y \text{ resp. and } \varphi(h(S_1)) = \varphi(h(S_2))\}$
 - 3: Choose an arbitrary subset $C' \subseteq C_\ell^{\mathcal{H}}$ of size $\min\{4nL, |C_\ell^{\mathcal{H}}|\}$
 - 4: Compute $\text{sk}_\varepsilon(\cdot)$ sketches for all length- ℓ substrings of X, Y
 - 5: **for** $(S_1, S_2, h) \in C'$ **do**
 - 6: **if** $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2 \leq (1 + \varepsilon)k$ **then return** (YES, (S_1, S_2))
 - 7: Draw a collision $(S_1, S_2, h) \in C_\ell^{\mathcal{H}}$ uniformly at random
 - 8: **if** $d_H(S_1, S_2) \leq (1 + \varepsilon)k$ **then return** (YES, (S_1, S_2))
 - 9: **return** NO
-

We must explain how we compute $C_\ell^{\mathcal{H}}$ and choose the collisions that we test. We consider each hash function $h \in \mathcal{H}$ in turn. Let $h = (\pi_{a_1}, \dots, \pi_{a_m})$. Recall that for a string S of length ℓ we define $h(S)$ as $S[a_1]S[a_2] \cdots S[a_m]$. Consequently, $\varphi(h(S)) = (\sum_{i=1}^m r^{i-1} S[a_i]) \bmod q$. We create a vector U of length ℓ where each entry is initialised with 0. For each i , we add $r^{i-1} \bmod q$ to the a_i -th entry of U . Finally, we run the FFT algorithm [12] for U and X, Y in the field \mathbb{Z}_q , and sort the resulting values. We obtain a list of sorted values that we can use to generate the collisions. Namely, consider some fixed value z . Assume that there are x substrings of X and y substrings of Y of length ℓ such that the fingerprint of their projection is equal to z . The value z then gives xy collisions, and we can generate each one of them in constant time. This explains how to choose the subset C' in $\mathcal{O}(nL \log n)$ time.

To draw a collision from $C_\ell^{\mathcal{H}}$ uniformly at random, we could simply compute the total number of collisions across all functions $h \in \mathcal{H}$, draw a number in $[1, |C_\ell^{\mathcal{H}}|]$, and generate the corresponding collision. However, this would require to generate the collisions twice. Instead, we use the weighted reservoir sampling algorithm [9]. We divide all collisions into subsets according to the values of fingerprints. We assume that the weighted reservoir sampling algorithm receives the fingerprint values one-by-one, as well as the number of corresponding collisions. At all times, the algorithm maintains a “reservoir” containing one fingerprint value and a random collision corresponding to this value. When a new value z with xy collisions arrives, the algorithm replaces the value in the reservoir with z and a random collision with

16:8 Approximating Longest Common Substring with k Mismatches

some probability. Note that to select a random collision it suffices to choose a pair from $[1, x] \times [1, y]$ uniformly at random. It is guaranteed that if for a value z we have xy collisions, the algorithm will select z with probability $xy/|C_\ell^{\mathcal{H}}|$. Consequently, after processing all values, the reservoir will contain a collision chosen from $C_\ell^{\mathcal{H}}$ uniformly at random.

► **Lemma 13.** *Algorithm 2 uses $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n + \sigma n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space.*

Proof. Computing the sketches (Line 4) takes $\mathcal{O}(\sigma n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space. Computing the collisions and choosing the collisions to test takes $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$ time and $\mathcal{O}(n)$ space in total. Testing $\min\{4nL, |C_\ell^{\mathcal{H}}|\}$ collisions (Line 5) takes $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$ time and constant space. Computing the Hamming distance for a random collision (Line 8) takes $\mathcal{O}(\ell) = \mathcal{O}(n)$ time and constant space. ◀

► **Lemma 14.** *Let S_1 and S_2 be two length- ℓ substrings of X and Y , respectively, with $d_H(S_1, S_2) \leq k$. If $L = \Theta(n^{1/(1+\varepsilon)})$ is large enough, then, with probability at least $3/4$, there exists a function $h \in \mathcal{H}$ such that $h(S_1) = h(S_2)$.*

Proof. Consider a function $h = (\pi_{a_1}, \dots, \pi_{a_m})$ drawn from Π^m uniformly at random. The probability of $h(S_1) = h(S_2)$ is at least p_1^m . Due to $p_1 \leq 1$, we have

$$p_1^m = p_1^{\lceil \log_{p_2} \frac{1}{n} \rceil} \geq p_1^{1 + \log_{p_2} \frac{1}{n}} = p_1 \cdot n^{-\frac{\log p_1}{\log p_2}}.$$

Moreover, $p_1 = 1 - \frac{k}{\ell}$ and $(1+\varepsilon)k < \ell$ yield $p_1 > 1 - \frac{1}{1+\varepsilon} = \frac{\varepsilon}{1+\varepsilon}$, whereas Bernoulli's inequality implies $p_2 = 1 - (1+\varepsilon)\frac{k}{\ell} \leq (1 - \frac{k}{\ell})^{1+\varepsilon} = p_1^{1+\varepsilon}$, i.e., $\log p_2 \leq (1+\varepsilon) \log p_1$. Therefore,

$$p_1^m \geq p_1 \cdot n^{-\frac{\log p_1}{\log p_2}} \geq \frac{\varepsilon}{1+\varepsilon} \cdot n^{-\frac{1}{1+\varepsilon}}.$$

Hence, we can choose the constant in $L = |\mathcal{H}|$ so that the claim of the lemma holds. ◀

► **Lemma 15.** *If $|C_\ell^{\mathcal{H}}| > 4nL$ and (S_1, S_2, h) is a uniformly random element of $C_\ell^{\mathcal{H}}$, then $\Pr[d_H(S_1, S_2) \geq (1+\varepsilon)k] \leq \frac{1}{2}$.*

Proof. Consider length- ℓ substrings S_1, S_2 of X, Y , respectively, such that $d_H(S_1, S_2) \geq (1+\varepsilon)k$, and a hash function h . Let us bound the probability of $(S_1, S_2, h) \in C_\ell^{\mathcal{H}}$. There are two possible cases: either $h(S_1) \neq h(S_2)$ but $\varphi(h(S_1)) = \varphi(h(S_2))$, or $h(S_1) = h(S_2)$. The probability of the first event is bounded by the collision probability of Karp–Rabin fingerprints, which is at most $1/n$. Let us now bound the probability of the second event. Since $d_H(S_1, S_2) \geq (1+\varepsilon)k$, we have $\Pr[h(S_1) = h(S_2)] \leq p_2^m \leq 1/n$, where the last inequality follows from the definition of m . Therefore, the probability that for some function $h \in \mathcal{H}$ we have $\varphi(h(S_1)) = \varphi(h(S_2))$ is at most $2/n$.

In total, we have $n^2|\mathcal{H}|$ possible triples (S_1, S_2, h) so by linearity of expectation, we conclude that the expected number of such triples is at most $\frac{2}{n}n^2L = 2nL$. Therefore the probability to hit a triple (S_1, S_2, h) such that $d_H(S_1, S_2) \geq (1+\varepsilon)k$ when drawing from $C_\ell^{\mathcal{H}}$ uniformly at random is at most $2nL/|C_\ell^{\mathcal{H}}| \leq 2nL/4nL = 1/2$. ◀

Below, we combine the previous results to prove that, with constant probability, Algorithm 2 correctly solves the decision variant of the LCS with Approximately k Mismatches problem. Note that we can reduce the error probability to an arbitrarily small constant $\delta > 0$: it suffices to repeat the algorithm a constant number of times.

► **Corollary 16.** *With non-zero constant probability, Algorithm 2 solves the decision variant of LCS with Approximately k Mismatches correctly.*

Proof. Suppose first that $\ell \leq \text{LCS}_k(X, Y)$, which means that there are two length- ℓ substrings S_1, S_2 of X, Y such that $d_H(S_1, S_2) \leq k$. By Lemma 14, with probability at least $3/4$, there exists a function $h \in \mathcal{H}$ such that $h(S_1) = h(S_2)$. In other words, $(S_1, S_2, h) \in C_\ell^{\mathcal{H}}$ with probability at least $\frac{3}{4}$. If $|C_\ell^{\mathcal{H}}| < 4nL$, we will find this triple and it will pass the test with probability at least $1 - n^{-6}$. If $|C_\ell^{\mathcal{H}}| \geq 4nL$, then by Lemma 15 the Hamming distance between S_1, S_2 , where (S_1, S_2, h) was drawn from $C_\ell^{\mathcal{H}}$ uniformly at random, is at most $(1 + \varepsilon)k$ with probability $\geq 1/2$, and therefore this pair will pass the test with probability $\geq 1/2$. It follows that in this case the algorithm outputs YES with constant probability.

Suppose now that $\ell > \text{LCS}_{(1+\varepsilon)k}(X, Y)$. In this case, the Hamming distance between any pair of length- ℓ substrings of X and Y is at least $(1 + \varepsilon)k$, so none of them will ever pass the second test and none of them will pass the first test with constant probability. ◀

We now improve the space of the algorithm to linear. Note that the only reason why we needed $\mathcal{O}(n \log n)$ space is that we precompute and store the sketches for the Hamming distance. Below we explain how to overcome this technicality.

First, we do not precompute the sketches. Second, we process the collisions in C' in batches of size n . Consider one of the batches, \mathcal{B} . For each collision $(S_1, S_2, h) \in \mathcal{B}$ we must compute $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2$. We initialize a counter for every collision, setting it to zero initially. The number of rounds in the algorithm will be equal to the length of the sketches, and, in round i , the counter for a collision $(S_1, S_2, h) \in \mathcal{B}$ will contain the squared L_2 distance between the length- i prefixes of $\text{sk}_\varepsilon(S_1)$ and $\text{sk}_\varepsilon(S_2)$. In more detail, let \mathcal{S} be the set of all substrings of X, Y that participate in the collisions in \mathcal{B} . Recall that all these substrings have length ℓ . At round i , we compute the i -th coordinate of the sketches of the substrings in \mathcal{S} . By definition, the i -th coordinate is the dot product of the i -th row of $c \cdot M$, where c and M are as in Corollary 5, and a substring encoded using μ . Hence, we can compute the coordinate using the FFT algorithm [12] in $\mathcal{O}(\sigma n \log n)$ time and $\mathcal{O}(n)$ space. When we have the coordinate i computed, we update the counters for the collisions and repeat.

At any time, the algorithm uses $\mathcal{O}(n)$ space. Compared to the time consumption proven in Lemma 13, the algorithm spends an additional $\mathcal{O}(\sigma n^{1+1/(1+\varepsilon)} \log^2 n)$ time for computing the coordinates of the sketches. Therefore, in total the algorithm uses $\mathcal{O}(\sigma n^{1+1/(1+\varepsilon)} \log^2 n) = \mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time and $\mathcal{O}(n)$ space. For constant-size alphabets, this completes the proof of Theorem 12. For alphabets of arbitrary size, we replace the sketches from Section 2 with the sketches defined in [19] to achieve the desired complexity. We note that we could use the sketches [19] for small-size alphabets as well, but their lengths hide a large constant.

4 Experiments

We now present results of experimental evaluation of the second solution of Theorem 2.

Methodology and test environment. The baselines and our solution are written in C++11 and compiled with optimizations using gcc 7.4.0. The experimental results were generated on an Intel Xeon E5-2630 CPU using 128 GiB RAM. To ensure the reproducibility of our results, our complete experimental setup, including data files, is available at https://github.com/fnareoh/LCS_Approx_k_mis.

Baseline. The only other solution to the LCS with Approximately k Mismatches problem was presented in [19]. However, it has a worse complexity and is likely to be unpractical because it uses a very complex class of hash functions. We therefore chose to compare our algorithm against algorithms for the LCS with k Mismatches problem. To the best of our knowledge,

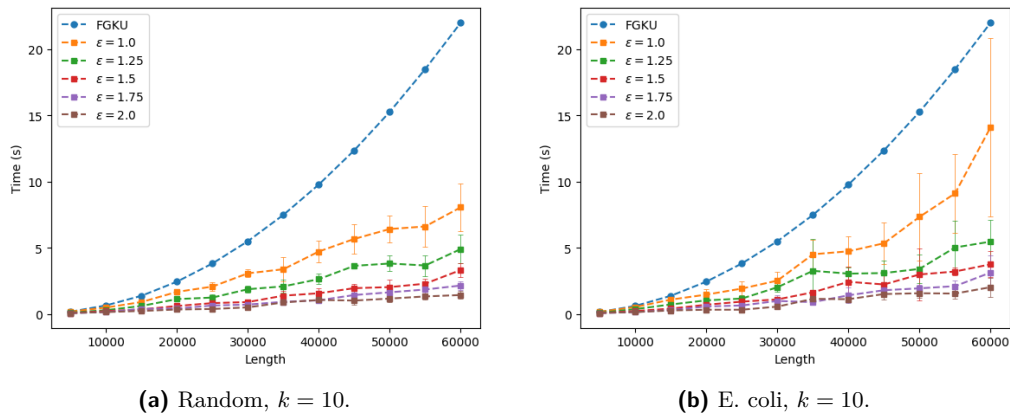
16:10 Approximating Longest Common Substring with k Mismatches

none of the existing algorithms has been implemented. We implemented the solution to LCS with k Mismatches by Flouri et al., which we refer to as FGKU [13]. (The other algorithms seem too complex to be efficient in practice.) The main idea of the algorithm of Flouri et al. is that if we know that the longest common substring with k mismatches is obtained by a substring of X that starts at a position p and a substring of Y that starts at a position $p + i$, then we can find it by scanning X and $Y[i, |Y|]$ in linear time.

Details of implementation. We made several adjustments to the theoretical algorithm we described. First, we use the fact that $A = \text{LCS}(X, Y) + k \leq \text{LCS}_k(X, Y) \leq B = (k + 1) \cdot \text{LCS}(X, Y) + k$ to bound the interval in the Twenty Questions game. We also treated the number of questions in the Twenty Questions game and L , the size of the set of hash functions \mathcal{H} , as parameters that trade time for accuracy, and put the number of questions to $2 \log(B - A)$ in the Twenty Questions game and $L = n^{1/(1+\varepsilon)}/16$. In Line 6 of Algorithm 2, we used sketches to estimate the Hamming distance. In practice, we computed the Hamming distance via character-by-character comparison when ℓ is small compared to k and via kangaroo jumps [14] otherwise. Also, when $\ell \leq 2 \log n$ in Algorithm 2, we computed the hash values of the length- ℓ substrings of S_1 and S_2 naively, instead of using the FFT algorithm [12].

Data sets and results. We considered $k \in \{10, 25, 50\}$ and $\varepsilon \in \{1.0, 1.25, 1.5, 1.75, 2.0\}$. We tested the algorithms on pairs of random strings (each character is selected independently and uniformly from a four-character alphabet $\{A, T, G, C\}$) and on pairs of strings extracted at random from the E. coli genome. The lengths of the strings in each pair are equal and vary from 0 to 60000 with a step of 5000. All timings reported are averaged over ten runs. Figures 1–3 show the results for $k = 10, 25, 50$. We note that for $\varepsilon = 1$ and $k = 10, 25$, the standard deviation of the running time on the E. coli data set is quite large, which is probably caused by our choice of the method to compute the Hamming distance between substrings, but for all other parameter combinations it is within the standard range. We can see that the time decreases when ε grows, which is coherent with the theoretical complexity.

As for the accuracy, note that our algorithm cannot return a pair of strings at Hamming distance more than $(1 + \varepsilon)k$, and so the only risk is returning strings which are too short. Consequently, we measured the accuracy of our implementation by the ratio of the length



■ **Figure 1** Comparison of the FGKU algorithm versus our algorithm for $k = 10$ and different values of ε . Large standard deviation for length 60000 is caused by an outlier with very long longest common substring with k mismatches.

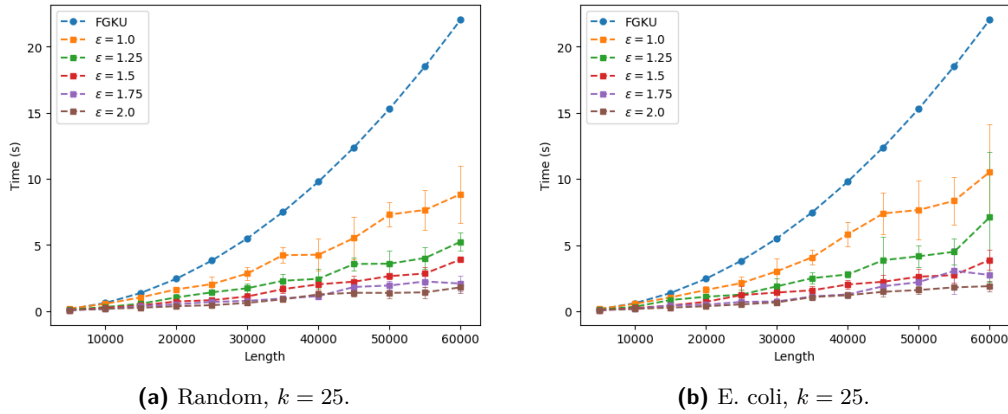


Figure 2 Comparison of the FGKU algorithm versus our algorithm for $k = 25$ and different values of ϵ .

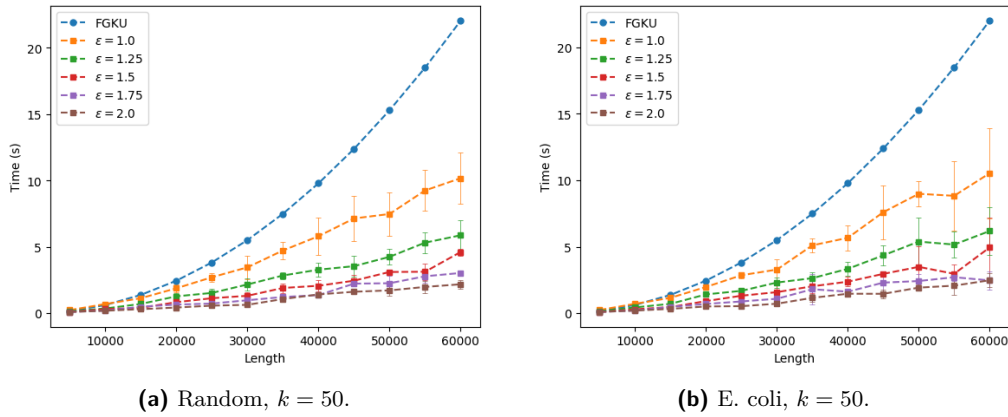


Figure 3 Comparison of the FGKU algorithm versus our algorithm for $k = 50$ and different values of ϵ .

Table 1 Accuracy of the LCS with Approximately k Mismatches algorithm. For each k and ϵ , we show $r_{\min}(\epsilon, k)$, $r_{\max}(\epsilon, k)$, as well as the error rate.

	Random						E. coli					
	$k = 10$		$k = 25$		$k = 50$		$k = 10$		$k = 25$		$k = 50$	
$\epsilon = 1.0$	0.95	1.41	1.12	1.46	1.27	1.54	0.89	1.34	0.94	1.48	0.97	1.59
	error = 3%		error = 0%		error = 0%		error = 33%		error = 13%		error = 3%	
$\epsilon = 1.25$	0.97	1.47	1.15	1.63	1.44	1.78	0.88	1.48	0.98	1.56	0.99	1.73
	error = 1%		error = 0%		error = 0%		error = 28%		error = 5%		error = 3%	
$\epsilon = 1.5$	1.05	1.57	1.37	1.76	1.55	1.91	0.88	1.45	0.96	1.67	0.99	1.89
	error = 0%		error = 0%		error = 0%		error = 17%		error = 3%		error = 3%	
$\epsilon = 1.75$	1.02	1.69	1.46	1.86	1.72	2.12	0.88	1.58	0.95	1.84	1.02	2.15
	error = 0%		error = 0%		error = 0%		error = 17%		error = 2%		error = 0%	
$\epsilon = 2.0$	1.10	1.72	1.59	2.00	1.89	2.24	0.91	1.77	1.01	2.10	1.00	2.19
	error = 0%		error = 0%		error = 0%		error = 9%		error = 0%		error = 1%	

16:12 Approximating Longest Common Substring with k Mismatches

$\text{LCS}_{\bar{k}}(X, Y)$ returned by our algorithm divided by $\text{LCS}_k(X, Y)$ computed by the dynamic programming. We estimate $r_{\min}(\varepsilon, k) = \min_{X, Y} (\text{LCS}_{\bar{k}}(X, Y) / \text{LCS}_k(X, Y))$ and $r_{\max}(\varepsilon, k) = \max_{X, Y} (\text{LCS}_{\bar{k}}(X, Y) / \text{LCS}_k(X, Y))$ by computing $\text{LCS}_{\bar{k}}(X, Y)$ and $\text{LCS}_k(X, Y)$ for 10 pairs of strings for each length from 5000 to 60000 with step of 5000, as well as the error rate, i.e., the percentage of experiments where $\text{LCS}_{\bar{k}}(X, Y) < \text{LCS}_k(X, Y)$ (see Table 1). Not surprisingly, r_{\min} and r_{\max} grow as k and ε grow, while the error rate drops. Even though there is no theoretical upper bound on r_{\max} , the latter is at most 2.24 at all times. We also note that even in the cases when the error rate is non-negligible, $\text{LCS}_{\bar{k}} \geq 0.86 \cdot \text{LCS}_k$; in other words, our algorithm returns a reasonable approximation of LCS_k .

5 Proof of Fact 3

We now show the lower bound of Fact 3 by a reduction from the $(1 + \gamma)$ -approximate Bichromatic Closest Pair problem.

► **Problem 17** ($(1 + \gamma)$ -approximate Bichromatic Closest Pair). *Given a constant $\gamma > 0$ and two sets of binary strings $(U_i)_{i \in [1, N]}$ and $(V_j)_{j \in [1, N]}$, each of length $d = \mathcal{O}(\log N)$, if the smallest Hamming distance between a pair $(U_i, V_j)_{i, j \in [1, N]}$ is h , we must output (possibly another) pair of binary strings (U_i, V_j) with Hamming distance in $[h, (1 + \gamma)h]$.*

Rubinfeld [22] proved that for every constant $\delta > 0$, there exists $\gamma = \gamma(\delta)$ such that any randomised algorithm that solves $(1 + \gamma)$ -approximate Bichromatic Closest Pair correctly with constant probability requires $\mathcal{O}(N^{2-\delta})$ time assuming SETH:

► **Hypothesis 18** (SETH). *For every $\delta > 0$, there exists an integer q such that SAT on q -CNF formulas with m clauses and n variables cannot be solved in $m^{\mathcal{O}(1)} 2^{(1-\delta)n}$ time even by a Monte-Carlo randomised algorithm (with error probability bounded by a small constant)².*

We show the lower bound by reducing a single instance of $(1 + \gamma)$ -approximate Bichromatic Closest Pair to a polylogarithmic number of instances of LCS with Approximately k Mismatches. We assume that U_i, V_j are over the alphabet $\{0, 1\}$. Let us introduce a string $H = (\mathbf{a}^d \mathbf{b})^{d+1}$ and construct $X = HU_1HU_2H \dots HU_NH$ and $Y = HV_1HV_2H \dots HV_NH$.

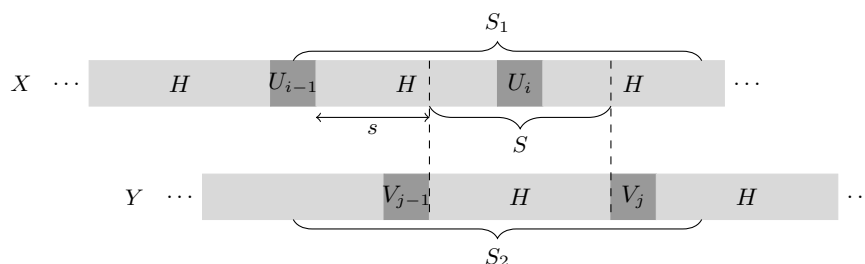
► **Observation 19**. *For every integer $k \geq 0$, if there exist $i, j \in [1, N]$ such that $d_H(U_i, V_j) \leq k$, then $\text{LCS}_k(X, Y) \geq 2(d + 1)^2 + d$.*

Proof. If $d_H(U_i, V_j) \leq k$ for some i, j , then $d_H(HU_iH, HV_jH) \leq k$ and $\text{LCS}_k(X, Y) \geq |HU_iH| = 2(d + 1)^2 + d$. ◀

► **Lemma 20**. *For every integer $0 \leq k \leq d$, if $\text{LCS}_k(X, Y) \geq 2(d + 1)^2 + d$, then there exist $i, j \in [1, N]$ such that $d_H(U_i, V_j) \leq k$.*

Proof. By the assumption of the lemma, there exist substrings S_1 and S_2 of X and Y , respectively, with $|S_1| = |S_2| \geq 2(d + 1)^2 + d$ and $d_H(S_1, S_2) \leq k$. The substring S_2 contains either HV_j or V_jH for some j . Without loss of generality, we can assume that S_2 contains a copy of H followed by V_j for some j . Let us consider the substring S of X aligned with the copy of H in S_2 . Below we will prove that $S = H$, and since S is followed by U_i for some i , this will imply that $d_H(HU_iH, HV_jH) \leq k$.

² Impagliazzo, Paturi, and Zane [16] stated the hypothesis for deterministic algorithms only, but nowadays it is common to extend SETH to allow randomisation. If we condition on the classic version of the hypothesis, we will obtain a lower bound for deterministic algorithms. See [27] for more discussion.



■ **Figure 4** Substrings S_1 and S_2 of X and Y , respectively, substring S aligned with a copy of H in S_2 , and the shift s .

Suppose that $S \neq H$, and let $0 < s < (d+1)^2 + d$ be the distance between the starting positions of S and the nearest copy of H from the left. If $s < d+1$ or $(d+1)^2 < s$, then each occurrence of b in H creates a mismatch. There are $d+1 > k$ of them, a contradiction. If $d+1 \leq s \leq (d+1)^2$, then S contains U_i , creating d mismatches with H . Since $|U_i| = d$ and $|H| = (d+1)^2$, we will have at least one more mismatch from the alignment of the copy of H in Y and the copies of H in X that surround U_i . Therefore, in total there are at least $d+1 > k$ mismatches, a contradiction. To conclude, both cases are impossible, and hence $s = 0$. The lemma follows as explained above. ◀

With this lemma, we can now proceed to prove Fact 3. Define $\varepsilon = \gamma/3$ and consider all $k = 1, (1+\varepsilon), (1+\varepsilon)^2, (1+\varepsilon)^3, \dots$ until $d/(1+\varepsilon)$. For each k , we run $\log \log n$ independent instances of an algorithm for LCS with Approximately k Mismatches. Let k_0 be the smallest k such that the identified longest common substring with approximately k mismatches has length at least $2(d+1)^2 + d$.

By the definition of k_0 , Observation 19 and Lemma 20, there do not exist $i, j \in [1, N]$ such that $d_H(U_i, V_j) \leq k_0/(1+\varepsilon)$, but there exist $i, j \in [1, N]$ such that $d_H(U_i, V_j) \leq k_0(1+\varepsilon)$. In the $(1+\gamma)$ -approximate Bichromatic Closest Pair problem, this translates to $k_0/(1+\varepsilon) < h \leq k_0(1+\varepsilon)$, where h is the minimal distance between all pairs U_i, V_j . This is equivalent to

$$h \leq k_0(1+\varepsilon) < h(1+\varepsilon)^2 = h\left(1 + \frac{2}{3}\gamma + \frac{1}{9}\gamma^2\right) \leq h(1+\gamma),$$

which means that the pair (U_i, V_j) found by the algorithm for k_0 is a valid solution for $(1+\gamma)$ -approximate Bichromatic Closest Pair. It follows that, for some k , the algorithm for LCS with Approximately k Mismatches must spend $\Omega(N^{2-\delta}/\log_{1+\varepsilon} \log N)$ time. We have $n = |X| = |Y| = \mathcal{O}(d^2 N) = \mathcal{O}(N \log^2 N)$, which implies $N = \Omega(n/\log^2 n)$. Fact 3 follows.


References

- 1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *SODA'15*, pages 218–230, 2015. doi:10.1137/1.9781611973730.17.
- 2 Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003. doi:10.1016/S0022-0000(03)00025-4.
- 3 Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004. doi:10.4007/annals.2004.160.781.
- 4 Alexandr Andoni and Piotr Indyk. Efficient algorithms for substring near neighbor problem. In *SODA'06*, pages 1203–1212, 2006. doi:10.1145/1109557.1109690.

- 5 Alexandr Andoni and Ilya Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *STOC'15*, pages 793–801, 2015. doi:10.1145/2746539.2746553.
- 6 Maxim Babenko and Tatiana Starikovskaya. Computing the longest common substring with one mismatch. *Problems of Information Transmission*, 47(1):28–33, 2011. doi:10.1134/S0032946011010030.
- 7 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.
- 8 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *FOCS'18*, pages 979–990, 2018. doi:10.1109/FOCS.2018.00096.
- 9 Min-Te Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, December 1982. doi:10.2307/2336002.
- 10 Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In *CPM'18*, pages 23:1–23:16, 2018. doi:10.4230/LIPIcs.CPM.2018.23.
- 11 Aditi Dhagat, Peter Gács, and Peter Winkler. On playing “Twenty questions” with a liar. In *SODA'92*, pages 16–22, 1992. URL: <http://dl.acm.org/citation.cfm?id=139404.139409>.
- 12 Michael J. Fischer and Michael S. Paterson. String matching and other products. In *Complexity of Computation*, pages 113–125, 1974.
- 13 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015.
- 14 Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, March 1986. doi:10.1145/8307.8309.
- 15 Szymon Grabowski. A note on the longest common substring with k -mismatches problem. *Information Processing Letters*, 115(6-8):640–642, 2015.
- 16 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.
- 17 William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Modern Analysis and Probability*, volume 26 of *Contemporary Mathematics*, pages 189–206, 1984.
- 18 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.
- 19 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. doi:10.1007/s00453-019-00548-x.
- 20 Chris-Andre Leimeister and Burkhard Morgenstern. kmacs: the k -mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014. doi:10.1093/bioinformatics/btu331.
- 21 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *FOCS'09*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.
- 22 Aviad Rubinfeld. Hardness of approximate nearest neighbor search. In *STOC'18*, pages 1260–1268, 2018. doi:10.1145/3188745.3188916.
- 23 Terence Tao, Ernest Croot III, and Harald Helfgott. Deterministic methods to find primes. *AMS Mathematics of Computation*, 81(278):1233–1246, 2012. doi:10.1090/S0025-5718-2011-02542-1.
- 24 Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k -mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016. doi:10.1089/cmb.2015.0235.

- 25 Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Alberto Apostolico, and Srinivas Aluru. ALFRED: A practical method for alignment-free distance computation. *Journal of Computational Biology*, 23(6):452–460, 2016. doi:10.1089/cmb.2015.0217.
- 26 Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC Bioinformatics*, 18(8):238, June 2017. doi:10.1186/s12859-017-1658-0.
- 27 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *ICM'18*, pages 3447–3487, 2018. doi:10.1142/9789813272880_0188.

String Factorizations Under Various Collision Constraints

Niels Grüttemeier 

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Germany
niegru@informatik.uni-marburg.de

Christian Komusiewicz 

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Germany
komusiewicz@informatik.uni-marburg.de

Nils Morawietz

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Germany
morawietz@informatik.uni-marburg.de

Frank Sommer 

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik, Germany
fsommer@informatik.uni-marburg.de

Abstract

In the NP-hard EQUALITY-FREE STRING FACTORIZATION problem, we are given a string S and ask whether S can be partitioned into k factors that are pairwise distinct. We describe a randomized algorithm for EQUALITY-FREE STRING FACTORIZATION with running time $2^k \cdot k^{\mathcal{O}(1)} + \mathcal{O}(n)$ improving over previous algorithms with running time $k^{\mathcal{O}(k)} + \mathcal{O}(n)$ [Schmid, TCS 2016; Mincu and Popa, Proc. SOFSEM 2020]. Our algorithm works for the generalization of EQUALITY-FREE STRING FACTORIZATION where equality can be replaced by an arbitrary polynomial-time computable equivalence relation on strings. We also consider two factorization problems to which this algorithm does not apply, namely PREFIX-FREE STRING FACTORIZATION where we ask for a factorization of size k such that no factor is a prefix of another factor and SUBSTRING-FREE STRING FACTORIZATION where we ask for a factorization of size k such that no factor is a substring of another factor. We show that these two problems are NP-hard as well. Then, we show that PREFIX-FREE STRING FACTORIZATION with the prefix-free relation is fixed-parameter tractable with respect to k by providing a polynomial problem kernel. Finally, we show a generic ILP formulation for R -FREE STRING FACTORIZATION where R is an arbitrary relation on strings. This formulation improves over a previous one for EQUALITY-FREE STRING FACTORIZATION in terms of the number of variables.

2012 ACM Subject Classification Theory of computation \rightarrow Parameterized complexity and exact algorithms; Theory of computation \rightarrow Pattern matching

Keywords and phrases NP-hard problem, fixed-parameter algorithms, collision-aware string partitioning

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.17

Funding *Frank Sommer*: Supported by the Deutsche Forschungsgemeinschaft (DFG), project MAGZ, KO 3669/4-1.

1 Introduction

In collision-aware string partitioning problems we are given a string S and want to compute a factorization of S , that is, a partition of S into substrings, called factors, such that no two factors of the factorization collide. Herein, two strings collide if they are too similar, for example if they are equal or if one is a prefix of the other [8]. These problems have applications in synthetic biology, where one important task is to assemble a DNA string S from some of its factors. To allow for an assembly of S , the factors from which S is built need



© Niels Grüttemeier, Christian Komusiewicz, Nils Morawietz, and Frank Sommer; licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 17; pp. 17:1–17:14

Leibniz International Proceedings in Informatics



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

to be sufficiently different from each other, as otherwise the assembly process will produce some unwanted string $S' \neq S$. The demand for pairwise inequality or pairwise prefix-freeness of the factors is an abstraction of the demand of sufficiently large differences [8]. Such demands are always fulfilled by the trivial factorization which consists of the single factor S but in the application described above, we aim to find small factors. Thus, the task is to find a factorization with the desired property such that each factor has bounded length [7, 8]. Another closely related variant of collision-aware string partitioning arises in the context of pattern matching with variables [11]. Here, the additional restriction is not on the length of the factors but instead on their number.

EQUALITY-FREE STRING FACTORIZATION

Input: A string S of length n and an integer k .

Question: Is there a factorization of S into k pairwise different factors?

EQUALITY-FREE STRING FACTORIZATION is NP-hard [11]. Motivated by this result, Schmid [16] initiated a parameterized complexity analysis with respect to parameters such as the alphabet size of S or the factorization size k . For the latter parameter, an algorithm with running time $\mathcal{O}(\binom{k^2+k}{2} + n)$ was proposed [16]. This algorithm relies on a combination of the brute-force algorithm with running time $\mathcal{O}(n^k)$ with the observation that instances with $n \geq \frac{k^2+k}{2} - 1$ are yes-instances. The running time was later improved to $\mathcal{O}(k^{k/2} + n)$ [15]. We continue the study of EQUALITY-FREE STRING FACTORIZATION with respect to the natural parameter k . Moreover, we consider several extensions and variants of EQUALITY-FREE STRING FACTORIZATION and study their classical and parameterized complexity.

Our Results. We present an improved randomized fixed-parameter algorithm for EQUALITY-FREE STRING FACTORIZATION with a running time of $2^k \cdot n^{\mathcal{O}(1)}$. This algorithm relies on a reduction to the problem of finding a path with k different colors in a directed graph G and on an algebraic algorithm for finding such a path. This is one of the few applications of algebraic algorithms to NP-hard string problems, another application was provided for MAXIMUM DUO STRING PARTITIONING [13]. Curiously, in both applications the first step is a reduction to a path-finding problem in an auxiliary graph. Unlike previous approaches which are tailored to EQUALITY-FREE STRING FACTORIZATION since they use the fact that strings with different length are unequal, our algorithm works for an arbitrary equivalence relation R over strings and thus for further notions of collision. To formulate our result precisely, we introduce the following generic problem.

R -FREE STRING FACTORIZATION

Input: A string S of length n and an integer k .

Question: Is there a factorization (w_1, w_2, \dots, w_k) of S such that $w_i \not\mathcal{R} w_j$ for all $i \neq j$?

► **Theorem 1.1.** *Let R be a polynomial-time computable equivalence relation over the set of all strings. Then, R -FREE STRING FACTORIZATION can be solved by a randomized algorithm with one-sided error and running time $2^k \cdot n^{\mathcal{O}(1)}$.*

Two natural examples for such an equivalence relation are to consider two strings as similar when they use the same set of letters, we denote this relation by $=_\Sigma$, and to consider two strings w and w' as similar when they have the same Parikh vector, we denote this relation by $=_{\Sigma, \#}$. The Parikh vector of a string w over alphabet $\Sigma = \{a_1, \dots, a_\sigma\}$ is the length- σ vector p_w where $p_w[i]$ is the number of occurrences of a_i in w . This notion of equivalence is used in JUMBLED PATTERN MATCHING [6]. Since $=_\Sigma$ and $=_{\Sigma, \#}$ are equivalence

relations, Theorem 1.1 directly implies an FPT algorithm for them. It is a priori not clear, however, whether R -FREE STRING FACTORIZATION is even NP-hard for these particular special cases of R . To show NP-hardness of these two special cases, we revisit the NP-hardness proof for EQUALITY-FREE STRING FACTORIZATION [11].

Motivated by the different notions of string collision that have been formulated previously [7,8] we then consider three cases of R that are not equivalence relations: In the *prefix relation* \preceq_p we have $w \preceq_p w'$ if w is a prefix of w' , in the *suffix relation* \preceq_s we have $w \preceq_s w'$ if w is a suffix of w' , and in the *substring relation* \preceq we have $w \preceq w'$ if w is a substring of w' . By slightly adapting the known hardness reduction for EQUALITY-FREE STRING FACTORIZATION [11], we obtain the following.

► **Theorem 1.2.** *For each $R \in \{=\Sigma, =\Sigma, \#, \preceq_p, \preceq_s, \preceq\}$, R -FREE STRING FACTORIZATION is NP-complete and cannot be solved in time $2^{o(n)}$ unless the ETH fails.*

Our second main technical result is a problem kernel for PREFIX-FREE STRING FACTORIZATION, that is, for the special case where R is the prefix relation \preceq_p . This kernel proves that PREFIX-FREE STRING FACTORIZATION is fixed-parameter tractable for the parameter k despite the fact that Theorem 1.1 does not apply. The main idea of the kernelization is to shrink highly repetitive regions of the string S . Moreover, this result also implies that SUFFIX-FREE STRING FACTORIZATION, which is the special case of R -FREE STRING FACTORIZATION where $R = \preceq_s$, is fixed-parameter tractable for the parameter k . Finally, as a side result we obtain an ILP formulation with $\mathcal{O}(n)$ variables for R -FREE STRING FACTORIZATION for all relations R that can be computed in polynomial time. This improves, in terms of the number of variables, upon a previous formulation for EQUALITY-FREE STRING FACTORIZATION [15].

Related Work. Fernau et al. [11] introduced the problem of maximizing the number of factors and showed that it is NP-hard. The NP-hardness reduction also implies that, assuming the ETH, EQUALITY-FREE STRING FACTORIZATION cannot be solved in $2^{o(n)}$ time (this uses the fact that 3D MATCHING cannot be solved in $2^{o(q)}$ time, where q is the instance size [12]). Mincu and Popa [15] introduced a version of EQUALITY-FREE STRING FACTORIZATION in which one allows gaps between the factors. That is, the aim is to find k disjoint factors of S such that no two are equal. A further related NP-hard factorization problem is DIVERSE PALINDROMIC FACTORIZATION, where we ask whether a given string has an equality-free factorization in which each factor is a palindrome [1].

Preliminaries. For $i \in \mathbb{N}$, we let $[i]$ denote the set $\{1, \dots, i\}$. For $i \in \mathbb{N}$ and $j \in \mathbb{N}$, where $i \leq j$, we let $[i, j]$ denote the set $\{i, \dots, j\}$.

The length of a string S is denoted by $|S|$. For a string S , we let $S[i]$, $1 \leq i \leq |S|$, denote the character at position i and $S[i, j]$, $1 \leq i \leq j \leq |S|$, denote the substring starting at position i and ending at position j . Given a string $S = S[1]S[2] \dots S[|S|]$, we define $S^1 := S$ and $S^i := S^{i-1}S$ for $i > 1$. Moreover, we let $\overleftarrow{S} := S[|S|]S[|S|-1] \dots S[1]$ denote the *reversed string* of S . A *period* of S is an integer p such that $S[i] = S[i+p]$ for all $i \in [|S| - p]$. The string $S[1, p]$ is also called period in this case. If a string S can be written as $w'w^i w''$ for some $i \geq 1$ and has period $|w|$, then we call w an *internal period* of S . A *border* of a string S is a suffix of S that is also a prefix of S . If a string has a border of length b , then it has a period of length $|S| - b$ [9]. Two substrings $S[i, j]$ and $S[i', j']$ *overlap* if $[i, j] \cap [i', j'] \neq \emptyset$, otherwise they are *disjoint*. A set \mathcal{P} of substrings of a string S is a *packing* if all substrings in \mathcal{P} are disjoint. A set \mathcal{S} of strings is *prefix-free* if no string in \mathcal{S} is a prefix of another string in \mathcal{S} . To avoid confusion, we will use the term *factor* only in combination with a factorization and not as a synonym of substring.

We consider directed graphs $D = (V, A)$ where V is a set of vertices and $A \subseteq V \times V$ is a set of directed edges called *arcs*. A *walk* of length k in a directed graph $D = (V, A)$ is a k -tuple (v_1, \dots, v_k) such that $v_i \in V$ for each $i \in [k]$ and $(v_i, v_{i+1}) \in A$ for each $i \in [k-1]$. A walk (v_1, \dots, v_k) is a (simple) *path* if $v_i \neq v_j$ for all $i \neq j$. A walk (v_1, \dots, v_k) is a *cycle* whenever $v_1 = v_k$.

For more details on parameterized algorithms and the Exponential Time Hypothesis (ETH), we refer the reader to the standard monographs. For an overview of the parameterized complexity of string problems, refer to the survey of Bulteau et al. [5]. For an introduction to algebraic algorithms including evaluation of polynomials over finite fields, refer to the monograph of Cygan et al. [10].

Due to lack of space, several proofs are deferred to the long version of this paper.

Lower-Bounding the Factorization Size. In the definition of R -FREE STRING FACTORIZATION we ask for a factorization into *exactly* k factors. An equally natural question would be to ask for a factorization into *at least* k factors. It is known that for EQUALITY-FREE STRING FACTORIZATION these two questions are equivalent: by merging a largest factor with a neighboring factor, any equality-free factorization of size k' can be transformed into one of size $k' - 1$. Such a property is also possible for the relations R under consideration in this work: For $=_{\Sigma}$, merge a factor with a maximal set of characters with one of its neighboring factors. For $=_{\Sigma, \#}$, merge the factor with the lexicographically largest Parikh vector with one of its neighboring factors. For \preceq_p , we make use of the following observation which will also be useful in the kernelization algorithm.

► **Lemma 1.3.** *Let \mathcal{S} be a prefix-free set of strings over an alphabet Σ and let $u \in \mathcal{S}$ be any string of \mathcal{S} and let $v \in \Sigma^*$. Then the set $\mathcal{S} \setminus \{u\} \cup \{uv\}$ is prefix-free.*

Proof. The string uv is not a prefix of any other string in \mathcal{S} since the string u is not a prefix of any other string in \mathcal{S} . Moreover, no other string w is a prefix of uv : Otherwise, if $|w| \leq |u|$, then w is a prefix of u and if $|w| > |u|$, then u is a prefix of w . In both cases, we have a contradiction to the fact that \mathcal{S} is prefix-free. ◀

Lemma 1.3 can now be used to argue that asking for a prefix-factorization of size k is equivalent to asking for one of size at least k : Given a prefix-free factorization $(f_1, \dots, f_{k'})$ where $k' > 1$, merge f_1 and f_2 into one factor. By Lemma 1.3 the factorization $(f_1 f_2, \dots, f_{k'})$ is prefix-free and has size $k' - 1$.

2 An Improved Parameterized Algorithm

A Reduction to a Rainbow Path Problem. The first step in the improved algorithm is to reduce R -FREE STRING FACTORIZATION to the following path problem.

RAINBOW- (s, t) -PATH

Input: A directed graph $D = (V, A)$, two vertices $s \in V$ and $t \in V$, and a vertex-coloring $c: V \rightarrow \{1, \dots, |V|\}$.

Question: Does G contain a *rainbow (s, t) -path* of length k , that is, a path on k vertices from s to t such that all vertices on this path have pairwise different colors?

► **Lemma 2.1.** *There is a parameterized polynomial-time reduction from R -FREE STRING FACTORIZATION parameterized by k to RAINBOW- (s, t) -PATH parameterized by k . Instances with parameter value k are mapped to instances with parameter value $k + 2$ and the graph produced by the reduction is a DAG.*

Proof. For each substring $S[i, j]$, create a vertex $v_{i,j}$. For each vertex $v_{i,j}$ add an arc to each vertex $v_{j+1,q}$, $j+1 \leq q \leq n$. In other words, arcs are added between vertices that represent neighboring substrings. For each equivalence class C of R that has at least one representative in the set of substrings of S , we introduce one color c_C . Each vertex $v_{i,j}$ is colored with the color of its equivalence class $c_{[v_{i,j}]}$. Finally, we add one vertex s with a unique color c_s and a vertex t with a unique color c_t and the arcs $(s, v_{1,j})$, $1 \leq j \leq n$, and $(v_{p,n}, t)$, $1 \leq p \leq n$.

The correctness of the reduction can be seen as follows. If S has a size- k factorization F such that for all factors w and w' of F we have $w \mathcal{R} w'$, then D has a rainbow (s, t) -path of length $k+2$: The vertices corresponding to the factors of F form a path of length k and s has an arc to the first vertex of the path and the last vertex of the path has an arc to t . Moreover, the vertices on the path have pairwise different colors since each color corresponds to an equivalence class of R . Conversely, any (s, t) -path in D corresponds to a factorization of S . Moreover, if the vertices of the factorization have a different color, then the corresponding factors are in different equivalence classes and thus not in relation with respect to R . If the length of the path is $k+2$, then the number of factors in the factorization is k . ◀

Observe that one approach to obtain a parameterized algorithm for RAINBOW- (s, t) -PATH parameterized by k is via color coding: randomly map the $\mathcal{O}(n^2)$ colors in D to a set of k labels and then find an (s, t) -path with k different labels, if it exists, via dynamic programming. Using standard derandomization techniques [10], this algorithm can be derandomized with a slight running time overhead, resulting in a deterministic algorithm with running time $(2e)^k \cdot k^{\mathcal{O}(\log k)} \cdot n^{\mathcal{O}(1)}$, improving over the previous fastest algorithm of Mincu and Popa [15]. We omit the description of this algorithm in favor of a substantially faster randomized algorithm.

Detecting Rainbow (s, t) -Paths. To solve RAINBOW- (s, t) -PATH, we reduce it to the problem of testing whether a polynomial can be evaluated to zero in some finite field. This technique has led to the currently fastest randomized algorithms for several longest path problems parameterized by the path length [2, 3]. We adapt the technique to handle the constraint that the path should be rainbow. Observe that there is a previous algorithm for RAINBOW- (s, t) -PATH on undirected graphs [14] which uses the number of colors as parameter. Since in our application the number of colors can be superlinear in n , we may not use this algorithm to obtain a fixed-parameter algorithm for EQUALITY-FREE STRING FACTORIZATION.

The algorithm to detect rainbow (s, t) -paths is a slight adaption of the algorithm for LONGEST PATH in directed graphs [10]: Associate a set of monomials (that is, a polynomial with only one summand) with every walk of length k of the graph and then consider the walk polynomial which is the sum of the walk monomials. More precisely, we will introduce one monomial $M_{W,\ell}$ for each walk $W = (v_1, \dots, v_k)$ and each bijective labeling $\ell : [k] \rightarrow [k]$. The labeling ℓ will assign a label $\ell(i)$ to the i th color occurring in the walk and is the main trick to establish the canceling property for walks that are not rainbow. If we ensure that nonsimple walks cancel out, then the polynomial will not be identically zero if and only if there is at least one rainbow path of length k .

For each edge (u, v) we introduce a variable $x_{u,v}$ which represents that the edge (u, v) is traversed in a walk. For each color c of G , we introduce k variables $y_{c,i}$, $i \in [k]$, each representing that a vertex with color c receives the label i . The monomial associated with the pair (W, ℓ) is now

$$M_{W,\ell}(\mathbf{x}, \mathbf{y}) := \prod_{i=1}^{k-1} x_{v_i, v_{i+1}} \prod_{i=1}^k y_{c(v_i), \ell(i)}.$$

17:6 String Factorizations Under Various Collision Constraints

To ensure that the path starts with s and ends in t , we consider only walks that start in s and end in t . Accordingly, the polynomial is defined as

$$P(\mathbf{x}, \mathbf{y}) := \sum_{\text{walk } W=(s=v_1, \dots, v_k=t)} \sum_{\text{bijective } \ell: [k] \rightarrow [k]} M_{W, \ell}(\mathbf{x}, \mathbf{y}).$$

As mentioned above, the idea of the construction is that the monomials for walks which have some label twice will cancel out. To enable this, the polynomial is evaluated over a field of characteristic 2, that is, addition of some value to itself will give 0. Thus, the walks that are not paths will cancel out if their monomials can be partitioned into pairs such that each pair will have the same variables. Observe that the polynomial will never be constructed explicitly but instead evaluated via dynamic programming.

► **Lemma 2.2.** *$P(\mathbf{x}, \mathbf{y})$ is not identically zero in a finite field with characteristic 2 if and only if there is a rainbow (s, t) -path of length k in G .*

Proof. Consider an (s, t) -walk $W = (s = v_1, \dots, v_k = t)$ that is not rainbow and any monomial $M_{W, \ell}$. Since W is not rainbow, there are two vertices v_i and v_j such that $c(v_i) = c(v_j)$. Take the lexicographically smallest pair of indices i and j for which this is true. Consider the labeling $\ell_{i \leftrightarrow j}$ defined as follows: $\ell_{i \leftrightarrow j}(i) := \ell(j)$, $\ell_{i \leftrightarrow j}(j) := \ell(i)$, and $\ell_{i \leftrightarrow j}(q) := \ell(q)$ for all $q \in [k] \setminus \{i, j\}$. In other words, $\ell_{i \leftrightarrow j}$ is obtained by swapping the i th and j th elements in the permutation corresponding to ℓ . The monomial $M_{W, \ell_{i \leftrightarrow j}}$ is the same as $M_{W, \ell}$ and, hence, $M_{W, \ell} + M_{W, \ell_{i \leftrightarrow j}}$ is identically zero. Moreover, since $(\ell_{i \leftrightarrow j})_{i \leftrightarrow j} = \ell$, we have a partition of all monomials corresponding to walks that are not rainbow paths into pairs $\{\ell, \ell_{i \leftrightarrow j}\}$ such that for each pair, the variables of the monomial are the same. Thus, these monomials cancel out and P can be written as the sum over all walks that are in fact rainbow paths.

It remains to show that the monomials that correspond to rainbow (s, t) -paths do not cancel out, by showing that they have pairwise different variable sets. This is obvious for two monomials that correspond to different walks. Thus, consider a rainbow (s, t) -path W and two monomials $M_{W, \ell}$ and $M_{W, \ell'}$ where ℓ and ℓ' are two different labelings. Moreover, choose $i \in [k]$ such that $\ell(i) \neq \ell'(i)$. Then, $M_{W, \ell}$ and $M_{W, \ell'}$ differ in at least two variables: the variable $y_{c(v_i), \ell(i)}$ occurs in $M_{W, \ell'}$ and not in $M_{W, \ell}$: First, $y_{c(v_i), \ell'(i)}$ is a different variable since $\ell(i) \neq \ell'(i)$ and each other y -variable in $M_{W, \ell'}$ is of the form $y_{c', q}$ for some $c' \neq c(v_i)$ since W is rainbow. ◀

The polynomial $P(\mathbf{x}, \mathbf{y})$ can be efficiently evaluated via dynamic programming.

► **Lemma 2.3.** *The polynomial $P(\mathbf{x}, \mathbf{y})$ can be evaluated in $2^k \cdot k^{\mathcal{O}(1)} \cdot |A|$ time over the field $\text{GF}(2^{\lceil \log 4k \rceil})$.*

Proof. We follow the exposition of Cygan et al. [10] and present the details only for the sake of completeness. Fix a walk W , then the sum of the monomials $M_{W, \ell}$ over all bijective labelings ℓ can be written as

$$\sum_{\text{surjective } \ell: [k] \rightarrow [k]} M_{W, \ell} = \sum_{\ell \in \bigcap_{i \in [k]} A_i} M_{W, \ell}(\mathbf{x}, \mathbf{y})$$

where A_i is the set of labelings such that $\ell(j) = i$ for some $j \in [k]$. Now, let U denote the set of all mappings $\ell: [k] \rightarrow [k]$. Using the inclusion–exclusion principle, the latter term can

be rewritten as.

$$\begin{aligned}
 \sum_{\ell \in \bigcap_{i \in [k]} A_i} M_{W,\ell}(\mathbf{x}, \mathbf{y}) &= \sum_{X \subseteq [k]} (-1)^{|X|} \cdot \sum_{\ell \in \bigcap_{i \in X} U \setminus A_i} M_{W,\ell}(\mathbf{x}, \mathbf{y}) \\
 &= \sum_{X \subseteq [k]} \sum_{\ell \in \bigcap_{i \in X} U \setminus A_i} M_{W,\ell}(\mathbf{x}, \mathbf{y}) \\
 &= \sum_{X \subseteq [k]} \sum_{\ell: [k] \rightarrow [k] \setminus X} M_{W,\ell}(\mathbf{x}, \mathbf{y}) \\
 &= \sum_{X \subseteq [k]} \sum_{\ell: [k] \rightarrow X} M_{W,\ell}(\mathbf{x}, \mathbf{y}).
 \end{aligned}$$

The equalities follow from the inclusion–exclusion principle, the fact that the field has characteristic 2, the fact that $\bigcap_{i \in X} U \setminus A_i$ is the set of all labelings $\ell : [k] \rightarrow [k]$ that do not map to any element in X , and a replacement of X by its complement $U \setminus X$ in the sum, respectively.

Thus, we have

$$\begin{aligned}
 P(\mathbf{x}, \mathbf{y}) &= \sum_{\text{walk } W=(s=v_1, \dots, v_k=t)} \sum_{\text{bijective } \ell: [k] \rightarrow [k]} M_{W,\ell}(\mathbf{x}, \mathbf{y}) \\
 &= \sum_{\text{walk } W=(s=v_1, \dots, v_k=t)} \sum_{X \subseteq [k]} \sum_{\ell: [k] \rightarrow X} M_{W,\ell}(\mathbf{x}, \mathbf{y}) \\
 &= \sum_{X \subseteq [k]} \sum_{\text{walk } W=(s=v_1, \dots, v_k=t)} \sum_{\ell: [k] \rightarrow X} M_{W,\ell}(\mathbf{x}, \mathbf{y})
 \end{aligned}$$

It is thus sufficient to show that $\sum_{\ell: [k] \rightarrow X} M_{W,\ell}(\mathbf{x}, \mathbf{y})$ can be computed in $k^{\mathcal{O}(1)} \cdot |A|$ time. This can be done by dynamic programming, building up the domain of the labeling ℓ from $[1]$ to $[k]$. More precisely, one may fill a table with entries of the type $T_X[v, d]$ where v is a vertex of D and $d \in [k]$ such that

$$T_X[v, d] := \sum_{\text{walk } W=(v=v_1, \dots, v_d=t)} \sum_{\ell: [d] \rightarrow X} M_{W,\ell}(\mathbf{x}, \mathbf{y})$$

as follows. For each the vertex t , $T[t, 1] = y_{c(t),i}$ as the walk (t) does not contain any edges, and we may consider all possible labels for $y_{c(t),\ell}$. For $d > 1$, we have

$$T_X[v, d] = \sum_{i \in X} y_{c(v),i} \sum_{(v,w) \in A} x_{v,w} \cdot T[w, d-1]$$

since this sum considers all possibilities for the label of $c(v)$, all outgoing edges from v , and multiplies each with the number of possibilities to continue the walk. Here, we exploit that $T[w, d]$ is not just the sum over all walks and all labelings $\ell : [d] \rightarrow X$ but due to symmetry, the sum over all walks and all labelings $\ell : X' \rightarrow X$ for every $X' \subseteq X$ such that $|X'| = d$.

The value of $P(\mathbf{x}, \mathbf{y})$ can thus be computed using $\mathcal{O}(2^k \cdot k|A|)$ field operations since

$$P(\mathbf{x}, \mathbf{y}) = \sum_{X \subseteq [k]} \sum_{\text{walk } W=(s=v_1, \dots, v_k=t)} \sum_{\ell: [k] \rightarrow X} M_{W, \ell}(\mathbf{x}, \mathbf{y}) \quad (1)$$

$$= \sum_{X \subseteq [k]} T_X[s, k]. \quad (2)$$

Since the order of the field is $2^{\lceil \log 4k \rceil} \leq 8k$, each field operation can be performed in $k^{\mathcal{O}(1)}$ time. \blacktriangleleft

Now we obtain a randomized algorithm for deciding whether D contains a rainbow (s, t) -path by evaluating P at a random vector (\mathbf{x}, \mathbf{y}) of elements of the field $\text{GF}(2^{\lceil \log 4k \rceil})$. If $P(\mathbf{x}, \mathbf{y}) = 0$, then the algorithm returns that D has no rainbow (s, t) -path of length k , otherwise it returns that D contains such a path. If the graph contains no rainbow (s, t) -path of length k , then P is identically zero and the algorithm answers correctly. Otherwise, if the graph contains a rainbow (s, t) -path of length k , then P is not identically zero. Since the maximum degree of P is at most $2k - 1$ and since the field has order at least $4k$, the Schwartz-Zippel Lemma now implies that (\mathbf{x}, \mathbf{y}) is a root with probability at most $1/2$. Thus, the error probability is at most $1/2$. By repeating this procedure $\log(1/\epsilon)$ times, we may achieve an error probability of at most ϵ for any $\epsilon > 0$.

This algorithm in combination with the reduction from R -FREE STRING FACTORIZATION to RAINBOW- (s, t) -PATH gives Theorem 1.1. Observe that Theorem 1.1 only refers to the decision version of the problem but in the applications we may want to output the factorization if it exists. This can be done via applying the framework of Björklund et al. [4] which only relies on two facts: The witness which we wish to extract has size k and the decision algorithm has one-sided error; the running time overhead is a factor of $\mathcal{O}(k \log n)$.

As a final remark, in our framework of solving R -FREE STRING FACTORIZATION via reduction to RAINBOW- (s, t) -PATH, we may put any further polynomial-time computable restriction on the factors: the only additional step is to add only those vertices that fulfill this restriction to the graph D . For example, we may demand that every factor has length at least q and at most r for some integers q and r , or we may demand that it contains every letter of the alphabet. Thus, we may apply the algorithm also when we search for length-bounded factorizations [8] when the parameter is the factorization size k . We can also use this framework to solve DIVERSE PALINDROMIC FACTORIZATION where each factor should be a palindrome [1].

► **Corollary 2.4.** *There is a randomized algorithm with one-sided error that decides in time $2^k \cdot n^{\mathcal{O}(1)}$ whether a string has a palindromic factorization with exactly k factors.*

3 Further String Relations

Hardness Results. First, we prove Theorem 1.2. That is, we show that for each $R \in \{=\Sigma, =\Sigma, \#, \preceq_p, \preceq_s, \preceq\}$, R -FREE STRING FACTORIZATION is NP-complete and cannot be solved in time $2^{o(n)}$ unless the ETH fails.

First, we show this result for $R \in \{\preceq_p, \preceq\}$.

► **Lemma 3.1.** *For each $R \in \{\preceq_p, \preceq\}$, R -FREE STRING FACTORIZATION is NP-complete and cannot be solved in time $2^{o(n)}$ unless the ETH fails.*

Observe that the hardness result for $R = \preceq_p$ also implies hardness for $R = \preceq_s$ by the following simple reduction: Let (S, k) be an instance of R -FREE STRING FACTORIZATION for $R = \preceq_p$. Then, the instance (\overleftarrow{S}, k) is an equivalent instance for $R = \preceq_s$. The equivalence can be seen as follows: (f_1, f_2, \dots, f_k) is a prefix-free string factorization of S if and only if $(\overleftarrow{f_k}, \dots, \overleftarrow{f_2}, \overleftarrow{f_1})$ is a suffix-free string factorization of \overleftarrow{S} . Hence, the following holds.

► **Lemma 3.2.** *If $R = \preceq_s$, then R -FREE STRING FACTORIZATION is NP-complete and cannot be solved in time $2^{o(n)}$ unless the ETH fails.*

Next, we show NP-hardness for each $R \in \{=\Sigma, =\Sigma, \#\}$; altogether this shows Theorem 1.2.

► **Lemma 3.3.** *For each $R \in \{=\Sigma, =\Sigma, \#\}$, R -FREE STRING FACTORIZATION is NP-complete and cannot be solved in time $2^{o(n)}$ unless the ETH fails.*

Prefix-Free String Factorization. In this paragraph we provide a problem kernel for PREFIX-FREE STRING FACTORIZATION parameterized by the number k of factors. Recall that PREFIX-FREE STRING FACTORIZATION is the special case of R -FREE STRING FACTORIZATION, where $R = \preceq_p$. For EQUALITY-FREE STRING FACTORIZATION, it is very easy to obtain a problem kernel: since factors with different length are trivially unequal, one may simply choose strings of increasing length starting with length 1. This implies that instances with $n \geq \frac{k^2+k}{2}$ are yes-instances and thus we have a quadratic kernel for EQUALITY-FREE STRING FACTORIZATION. For PREFIX-FREE STRING FACTORIZATION, this argument does not hold. Consider for example the string a^n . The maximum prefix-free factorization has size one because of the periodicity of a^n .

To describe the kernelization we first need to establish some notation. Let (S, k) be an instance of PREFIX-FREE STRING FACTORIZATION. For any string w , a substring $S[i, j]$ of S is called w -periodic if $S[i, j] = w^t$ for some integer $t \geq 0$ and if $|w|$ is the shortest period of $S[i, j]$. Moreover, $S[i, j]$ is called *maximal w -periodic* if there is no substring $S[i', j'] = w^{t'}$ with $[i, j] \subsetneq [i', j']$. We define $R(w) := \{t \mid w^t \text{ is a maximal } w\text{-periodic substring of } S\}$. The central rule of this kernelization reduces the length of maximal w -periodic substrings of S . For fixed w , the maximal w -periodic substrings of S are uniquely defined since each begins with w and $|w|$ is the shortest period. Hence, for each w we can find all maximal w -periodic substrings in linear time. We first show that we may assume that for every w the size of $R(w)$ is bounded, which we need to give a bound for the kernel size.

► **Lemma 3.4.** *Let (S, k) be an instance of PREFIX-FREE STRING FACTORIZATION. If there exists a string w such that $|R(w)| \geq 2k + 3$, then (S, k) is a yes-instance.*

Proof. Let w be a substring of S such that $|R(w)| \geq 2k + 3$. Then, there are distinct $t_1, t_2, \dots, t_{2k+2} \in R(w)$ such that $t_i \geq 2$ for each $i \in [2k + 1]$. We show that we can use maximal w -periodic occurrences of the w^{t_i} to find at least k prefix free-factors. To this end, let $X = \{(p_1, q_1), \dots, (p_{2k+2}, q_{2k+2})\}$ be a set containing the start positions p_i and the end positions q_i of one maximal w -periodic occurrence of w^{t_i} for each $i \in \{1, \dots, 2k + 2\}$. By the definition of maximal w -periodic substrings no element $S[p_i, q_i]$ includes another element $S[p_j, q_j]$ in X . Hence, $p_i \neq p_j$ if $i \neq j$. Without loss of generality we assume that $p_1 < p_2 < \dots < p_{2k+2}$.

Note that in S two strings $S[p_i, q_i]$ and $S[p_j, q_j]$ with $i \neq j$ overlap with less than $|w|$ characters, since otherwise this contradicts the fact that these strings are maximal w -periodic. To obtain the prefix-free factors from X , we first show the following.

17:10 String Factorizations Under Various Collision Constraints

▷ **Claim 3.5.** In S , every string $S[p_i, q_i]$ overlaps with at most two other strings $S[p_j, q_j]$, and $S[p_t, q_t]$.

Proof. Let $S[p_i, q_i]$ overlap with $S[p_j, q_j]$, and $S[p_t, q_t]$. To prove the claim we show that this implies that $S[p_j, q_j]$ and $S[p_t, q_t]$ do not overlap in S .

Without loss of generality assume $p_j < p_i < p_t$. Since $S[p_i, q_i]$ overlaps with $S[p_j, q_j]$ and $S[p_t, q_t]$, and do not overlap in at least $|w|$ characters as discussed above, we have $0 \leq q_j - p_i < |w| - 1$ and $0 \leq q_i - p_t < |w| - 1$. Since $t_2 \geq 2$ by the definition of X it also holds that $q_i - p_i \geq 2|w| - 1$ and therefore $q_i - p_i > q_j - p_i + q_i - p_t$. Consequently it holds that $q_j < p_t$ and therefore $S[p_j, q_j]$ and $S[p_t, q_t]$ do not overlap in S . ◀

We next use Claim 3.5 to define the factors. We define $k + 1$ disjoint substrings of S as follows: $f_0 := S[1, p_2 - 1]$, $f_i := S[p_{2i+1}, p_{2i+3} - 1]$ for all $i \in [k - 1]$, and $f_k := S[p_{2k+1}, |S|]$. Claim 3.5 guarantees that $w^{t_{2i+1}}$ is a prefix of f_i if $i \geq 1$. Then, for every distinct $f, f' \in \{f_1, \dots, f_k\}$, the substring f is not a prefix of f' since f and f' start with substrings of period $|w|$ that have distinct lengths. Next, consider the following cases for f_0 .

Case 1: There exists no f_i with $i \geq 1$ such that f_0 is a prefix of f_i or vice versa. Then, (f_0, \dots, f_k) is a prefix-free string factorization of size $k + 1$ and nothing more needs to be shown.

Case 2: There exists some f_i with $i \geq 1$ such that f_0 is a prefix of f_i or vice versa. Then, f_0 starts with a maximal w -periodic substring $w^{t_{2i+1}}$. Hence, for every $j \neq i$ it holds that f_0 is not a prefix of f_j and vice versa. We then discard f_{i-1} and f_i and instead consider their concatenation $f_{i-1}f_i$. We end up with a prefix-free factorization of S containing at least k factors. Hence, (S, k) is a yes-instance. ◀

For the rest of this section, we assume that, given an instance (S, k) , we have $|R(w)| \leq 2k + 3$ for each substring w of S since otherwise (S, k) is a trivial yes-instance due to Lemma 3.4. The main idea of the kernelization is thus to reduce the number of maximum repetitions of every substring of the input string S .

► **Rule 3.1.** Let w be a substring in S such that S has

1. at least one maximal w -periodic substring $S[q, r] = w^d$ with $d \geq 2k^2 + 2$ and
 2. for each $\psi \in [k^2]$ no maximal w -periodic substring $S[q', r'] = w^{d-\psi}$.
- Then, for each $p \geq d$, replace each maximal w -periodic substring $S[q, r] = w^p$ by w^{p-1} .

The rule can be applied in polynomial time by checking for each substring w of S whether it fulfills the conditions of the rule. The correctness proof of the rule is quite technical and due to lack of space deferred to the long version. The proof idea is as follows. When the rule shortens a maximal w -periodic substring w^p , one of the k factors that overlaps with this factor must be shortened by one occurrence of w . This may, however, lead to a factorization that is not prefix-free. To reestablish prefix-freeness, we may need to remove w from another factor f that overlaps some other maximal w -periodic substring $w^{p'}$. Since $w^{p'}$ is not necessarily shortened by the rule, we must add w to some other factor f' that overlaps $w^{p'}$ in order to reestablish that we have a factorization. Due to Lemma 1.3, we may add w safely to the factor f' that starts before $w^{p'}$ and ends either right before the first position of $w^{p'}$ or overlaps with $w^{p'}$.

Let w be a substring of S . According to Lemma 3.4, there are at most $2k + 2$ different repetition numbers in $R(w)$. Let $d \geq 2k^2 + 1$ be a repetition number of w such that for each $\psi \in [k^2]$ we have $d - \psi \notin R(w)$. Then Reduction Rule 3.1 reduces each repetition number $p \geq d$ by 1. Hence, the largest repetition number of w is at most $(2k^2 + 1) + k^2 \cdot (2k + 2) = 2k^3 + 4k^2 + 1$ if Reduction Rule 3.1 has been applied exhaustively.

► **Corollary 3.6.** *Let S be an instance to which Reduction Rule 3.1 has been applied exhaustively and let w be a substring of S . Then the maximal repetition number of w in S is $2k^3 + 4k^2 + 1$.*

We now show that bounding the number of repetition numbers and their size for all substrings of S results in an instance that is a yes-instance if $|S|$ is too big. This implies the problem kernel.

► **Theorem 3.7.** *PREFIX-FREE STRING FACTORIZATION has a problem kernel of size $\mathcal{O}(k^{10})$.*

Proof. Let (S, k) be an instance of PREFIX-FREE STRING FACTORIZATION that is reduced exhaustively with respect to Rule 3.1. We show that if $|S| > 12k(k+1)(2k^4 + 4k^3 + 2k)^2$, then (S, k) is a yes-instance.

Let (S, k) be a no-instance. Let $d(w)$ be the maximum size of disjoint occurrences of a substring w in S . To show $|S| \leq 12k(k+1)(2k^4 + 4k^3 + 2k)^2$ we prove that for every substring with length $2k^4 + 4k^3 + 2k$ we have $d(w) \leq 2k$. Afterwards, we use this upper bound on the number of occurrences to give an upper bound for the size of S .

Let w be a substring of S such that $|w| = 2k^4 + 4k^3 + 2k$. Moreover, let $\mathcal{P} := \{S[p_1, q_1], S[p_2, q_2], \dots, S[p_{|\mathcal{P}|}, q_{|\mathcal{P}|}]\}$ with $p_1 < q_1 < p_2 < \dots < p_{|\mathcal{P}|} < q_{|\mathcal{P}|}$ be a maximum packing of occurrences of w in S . Without loss of generality, we also assume that $S[p_1, q_1]$ is the first occurrence of w in S , since otherwise, we can replace p_1 and q_1 by the start and endpoint of the first occurrence. Assume towards a contradiction that $|\mathcal{P}| \geq 2k + 1$. We define the subpacking

$$\mathcal{P}' := \{S[p_{2i+1}, q_{2i+1}] \mid i \in [0, k]\} \subseteq \mathcal{P}$$

containing $k+1$ elements from \mathcal{P} . For every $i \in \{1, \dots, k-1\}$ we define the substring $V_{2i+1} := S[p_{2i+1} - k, p_{2i+1} - 1]$ which contains the last k characters before p_{2i+1} . Since $|w| = 2k^4 + 4k^3 + 2k$ and \mathcal{P}' contains every second element of \mathcal{P} it holds that no V_{2i+1} overlaps with $S[p_{2i-1}, q_{2i-1}] \in \mathcal{P}'$.

We first show that no occurrence of w starts in some V_{2i+1} . Assume towards a contradiction that there is one such occurrence starting in some V_{2i+1} . Then, since $|w| = 2k^4 + 4k^3 + 2k$ and $|V_{2i+1}| = k$, the string w has a border of size at least $2k^4 + 4k^3 + k$. Hence, w has a period of length at most k [9] and therefore there exists some z such that there is a maximum z -periodic substring z^p with $p \geq 2k^3 + 4k^2 + 2$ of S . Together with Corollary 3.6, this contradicts the fact that (S, k) is reduced exhaustively regarding Rule 3.1. Hence, we can assume that no occurrence of w starts in some V_{2i+1} . In the following case distinction we consider the possible values of p_1 and show that in each case we can define a prefix-free string factorization of size at least k for S , which then contradicts the fact that (S, k) is a no-instance.

Case 1: $p_1 \geq k + 1$. We define the factors $f_{\text{start}} := S[1, p_3 - 2]$, $f_i := S[p_{2i+1} - i, p_{2i+3} - (i + 2)]$ for all $i \in [k - 1]$, and $f_{\text{end}} := S[p_{2k+1} - k, |S|]$. Note that these $k + 1$ factors cover all of S and w is a substring of each such factor. We next show that none of these factors is the prefix of another factor. To this end, we consider the first occurrence of w in all factors.

Since $p_1 \geq k + 1$ and we assumed that $S[p_1, q_1]$ is the first occurrence of w in S we conclude that in f_{start} there is no occurrence of w starting in the first k positions of f_{start} . Next, the fact that no occurrence of w starts in some V_{2i+1} implies that the first occurrence of w starts at position $i + 1$ of each f_i , $i \in [k - 1]$. Moreover, in f_{end} , the first occurrence of w starts at position $k + 1$ by the same argument. Since the first occurrence of w starts at distinct positions in each of the factors, no factor is a prefix of one of the other factors.

17:12 String Factorizations Under Various Collision Constraints

Hence, S has a prefix-free factorization with $k+1$ factors contradicting the fact that (S, k) is a no-instance.

Case 2: $p_1 \in \{1, 2\}$. We define the factors $f_{\text{start}} := S[1, p_5 - 3]$, $f_i := S[p_{2i+1} - i, p_{2i+3} - (i+2)]$ for all $i \in [2, k-1]$, and $f_{\text{end}} := S[p_{2k+1} - k, |S|]$. Intuitively, these are the $k+1$ factors we defined in Case 1, but we concatenated the first two factors. We obtain k factors that cover all of S and w is a substring of each of the factors. Again we prove prefix-freeness by showing that in each pair of factor the first occurrence of w starts at different positions.

In f_{start} , the first occurrence of w starts at position 1 or 2 since $p_1 \in \{1, 2\}$. Since no occurrence of w starts in some V_{2i+1} , the first occurrence of w starts at position $i+1$ in each f_i and at position $k+1$ in f_{end} . Observe that since $i \in [2, \dots, k-2]$ it holds that $i+1 \geq 3$. Again, this contradicts the fact that (S, k) is a no-instance.

Case 3: $p_1 \in [3, k-1]$. We define the factors $f_{\text{start}} := S[1, p_3 - 2]$, $f_i := S[p_{2i+1} - i, p_{2i+3} - (i+2)]$ for all $i \in [k-1] \setminus \{p_1 - 2, p_1 - 1\}$, $f_{\text{merge}} := S[p_{2p_1-3} - (p_1 - 2), p_{2p_1+1} - (p_1 + 1)]$, and $f_{\text{end}} := S[p_{2k+1} - k, |S|]$. Again, these are k factors covering S containing w as substring. It remains to check for each factor, when the first occurrence of w starts.

In f_{start} , the first occurrence of w starts at p_1 . In each f_i the first occurrence of w starts at position $i+1$. Note that $i+1 \notin \{p_1, p_1 - 1\}$. In f_{merge} , the first occurrence of w starts at position $p_1 - 1$. Finally, in f_{end} , the first occurrence of w starts at position $k+1$. Again, this contradicts the fact that (S, k) is a no-instance.

Case 4: $p_1 = k$. We define $f_{\text{start}} := S[1, p_3 - 2]$, $f_i := S[p_{2i+1} - i, p_{2i+3} - (i+2)]$ for all $i \in [k-3]$, $f_i := S[p_{2(k-2)+1} - (k-2), p_{2(k-2)+3} - (k-1)]$, and $f_{\text{end}} := S[p_{2k-1} - k, |S|]$. Again, these are k factors covering S containing w as substring. It remains to check for each factor, when the first occurrence of w starts.

The first occurrence of w in f_{start} starts in position k . The first occurrence of w in each f_i starts in position $i+1$ and the first occurrence of w in f_{end} starts at position $k+1$. Again, this contradicts the fact that (S, k) is a no-instance.

Since all cases are contradictory we know that every substring of size $2k^4 + 4k^3 + 2k$ in S has at most $2k$ disjoint occurrences in S . We next use this fact to prove $|S| \leq 12k(k+1)(2k^4 + 4k^3 + 2k)^2$. To this end, let $X := \{w \mid w \text{ is a substring of } S \text{ and } |w| = 2k^4 + 4k^3 + 2k\}$. We first show that $|X| \leq 2(k+1)(2k^4 + 4k^3 + 2k)$. Assume towards a contradiction that $|X| > 2(k+1)(2k^4 + 4k^3 + 2k)$. Let $w \in X$. Observe that, since $|w| = 2k^4 + 4k^3 + 2k$, every occurrence of w in S overlaps with at most $2(2k^4 + 4k^3 + 2k - 1)$ occurrences of other strings in X . Then, since $|X| > 2(k+1)(2k^4 + 4k^3 + 2k)$, there exists a packing of $k+1$ elements of X . Let $\mathcal{X} := \{w_1, \dots, w_{k+1}\}$ be such packing and for each $i \in [k+1]$, let p_i be the position in S where w_i starts. We then define $k+1$ disjoint substrings of S as follows: $f_1 := S[1, p_2 - 1]$, $f_i := S[p_i, p_{i+1} - 1]$ for $i = 2, \dots, k$, and $f_{k+1} := S[p_k, |S|]$. Clearly, for every distinct $f, f' \in \{f_2, f_3, \dots, f_k, f_{k+1}\}$, the substring f is not a prefix of f' since f and f' start with distinct words of length $2k^4 + 4k^3 + 2k$. Consider f_1 and f_i for $i > 1$. If f_1 is a prefix of f_i or vice versa, we discard f_{i-1} and f_i and instead consider their concatenation $f_{i-1}f_i$. Since $|f_1| \geq 2k^4 + 4k^3 + 2k$ we end up with a prefix-free factorization of S containing at least k factors which contradicts the fact that (S, k) is a no-instance. Hence, $|X| \leq 2(k+1)(2k^4 + 4k^3 + 2k)$.

We can now give a bound for $|S|$. Recall that $d(w)$ is the maximum size of disjoint occurrences of w in S and that $d(w) \in [2k]$ for every $w \in X$. Given a fixed $w \in X$, for each of the $d(w)$ disjoint occurrences $S[j, j + |w| - 1]$ of w in S there can be occurrences of w in $S[j - |w|, j + 2|w|]$ that overlap with $S[j, j + |w| - 1]$. Hence, for every $w \in X$, the number of symbols of S in occurrences of w is at most $3|w| \cdot d(w)$. It then holds that $|S| \leq \sum_{w \in X} 3|w| \cdot d(w) \leq 12k(k+1)(2k^4 + 4k^3 + 2k)^2 \in \mathcal{O}(k^{10})$. ◀

The above problem kernelization for PREFIX-FREE STRING FACTORIZATION also implies a problem kernelization for SUFFIX-FREE STRING FACTORIZATION. To compute a problem kernel for an instance (S, k) , we first apply the problem kernelization for PREFIX-FREE STRING FACTORIZATION on (\overleftarrow{S}, k) . Reversing the string of the resulting instance once more gives an instance of the original problem of size $\mathcal{O}(k^{10})$.

► **Corollary 3.8.** *SUFFIX-FREE STRING FACTORIZATION has a problem kernel of size $\mathcal{O}(k^{10})$.*

4 An ILP formulation with $\mathcal{O}(n)$ variables

As a side result, we provide an ILP formulation which is better than a previous one in terms of the number of variables. Mincu and Popa [15] described a 0/1-ILP with $\mathcal{O}(n\sqrt{n})$ variables based on the following idea: Introduce a binary variable $x_{i,j}$ for each candidate factor $S[i, j]$ of S such that $x_{i,j} = 1$ precisely if the factor $S[i, j]$ is one of the factors of the factorization. The goal is to maximize $\sum_{1 \leq i \leq j \leq n} x_{i,j}$. The constraints of the ILP ensure that no two equal factors are chosen, no two overlapping factors are chosen, and that if we choose some factor $x_{i,j}$ where $j < n$, then a factor $x_{j+1,\ell}$ must be chosen as well. The number of variables is $\mathcal{O}(n\sqrt{n})$ since, due to an observation of Mincu and Popa [15], there is an optimal equality-free string factorization with factors of length $\mathcal{O}(\sqrt{n})$. The latter observation does not hold for other equivalence relations. For example, if we consider $=_{\Sigma}$ on binary strings, then we may have at most three factors and thus some factor must have length $\Omega(n)$.

We provide an alternative 0/1-ILP that has $\mathcal{O}(n)$ variables and works for all string relations. For each $i \in \{1, \dots, n+1\}$, we introduce one variable x_i . This variable will have the value 1 if some factor starts at $S[i]$, that is, the factorization cuts between positions $i-1$ and i . The variable x_1 will correspond to the start of the first factor and the variable x_{n+1} will correspond to the end of the last factor, these variables will be set to 1 and we just introduce them to make the formulation more concise. The whole ILP reads as follows.

$$\max. \sum_{i \in [n+1]} x_i \text{ subject to} \tag{3}$$

$$-x_i - x_{j+1} - x_p - x_{q+1} + \sum_{\ell \in [i+1, j] \cup [p+1, q]} x_{\ell} > -4 \quad \forall i < j < p < q \text{ where } (S[i, j] R S[p, q]) \tag{4}$$

$$x_i \in \{0, 1\} \quad \forall i \in [2, n] \tag{5}$$

$$x_i = 1 \quad \forall i \in \{1, n+1\} \tag{6}$$

Every assignment to the variables directly corresponds to the factorization of the input string where $x_i = 1$ means that some factor starts at position i . The number of factors is one less than the objective function value. It remains to show that no two factors of the factorization are in relation. This is ensured by Constraint 4. Let $S[i, j]$ and $S[p, q]$ be nonoverlapping equivalent candidate factors. Then Constraint 4 for this index set is fulfilled when either one of $x_i, x_{j+1}, x_p,$ and x_{q+1} has value 0, or when one of the variables in the sum has value 1. In the first, case one of the two strings $S[i, j]$ and $S[p, q]$ is not a factor of the factorization, since one of the four factor endpoints is not selected by the solution. In the second case, one of the two candidate factors is not part of the solution since some factor starts after i and before j or after p and before q . Thus, the factorization produced by the ILP is equality-free. Conversely, any equality-free factorization corresponds to a feasible solution of the ILP.

References

- 1 Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, and Shiho Sugimoto. Diverse palindromic factorization is NP-complete. *Int. J. Found. Comput. Sci.*, 29(2):143–164, 2018.
- 2 Andreas Björklund. Determinant sums for undirected hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014.
- 3 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017.
- 4 Andreas Björklund, Petteri Kaski, and Lukasz Kowalik. Fast witness extraction using a decision oracle. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA '14)*, volume 8737 of *LNCS*, pages 149–160. Springer, 2014.
- 5 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, 114, 2014.
- 6 Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Algorithms for jumbled pattern matching in strings. *Int. J. Found. Comput. Sci.*, 23(2):357–374, 2012.
- 7 Anne Condon, Ján Manuch, and Chris Thachuk. Complexity of a collision-aware string partition problem and its relation to oligo design for gene synthesis. In *Proceedings of the 14th International Computing and Combinatorics Conference (COCOON '08)*, volume 5092 of *LNCS*, pages 265–275. Springer, 2008.
- 8 Anne Condon, Ján Manuch, and Chris Thachuk. The complexity of string partitioning. *J. Discrete Algorithms*, 32:24–43, 2015.
- 9 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002.
- 10 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- 11 Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. Pattern matching with variables: Fast algorithms and new hardness results. In *Proceedings of the 32nd International Symposium on Theoretical Aspects of Computer Science (STACS '15)*, volume 30 of *LIPICs*, pages 302–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- 12 Klaus Jansen, Felix Land, and Kati Land. Bounding the running time of algorithms for scheduling and packing problems. *SIAM J. Discrete Math.*, 30(1):343–366, 2016.
- 13 Christian Komusiewicz, Mateus de Oliveira Oliveira, and Meirav Zehavi. Revisiting the parameterized complexity of maximum-duo preservation string mapping. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM '17)*, volume 78 of *LIPICs*, pages 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- 14 Lukasz Kowalik and Juho Lauri. On finding rainbow and colorful paths. *Theor. Comput. Sci.*, 628:110–114, 2016.
- 15 Radu Stefan Mincu and Alexandru Popa. The maximum equality-free string factorization problem: Gaps vs. no gaps. In *Proceedings of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '20)*, volume 12011 of *LNCS*, pages 531–543. Springer, 2020.
- 16 Markus L. Schmid. Computing equality-free and repetitive string factorisations. *Theor. Comput. Sci.*, 618:42–51, 2016.

k -Approximate Quasiperiodicity under Hamming and Edit Distance

Aleksander Kędzierski 

Institute of Informatics, University of Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
aa.kedzierski2@uw.edu.pl

Jakub Radoszewski 

Institute of Informatics, University of Warsaw, Poland
Samsung R&D Institute, Warsaw, Poland
jrad@mimuw.edu.pl

Abstract

Quasiperiodicity in strings was introduced almost 30 years ago as an extension of string periodicity. The basic notions of quasiperiodicity are cover and seed. A cover of a text T is a string whose occurrences in T cover all positions of T . A seed of text T is a cover of a superstring of T . In various applications exact quasiperiodicity is still not sufficient due to the presence of errors. We consider approximate notions of quasiperiodicity, for which we allow approximate occurrences in T with a small Hamming, Levenshtein or weighted edit distance.

In previous work Sip et al. (2002) and Christodoulakis et al. (2005) showed that computing approximate covers and seeds, respectively, under weighted edit distance is NP-hard. They, therefore, considered restricted approximate covers and seeds which need to be factors of the original string T and presented polynomial-time algorithms for computing them. Further algorithms, considering approximate occurrences with Hamming distance bounded by k , were given in several contributions by Guth et al. They also studied relaxed approximate quasiperiods that do not need to cover all positions of T .

In case of large data the exponents in polynomial time complexity play a crucial role. We present more efficient algorithms for computing restricted approximate covers and seeds. In particular, we improve upon the complexities of many of the aforementioned algorithms, also for relaxed quasiperiods. Our solutions are especially efficient if the number (or total cost) of allowed errors is bounded. We also show NP-hardness of computing non-restricted approximate covers and seeds under Hamming distance.

Approximate covers were studied in three recent contributions at CPM over the last three years. However, these works consider a different definition of an approximate cover of T , that is, the shortest exact cover of a string T' with the smallest Hamming distance from T .

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases approximate cover, approximate seed, enhanced cover, Hamming distance, edit distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.18

Related Version <https://arxiv.org/abs/2005.06329>

Funding *Jakub Radoszewski*: Supported by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

1 Introduction

Quasiperiodicity was introduced as an extension of periodicity [6]. Its aim is to capture repetitive structure of strings that do not have an exact period. The basic notions of quasiperiodicity are cover (also called quasiperiod) and seed. A *cover* of a string T is a string C whose occurrences cover all positions of T . A *seed* of string T is a cover of a superstring



© Aleksander Kędzierski and Jakub Radoszewski;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 18; pp. 18:1–18:15

Leibniz International Proceedings in Informatics



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

of T . Covers and seeds were first considered in [7] and [21], respectively, and linear-time algorithms computing them are known; see [9, 21, 29, 30, 31] and [24].

A cover is necessarily a *border*, that is, a prefix and a suffix of the string. A seed C of T covers all positions of T by its occurrences or by left- or right-overhangs, that is, by suffixes of C being prefixes of T and prefixes of C being suffixes of T . In order to avoid extreme cases one usually assumes that covers C of T need to satisfy $|C| < |T|$ and seeds C need to satisfy $2|C| \leq |T|$ (so a seed needs to be a factor of T). Seeds, unlike covers, preserve an important property of periods that if T has a period or a seed, then every (sufficiently long) factor of T has the same period or seed, respectively.

The classic notions of quasiperiodicity may not capture repetitive structure of strings in practical settings; it was also confirmed by a recent experimental study [12]. In order to tackle this problem, further types of quasiperiodicity were studied that require that only a certain number of positions in a string are covered. This way notions of enhanced cover, partial cover and partial seed were introduced. A *partial cover* and *partial seed* are required to cover a given number of positions of a string, where for the partial seed overhangs are allowed, and an *enhanced cover* is a partial cover with an additional requirement of being a border of the string. $\mathcal{O}(n \log n)$ -time algorithms for computing shortest partial covers and seeds were shown in [26] and [25], respectively, whereas a linear-time algorithm for computing a proper enhanced cover that covers the maximum number of positions in T was presented (among other variations of the problem) in [14].

Further study has led to *approximate quasiperiodicity* in which approximate occurrences of a quasiperiod are allowed. In particular, Hamming, Levenshtein and weighted edit distance were considered. A *k*-approximate cover of string T is a string C whose approximate occurrences with distance at most k cover T . Similarly one can define a *k*-approximate seed, allowing overhangs. These notions were introduced by Sip et al. [33] and Christodoulakis et al. [10], respectively, who showed that the problem of checking if a string T has a *k*-approximate cover and *k*-approximate seed, respectively, for a given k is NP-complete under weighted edit distance. (Their proof used arbitrary integer weights and a constant-sized – 12 letters in the case of approximate seeds – alphabet.) Therefore, they considered a *restricted* version of the problem in which the approximate cover or seed is required to be a factor of T . Formally, the problem is to compute, for every factor of T , the smallest k for which it is a *k*-approximate cover or seed of T . For this version of the problem, they presented an $\mathcal{O}(n^3)$ -time algorithm for the Hamming distance and an $\mathcal{O}(n^4)$ -time algorithm for the edit distance¹. The same problems under Hamming distance were considered by Guth et al. [19] and Guth and Melichar [18]. They studied a *k*-restricted version of the problems, in which we are only interested in factors of T being ℓ -approximate covers or seeds for $\ell \leq k$, and developed $\mathcal{O}(n^3(|\Sigma| + k))$ -time and $\mathcal{O}(n^3|\Sigma|k)$ -time automata-based algorithms for *k*-restricted approximate covers and seeds, respectively. Experimental evaluation of these algorithms was performed by Guth [16].

Recently, Guth [17] extended this study to *k*-approximate restricted enhanced covers under Hamming distance. In this problem, we search for a border of T whose *k*-approximate occurrences cover the maximum number of text positions. In another variant of the problem, which one could see as approximate partial cover problem, we only require the approximate

¹ In fact, they consider *relative* Hamming and Levenshtein distances which are inversely proportional to the length of the candidate factor and seek for an approximate cover/seed that minimizes such distance. However, their algorithms actually compute the minimum distance k for every factor of T under the standard distance definitions.

enhanced cover to be a k -approximate border of T , but still to be a factor of T . Guth [17] proposed $\mathcal{O}(n^2)$ -time and $\mathcal{O}(n^3(|\Sigma| + k))$ -time algorithms for the two respective variants.

We improve upon previous results on restricted approximate quasiperiodicity. We introduce a general notion of k -coverage of a string S in a string T , defined as the number of positions in T that are covered by k -approximate occurrences of S . Efficient algorithms computing the k -coverage for factors of T are presented. We also show NP-hardness for non-restricted approximate covers and seeds under the Hamming distance. A detailed list of our results is as follows.

1. The Hamming k -coverage for every prefix and for every factor of a string of length n can be computed in $\mathcal{O}(nk^{2/3} \log^{1/3} n \log k)$ time (for a string over an integer alphabet) and $\mathcal{O}(n^2)$ time, respectively. (See Section 3.)

With this result we obtain algorithms with the same time complexities for the two versions of k -approximate restricted enhanced covers that were proposed by Guth [17] and an $\mathcal{O}(n^2k)$ -time algorithm computing k -restricted approximate covers and seeds. Our algorithm for prefixes actually works in linear time assuming that a k -mismatch version of the PREF table [11] is given. Thus, as a by-product, for $k = 0$, we obtain an alternative linear-time algorithm for computing all (exact) enhanced covers of a string. (A different linear-time algorithm for this problem was given in [14]).

The complexities come from using tools of Kaplan et al. [22] and Flouri et al. [13], respectively.

2. The k -coverage under Levenshtein distance and weighted edit distance for every factor of a string of length n can be computed in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^3 \sqrt{n \log n})$ time, respectively. (See Section 4.)

We also show in Section 4 how our approach can be used to compute restricted approximate covers and seeds under weighted edit distance in $\mathcal{O}(n^3 \sqrt{n \log n})$ time, thus improving upon the previous $\mathcal{O}(n^4)$ -time algorithms of Sip et al. [33] and Christodoulakis et al. [10]. Our algorithm for Levenshtein distance uses incremental string comparison [27].

3. Under Hamming distance, it is NP-hard to check if a given string of length n has a k -approximate cover or a k -approximate seed of a given length c . This statement holds even for strings over a binary alphabet. (See Section 5.)

This result extends the previous proofs of Sip et al. [33] and Christodoulakis et al. [10] which worked for the weighted edit distance.

A different notion of approximate cover, which we do not consider in this work, was recently studied in [1, 2, 3, 4, 5]. This work assumed that the string T may not have a cover, but it is at a small Hamming distance from a string T' that has a proper cover. They defined an approximate cover of T as the shortest cover of a string T' that is closest to T under Hamming distance. Interestingly, this problem was also shown to be NP-hard [2] and an $\mathcal{O}(n^4)$ -time algorithm was developed for it in the restricted case that the approximate cover is a factor of the string T [4]. Our work can be viewed as complementary to this study as “the natural definition of an approximate repetition is not clear” [4].

2 Preliminaries

We consider strings over an alphabet Σ . The empty string is denoted by ε . For a string T , by $|T|$ we denote its length and by $T[0], \dots, T[|T| - 1]$ its subsequent letters. By $T[i, j]$ we denote the string $T[i] \dots T[j]$ which we call a *factor* of T . If $i = 0$, it is a *prefix* of T , and if $j = |T| - 1$, it is a *suffix* of T . A string that is both a prefix and a suffix of T is called

18:4 k -Approximate Quasiperiodicity under Hamming and Edit Distance

a *border* of T . For a string $T = XY$ such that $|X| = b$, by $\text{rot}_b(T)$ we denote YX , called a *cyclic shift* of T .

For equal-length strings U and V , by $\text{Ham}(U, V)$ we denote their *Hamming distance*, that is, the number of positions where they do not match. For strings U and V , by $\text{ed}(U, V)$ we denote their *edit distance*, that is, the minimum cost of edit operations (insertions, deletions, substitutions) that allow to transform U to V . Here the cost of an edit operation can vary depending both on the type of the operation and on the letters that take part in it. In case that all edit operations have unit cost, the edit distance is also called *Levenshtein distance* and denoted here as $\text{Lev}(U, V)$.

For two strings S and T and metric d , we denote by

$$\text{Occ}_k^d(S, T) = \{[i, j] : d(S, T[i, j]) \leq k\}$$

the set of approximate occurrences of S in T , represented as intervals, under the metric d . We then denote by

$$\text{Covered}_k^d(S, T) = |\bigcup \text{Occ}_k^d(S, T)|$$

the k -coverage of S in T . In case of Hamming or Levenshtein distances, $k \leq n$, but for the weighted edit distance k can be arbitrarily large. Moreover, by $\text{StartOcc}_k^d(S, T)$ we denote the set of left endpoints of the intervals in $\text{Occ}_k^d(S, T)$.

► **Definition 1.** Let d be a metric and T be a string. We say that string C , $|C| < |T|$, is a k -approximate cover of T under metric d if $\text{Covered}_k^d(C, T) = |T|$.

We say that string C , $2|C| \leq |T|$, is a k -approximate seed of T if it is a k -approximate cover of some string T' whose factor is T . Let \diamond be a wildcard symbol that matches every other symbol of the alphabet. Strings over $\Sigma \cup \{\diamond\}$ are also called *partial words*. In order to compute k -approximate seeds, it suffices to consider k -approximate covers of $\diamond^{|T|}T\diamond^{|T|}$.

The main problems in scope can now be stated as follows.

GENERAL k -APPROXIMATE COVER/SEED

Input: String T of length n , metric d , integer $c \in \{1, \dots, n-1\}$ and number k

Output: A string C of length c that is a k -approximate cover/seed of T under d

PREFIX/FACTOR k -COVERAGE

Input: String T of length n , metric d and number k

Output: For every prefix/factor of T , compute its k -coverage under d

RESTRICTED APPROXIMATE COVERS/SEEDS

Input: String T of length n and metric d

Output: Compute, for every factor C of T , the smallest k such that C is a k -approximate cover/seed of T under d

2.1 Algorithmic Toolbox for Hamming Distance

For a string T of length n , by $\text{lcp}_k(i, j)$ we denote the length of the longest common prefix with at most k mismatches of the suffixes $T[i, n-1]$ and $T[j, n-1]$. Flouri et al. [13] proposed an $\mathcal{O}(n^2)$ -time algorithm to compute the longest common factor of two strings T_1, T_2 with at most k mismatches. Their algorithm actually computes the lengths of the longest common prefixes with at most k mismatches of every two suffixes $T_1[i, |T_1|-1]$ and $T_2[j, |T_2|-1]$ and returns the maximum among them. Applied for $T_1 = T_2$, it gives the following result.

► **Lemma 2** ([13]). *For a string of length n , values $\text{lcp}_k(i, j)$ for all $i, j = 0, \dots, n - 1$ can be computed in $\mathcal{O}(n^2)$ time.*

We also use a table PREF_k such that $\text{PREF}_k[i] = \text{lcp}_k(0, i)$. LCP-queries with mismatches can be answered in $\mathcal{O}(k)$ time after linear-time preprocessing using the kangaroo method [28]. In particular, this allows to compute the PREF_k table in $\mathcal{O}(nk)$ time. Kaplan et al. [22] presented an algorithm that, given a pattern P of length m , a text T of length n over an integer alphabet $\Sigma \subseteq \{1, \dots, n^{\mathcal{O}(1)}\}$, and an integer k , finds in $\mathcal{O}(nk^{2/3} \log^{1/3} m \log k)$ time for all positions j of T , the index of the k -th mismatch of P with the suffix $T[j, n - 1]$. Applied for $P = T$, it gives the following result.

► **Lemma 3** ([22]). *The PREF_k table of a string of length n over an integer alphabet can be computed in $\mathcal{O}(nk^{2/3} \log^{1/3} n \log k)$ time.*

We say that strings U and V have a k -mismatch prefix-suffix of length p if U has a prefix U' of length p and V has a suffix V' of length p such that $\text{Ham}(U', V') \leq k$.

2.2 Algorithmic Toolbox for Edit Distance

For $x, y \in \Sigma$, let $c(x, y)$, $c(\varepsilon, x)$ and $c(x, \varepsilon)$ be the costs of substituting letter x by letter y (equal to 0 if $x = y$), inserting letter x and deleting letter x , respectively. They are usually specified by a penalty matrix c ; it implies a metric if certain conditions are satisfied (identity of indiscernibles, symmetry, triangle inequality).

The classic dynamic programming solution to the edit distance problem (see [34]) for strings T_1 and T_2 uses the so-called D -table such that $D[i, j]$ is the edit distance between prefixes $T_1[0, i]$ and $T_2[0, j]$. Initially $D[-1, -1] = 0$, $D[i, -1] = D[i - 1, -1] + c(T_1[i], \varepsilon)$ for $i \geq 0$ and $D[-1, j] = D[-1, j - 1] + c(\varepsilon, T_2[j])$ for $j \geq 0$. For $i, j \geq 0$, $D[i, j]$ can be computed as follows:

$$D[i, j] = \min(D[i - 1, j - 1] + c(T_1[i], T_2[j]), D[i, j - 1] + c(\varepsilon, T_2[j]), D[i - 1, j] + c(T_1[i], \varepsilon)).$$

Given a threshold h on the Levenshtein distance, Landau et al. [27] show how to compute the Levenshtein distance between T_1 and bT_2 , for any $b \in \Sigma$, in $\mathcal{O}(h)$ time using previously computed solution for T_1 and T_2 (another solution was given later by Kim and Park [23]). They define an h -wave that contains indices of the last value h in diagonals of the D -table. Let $L^h(d) = \max\{i : D[i, i + d] = h\}$. Formally an h -wave is:

$$L^h = [L^h(-h), L^h(-h + 1), \dots, L^h(h - 1), L^h(h)].$$

Landau et al. [27] show how to update the h -wave when string T_2 is prepended by a single letter in $\mathcal{O}(h)$ time. This method was introduced to approximate periodicity in [32].

3 Computing k -Coverage under Hamming Distance

Let T be a string of length n and assume that its PREF_k table is given. We will show a linear-time algorithm for computing the k -coverage of every prefix of T under the Hamming distance.

In the algorithm we consider all prefix lengths $\ell = 1, \dots, n$. At each step of the algorithm, a linked list \mathcal{L} is stored that contains all positions i such that $\text{PREF}_k[i] \geq \ell$ and a sentinel value n , in an increasing order. The list is stored together with a table $A(\mathcal{L})[0..n - 1]$ such that $A(\mathcal{L})[i]$ is a link to the occurrence of i in \mathcal{L} or **nil** if $i \notin \mathcal{L}$. It can be used to access and remove a given element of \mathcal{L} in $\mathcal{O}(1)$ time. Before the start of the algorithm, \mathcal{L} contains all numbers $0, \dots, n$.

If $i \in \mathcal{L}$ and j is the successor of i in \mathcal{L} , then the approximate occurrence of $T[0, \ell - 1]$ at position i accounts for $\min(\ell, j - i)$ positions that are covered in T . A pair of adjacent elements $i < j$ in \mathcal{L} is called *overlapping* if $j - i < \ell$ and *non-overlapping* otherwise. Hence, each non-overlapping adjacent pair adds the same amount to the number of covered positions.

All pairs of adjacent elements of \mathcal{L} are partitioned in two data structures, \mathcal{D}_o and \mathcal{D}_{no} , that store overlapping and non-overlapping pairs, respectively. Data structure \mathcal{D}_{no} stores non-overlapping pairs (i, j) in buckets that correspond to $j - i$, in a table $B(\mathcal{D}_{no})$ indexed from 1 to n . It also stores a table $A(\mathcal{D}_{no})$ indexed 0 through $n - 1$ such that $A(\mathcal{D}_{no})[i]$ points to the location of (i, j) in its bucket, provided that such a pair exists for some j , or **nil** otherwise. Finally, it remembers the number $num(\mathcal{D}_{no})$ of stored adjacent pairs. \mathcal{D}_o does not store the overlapping adjacent pairs (i, j) explicitly, just the sum of values $j - i$, as $sum(\mathcal{D}_o)$. Then

$$\text{Covered}_k^{Ham}(T[0, \ell - 1], T) = sum(\mathcal{D}_o) + num(\mathcal{D}_{no}) \cdot \ell. \quad (1)$$

Now we need to describe how the data structures are updated when ℓ is incremented.

In the algorithm we store a table $Q[0..n]$ of buckets containing pairs $(\text{PREFIX}_k[i], i)$ grouped by the first component. When ℓ changes to $\ell + 1$, the second components of all pairs from $Q[\ell]$ are removed, one by one, from the list \mathcal{L} (using the table $A(\mathcal{L})$).

Let us describe what happens when element q is removed from \mathcal{L} . Let q_1 and q_2 be its predecessor and successor in \mathcal{L} . (They exist because 0 and n are never removed from \mathcal{L} .) Then each of the pairs (q_1, q) and (q, q_2) is removed from the respective data structure \mathcal{D}_o or \mathcal{D}_{no} , depending on the difference of elements. Removal of a pair (i, j) from \mathcal{D}_o simply consists in decreasing $sum(\mathcal{D}_o)$ by $j - i$, whereas to remove (i, j) from \mathcal{D}_{no} one needs to remove it from the right bucket (using the table $A(\mathcal{D}_{no})$) and decrement $num(\mathcal{D}_{no})$. In the end, the pair (q_1, q_2) is inserted to \mathcal{D}_o or to \mathcal{D}_{no} depending on $q_2 - q_1$. Insertion to \mathcal{D}_o and to \mathcal{D}_{no} is symmetric to deletion.

When ℓ is incremented, non-overlapping pairs (i, j) with $j - i = \ell$ become overlapping. Thus, all pairs from the bucket $B(\mathcal{D}_{no})[\ell]$ are removed from \mathcal{D}_{no} and inserted to \mathcal{D}_o .

This concludes the description of operations on the data structures. Correctness of the resulting algorithm follows from (1). We analyze its complexity in the following theorem.

► **Theorem 4.** *Let T be a string of length n . Assuming that the PREFIX_k table for string T is given, the k -coverage of every prefix of T under the Hamming distance can be computed in $\mathcal{O}(n)$ time.*

Proof. There are up to n removals from \mathcal{L} . Initially \mathcal{L} contains n adjacent pairs. Every removal from \mathcal{L} introduces one new adjacent pair, so the total number of adjacent pairs that are considered in the algorithm is $2n - 1$. Each adjacent pair is inserted to \mathcal{D}_o or to \mathcal{D}_{no} , then it may be moved from \mathcal{D}_{no} to \mathcal{D}_o , and finally it is removed from its data structure. In total, $\mathcal{O}(n)$ insertions and deletions are performed on the two data structures, in $\mathcal{O}(1)$ time each. This yields the desired time complexity of the algorithm. ◀

Let us note that in order to compute the k -coverage of all factors of T that start at a given position i , it suffices to use a table $[\text{lcp}_k(i, 0), \dots, \text{lcp}_k(i, n - 1)]$ instead of PREFIX_k . Together with Lemma 2 this gives the following result.

► **Corollary 5.** *Let T be a string of length n . The k -coverage of every factor of T under the Hamming distance can be computed in $\mathcal{O}(n^2)$ time.*

4 Computing k -Coverage under Edit Distance

Let us state an abstract problem that, to some extent, is a generalization of the k -mismatch lcp-queries to the edit distance.

LONGEST APPROXIMATE PREFIX PROBLEM

Input: A string T of length n , a metric d and a number k

Output: A table P_k^d such that $P_k^d[a, b, a']$ is the maximum $b' \geq a' - 1$ such that $d(T[a, b], T[a', b']) \leq k$ or -1 if no such b' exists.

Having the table P_k^d , one can easily compute the k -coverage of a factor $T[a, b]$ under metric d as:

$$\text{Covered}_k^d(T[a, b], T) = \left| \bigcup_{a'=0}^{n-1} [a', P_k^d[a, b, a']] \right|, \quad (2)$$

where an interval of the form $[a', b']$ for $b' < a'$ is considered to be empty. The size of the union of n intervals can be computed in $\mathcal{O}(n)$ time, which gives $\mathcal{O}(n^3)$ time over all factors.

In Section 4.1 and 4.2 we show how to compute the tables P_k^{Lev} and P_k^{ed} for a given threshold k in $\mathcal{O}(n^3)$ and $\mathcal{O}(n^3 \sqrt{n \log n})$ time, respectively. Then in Section 4.3 we apply the techniques of Section 4.2 to obtain an $\mathcal{O}(n^3 \sqrt{n \log n})$ -time algorithm for computing restricted approximate covers and seeds under the edit distance.

4.1 Longest Approximate Prefix under Levenshtein Distance

Let $H_{i,j}$ be the h -wave for strings $T[i, n-1]$ and $T[j, n-1]$ and $h = k$. Then we can compute P_k^{Lev} with Algorithm 1. The algorithm basically takes the rightmost diagonal of D -table in which the value in row $b - a + 1$ is less than or equal to k .

Algorithm 1 Computing P_k^{Lev} table.

```

1 for  $a' := n - 1$  down to 0 do
2   Compute  $H_{n-1, a'}$ ;
3   for  $a := n - 1$  down to 0 do
4     if  $a < n$  then
5       Compute  $H_{a, a'}$  from  $H_{a+1, a'}$ ;
6        $d := k$ ;
7       for  $b := a$  to  $n - 1$  do
8          $i := b - a + 1$ ;
9         while  $d \geq -k$  and  $H_{a, a'}(d) < i$  do
10           $d := d - 1$ ;
11          if  $d < -k$  then  $P_k^{Lev}[a, b, a'] := -1$ ;
12          else  $P_k^{Lev}[a, b, a'] := a' + i + d$ ;

```

The while-loop can run up to $2k$ times for given a and a' . Computing $H_{n-1, a'}$ takes $\mathcal{O}(k^2)$ time and updating $H_{a, a'}$ takes $\mathcal{O}(k)$ time. It makes the algorithm run in $\mathcal{O}(n^3)$ time. Together with Equation (2) this yields the following result.

► **Proposition 6.** *Let T be a string of length n . The k -coverage of every factor of T under the Levenshtein distance can be computed in $\mathcal{O}(n^3)$ time.*

A similar method could be used in case of constant edit operation costs, by applying the work of [20]. In the following section we develop a solution for arbitrary costs.

4.2 Longest Approximate Prefix under Edit Distance

For indices $a, a' \in [0, n]$ we define a table $D_{a,a'}$ such that $D_{a,a'}[b, b']$ is the edit distance between $T[a, b]$ and $T[a', b']$, for $b \in [a - 1, n - 1]$ and $b' \in [a' - 1, n - 1]$. For other indices we set $D_{a,a'}[b, b'] = \infty$. The $D_{a,a'}$ table corresponds to the D -table for $T[a, n - 1]$ and $T[a', n - 1]$ and so it can be computed in $\mathcal{O}(n^2)$ time.

We say that pair (d, b) (Pareto-)dominates pair (d', b') if $(d, b) \neq (d', b')$, $d \leq d'$ and $b \geq b'$. Let us introduce a data structure $L_{a,a'}[b]$ being a table of all among pairs $(D_{a,a'}[b, b'], b')$ that are maximal in this sense (i.e., are not dominated by other pairs), sorted by increasing first component. Using a folklore stack-based algorithm (Algorithm 2), this data structure can be computed from $D_{a,a'}[b, a' - 1], \dots, D_{a,a'}[b, n - 1]$ in linear time.

■ **Algorithm 2** Computing $L_{a,a'}[b]$ from $D_{a,a'}[b, \cdot]$.

```

1  $Q :=$  empty stack;
2 for  $b' := a' - 1$  to  $n - 1$  do
3    $d := D_{a,a'}[b, b'];$ 
4   while  $Q$  not empty do
5      $(d', x) := \text{top}(Q);$ 
6     if  $d' \geq d$  then  $\text{pop}(Q);$ 
7     else break;
8    $\text{push}(Q, (d, b'));$ 
9  $L_{a,a'}[b] := Q;$ 

```

Every multiple of $M = \lfloor \sqrt{n/\log n} \rfloor$ will be called a *special point*. In our algorithm we first compute the following data structures:

- (a) all $L_{a,a'}[b]$ lists where a or a' is a special point, for $a, a' \in [0, n - 1]$ and $b \in [a - 1, n - 1]$ (if $a \geq n$ or $a' \geq n$, the list is empty); and
- (b) all cells $D_{a,a'}[b, b']$ of all $D_{a,a'}$ tables for $a, a' \in [0, n]$ and $-1 \leq b - a, b' - a' < M - 1$. Computing part (a) takes $\mathcal{O}(n^4/M) = \mathcal{O}(n^3\sqrt{n\log n})$ time, whereas part (b) can be computed in $\mathcal{O}(n^4/M^2) = \mathcal{O}(n^3\log n)$ time. The intuition behind this data structure is shown in the following lemma.

► **Lemma 7.** *Assume that $b - a \geq M - 1$ or $b' - a' \geq M - 1$. Then there exists a pair of positions c, c' such that the following conditions hold:*

- $a \leq c \leq b + 1$ and $a' \leq c' \leq b' + 1$, and
- $c - a, c' - a' < M$, and
- $\text{ed}(T[a, b], T[a', b']) = \text{ed}(T[a, c - 1], T[a', c' - 1]) + \text{ed}(T[c, b], T[c', b']),$ and
- at least one of c, c' is a special point.

Moreover, if c (c') is the special point, then $c \leq b$ ($c' \leq b'$, respectively).

Proof. By the assumption, at least one of the intervals $[a, b]$ and $[a', b']$ contains a special point. Let $p \in [a, b]$ and $p' \in [a', b']$ be the smallest among them; we have $p - a, p' - a' < M$ provided that p or p' exists, respectively (otherwise p or p' is set to ∞). Let us consider the table $D_{a,a'}$ and how its cell $D_{a,a'}[b, b']$ is computed. We can trace the path of parents in the dynamic programming from $D_{a,a'}[b, b']$ to the origin $(D_{a,a'}[a - 1, a' - 1])$. Let us traverse this path in the reverse direction until the first dimension of the table reaches p or the second

dimension reaches p' . Say that just before this step we are at $D_{a,a'}[q, q']$. If $q + 1 = p$ and $q' < p'$, then we set $c = q + 1$ and $c' = q' + 1$. Indeed $c = p$ is a special point,

$$ed(T[a, b], T[a', b']) = ed(T[a, c - 1], T[a', c' - 1]) + ed(T[c, b], T[c', b'])$$

and $c - a, c' - a' < M$. Moreover, $q' \in [a' - 1, b']$, so $c' \in [a', b' + 1]$. The opposite case (that $q' + 1 = p'$) is symmetric. \blacktriangleleft

If $P_k^{ed}[a, b, a'] - a' < M - 1$, then it can be computed using one of the $M \times M$ prefix fragments of the $D_{a,a'}$ tables. Otherwise, according to the statement of the lemma, one of the $L_{c,c'}[b]$ lists can be used, where $c - a, c' - a' < M$, as shown in Algorithm 3. The algorithm uses a predecessor operation $Pred(x, L)$ which for a number x and a list $L = L_{c,c'}[b]$ returns the maximal pair whose first component does not exceed x , or (∞, ∞) if no such pair exists. This operation can be implemented in $\mathcal{O}(\log n)$ time via binary search.

■ **Algorithm 3** Computing $P_k^{ed}[a, b, a']$.

```

1  res := -1;
2  if b - a < M - 1 then
3    for b' := a' - 1 to a' + M - 2 do
4      if D_{a,a'}[b, b'] ≤ k then
5        res := b';
6  s := a + ((-a) mod M); s' := a' + ((-a') mod M); // closest special pts
7  foreach (c, c') in ({s} × [a', a' + M - 1]) ∪ ([a, a + M - 1] × {s'}) do
8    (d', b') := Pred(k - D_{a,a'}[c - 1, c' - 1], L_{c,c'}[b]);
9    if d' ≠ ∞ then
10     res := max(res, b');
11 P_k^{ed}[a, b, a'] := res;

```

► **Theorem 8.** *Let T be a string of length n . The k -coverage of every factor of T under the edit distance can be computed in $\mathcal{O}(n^3 \sqrt{n \log n})$ time.*

Proof. We want to show that Algorithm 3 correctly computes $P_k^{ed}[a, b, a']$. Let us first check that the result $b' = res$ of Algorithm 3 satisfies $D_{a,a'}[b, b'] \leq k$. It is clear if the algorithm computes b' in line 5. Otherwise, it is computed in line 10. This means that $L_{c,c'}[b]$ contains a pair $(D_{c,c'}[b, b'], b')$ such that

$$k \geq D_{c,c'}[b, b'] + D_{a,a'}[c - 1, c' - 1] \geq D_{a,a'}[b, b'].$$

Now we show that the returned value res is at least $x = P_k^{ed}[a, b, a']$. If $b - a < M - 1$ and $x - a' < M - 1$, then the condition in line 4 holds for $b' = x$, so indeed $res \geq x$. Otherwise, the condition of Lemma 7 is satisfied. The lemma implies two positions c, c' such that at least one of them is special and that satisfy additional constraints.

If c is special, then the constraints $a \leq c$ and $c - a < M$ imply that $c = s$, as defined in line 6. Additionally, $a' \leq c' \leq a' + M - 1$, so (c, c') will be considered in the loop from line 7. By the lemma and the definition of x , we have

$$D_{c,c'}[b, x] = D_{a,a'}[b, x] - D_{a,a'}[c - 1, c' - 1] \leq k - D_{a,a'}[c - 1, c' - 1]. \quad (3)$$

18:10 k -Approximate Quasiperiodicity under Hamming and Edit Distance

The list $L_{c,c'}[b]$ either contains the pair $(D_{c,c'}[b,x], x)$, or a pair $(D_{c,c'}[b,x'], x')$ such that $D_{c,c'}[b,x'] \leq D_{c,c'}[b,x]$ and $x' > x$. In the latter case by (3) we would have

$$k \geq D_{a,a'}[c-1, c'-1] + D_{c,c'}[b,x] \geq D_{a,a'}[c-1, c'-1] + D_{c,c'}[b,x'] \geq D_{a,a'}[b,x']$$

and $x' > x$. In both cases the predecessor computed in line 8 returns a value res such that $res \geq x$ and $res \neq \infty$. The case that c' is special admits an analogous argument.

Combining Algorithm 3 with Equation (2), we obtain correctness of the computation.

As for complexity, Algorithm 3 computes $P_k^{ed}[a, b, a']$ in $\mathcal{O}(M \log n) = \mathcal{O}(\sqrt{n \log n})$ time and the pre-computations take $\mathcal{O}(n^3 \sqrt{n \log n})$ total time. \blacktriangleleft

4.3 Restricted Approximate Covers and Seeds under Edit Distance

The techniques that were developed in Section 4.2 can be used to improve upon the $\mathcal{O}(n^4)$ time complexity of the algorithms for computing the restricted approximate covers and seeds under the edit distance [10, 33]. We describe our solution only for restricted approximate covers; the solution for restricted approximate seeds follows by considering the text $\diamond^{|T|} T \diamond^{|T|}$.

Let us first note that the techniques from the previous subsection can be used as a black box to solve the problem in scope in $\mathcal{O}(n^3 \sqrt{n \log n} \log(nw))$ time, where w is the maximum cost of an edit operation. Indeed, for every factor $T[a, b]$ we binary search for the smallest k for which $T[a, b]$ is a k -approximate cover of T . A given value k is tested by computing the tables $P_k^{ed}[a, b, a']$ for all $a' = 0, \dots, n-1$ and checking if $\text{Covered}_k^d(T[a, b], T) = n$ using Equation (2).

Now we proceed to a more efficient solution. Same as in the algorithms from [10, 33] we compute, for every factor $T[a, b]$, a table $Q_{a,b}[0..n]$ such that $Q_{a,b}[i]$ is the minimum edit distance threshold k for which $T[a, b]$ is a k -approximate cover of $T[i, n-1]$. In the end, all factors $T[a, b]$ for which $Q_{a,b}[0]$ is minimal need to be reported as restricted approximate covers of T . We will show how, given the data structures (a) and (b) of the previous section, we can compute this table in $\mathcal{O}(nM \log n)$ time.

A dynamic programming algorithm for computing the $Q_{a,b}$ table, similar to the one in [10], is shown in Algorithm 4. Computing $Q_{a,b}$ takes $\mathcal{O}(n^2)$ time provided that all $D_{a,b}$ arrays, of total size $\mathcal{O}(n^4)$, are available. The algorithm considers all possibilities for the approximate occurrence $T[i, j]$ of $T[a, b]$.

■ **Algorithm 4** Computing $Q_{a,b}$ in quadratic time.

```

1  $Q_{a,b}[n] := 0;$ 
2 for  $i := n - 1$  down to 0 do
3    $Q_{a,b}[i] := \infty;$ 
4    $minQ := \infty;$ 
5   for  $j := i$  to  $n - 1$  do
6      $minQ := \min(minQ, Q_{a,b}[j + 1]);$  //  $minQ = \min Q_{a,b}[i + 1..j + 1]$ 
7      $Q_{a,b}[i] := \min(Q_{a,b}[i], \max(D_{a,i}[b, j], minQ));$ 

```

During the computation of $Q_{a,b}$, we will compute a data structure for on-line range-minimum queries over the table. We can use the following simple data structure with $\mathcal{O}(n \log n)$ total construction time and $\mathcal{O}(1)$ -time queries. For every position i and power of two 2^p , we store as $RM[i, p]$ the minimal value in the table $Q_{a,b}$ on the interval $[i, i + 2^p - 1]$. When a new value $Q_{a,b}[i]$ is computed, we compute $RM[i, 0] = Q_{a,b}[i]$ and $RM[i, p]$ for all $0 < p \leq \log_2(n - i)$ using the formula $RM[i, p] = \min(RM[i, p - 1], RM[i + 2^{p-1}, p - 1])$.

Then a range-minimum query over an interval $[i, j]$ of $Q_{a,b}$ can be answered by inspecting up to two cells of the RM table for p such that $2^p \leq j - i + 1 < 2^{p+1}$.

Let us note that the variable $\min Q$, which denotes the minimum of a growing segment in the $Q_{a,b}$ table, can only decrease. We would like to make the second argument of \max in line 7 non-decreasing for increasing j . The values $ed(T[a, b], T[i, j]) = D_{a,i}[b, j]$ may increase or decrease as j grows. However, it is sufficient to consider only those values of j for which $(D_{a,i}[b, j], j)$ is not (Pareto-)dominated (as in Section 4.2), i.e., the elements of the list $L_{a,i}[b]$. For these values, $D_{a,i}[b, j]$ is indeed increasing for increasing j . The next observation follows from this monotonicity and the monotonicity of $\min Q_{a,b}[i+1..j+1]$.

► **Observation 9.** *Let $(D_{a,i}[b, j'], j')$ be the first element on the list $L_{a,i}[b]$ such that*

$$\min Q_{a,b}[i+1..j'+1] \leq D_{a,i}[b, j'].$$

If j' does not exist, we simply take the last element of $L_{a,i}[b]$. Further let $(D_{a,i}[b, j''], j'')$ be the predecessor of $(D_{a,i}[b, j'], j')$ in $L_{a,i}[b]$ (if it exists). Then $j \in \{j', j''\}$ minimizes the value of the expression $\max(\min Q_{a,b}[i+1..j+1], D_{a,i}[b, j])$.

If we had access to the list $L_{a,i}[b]$, we could use binary search to locate the index j' defined in the observation. However, we only store the lists $L_{a,i}[b]$ for a and i such that at least one of them is a special point. We can cope with this issue by separately considering all j such that $j < i + M - 1$ and then performing binary search on every of $\mathcal{O}(M)$ lists $L_{c,c'}[b]$ where $a \leq c < a + M$, $i \leq c' < i + M$ and at least one of c, c' is a special point, just as in Algorithm 3. A pseudocode of the resulting algorithm is given as Algorithm 5.

■ **Algorithm 5** Computing $Q_{a,b}$ in $\mathcal{O}(n\sqrt{n \log n})$ time using pre-computed data structures.

```

1  $Q_{a,b}[n] := 0;$ 
2 for  $i := n - 1$  down to 0 do
3    $Q_{a,b}[i] := \infty;$ 
4    $\min Q := \infty;$ 
5   if  $b - a < M - 1$  then
6     for  $j := i$  to  $i + M - 2$  do
7        $\min Q := \min(\min Q, Q_{a,b}[j + 1]);$ 
8        $Q_{a,b}[i] := \min(Q_{a,b}[i], \max(D_{a,i}[b, j], \min Q));$ 
9    $s := a + ((-a) \bmod M); s' := i + ((-i) \bmod M);$ 
10  foreach  $(c, c')$  in  $(\{s\} \times [i, i + M - 1]) \cup ([a, a + M - 1] \times \{s'\})$  do
11    if  $L_{c,c'}[b]$  is empty then continue;
12    /* Binary search */
13     $(d_{j'}, j') :=$  the first pair in  $L_{c,c'}[b]$  such that
14     $\min Q_{a,b}[i + 1..j' + 1] \leq D_{a,i}[c - 1, c' - 1] + d_{j'}$  or the last pair;
15     $(d_{j''}, j'') :=$  predecessor of  $(d_{j'}, j')$  in  $L_{c,c'}[b]$  or  $(d_{j'}, j')$  if there is none;
16    foreach  $j$  in  $\{j', j''\}$  do
17       $Q_{a,b}[i] := \min(Q_{a,b}[i], \max(D_{a,i}[c - 1, c' - 1] + d_j, \min Q_{a,b}[i + 1..j + 1]));$ 

```

Let us summarize the complexity of the algorithm. Pre-computation of auxiliary data structures requires $\mathcal{O}(n^3 \sqrt{n \log n})$ time. Then for every factor $T[a, b]$ we compute the table $Q_{a,b}$. The data structure for constant-time range-minimum queries over the table costs only additional $\mathcal{O}(n \log n)$ space and computation time. When computing $Q_{a,b}[i]$ using dynamic programming, we may separately consider first $M - 1$ indices j , and then we perform a binary search in $\mathcal{O}(M)$ lists $L_{c,c'}[b]$. In total, the time to compute $Q_{a,b}[i]$ given a, b, i is $\mathcal{O}(M \log n) = \mathcal{O}(\sqrt{n \log n})$.

► **Theorem 10.** *Let T be a string of length n . All restricted approximate covers and seeds of T under the edit distance can be computed in $\mathcal{O}(n^3\sqrt{n\log n})$ time.*

The work of [10, 33] on approximate covers and seeds originates from a study of approximate periods [32]. Interestingly, while our algorithm improves upon the algorithms for computing approximate covers and seeds, it does not work for approximate periods.

5 NP-hardness of General Hamming k -Approximate Cover and Seed

We make a reduction from the following problem.

HAMMING STRING CONSENSUS

Input: Strings S_1, \dots, S_m , each of length ℓ , and an integer $k < \ell$

Output: A string S , called consensus string, such that $\text{Ham}(S, S_i) \leq k$ for all $i = 1, \dots, m$

The following fact is known.

► **Fact 11** ([15]). *HAMMING STRING CONSENSUS is NP-complete even for the binary alphabet.*

Let strings S_1, \dots, S_m of length ℓ over the alphabet $\Sigma = \{0, 1\}$ and integer k be an instance of HAMMING STRING CONSENSUS. We introduce a morphism ϕ such that

$$\phi(0) = 0^{2k+4} 1010 0^{2k+4}, \quad \phi(1) = 0^{2k+4} 1011 0^{2k+4}.$$

We will exploit the following simple property of this morphism.

► **Observation 12.** *For every string S , every length- $(2k+4)$ factor of $\phi(S)$ contains at most three ones.*

We set $\gamma_i = 1^{2k+4}\phi(S_i)$ and $T = \gamma_1 \dots \gamma_m$. Further let $\psi(U)$ be an operation that reverses this encoding, i.e., $\psi(\gamma_i) = S_i$. Formally, it takes as input a string U and outputs $U[4k+12-1]U[2 \cdot (4k+12)-1] \dots U[(\ell-1)(4k+12)-1]$.

► **Lemma 13.** *Strings γ_i and γ_j , for any $i, j \in \{1, \dots, m\}$, have no $2k$ -mismatch prefix-suffix of length $p \in \{2k+4, \dots, |\gamma_i|-1\}$.*

Proof. We will show that the prefix U of γ_i of length p and the suffix V of γ_j of length p have at least $2k+1$ mismatches. Let us note that U starts with 1^{2k+4} . The proof depends on the value $d = |\gamma_i| - p$; we have $1 \leq d \leq |\gamma_i| - 2k - 4$. Let us start with the following observation that can be readily verified.

► **Observation 14.** *For $A, B \in \{1010, 1011\}$, the strings $A0^4$ and 0^4B have no 1-mismatch prefix-suffix of length in $\{5, \dots, 8\}$.*

If $1 \leq d \leq 4$, then U and V have a mismatch at position $2k+4$ since V starts with $1^{2k+4-d}0$. Moreover, they have at least $2d$ mismatches by the observation (applied for the prefix-suffix length $d+4$). In total, $\text{Ham}(U, V) \geq 2d+1 \geq 2k+1$.

If $4 < d < 2k+4$, then every block 1010 or 1011 in γ_i and in γ_j is matched against a block of zeroes in the other string, which gives at least $4d$ mismatches. Hence, $\text{Ham}(U, V) \geq 4d \geq 2k+1$.

Finally, if $2k+4 \leq d \leq |\gamma_i| - 2k - 4$, then U starts with 1^{2k+4} and every factor of V of length $2k+4$ has at most three ones (see Observation 12). Hence, $\text{Ham}(U, V) \geq 2k+1$. ◀

The following lemma gives the reduction.

► **Lemma 15.** *If HAMMING STRING CONSENSUS for S_1, \dots, S_m has a positive answer, then the GENERAL k -APPROXIMATE COVER under Hamming distance for T , k , and $c = |\gamma_i|$ returns a k -approximate cover C such that $S = \psi(C)$ is a Hamming consensus string for S_1, \dots, S_m .*

Proof. By Lemma 13, if C is a k -approximate cover of T of length c , then every position $a \in \text{StartOcc}_k^H(C, T)$ satisfies $c \mid a$. Hence, $\text{StartOcc}_k^H(C, T) = \{0, c, 2c, \dots, (m-1)c\}$.

If HAMMING STRING CONSENSUS for S_1, \dots, S_m has a positive answer S , then $1^{2k+4}\phi(S)$ is a k -approximate cover of T of length c . Moreover, if T has a k -approximate cover C of length c , then for $S = \psi(C)$ and for each $i = 1, \dots, m$, we have that

$$\text{Ham}(C, T[(i-1)c, ic-1]) \geq \text{Ham}(S, S_i),$$

so S is a consensus string for S_1, \dots, S_m . This completes the proof. ◀

Lemma 15 and Fact 11 imply that computing k -approximate covers is NP-hard. Obviously, it is in NP.

► **Theorem 16.** *GENERAL k -APPROXIMATE COVER under the Hamming distance is NP-complete even over a binary alphabet.*

A lemma that is similar to Lemma 15 can be shown for approximate seeds. We leave its technical proof for the full version of the paper. Let $T' = \gamma_1\gamma_1 \dots \gamma_m 1^{2k+4} \gamma_m 1^{2k+4}$.

► **Lemma 17.** *If HAMMING STRING CONSENSUS for S_1, \dots, S_m has a positive answer, then the GENERAL k -APPROXIMATE SEED under Hamming distance for T' , k , and $c = |\gamma_1| + 2k + 4$ returns a k -approximate seed C such that $S = \psi(C')$ is a Hamming consensus string for S_1, \dots, S_m for some cyclic shift C' of C .*

► **Theorem 18.** *GENERAL k -APPROXIMATE SEED under the Hamming distance is NP-complete even over a binary alphabet.*

6 Conclusions

We have presented several polynomial-time algorithms for computing restricted approximate covers and seeds and k -coverage under Hamming, Levenshtein and weighted edit distances and shown NP-hardness of non-restricted variants of these problems under the Hamming distance. It is not clear if any of the algorithms are optimal. The only known related conditional lower bound shows hardness of computing the Levenshtein distance of two strings in strongly subquadratic time [8]; however, our algorithms for approximate covers under edit distance work in $\Omega(n^3)$ time. An interesting open problem is if restricted approximate covers or seeds under Hamming distance, as defined in [10, 33], can be computed in $\mathcal{O}(n^{3-\epsilon})$ time, for any $\epsilon > 0$. Here we have shown an efficient solution for k -restricted versions of these problems.

References

- 1 Amihod Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 25:1–25:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPICs.CPM.2017.25.

- 2 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? *Algorithmica*, 81(7):2857–2875, 2019. doi:10.1007/s00453-019-00559-8.
- 3 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPICs*, pages 26:1–26:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.26.
- 4 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. *Theoretical Computer Science*, 793:59–69, 2019. doi:10.1016/j.tcs.2019.05.020.
- 5 Amihood Amir, Avivit Levy, and Ely Porat. Quasi-periodicity under mismatch errors. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, volume 105 of *LIPICs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPICs.CPM.2018.4.
- 6 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.
- 7 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.
- 8 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.
- 9 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.
- 10 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *Journal of Automata, Languages and Combinatorics*, 10(5/6):609–626, 2005. doi:10.25596/jalc-2005-609.
- 11 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2003. doi:10.1142/4838.
- 12 Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *CoRR (accepted to Theoretical Computer Science)*, 2019. arXiv:1909.11336.
- 13 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015. doi:10.1016/j.ipl.2015.03.006.
- 14 Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theoretical Computer Science*, 506:102–114, 2013. doi:10.1016/j.tcs.2013.08.013.
- 15 Moti Frances and Ami Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997. doi:10.1007/s002240000044.
- 16 Ondřej Guth. *Searching Regularities in Strings using Finite Automata*. PhD thesis, Czech Technical University in Prague, 2014. URL: https://fit.cvut.cz/sites/default/files/PhDThesis_Guth.pdf.
- 17 Ondřej Guth. On approximate enhanced covers under Hamming distance. *Discrete Applied Mathematics*, 274:67–80, 2020. doi:10.1016/j.dam.2019.01.015.
- 18 Ondřej Guth and Bořivoj Melichar. Using finite automata approach for searching approximate seeds of strings. In Xu Huang, Sio-Iong Ao, and Oscar Castillo, editors, *Intelligent Automation and Computer Engineering*, pages 347–360, Dordrecht, 2010. Springer Netherlands. doi:10.1007/978-90-481-3517-2_27.
- 19 Ondřej Guth, Bořivoj Melichar, and Miroslav Balík. Searching all approximate covers and their distance using finite automata. In Peter Vojtás, editor, *Proceedings of the Conference on Theory and Practice of Information Technologies, ITAT 2008*, volume 414 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL: <http://ceur-ws.org/Vol-414/paper4.pdf>.

- 20 Heikki Hyvrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *Journal of Discrete Algorithms*, 34:2–17, 2015. doi:10.1016/j.jda.2015.05.007.
- 21 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. doi:10.1007/BF01955677.
- 22 Haim Kaplan, Ely Porat, and Nira Shafrir. Finding the position of the k -mismatch and approximate tandem repeats. In Lars Arge and Rusins Freivalds, editors, *Algorithm Theory - SWAT 2006, 10th Scandinavian Workshop on Algorithm Theory*, volume 4059 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2006. doi:10.1007/11785293_11.
- 23 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.
- 24 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):Article 27, April 2020. doi:10.1145/3386369.
- 25 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient algorithms for shortest partial seeds in words. *Theoretical Computer Science*, 710:139–147, 2018. doi:10.1016/j.tcs.2016.11.035.
- 26 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. doi:10.1007/s00453-014-9915-3.
- 27 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.
- 28 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.
- 29 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.
- 30 Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.
- 31 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.
- 32 Jeong Seop Sim, Costas S. Iliopoulos, Kunsoo Park, and William F. Smyth. Approximate periods of strings. *Theoretical Computer Science*, 262(1):557–568, 2001. doi:10.1016/S0304-3975(00)00365-0.
- 33 Jeong Seop Sim, Kunsoo Park, Sung-Ryul Kim, and Jee-Soo Lee. Finding approximate covers of strings. *Journal of Korea Information Science Society*, 29(1):16–21, 2002. URL: http://www.koreascience.or.kr/article/ArticleFullRecord.jsp?cn=JBGHG6_2002_v29n1_16.
- 34 R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

Longest Common Subsequence on Weighted Sequences

Evangelos Kipouridis 

Basic Algorithms Research Copenhagen (BARC), University of Copenhagen, Denmark
kipouridis@di.ku.dk

Kostas Tsichlas

Computer Engineering and Informatics Department, University of Patras, Greece
ktsichlas@ceid.upatras.gr

Abstract


We consider the general problem of the Longest Common Subsequence (*LCS*) on weighted sequences. Weighted sequences are an extension of classical strings, where in each position every letter of the alphabet may occur with some probability. Previous results presented a *PTAS* and noticed that no *FPTAS* is possible unless $P = NP$. In this paper we essentially close the gap between upper and lower bounds by improving both. First of all, we provide an *EPTAS* for bounded alphabets (which is the most natural case), and prove that there does not exist any *EPTAS* for unbounded alphabets unless $FPT = W[1]$. Furthermore, under the Exponential Time Hypothesis, we provide a lower bound which shows that no significantly better *PTAS* can exist for unbounded alphabets. As a side note, we prove that it is sufficient to work with only one threshold in the general variant of the problem.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis; Theory of computation → W hierarchy; Theory of computation → Problems, reductions and completeness

Keywords and phrases WLCS, LCS, weighted sequences, approximation algorithms, lower bound

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.19

Related Version Full version available at arXiv: <https://arxiv.org/abs/1901.04068>.

Funding *Evangelos Kipouridis*: Thorup's Investigator Grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation, and European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 801199. 

Acknowledgements We would like to thank the anonymous reviewers for their careful reading of our paper and their many insightful comments and suggestions.

1 Introduction

1.1 General concepts

We consider the problem of determining the *LCS* (Longest Common Subsequence) on weighted sequences. Weighted sequences, also known as p -weighted sequences or Position Weighted Matrices (PWM) [3, 36] are probabilistic sequences which extend the notion of strings, in the sense that in each position there is some probability for each letter of an alphabet Σ to occur there.

Weighted sequences were introduced as a tool for motif discovery and local alignment and are extensively used in molecular biology [23]. They have been studied both in the context of short sequences (binding sites, sequences resulting from multiple alignment, etc.) and on large sequences, such as complete chromosome sequences that have been obtained



© Evangelos Kipouridis and Kostas Tsichlas;
licensed under Creative Commons License CC-BY
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 19; pp. 19:1–19:15



Leibniz International Proceedings in Informatics

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

using a whole-genome shotgun strategy [32, 37]. Weighted sequences are able to keep all the information produced by such strategies, while classical strings impose restrictions that oversimplify the original data.

Basic concepts concerning the combinatorics of weighted sequences (like pattern matching, repeats discovery and cover computation) were studied using weighted suffix trees [26], Crochemore's partitioning [9, 11, 18], the Karp-Miller-Rabin algorithm [18], and other approaches [43, 30]. Other interesting results include approximate and gapped pattern matching [6, 41, 34], online pattern matching [16], weighted indexing [2, 10], swapped matching [40], the all-covers and all-seeds problem [39, 42], extracting motifs [28], and the weighted shortest common supersequence problem [4, 17]. There are also some more practical results on mapping short weighted sequences to a reference genome [7] (also studied in the parallel setting [27]), as well as on the reporting version of the problem which we also consider in this paper [11].

The Longest Common Subsequence (*LCS*) problem is a well-known measure of similarity between two strings. Given two strings, the output should be the length of the longest subsequence common to both strings. Dynamic programming solutions [25, 38] for this problem are classical textbook algorithms in Computer Science. *LCS* has been applied in computational biology for measuring the commonality of DNA molecules or proteins which may yield similar functionality. A very interesting survey on algorithms for the *LCS* can be found in [13]. The current *LCS* algorithms are considered optimal, since matching lower bounds (under the Strong Exponential Time Hypothesis) were proven [1, 14].

Extensions of this problem on more general structures have also been investigated (trees and matrices [5], run-length encoded strings [8], and more). One interesting variant of the *LCS* is the Heaviest Common Subsequence (*HCS*) where the matching between different letters is assigned a different weight, and the goal is to maximize the weight of the common subsequence, rather than its length.

1.2 Weighted LCS

The problem studied in this paper is the weighted *LCS* (WLCS) problem. It was introduced by Amir et al. [3] as an extension of the classical *LCS* problem on weighted sequences. Given two weighted sequences, the goal is to find a longest string which has a high probability of appearing in both sequences. Amir et al. initially solved an easier version of this problem in polynomial time, but unfortunately its applications are limited. As far as the general problem is concerned, they hinted its NP-Hardness by giving an NP-Hardness result on a closely related problem, which they call the log-probability version of WLCS. In short, the problem is the same, but all products in its definition are replaced with sums. Their proof is based on a Turing reduction and only works for unbounded alphabets. Finally, Amir et al. provide an $\frac{1}{|\Sigma|}$ -approximation algorithm for the WLCS problem.

Cygan et al. [19] strengthened the evidence that WLCS is NP-Hard by providing an NP-Completeness result on the decision log-probability version of WLCS (informally introduced in the previous paragraph), already for alphabets of size 2, using a Karp reduction; for alphabets of size 1 the solution is trivial since there is no uncertainty. They also gave an $\frac{1}{2}$ -approximation algorithm and a *PTAS*, while also noticing that an *FPTAS* cannot exist, assuming WLCS is indeed NP-Hard, as hinted by their evidence, and that $P \neq NP$. Finally, they proved that every instance of the problem can be reduced to a more restricted class of instances. However, for this to be achieved their algorithm needs to perform exact computations of roots and logarithms that may make the algorithm to err.

Finally, it is worth noting that Charalampopoulos et al. [17], proved that unless $P=NP$, WLCS cannot be solved in $\mathcal{O}(n^{f(a)})$ time, for any function $f(a)$, where a is the cut-off probability. We note that this result concerns exact computations rather than approximations.

1.3 Our results

In this paper we essentially close the gap between upper and lower bounds for WLCS by improving both; we prove that the problem is indeed NP-Hard even for alphabets of size 2. Furthermore, we provide an *EPTAS* for bounded alphabets. These two results, along with the *FPTAS* observation by Cygan et al. completely characterize the complexity of WLCS for bounded alphabets. For unbounded alphabets, a *PTAS* was already known by Cygan et al. [19]. We show matching lower bounds, both by ruling out the possibility of an *EPTAS*, and by showing that, under the Exponential Time Hypothesis, no significantly better *PTAS* can exist. We also prove that every instance of WLCS can be reduced to a restricted class of instances without using roots and logarithms, thus being able to actually achieve exact computations without rounding errors that can make the algorithm err.

As noted in the previous paragraph, apart from essentially closing the gap between hardness results and faster algorithms we also circumvent the need to work with roots and logarithms as the previous results did. In short, by taking advantage of the property that $(ab)^c = a^c b^c$ and setting c to be an appropriate logarithm, previous results transformed any instance to a more manageable form. However, this transformation introduces an error that may make the algorithm err as noted in the full version of the paper [29]. Table 1 summarizes the above discussion. Table 2 summarizes our results depending on the alphabet-size.

A short discussion is in order with respect to what new insights on weighted *LCS* enabled us to achieve progress. Our most crucial observation is the fact that the problem behaves differently in the natural case of a bounded alphabet, and in the case of an unbounded alphabet. Without this distinction, closing the gap between upper and lower bounds was unlikely. That's because, on the one hand, no *EPTAS* for the general case could be found, as none existed. On the other hand, proving that no *EPTAS* exists requires reductions that work only on unbounded alphabets. The aforementioned distinction is what enabled us to understand that modifying the existing reductions, which work for alphabets of size 2, would be futile in proving $W[1]$ -Hardness.

Furthermore, it was crucial to identify that working with products is the core difficulty in proving NP-Hardness of weighted *LCS*. The introduction of the log-probability version of the weighted *LCS* reflects the assumption that the difference between working with sums and working with products is just a technicality. After [3] and [19] successfully proved NP-Hardness for the log-probability version, it was natural to attempt reducing from it for proving NP-Hardness of the weighted *LCS* problem. Despite the apparent similarities between the two problems, their difference did not allow us to craft such a reduction. For the same reason, Cygan et al. used a model that assumed infinite precision computations with reals, while we are able to avoid such a strong assumption.

1.4 Organization of the paper

The rest of the paper is organized as follows. In Section 2, we provide necessary definitions and discuss the model of computation. In Section 3, we show that WLCS is NP-Complete while in Section 4, we provide the *EPTAS* algorithm for bounded alphabets, which is also an improved *PTAS* for unbounded alphabets. In Section 5, we show that there can be no *EPTAS* for unbounded alphabets by showing that this problem is $W[1]$ -hard and in Section 6, we describe the matching conditional lower bound. We conclude in Section 7.

19:4 Longest Common Subsequence on Weighted Sequences

Due to space constraints, some proofs and technical discussions are only available on the full version of the paper [29]. More specifically, in the full version we present an algorithm that transforms any instance of our problem to an equivalent, but easier to handle, instance. We also show that the rounding errors introduced by working with reals (logarithms and roots) may cause a similar algorithm by Cygan et al. [19] to err if standard rounding is used.

■ **Table 1** Results on WLCS.

	Amir et al.	Cygan et al.	Our results
NP-Hardness of WLCS	Hinted, by NP-Hardness of Log-probability version (Turing reduction - only for unbounded alphabets)	Hinted, by NP-Hardness of Log-probability version (Karp reduction - already from alphabets size 2)	Proved (Karp Reduction - already from alphabets of size 2)
Approximation Algorithms	$\frac{1}{\Sigma}$ -Approximation	<i>PTAS</i>	<i>EPTAS</i> for bounded alphabets, Improved <i>PTAS</i> for unbounded
Proof that no <i>EPTAS</i> exists for unbounded alphabets	No	No	Yes
Lower bound on any <i>PTAS</i>	No	No	Matching the upper bound, under <i>ETH</i>
Reduction to a restricted class of instances	No	Yes, by assuming exact computations of logarithms	Yes, without any extra assumptions

■ **Table 2** Results depending on the Alphabet Size.

Alphabet Size	Previous Results	Our results
1	Trivial	Trivial
Constant Size	No <i>FPTAS</i> possible	Achieved <i>EPTAS</i>
Depending on the input	Achieved <i>PTAS</i>	No <i>EPTAS</i> possible, Improved <i>PTAS</i> , Matching Lower Bound

2 Preliminaries

2.1 Basic Definitions

Let $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_K\}$ be a finite alphabet. We deal both with bounded ($K = O(1)$) and unbounded alphabets. Σ^d denotes the set of all words of length d over Σ . Σ^* denotes the set of all words over Σ .

► **Definition 1** (Weighted Sequence). *A weighted sequence X is a sequence of functions $p_1^{(X)}, \dots, p_{|X|}^{(X)}$, where each function assigns a probability to each letter from Σ . We thus have $\sum_{j=1}^K p_i^{(X)}(\sigma_j) = 1$ for all i , and $p_i^{(X)}(\sigma_j) \geq 0$ for all i, j .*

By $WS(\Sigma)$ we denote the set of all weighted sequences over Σ . Let $X \in WS(\Sigma)$. Let $Seq_d^{|X|}$ be the set of all increasing sequences of d positions in X . For a string $s \in \Sigma^d$ and $\pi \in Seq_d^{|X|}$, define $P_X(\pi, s)$ as the probability that the subsequence on positions corresponding to π in X equals s . More formally, if $\pi = (i_1, i_2, \dots, i_d)$ and s_k denotes the k -th letter of s , then

$$P_X(\pi, s) = \prod_{k=1}^d p_{i_k}^{(X)}(s_k)$$

Denote

$$SUBS(X, a) = \{s \in \Sigma^* \mid \exists \pi \in Seq_{|s|}^{|X|} \text{ such that } P_X(\pi, s) \geq a\}$$

That is, $SUBS(X, a)$ is the set of deterministic strings which match a subsequence of X with probability at least a . Every $s \in SUBS(X, a)$ is called an a -subsequence of X .

Let us give a clarifying example. If $\Sigma = \{\sigma_1, \sigma_2\}$ and X is a long weighted sequence, where in each position the probability for each letter to appear is 0.5, then $SUBS(X, 0.3)$ does not contain $s = \sigma_1\sigma_1$, as, for any increasing subsequence of 2 positions, the probability of s appearing is $0.25 < 0.3$.

The decision problem we consider is the following:

► **Definition 2** ((a_1, a_2) -WLCS decision problem). *Given two weighted sequences X, Y , two cut-off probabilities a_1, a_2 and a number k , find if the longest string s contained in $SUBS(X, a_1) \cap SUBS(Y, a_2)$ has length at least k .*

Naturally, the respective optimization problem is the following:

► **Definition 3** ((a_1, a_2) -WLCS optimization problem). *Given two weighted sequences X, Y , and two cut-off probabilities a_1, a_2 , find the length of the longest string contained in $SUBS(X, a_1) \cap SUBS(Y, a_2)$.*

Both in the decision and the optimization version, the WLCS problem is the (a_1, a_2) -WLCS problem, where $a_1 = a_2$. We denote these (equal) probabilities by a ($a = a_1 = a_2$) for concreteness.

Let us note that the problem is only interesting if $|\Sigma| \geq 2$. For $|\Sigma| = 1$ the problem is trivial since there is no uncertainty at all. The same letter appears in every position in both strings with probability 1, and thus the answer is simply the length of the shorter weighted sequence.

Finally, let us also state that the Log-Probability version of the WLCS, studied in previous papers, is the same as the original WLCS if we replace $P_X(\pi, s) = \prod_{k=1}^d p_{i_k}^{(X)}(s_k)$ by $P_X(\pi, s) = \sum_{k=1}^d p_{i_k}^{(X)}(s_k)$.

2.2 Model of Computation

Our model of computation is the standard word *RAM*, introduced by Fredman and Willard [20] to simulate programming languages like C. The word size is $w = \Omega(\log I)$, where I is the input size in bits, so as to allow random access indexing of the whole input. Thus, arithmetic operations between words take constant time. However, due to the nature of our problem, it is necessary to compute products of many numbers. This can produce numbers that are much larger than the word size. We even allow numbers in the input to be larger than 2^w (these numbers just need to use more than one word to be represented). We generally assume that each number in the input is represented by at most B bits, but we do not pose any constraint on B other than the trivial one that $B < I$. Of course, in cases where we deal with numbers that occupy many words, we no longer have unit-cost arithmetic operations; we guarantee, however, that our results only use linear or near-linear time operations (like comparisons and multiplications) on numbers polynomial in the input size. Thus, although we do not enjoy the unit-cost assumption for arbitrary numbers, we always stay within the polynomial-time regime.

2.3 Basic Operations

In this subsection we discuss the multiplication of two B -bit input numbers in (polynomial) $Mul_w(B)$ time, where w is the word-size. For example, for integers there exists a multiplication algorithm by Harvey and van der Hoeven [24] with time complexity $Mul_w(B) = \mathcal{O}(B \log B)$ (generally the running time can also depend on w , although in this case it does not). Let us notice that although the result is unpublished yet, we use it due to its easy to read time complexity; it is trivial to use other algorithms instead, like the one from Fürer [21], or the more practical one by Schönhage and Strassen [35]. We establish the complexity of multiplying x B -bit numbers. Our divide and conquer algorithm splits the numbers into two (equal sized) groups, recursively multiplies each, and multiplies the results in $Mul_w(\frac{xB}{2})$ time. By a direct application of the Master Theorem by Bentley et al. [12] we prove the following lemma.

- **Lemma 4.** *Multiplying x B -bit numbers costs*
- $\mathcal{O}(Mul_w(xB) \log(xB))$ time if $Mul_w(xB) = \Theta(xB \log^k(xB))$ for some constant k ,
 - $\mathcal{O}((xB)^c)$ else if $Mul_w(xB) = \mathcal{O}((xB)^c)$ for some constant $c \geq 1$,
- assuming that $Mul_w(N)$ is a polynomial time algorithm that multiplies two N -bit numbers.

Proof. The algorithm simply splits the numbers in two equal-sized groups, recursively multiplies each, and then multiplies the results. Let $N = xB$. We have that the running time for multiplying x B -bit numbers is $T(N) = 2T(\frac{N}{2}) + Mul_w(N)$. Since $c_{crit} = \log_2 2 = 1$, and $Mul_w(N) = \Omega(N)$, the Master Theorem [12] gives two cases. Either $Mul_w(N) = \Theta(N \log^k(N))$ for some constant k , in which case $T(N) = \mathcal{O}(Mul_w(N) \log N)$, or else $Mul_w(N) = \mathcal{O}(N^c)$ for some constant $c \geq 1$ (such a constant exists since we assume polynomial time multiplications). In this case, since it holds that $2Mul_w(\frac{N}{2}) \leq 2Mul_w(N)$, we get that $T(N) = Mul_w(N)$ if $c > c_{crit} = 1$. Notice that we handled all cases, since $Mul_w(N) = N$ is handled by the first case with $k = 0$, and whatever does not fit in the first case, definitely fits in the second, since we assumed that $Mul_w(N)$ is polynomial in N . ◀

- **Corollary 5.** *Multiplying x B -bit numbers costs polynomial time by using any polynomial time algorithm for multiplying two B -bit numbers as a black box. Especially if we use Harvey and Van Der Hoeven's algorithm, the time cost is $\mathcal{O}(xB \log^2(xB))$.*

Let us also notice that the way to divide two B -bit numbers is simply storing both the numerator and the denominator. Comparing two numbers $x_1 = \frac{num_1}{den_1}$ and $x_2 = \frac{num_2}{den_2}$ can be done by comparing $num_1 \times den_2$ and $num_2 \times den_1$. The only other operation we need when working with such fractions is subtracting a B -bit number $x = \frac{num}{den}$ from 1. This is simply $\frac{den-num}{den}$.

3 NP-Completeness

An NP-Completeness proof for the integer log-probability version of the WLCS problem has been given in [19]. This is a closely related problem, with the main difference being that products are replaced with sums. We do not know of any way to reduce from this log-probability version to WLCS other than exponentiating. As stated in the explanation of our model of computation in Section 2, there is no limit on the number of bits needed to represent a single number (it just occupies a lot of words). This means that, if the input consisted of I bits, and there was a number (probability) represented with $\frac{I}{100}$ bits, exponentiating would result in a number with $2^{\frac{I}{100}}$ bits, meaning the reduction would not

be a polynomial-time one. For this reason, we believe that although it is easier to prove NP-Completeness for the integer log-probability version of the problem, there is no easy way to use it for proving NP-Completeness for the general version. We, thus, give a reduction from the NP-Complete problem Subset Product [22] which proves NP-Completeness directly for the general problem.

Notice that for alphabets consisting of one letter, the problem is trivial since there is no uncertainty at all. In the following, we prove that even for alphabets consisting of two letters, the problem is NP-Complete.

► **Definition 6 (Subset Product).** *Given a set L of n integers and an integer P , find if there exists a subset of the numbers in L with product P .*

► **Lemma 7.** *WLCS is NP-Complete, even for alphabets of size 2.*

Proof. Obviously $WLCS \in NP$ since the increasing subsequences π_1, π_2 and the string s for which $P_X(\pi_1, s) \geq a, P_Y(\pi_2, s) \geq a$ are a certificate which, along with the input, can be used to verify in polynomial time that the problem has a solution.

Let (L, P) be an instance of Subset Product and let $n = |L|$. By L_i we denote the i -th number of the set L , assuming any fixed ordering of the n numbers of L . We give a polynomial-time reduction to a (X, Y, a, k) instance of WLCS, with alphabet size 2 (we call the letters ' A ' and ' B ').

The core idea is the following: The weighted sequences have n positions (plus 2 more for technical reasons related to the threshold a). The number k is equal to the length of the sequences, meaning that we pick every position, and the only question is whether we picked letter ' A ' or letter ' B '. Letter ' A ' in position i corresponds to picking the i -th number in the original Subset Product, while letter ' B ' corresponds to not picking it. Finally, the letters ' A ' picked in X form an inequality of the form: "some product is $\geq P$ ", while the same letters in Y form the inequality: "the same product is $\leq P$ ". For these two to hold simultaneously, it must be the case that we found some product equal to P , which is the goal of the original Subset Product.

More formally, the weighted sequences have size $n + 2$. Let $c_i = \frac{1}{1+L_i}$ and $d_i = \frac{1}{1+\frac{1}{L_i}}$.

$$\begin{aligned} p_i^{(X)}('A') &= c_i L_i, 1 \leq i \leq n & p_i^{(Y)}('A') &= \frac{d_i}{L_i}, 1 \leq i \leq n \\ p_{n+1}^{(X)}('A') &= 1 & p_{n+1}^{(Y)}('A') &= \prod_{j=1}^n \frac{1}{L_j} = \frac{\prod_{j=1}^n c_j}{\prod_{j=1}^n d_j} \\ p_{n+2}^{(X)}('A') &= \frac{1}{P^2} & p_{n+2}^{(Y)}('A') &= 1 \end{aligned}$$

where $p_i^{(X)}('B') = 1 - p_i^{(X)}('A')$ for all i , and similarly for Y . Notice that, in particular, $p_i^{(X)}('B') = c_i, 1 \leq i \leq n$ and $p_i^{(Y)}('B') = d_i, 1 \leq i \leq n$. Finally, we set $k = n + 2$ and $a = \frac{\prod_{j=1}^n c_j}{P}$.

First of all, notice that since we must find a string of length $n + 2$, we must choose a letter from every position. Thus, a choice of letter at some position on X corresponds to the same choice of letter at that position on Y . The choice of letter on positions $n + 1$ and $n + 2$ is ' A ' in both cases since

$$p_{n+1}^{(X)}('B') = p_{n+2}^{(Y)}('B') = 0$$

19:8 Longest Common Subsequence on Weighted Sequences

Suppose that the numbers at positions $\{i_1, \dots, i_\ell\}$ give product P :

$$\prod_{j=1}^{\ell} L_{i_j} = P$$

Then, we form the string s by picking 'A' at positions $\{i_1, \dots, i_\ell, n+1, n+2\}$ and 'B' at all other positions. Thus

$$P_X(\{1, 2, \dots, n+2\}, s) = \frac{\prod_{j=1}^{\ell} L_{i_j} \prod_{j=1}^n c_i}{P^2} = \frac{\prod_{j=1}^n c_i}{P} = a$$

$$P_Y(\{1, 2, \dots, n+2\}, s) = \frac{\prod_{j=1}^n d_i \prod_{j=1}^n c_i}{\prod_{j=1}^{\ell} L_{i_j} \prod_{j=1}^n d_i} = \frac{\prod_{j=1}^n c_i}{P} = a$$

Conversely, suppose a solution for the WLCS problem, where the string s is formed by picking 'A' at positions $\{i_1, \dots, i_\ell, n+1, n+2\}$ and 'B' at all other positions. It holds that:

$$P_X(\{1, 2, \dots, n+2\}, s) = \frac{\prod_{j=1}^{\ell} L_{i_j} \prod_{j=1}^n c_i}{P^2} \geq a \implies \prod_{j=1}^{\ell} L_{i_j} \geq P$$

$$P_Y(\{1, 2, \dots, n+2\}, s) = \frac{\prod_{j=1}^n d_i \prod_{j=1}^n c_i}{\prod_{j=1}^{\ell} L_{i_j} \prod_{j=1}^n d_i} \geq a \implies \prod_{j=1}^{\ell} L_{i_j} \leq P$$

The above imply that $\prod_{j=1}^{\ell} L_{i_j} = P$. Finally, notice that all computations are done in polynomial time, due to Corollary 5. \blacktriangleleft

4 EPTAS for Bounded Alphabets, Improved PTAS for Unbounded Alphabets

We now give an Efficient Polynomial Time Approximation Scheme (*EPTAS*) for the case where our alphabet size is bounded ($|\Sigma| = O(1)$). Let us notice that this is the case when working with DNA sequences ($|\Sigma| = 4$), the most usual application of weighted sequences. The same algorithm is an improved (when compared to [19]) *PTAS* in the case of unbounded alphabets. This means that the WLCS problem is Fixed-Parameter Tractable for constant size alphabets and thus belongs to the corresponding complexity class *FPT* as shown in Corollary 11.

The authors in [19] first noted that there is no *FPTAS* unless $P = NP$, and so we can only hope for an *EPTAS*. Our result relies on their following result:

► **Lemma 8** (Lemma 4.6 of [19]). *It is possible to find, in polynomial time, a solution of size d to the WLCS optimization problem such that the optimal value OPT is guaranteed to be either d or $d+1$ (however we do not know which one holds).*

Their *PTAS* uses the above result and in case the approximation is guaranteed to be good enough ($d > (1 - \epsilon)(d+1)$, which implies that $d > (1 - \epsilon)OPT$), it stops. Else, it holds that $\frac{1}{\epsilon} \geq d+1 \geq OPT$, and the *PTAS* exhaustively searches all subsequences of X , all subsequences of Y , and all possible strings of length $d+1$, for a total complexity of

$$\mathcal{O}\left(\text{Mul}_w\left(\frac{B}{\epsilon}\right) \log\left(\frac{B}{\epsilon}\right) |\Sigma|^{\frac{1}{\epsilon}} \binom{n}{\frac{1}{\epsilon}}^2\right)$$

$Mul_w(\frac{B}{\epsilon}) \log(\frac{B}{\epsilon})$ is the time needed to multiply $d + 1$ numbers with at most B -bits each, by Lemma 4, and is insignificant compared to the other terms. Our *EPTAS* improves the exhaustive search part to

$$\mathcal{O}\left(Mul_w\left(\frac{B}{\epsilon}\right) \frac{n}{\epsilon} |\Sigma|^{\frac{1}{\epsilon}}\right)$$

which is polynomial in the input size, in case of bounded alphabets. The following lemma is needed.

► **Lemma 9.** *Given a weighted sequence X of length n , and a string s of length d , it is possible to find the maximum number a such that there exists an increasing subsequence π of length d for which $P_X(\pi, s) = a$. The running time of the algorithm is $O(Mul_w(dB)nd)$, where B is the maximum number of bits needed to represent each probability in X .*

Proof. We use dynamic programming. Let s_j be the string formed by the first j letters of s , c_j be the j -th letter of s and $opt_X(i, j)$ be the maximum number such that there exists an increasing subsequence π' of length j whose last term π'_j is at most i and for which $P_X(\pi', s_j) = opt_X(i, j)$. Since we choose whether c_j is picked from the i -th position of X , it holds that:

$$opt_X(i, j) = \max\{opt_X(i-1, j), opt_X(i-1, j-1)p_i^{(X)}(c_j)\}$$

For the base cases, $opt_X(i, 0) = 1$ for all i (we can always form the empty string with certainty, by not picking anything), and $opt_X(0, j) = 0$ for $j > 0$ (not picking anything never gives us a non-empty string). We are interested in the value $opt_X(|X|, |s|)$. ◀

Now we are ready to give our *EPTAS*.

► **Theorem 10.** *For any value $\epsilon \in (0, 1]$ there exists an $(1 - \epsilon)$ -approximation algorithm for the WLCS problem which runs in $\mathcal{O}\left(\text{poly}(I) + \frac{n}{\epsilon} Mul_w\left(\frac{B}{\epsilon}\right) |\Sigma|^{\frac{1}{\epsilon}}\right)$ time and uses $\mathcal{O}(\text{poly}(I))$ space, where I is the input size, $n = |X| + |Y|$ and B is the maximum number of bits needed to represent a probability in X and Y . Consequently, the WLCS problem admits an *EPTAS* for bounded alphabets.*

Proof. We begin by using Lemma 8 to find an a -subsequence of length d , such that the optimal solution is at most $d + 1$. If $d + 1 \geq \frac{1}{\epsilon}$, we are done, since in that case we have a $\frac{d}{d+1} = 1 - \frac{1}{d+1} \geq (1 - \epsilon)$ approximation. Otherwise, we try all possible strings $s \in |\Sigma|^{d+1}$, and use Lemma 9 to check if any one of them can appear in both weighted sequences with probability at least a . ◀

► **Corollary 11.** *WLCS \in FPT for bounded alphabets, parameterized by the solution length.*

Proof. Follows directly from [31], Proposition 2. ◀

5 No EPTAS for Unbounded Alphabets

We have already seen that there is no *FPTAS* for WLCS, even for alphabets of size 2, unless $P = NP$. We have also shown an *EPTAS* for bounded alphabets and a *PTAS* for unbounded alphabets. The natural question that arises is: Is it possible to give an *EPTAS* for unbounded alphabets?

We answer this question negatively, by proving that WLCS is $W[1]$ -hard, meaning that it does not admit an *EPTAS* (and is in fact not even in *FPT*) unless $FPT = W[1]$ ([31], Corollary 1). To show this, we give a 2-step *FPT*-reduction from Perfect Code, which

19:10 Longest Common Subsequence on Weighted Sequences

was shown to be $W[1]$ -Complete in [15], to k -sized Subset Product and then to WLCS. The k -sized Subset Product problem is the Subset Product problem with the additional constraint that the target subset must be of size k .

► **Definition 12** (Perfect Code). *A perfect code is a set of vertices $V' \subseteq V$ with the property that for each vertex $u \in V$ there is precisely one vertex in $N_G(u) \cap V'$, where $N_G(u)$ is the set of adjacent nodes of u in G .*

In the perfect code problem, we are given an undirected graph G and a positive integer k , and we need to decide whether G has a k -element perfect code. Notice that the definition of a perfect code implies that there is a perfect code iff there is a set $V' \subseteq V$ for which $\bigcup_{u \in V'} N_G(u) = V$ and $N_G(u) \cap N_G(v) = \emptyset$ for all $u, v \in V', u \neq v$. First we show that k -sized Subset Product is $W[1]$ -hard.

► **Lemma 13.** *k -sized Subset Product is $W[1]$ -hard.*

Proof. Let $(G = (V, E), k)$ be an instance of Perfect Code. Suppose that the vertices are $V = \{1, \dots, n\}$. First of all, we compute the first n prime numbers using the Sieve of Eratosthenes. We denote the i -th prime number as p_i . The set of positive integers $L = \{L_1, L_2, \dots, L_n\}$ as well as the positive integer P are defined as follows:

$$L_v = \prod_{u \in N_G(v)} p_u, \quad P = \prod_{v=1}^n p_v$$

Notice that due to the unique prime factorization theorem, a subset of k numbers from the set L have product P iff G has a k -element Perfect Code.

The size of our primes is $O(n \log n)$ due to the prime number theorem. Thus, they require $O(\log n)$ bits to be represented. Each integer in L , as well as in P , is computed using Corollary 5 in $O(n \log^3 n)$ time, for an overall $O(n^2 \log^3 n)$ complexity for our reduction. Since the new parameter k is the same as the old one (no dependence on n), our reduction is in fact an FPT -reduction. ◀

Our result for this section is the following.

► **Theorem 14.** *WLCS, parameterized by the length of the solution, is $W[1]$ -hard.*

Proof. To prove the theorem we create diagonal weighted sequences. That is, we require each letter to appear only in one position and vice-versa. In this way, the subsequences picked for X and Y are the same. The above rule is broken by the addition of two auxiliary letters that are there to make the probabilities add up to 1 in each position. This creates no problem because we make sure that these letters are never picked. Finally, we force the product to be equal to our target, by forcing it to be at most our target and at least our target at the same time.

More formally, let $(L = \{L_1, L_2, \dots, L_n\}, k, P)$ be an instance of the k -sized Subset Product problem and let $M = m^{k+1}$, where m is the maximum number in set L . Notice that if $m^k \leq P$ then we only need to check the product of the highest k numbers of L , which means the problem is solvable in polynomial time. Thus we can assume that $M \geq m^k > P$. The alphabet of X, Y is $\Sigma = \{1, 2, \dots, n, n+1, n+2, n+3\}$ and we set $a = \frac{1}{PM^k}$.

$$\begin{aligned} p_i^{(X)}(i) &= \frac{L_i}{M}, \quad 1 \leq i \leq n & p_i^{(Y)}(i) &= \frac{1}{ML_i}, \quad 1 \leq i \leq n \\ p_{n+1}^{(X)}(n+1) &= \frac{1}{P^2} & p_{n+1}^{(Y)}(n+1) &= 1 \\ p_i^{(X)}(n+2) &= 1 - p_i^{(X)}(i), \quad 1 \leq i \leq n+1 & p_i^{(Y)}(n+3) &= 1 - p_i^{(Y)}(i), \quad 1 \leq i \leq n+1 \end{aligned}$$

All non-specified probabilities are equal to 0. Notice that symbols $n + 2$ and $n + 3$ are used to guarantee that probabilities sum up to 1.

We show that the instance $(X, Y, a, k + 1)$ has a solution iff (L, k, P) has a solution. Suppose there exists a solution to (L, k, P) . Then, there exists an increasing subsequence $\pi = (i_1, \dots, i_k)$ such that $\prod_{j=1}^k L_{i_j} = P$. Let π' be π extended by the number $i_{k+1} = n + 1$ and s be the string $i_1 i_2 \dots i_{k+1}$. It holds that $P_X(\pi', s) = P_Y(\pi', s) = a$.

Conversely, suppose there exists a solution to $(X, Y, a, k + 1)$. Then there exist increasing subsequences $\pi = (i_1, \dots, i_{k+1}), \pi' = (j_1, \dots, j_{k+1})$ and a string s such that $P_X(\pi, s) \geq a, P_Y(\pi', s) \geq a$. First of all, notice that, due to $p_i^{(X)}(n + 3) = p_i^{(Y)}(n + 2) = 0$ for all i , s does not contain letters $n + 2$ and $n + 3$, which leaves only one choice for every position. Also each letter appears only once in each sequence, and in the same position. Thus, $\pi = \pi'$, and due to our construction the i -th letter of s is the i -th member of π . Finally, not picking position $n + 1$ would result in $P_Y(\pi, s) < a$ due to the fact that $P < M$. Thus, the last letter of s is $n + 1$. It holds that:

$$P_X(\{i_1, \dots, i_{k+1}\}, s) \geq a \implies \frac{\prod_{i=1}^k L_{\pi_i}}{P^2 M^k} \geq \frac{1}{P M^k} \implies \prod_{i=1}^k L_{\pi_i} \geq P$$

$$P_Y(\{i_1, \dots, i_{k+1}\}, s) \geq a \implies \frac{1}{M^k \prod_{i=1}^k L_{\pi_i}} \geq \frac{1}{P M^k} \implies \prod_{i=1}^k L_{\pi_i} \leq P$$

The above two inequalities imply a k -sized subset of L with product equal to P .

The reduction is a polynomial-time one, due to Corollary 5. More than that, it is an *FPT*-reduction since the new parameter k is equal to the old parameter incremented by one, and thus has no dependence on n . ◀

6 Matching Conditional Lower Bound on any PTAS

In the d -SUM problem, we are given N numbers and need to decide whether there exists a d -tuple that sums to zero. Patrascu and Williams [33] proved that any algorithm for solving the d -SUM problem requires $n^{\Omega(d)}$ time, unless the Exponential Time Hypothesis (*ETH*) fails. To show this, they first proved a hardness result for a variant of 3-SAT, the sparse 1-in-3 SAT.

► **Definition 15** (Sparse 1-in-3 SAT). *Given a boolean formula with n variables and $O(n)$ clauses in 3 CNF form, where each variable appears in a constant number of clauses, determine whether there exists an assignment of the variables such that each clause is satisfied by exactly one variable.*

They first prove the following hardness result under *ETH*.

► **Proposition 16.** *Under ETH, there is an (unknown) constant s_3 such that there exists no algorithm to solve sparse 1-in-3 SAT in $\mathcal{O}(2^{\delta n})$ time for $\delta < s_3$.*

By assuming an $n^{\mathcal{O}(d)}$ time algorithm for d -SUM they disproved the above fact, which cannot happen under *ETH*. We use the same technique for proving an $n^{\Omega(k)}$ lower bound for k -sized Subset Product.

► **Lemma 17.** *Assuming the ETH, the problem of k -sized Subset Product cannot be solved in $\mathcal{O}(n^{\frac{s_3 k}{101}})$ time on instances satisfying $k < n^{0.99}$ and each number in the input set L has $\mathcal{O}(\log n(\log k + \log \log n))$ bits, where n is the size of L , and P is the target which can be arbitrarily big.*

19:12 Longest Common Subsequence on Weighted Sequences

Proof. Let f be a sparse 1-in-3 SAT instance with N variables and $M = \mathcal{O}(N)$ clauses, and $k > \frac{1}{s_3}$. Conceptually, we split the variables of f into k blocks of equal size - apart from the last block that may have smaller size. Each block contains at most $\lceil \frac{N}{k} \rceil$ variables, and thus there are at most $2^{\lceil \frac{N}{k} \rceil}$ different assignments of values to the group-of-variables within a block. For each block and for each one of these assignments we generate a number which serves as an identifier of the corresponding block and assignment. Thus, there are $n = k2^{\lceil \frac{N}{k} \rceil}$ different identifiers.

Let p_i be the i -th prime number. In order to compute an identifier, we initialize it to p_b , where b is the index of the identifier's corresponding block. Then, we run through all of the $M = \mathcal{O}(N)$ clauses and do the following: suppose we process the i -th clause and let $0 \leq j \leq 3$ be the number of variables of the identifier's corresponding assignment that satisfy the clause. We update the identifier by multiplying it with p_{k+i}^j .

Since each variable appears only in a constant number of clauses, each identifier is a product of $\mathcal{O}(\frac{N}{k})$ numbers. The prime number theorem guarantees $\mathcal{O}(\log N)$ bits to represent each factor, which means the identifiers have $\mathcal{O}(\frac{N}{k} \log N)$ bits. Using the fact that $n = k2^{\lceil \frac{N}{k} \rceil}$, each identifier is represented by $\mathcal{O}(\log n(\log k + \log \log n))$ bits.

These n identifiers, along with the target $P = \prod_{i=1}^{k+M} p_i$ (recall that p_i is the i -th prime number), form a k -sized Subset Product instance. This preprocessing step costs $\mathcal{O}(2^{\frac{N}{k}})$ time, ignoring polynomial terms, which is more efficient than $\mathcal{O}(2^{s_3 N})$.

Due to the unique prime factorization, a solution to the k -sized Subset Product corresponds to a solution in f and vice-versa. If the running time of the k -sized Subset Product was $\mathcal{O}(n^{\frac{s_3 k}{101}})$ then we could solve the above instance in $\mathcal{O}((k2^{\frac{N}{k}})^{\frac{s_3 k}{101}})$ time.

Since $k = \frac{n}{2^{\lceil \frac{N}{k} \rceil}}$ and $k < n^{0.99}$, it follows that $\frac{n}{2^{\lceil \frac{N}{k} \rceil}} < n^{0.99} \implies n^{0.99} < 2^{99 \lceil \frac{N}{k} \rceil}$. But $k < n^{0.99}$, which means $k < 2^{99 \lceil \frac{N}{k} \rceil}$.

Thus the previous running time becomes $\mathcal{O}(2^{\frac{100}{101} s_3 N})$. Both the preprocessing step and the solution of the k -sized Subset Product can be achieved in time $\mathcal{O}(2^{\delta N})$, where $\delta < s_3$. However, this would violate Proposition 16. \blacktriangleleft

Using the above, we are ready to prove our (matching) lower bound, conditional on *ETH*.

► **Theorem 18.** *Under ETH, there is no PTAS for WLCS with running time $|I|^{\mathcal{O}(\frac{1}{\epsilon})}$, where $|I|$ is the input size in bits.*

Proof. Suppose that such an algorithm $A(I, \epsilon)$ existed. Let $R()$ be the polynomial time reduction from k -sized Subset Product to WLCS given in the proof of Theorem 14. Then, there is a solution to k -sized Subset Product iff there is a solution to WLCS of size $k + 1$, or, equivalently, iff the optimal solution to WLCS is at least $k + 1$.

Using the hypothetical $A(I, \epsilon)$ with an appropriate value of ϵ , we solve k -sized Subset Product more efficiently than possible, thus reaching a contradiction.

Consider the following algorithm for k -sized Subset Product, where there are $|L|$ numbers in the input, each having $\mathcal{O}(\log |L|(\log k + \log \log |L|))$ bits and $k < |L|^{0.99}$. Given an instance (L, k, P) , we define the instance for the WLCS to be $I = R(L, k, P)$. We run $A(I, \frac{1}{2(k+1)})$ and if the output is at least $k + 1$ we return that (L, k, P) is satisfied, otherwise we return that it cannot be satisfied.

Note that if k -sized Subset Product is solvable, then $OPT(I) \geq k + 1$, and the value output by A is at least $(1 - \frac{1}{2(k+1)})(k+1) = k + \frac{1}{2} > k$. Thus, the value output by A is at least $k + 1$. On the other hand, if k -sized Subset Product is not solvable, then $OPT(I) < k + 1$, and obviously the value output by A is at most k .

Thus we found an algorithm for k -sized Subset Product whose running time is $|I|^{\mathcal{O}(k)}$. Since I is obtained by a polynomial time reduction, its size is bounded by a polynomial in $|(L, k, P)|$. Therefore, the above running time becomes $|(L, k, P)|^{\mathcal{O}(k)}$. Under our assumptions, this becomes $|L|^{\mathcal{O}(k)}$, which is not feasible under *ETH*, due to Lemma 17. ◀

7 Conclusion

In this paper we prove NP-Completeness for the WLCS decision problem, and give a *PTAS* along with a matching conditional lower bound for the optimization problem. In the most usual setting, where the alphabet size is constant, the above *PTAS* is in fact an *EPTAS*, and it is known that no *FPTAS* can exist unless $P = NP$. In the full version of the paper [29], we give a transformation such that algorithms for the WLCS problem can be applied for the (a_1, a_2) -WLCS problem.

In proving that WLCS does not admit any *EPTAS*, we proved that it is $W[1]$ – *hard*. It may be interesting to determine the exact complexity of WLCS in the W – *hierarchy*.


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, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.
- 2 Amihod Amir, Eran Chencinski, Costas S. Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property matching and weighted matching. *Theoretical Computer Science*, 395(2-3):298–310, 2008. doi:10.1016/j.tcs.2008.01.006.
- 3 Amihod Amir, Zvi Gotthilf, and B. Riva Shalom. Weighted LCS. *Journal of Discrete Algorithms*, 8(3):273–281, 2010. doi:10.1016/j.jda.2010.02.001.
- 4 Amihod Amir, Zvi Gotthilf, and B. Riva Shalom. Weighted shortest common supersequence. In *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, pages 44–54, 2011. doi:10.1007/978-3-642-24583-1_6.
- 5 Amihod Amir, Tzvika Hartman, Oren Kapah, B. Riva Shalom, and Dekel Tsur. Generalized LCS. *Theoretical Computer Science*, 409(3):438–449, 2008. doi:10.1016/j.tcs.2008.08.037.
- 6 Amihod Amir, Costas S. Iliopoulos, Oren Kapah, and Ely Porat. Approximate matching in weighted sequences. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, pages 365–376, 2006. doi:10.1007/11780441_33.
- 7 Pavlos Antoniou, Costas S. Iliopoulos, Laurent Mouchard, and Solon P. Pissis. Algorithms for mapping short degenerate and weighted sequences to a reference genome. *International Journal of Computational Biology and Drug Design*, 2(4):385–397, 2009. doi:10.1504/IJCBD.2009.030768.
- 8 Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999. doi:10.1006/jcom.1998.0493.
- 9 Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Optimal computation of all tandem repeats in a weighted sequence. *Algorithms for Molecular Biology*, 9:21, 2014. doi:10.1186/s13015-014-0021-5.
- 10 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Information and Computation*, 270, 2020. doi:10.1016/j.ic.2019.104462.
- 11 Carl Barton and Solon P. Pissis. Crochemore’s partitioning on weighted strings and applications. *Algorithmica*, 80(2):496–514, 2018. doi:10.1007/s00453-016-0266-0.

- 12 Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, September 1980. doi:10.1145/1008861.1008865.
- 13 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, A Coruña, Spain, September 27-29, 2000*, pages 39–48, 2000. doi:10.1109/SPIRE.2000.878178.
- 14 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, New Orleans, LA, USA, January 7-10, 2018*, pages 1216–1235, 2018. doi:10.1137/1.9781611975031.79.
- 15 Marco Cesati. Perfect code is W[1]-complete. *Information Processing Letters*, 81(3):163–168, 2002. doi:10.1016/S0020-0190(01)00207-1.
- 16 Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. On-line weighted pattern matching. *Information and Computation*, 266:49–59, 2019. doi:10.1016/j.ic.2019.01.001.
- 17 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Weighted shortest common supersequence problem revisited. 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 221–238. Springer, 2019. doi:10.1007/978-3-030-32686-9_16.
- 18 Manolis Christodoulakis, Costas S. Iliopoulos, Laurent Mouchard, Katerina Perdikuri, Athanasios K. Tsakalidis, and Kostas Tsichlas. Computation of repetitions and regularities of biologically weighted sequences. *Journal of Computational Biology*, 13(6):1214–1231, 2006. doi:10.1089/cmb.2006.13.1214.
- 19 Marek Cygan, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Polynomial-time approximation algorithms for weighted LCS problem. *Discrete Applied Mathematics*, 204:38–48, 2016. doi:10.1016/j.dam.2015.11.011.
- 20 Michael L. Fredman and Dan E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 1–7, 1990. doi:10.1145/100216.100217.
- 21 Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009. doi:10.1137/070711761.
- 22 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- 23 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 24 David Harvey and Joris Van Der Hoeven. Integer multiplication in time $O(n \log n)$. working paper or preprint, March 2019. URL: <https://hal.archives-ouvertes.fr/hal-02070778>.
- 25 Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.
- 26 Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. Efficient algorithms for handling molecular weighted sequences. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, pages 265–278, 2004. doi:10.1007/1-4020-8141-3_22.
- 27 Costas S. Iliopoulos, Mirka Miller, and Solon P. Pissis. Parallel algorithms for mapping short degenerate and weighted DNA sequences to a reference genome. *International Journal of Foundations of Computer Science*, 23(2):249–259, 2012. doi:10.1142/S0129054112400114.

- 28 Costas S. Iliopoulos, Katerina Perdikuri, Evangelos Theodoridis, Athanasios K. Tsakalidis, and Kostas Tsichlas. Motif extraction from weighted sequences. In *String Processing and Information Retrieval, 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004, Proceedings*, pages 286–297, 2004. doi:10.1007/978-3-540-30213-1_41.
- 29 Evangelos Kipouridis and Kostas Tsichlas. Longest common subsequence on weighted sequences. *CoRR*, abs/1901.04068, 2019. arXiv:1901.04068.
- 30 Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. *Theory of Computing Systems*, 63(3):506–542, 2019. doi:10.1007/s00224-018-9881-2.
- 31 Dániel Marx. Parameterized complexity and approximation algorithms. *The Computer Journal*, 51(1):60–78, 2008. doi:10.1093/comjnl/bxm048.
- 32 Gene Myers. The whole genome assembly of drosophila. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, page 753, 2000. URL: <http://dl.acm.org/citation.cfm?id=338219.338635>.
- 33 Mihai Patrascu and Ryan Williams. On the possibility of faster SAT algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1065–1075, 2010. doi:10.1137/1.9781611973075.86.
- 34 Jakub Radoszewski and Tatiana Starikovskaya. Streaming k -mismatch with error correcting and applications. *Information and Computation*, 271:104513, 2020. doi:10.1016/j.ic.2019.104513.
- 35 Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971. doi:10.1007/BF02242355.
- 36 Julie Thompson, Desmond G. Higgins, and Toby J. Gibson. W: Clustal. improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic acids research*, 22:4673–80, December 1994. doi:10.1093/nar/22.22.4673.
- 37 J. Craig Venter. Sequencing the human genome. In *RECOMB*, pages 309–309. ACM, 2002.
- 38 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
- 39 Hui Zhang, Qing Guo, Jing Fan, and Costas S. Iliopoulos. Loose and strict repeats in weighted sequences of proteins. *Protein and Peptide Letters*, 17(9):1136–1142, 2010.
- 40 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. String matching with swaps in a weighted sequence. In *CIS*, volume 3314 of *Lecture Notes in Computer Science*, pages 698–704. Springer, 2004.
- 41 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. An algorithmic framework for motif discovery problems in weighted sequences. In *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings*, pages 335–346, 2010. doi:10.1007/978-3-642-13073-1_30.
- 42 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Varieties of regularities in weighted sequences. In *AAIM*, volume 6124 of *Lecture Notes in Computer Science*, pages 271–280. Springer, 2010.
- 43 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Locating tandem repeats in weighted sequences in proteins. *BMC Bioinformatics*, 14(S-8):S2, 2013.

Parameterized Algorithms for Matrix Completion with Radius Constraints

Tomohiro Koana 

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
tomohiro.koana@tu-berlin.de

Vincent Froese

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
vincent.froese@tu-berlin.de

Rolf Niedermeier 

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
rolf.niedermeier@tu-berlin.de

Abstract

Considering matrices with missing entries, we study NP-hard matrix completion problems where the resulting completed matrix should have limited (local) radius. In the pure radius version, this means that the goal is to fill in the entries such that there exists a “center string” which has Hamming distance to all matrix rows as small as possible. In stringology, this problem is also known as CLOSEST STRING WITH WILDCARDS. In the local radius version, the requested center string must be one of the rows of the completed matrix.

Hermelin and Rozenberg [CPM 2014, TCS 2016] performed a parameterized complexity analysis for CLOSEST STRING WITH WILDCARDS. We answer one of their open questions, fix a bug concerning a fixed-parameter tractability result in their work, and improve some running time upper bounds. For the local radius case, we reveal a computational complexity dichotomy. In general, our results indicate that, although being NP-hard as well, this variant often allows for faster (fixed-parameter) algorithms.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Pattern matching

Keywords and phrases fixed-parameter tractability, consensus string problems, Closest String, Closest String with Wildcards

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.20

Funding *Tomohiro Koana*: Partially supported by the DFG project MATE (NI 369/17).

Acknowledgements We are grateful to Christian Komusiewicz for helpful feedback on an earlier version of this work and to Stefan Szeider for pointing us to reference [7].

1 Introduction

In many applications data can only be partially measured, which leads to incomplete data records with missing entries. A common problem in data mining, machine learning, and computational biology is to infer these missing entries. In this context, matrix completion problems play a central role. Here the goal is to fill in the unknown entries of an incomplete data matrix such that certain measures regarding the completed matrix are optimized. Ganian et al. [9] recently studied parameterized algorithms for two variants of matrix completion problems; their goal was to minimize either the rank or the number of distinct rows in the completed matrix. In this work, we focus our study on another variant, namely, minimizing the “radius” of the completed matrix. Indeed, this is closely related to the topic of consensus (string) problems, which received a lot of attention in stringology and particularly with respect to parameterized complexity studies [16, 15, 3]. Indeed, radius minimization for



© Tomohiro Koana, Vincent Froese, and Rolf Niedermeier;
licensed under Creative Commons License CC-BY

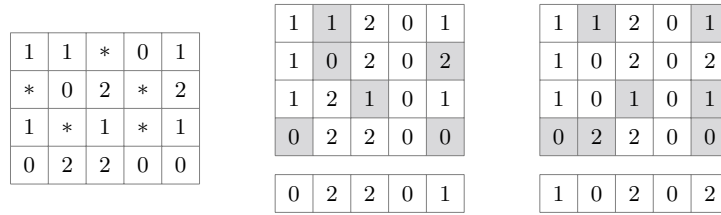
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 20; pp. 20:1–20:14

Leibniz International Proceedings in Informatics



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



■ **Figure 1** An example of MINRMC and MINLRMC. The incomplete input matrix is depicted in the left (a * denotes a missing entry). Optimal solutions for MINRMC ($d = 2$) and MINLRMC ($d = 3$) are shown in the middle and right. Entries differing from the solution vector are marked gray.

incomplete matrices is known in the stringology community under the name CLOSEST STRING WITH WILDCARDS [12], a generalization of the frequently studied CLOSEST STRING problem. Herein, an incomplete matrix shall be completed such that there exists a vector that is not too far from each row vector of the completed matrix in terms of the Hamming distance (that is, the completed matrix has small *radius*). We consider this radius minimization problem and also a local variant where all row vectors of the completed matrix must be close to a row vector in the matrix.

Given the close relation to CLOSEST STRING, which is NP-hard already for binary strings [8], all problems we study in this work are NP-hard in general. However, we provide several positive algorithmic results, namely fixed-parameter tractability with respect to natural parameters or even polynomial-time solvability for special cases. Formally, we study the following problems (see Figure 1 for illustrative examples):

MINIMUM RADIUS MATRIX COMPLETION (MINRMC)

Input: An incomplete matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d \in \mathbb{N}$.

Question: Is there a “completion” $\mathbf{T} \in \Sigma^{n \times \ell}$ of \mathbf{S} such that $\delta(v, \mathbf{T}) \leq d$ for some vector $v \in \Sigma^\ell$?

MINIMUM LOCAL RADIUS MATRIX COMPLETION (MINLRMC)

Input: An incomplete matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d \in \mathbb{N}$.

Question: Is there a “completion” $\mathbf{T} \in \Sigma^{n \times \ell}$ of \mathbf{S} such that $\delta(v, \mathbf{T}) \leq d$ for some vector $v \in \mathbf{T}$?

Here, missing entries are denoted by *. A completion of an incomplete matrix $\mathbf{S} \in \Sigma \cup \{*\}$ is a matrix obtained by replacing each missing entry with some symbol from Σ . We denote the Hamming distance by δ . The Hamming distance between a vector v and a matrix \mathbf{T} is the maximum Hamming distance between v and a vector in \mathbf{T} . In fact, our results for MINRMC also hold for the following more general problem:

CONSTRAINT RADIUS MATRIX COMPLETION (CONRMC)

Input: An incomplete matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Question: Is there a “completion” $\mathbf{T} \in \Sigma^{n \times \ell}$ of \mathbf{S} such that for some vector $v \in \Sigma^\ell$, $\delta(v, \mathbf{T}[i]) \leq d_i$ for each $i \in [n]$?

Related work

Our most important reference point is the work of Hermelin and Rozenberg [12] who analyzed the parameterized complexity of MINRMC (under the name CLOSEST STRING WITH WILDCARDS) with respect to several problem-specific parameters (also see Table 1). In

■ **Table 1** Overview of known results and our new results for MINRMC and MINLRMC. Notation: n – number of rows, ℓ – number of columns, $|\Sigma|$ – alphabet size, d – distance bound, k – maximum number of missing entries in any row vector. Note that all our results for MINRMC also hold for CONRMC.

Parameter	MINRMC	Reference	MINLRMC	Reference
n	$O^*(2^{2^{O(n \log n)}})$	[12]	$O^*(2^{O(n^2 \log n)})$	Corollary 3
	$O^*(2^{O(n^2 \log n)})$	[13]		
ℓ	$O^*(2^{\ell^2/2})$	[12]	$O^*(\ell^\ell)$	Corollary 7
	$O^*(\ell^\ell)$	Corollary 6		
$d = 1$	$O(n\ell^2)$ for $ \Sigma = 2$	[12]	$O(n^2\ell)$	Corollary 2
	$O(n\ell)$	Theorem 1		
$d = 2$	NP-hard for $ \Sigma = 2$	[12]	NP-hard for $ \Sigma = 2$	[12]
k	NP-hard for $k = 0$	[8]	$O^*(k^k)$	Corollary 7
$d + k$	$O^*((d + 1)^{d+k})$	Theorem 9		
$d + k + \Sigma $	$O^*(\Sigma ^k \cdot d^d)$	[12]	$O^*(\Sigma ^k)$	trivial
	$O^*(2^{4d+k} \cdot \Sigma ^{d+k})$	Theorem 12		

particular, they provided fixed-parameter tractability results for the parameters number n of rows, number ℓ of columns, and radius d combined with maximum number k of missing entries per row. However, we will show that their fixed-parameter tractability result for the combined parameter $d + k$ is flawed. Moreover, they showed a computational complexity dichotomy for binary inputs between radius 1 (polynomial time) and radius 2 (NP-hard) (see Table 1 for a complete overview).

As mentioned before, Ganian et al. [9] started research on the parameterized complexity of two related matrix completion problems (minimizing rank and minimizing number of distinct rows). Very recently, Eiben et al. [7] studied generalizations of our problems which demand that the completed matrix be clustered into several submatrices of small (local) radius – basically, our work studies the case of a single cluster. They proved fixed-parameter tractability (problem kernels of superexponential size) with respect to the combined parameter (c, d, r) . Here, c is the number of clusters and r is the minimum number of rows and columns covering all missing entries. On the negative side, they proved that dropping any of c , d , or r results in parameterized intractability even for binary alphabet. Amir et al. [1] showed that the clustering version of MINLRMC on complete matrices with unbounded alphabet size is NP-hard when restricted to only two columns. Note that fixed-parameter tractability for the clustering variant implies fixed-parameter tractability for MINRMC and MINLRMC with respect to $d + r$. Indeed, to reach for (more) practical algorithms, we consider an alternative parameterization by the maximum number k of missing entries in any row vector (which can be much smaller than r). Finally, let us also mention our companion work [14], in which we studied a matrix completion problem with diametrical constraints – all pairwise row distances must be within a specified range in the completion.

Our contributions

We survey our and previous results in Table 1. Notably, all of our results for MINRMC indeed also hold for the more general CONRMC when setting $d := \max\{d_1, d_2, \dots, d_n\}$.

Let us highlight a few of our new results in comparison with previous work. For MINRMC, we give a linear-time algorithm for the case radius $d = 1$ and arbitrary alphabet. This answers an open question of Hermelin and Rozenberg [12]. We also show fixed-parameter tractability

with respect to the combined parameter $d + k$, which was already claimed by Hermelin and Rozenberg [12] but was flawed, as we will point out by providing a counter-example to their algorithm. Lastly, inspired by known results for CLOSEST STRING, we give a more efficient algorithm for small alphabet size.

Regarding MINLRMC, we show that it can be solved in polynomial time when $d = 1$. This yields a computational complexity dichotomy since MINLRMC is NP-hard for $d = 2$. Moreover, we show that MINLRMC is fixed-parameter tractable with respect to the maximum number k of missing entries per row. Remarkably, this stands in sharp contrast to MINRMC, which is NP-hard even for $k = 0$.

2 Preliminaries

For $m \leq n \in \mathbb{N}$, let $[m, n] := \{m, \dots, n\}$ and $[n] := [1, n]$.

Let $\mathbf{T} \in \Sigma^{n \times \ell}$ be an $(n \times \ell)$ -matrix over a finite alphabet Σ . Let $i \in [n]$ and $j \in [\ell]$. We use $\mathbf{T}[i, j]$ to denote the character in the i -th row and j -th column of \mathbf{T} . We use $\mathbf{T}[i, :]$ (or $\mathbf{T}[i]$ in short) to denote the *row vector* $(\mathbf{T}[i, 1], \dots, \mathbf{T}[i, \ell])$ and $\mathbf{T}[:, j]$ to denote the *column vector* $(\mathbf{T}[1, j], \dots, \mathbf{T}[n, j])^T$. For any subsets $I \subseteq [n]$ and $J \subseteq [\ell]$, we write $\mathbf{T}[I, J]$ to denote the submatrix obtained by omitting rows in $[n] \setminus I$ and columns in $[\ell] \setminus J$ from \mathbf{T} . We abbreviate $\mathbf{T}[I, [\ell]]$ and $\mathbf{T}[[n], J]$ as $\mathbf{T}[I, :]$ (or $\mathbf{T}[I]$ for short) and $\mathbf{T}[:, J]$, respectively. We use the special character $*$ for a *missing* entry. A matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ that contains a missing entry is called *incomplete*. We say that $\mathbf{T} \in \Sigma^{n \times \ell}$ is a *completion* of $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ if either $\mathbf{S}[i, j] = *$ or $\mathbf{S}[i, j] = \mathbf{T}[i, j]$ holds for all $i \in [n]$ and $j \in [\ell]$.

Let $v, v' \in (\Sigma \cup \{*\})^\ell$ be row vectors and let $\sigma \in \Sigma \cup \{*\}$. We write $P_\sigma(v)$ to denote the set $\{j \in [\ell] \mid v[j] = \sigma\}$ of column indices where the corresponding entries of v are σ . We write $Q(v, v')$ to denote the set $\{j \in [\ell] \mid v[j] \neq v'[j] \wedge v[j] \neq * \wedge v'[j] \neq *\}$ of column indices where v and v' disagree (not considering positions with missing entries). The *Hamming distance* between v and v' is $\delta(v, v') := |Q(v, v')|$. For $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $v \in (\Sigma \cup \{*\})^\ell$, let $\delta(v, \mathbf{S}) := \max_{i \in [n]} \delta(v, \mathbf{S}[i])$. The binary operation $v \oplus v'$ replaces the missing entries of v with the character in v' in the corresponding position, given that v' contains no missing entry. We sometimes use string notation $\sigma_1 \sigma_2 \sigma_3$ to represent the row vector $(\sigma_1, \sigma_2, \sigma_3)$.

Parameterized Complexity

We sometimes use the O^* -notation which suppresses polynomial factors in the input size. A *parameterized problem* Π is a set of instances $(I, k) \in \Sigma^* \times \mathbb{N}$, where k is called the *parameter* of the instance. A parameterized problem is *fixed-parameter tractable* if $(I, k) \in \Pi$ can be determined in $f(k) \cdot |I|^{O(1)}$ time for an arbitrary computable function f . An algorithm with such a running time is called a *fixed-parameter algorithm*.

3 Linear-time algorithm for radius $d = 1$

Hermelin and Rozenberg [12, Theorem 6] gave a reduction from MINRMC to 2-SAT for the case $|\Sigma| = 2$ and $d = 1$, resulting in an $O(n\ell^2)$ -time algorithm. We provide a more efficient reduction to 2-SAT, exploiting the compact encoding $C_{\leq 1}$ of the “at-most-one” constraint by Sinz [20]. Let $L = \{l_1, \dots, l_m\}$ be a set of m literals. The encoding uses $m - 1$ additional

variables r_1, \dots, r_{m-1} and it is defined as follows:

$$C_{\leq 1}(L) = (\neg l_1 \vee r_1) \wedge (\neg l_m \vee \neg r_{m-1}) \\ \wedge \bigwedge_{2 \leq j \leq m-1} ((\neg l_j \vee \neg r_{j-1}) \wedge (\neg l_j \vee r_j) \wedge (\neg r_{j-1} \vee r_j)).$$

Note that if l_j is true for some $j \in [m]$, then r_j, \dots, r_{m-1} are all true and r_1, \dots, r_{j-1} are all false. Hence, at most one literal in L can be true.

Actually, our algorithm solves CONRMC, the generalization of MINRMC where the distance bound can be specified for each row vector individually.

► **Theorem 1.** *If $\max_{i \in [n]} d_i = 1$, then CONRMC can be solved in $O(n\ell)$ time.*

Proof. We reduce CONRMC to 2-SAT. Let $I_d := \{i \in [n] \mid d_i = d\}$ be the row indices for which the distance bound is d for $d \in \{0, 1\}$. We define a variable $x_{j,\sigma}$ for each $j \in [\ell]$ and $\sigma \in \Sigma$. The intuition behind our reduction is that the j -th entry of the solution vector v becomes σ when $x_{j,\sigma}$ is true. We give the construction of a 2-CNF formula ϕ in three parts ϕ_1, ϕ_2, ϕ_3 (that is, $\phi = \phi_1 \wedge \phi_2 \wedge \phi_3$).

■ Let $X_j = \{x_{j,\sigma} \mid \sigma \in \Sigma\}$ for each $j \in [\ell]$. The first subformula will ensure that at most one character is assigned to each entry of the solution vector v :

$$\phi_1 = \bigwedge_{j \in [\ell]} C_{\leq 1}(X_j).$$

■ Subformula ϕ_2 handles distance-0 constraints:

$$\phi_2 = \bigwedge_{i \in I_0} \bigwedge_{\substack{j \in [\ell] \\ \mathbf{S}[i,j] \neq *}} (x_{j,\mathbf{S}[i,j]}).$$

■ Finally, subformula ϕ_3 guarantees that the solution vector v deviates from each row vector of $\mathbf{S}[I_1]$ in at most one position.

$$\phi_3 = \bigwedge_{i \in I_1} C_{\leq 1}(\{\neg x_{j,\mathbf{S}[i,j]} \mid j \in [\ell], \mathbf{S}[i,j] \neq *\}).$$

Note that our construction uses $O(|\Sigma| \cdot \ell)$ variables and $O((n + |\Sigma|) \cdot \ell) = O(n\ell)$ clauses. We prove the correctness of the reduction.

(\Rightarrow) Suppose that there exists a vector $v \in \Sigma^\ell$ such that $\delta(v, \mathbf{S}[i]) \leq d_i$ holds for each $i \in [n]$. For each $j \in [\ell]$ and $\sigma \in \Sigma$, we set $x_{j,\sigma}$ to true if $v[j] = \sigma$, and false otherwise. It is easy to see that this truth assignment satisfies ϕ .

(\Leftarrow) Suppose that there exists a satisfying truth assignment φ . Let J^* denote the column indices $j \in [\ell]$ such that $\varphi(x_{j,\sigma}) = 0$ for all $\sigma \in \Sigma$. Note that at most one variable in X_j is set to true in φ for each $j \in [\ell]$. It follows that, for each $j \in [\ell] \setminus J^*$, there exists exactly one character $\sigma_j \in \Sigma$ satisfying $\varphi(x_{j,\sigma_j}) = 1$ and we assign $v[j] = \sigma_j$. For each $j \in J^*$, we set $v[j] = \sigma^*$ for some arbitrary character $\sigma^* \in \Sigma$. The formula ϕ_2 ensures that $\delta(v, \mathbf{S}[i]) = 0$ holds for each $i \in I_0$. Moreover, ϕ_3 ensures that there is at most one column index $j \in [\ell]$ such that $\mathbf{S}[i,j] \neq *$ and $\mathbf{S}[i,j] \neq v[j]$ for each $i \in I_1$. ◀

Note that MINRMC (and thus CONRMC) is NP-hard for $|\Sigma| = 2$ and $d = 2$ [12]. Thus, our result implies a complete complexity dichotomy regarding d . We remark that this dichotomy also holds for MINLRMC since there is a simple reduction from MINLRMC to CONRMC. To solve an instance (\mathbf{S}, d) of MINLRMC, we solve n instances of ConRMC: For

■ **Algorithm 1** Improved algorithm for CONRMC (based on Hermelin and Rozenberg [12]).

Input: An incomplete matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Task: Decide whether there exists a row vector $v \in \Sigma^\ell$ with $d(v, \mathbf{S}[i]) \leq d_i$ for all $i \in [n]$.

1: if $d_i < 0$ for some $i \in [n]$ then return **No**.

2: if $\ell - |P_*(\mathbf{S}[i])| \leq d_i$ for all $i \in [n]$ then return **Yes**.

▷ $|P_*(\mathbf{S}[i])|$ is the number of missing entries in $\mathbf{S}[i]$

3: Choose any $i \in [n]$ such that $\ell - |P_*(\mathbf{S}[i])| > d_i$.

4: Choose any $R \subseteq [\ell] \setminus P_*(\mathbf{S}[i])$ with $|R| = d_i + 1$.

5: for all $j \in R$ do

6: Let $\mathbf{S}' = \mathbf{S}[:, [\ell] \setminus \{j\}]$ and $d'_{i'} = d_i - \delta(\mathbf{S}[i, j], \mathbf{S}[i', j])$ for each $i' \in [n]$.

7: if recursion on $(\mathbf{S}', d'_1, \dots, d'_n)$ returns **Yes** then return **Yes**.

8: return **No**.

each $i \in [n]$, we solve the instance $(\mathbf{S}, d_1, \dots, d_n)$ where $d_{i'} = d$ for each $i' \in [n] \setminus \{i\}$ and $d_i = 0$. Clearly, (\mathbf{S}, d) is a **Yes**-instance if and only if at least one CONRMC-instance is a **Yes**-instance. This yields the following.

► **Corollary 2.** *MINLRMC can be solved in $O(n^2\ell)$ time when $d = 1$.*

Since CONRMC is solvable in $O^*(2^{O(n^2 \log n)})$ time [13], we also obtain the following result, where the running time bound only depends on the number n of rows.

► **Corollary 3.** *MINLRMC can be solved in $O^*(2^{O(n^2 \log n)})$ time.*

Finally, we remark that CONRMC can be solved in linear time for binary alphabet $\Sigma = \{0, 1\}$ if $d_i \geq \ell - 1$ for all $i \in [n]$ (the problem remains NP-hard in the case of unbounded alphabet size [17] even if $d_i \geq \ell - 1$ for all $i \in [n]$): First, we remove each row vector with distance bound ℓ . We also remove every row vector with at least one missing entry since it has distance at most $\ell - 1$ from any vector of length ℓ . We then remove every duplicate row vector. This can be achieved in linear time: We sort the row vectors lexicographically using radix sort and we compare each row vector to the adjacent row vectors in the sorted order. We return **Yes** if and only if there are at most $2^\ell - 1$ row vectors, because each distinct row vector $u \in \{0, 1\}^\ell$ excludes exactly one row vector $\bar{u} \in \{0, 1\}^\ell$ where $\bar{u}[j] = 1 - u[j]$ for each $j \in [\ell]$. Summarizing, we arrive at the following.

► **Proposition 4.** *If $\Sigma = \{0, 1\}$ and $d_i \geq \ell - 1$ for all $i \in [n]$, then CONRMC can be solved in linear time.*

4 Parameter number ℓ of columns

Hermelin and Rozenberg [12, Theorem 3] showed that one can solve MINRMC in $O(2^{\ell^2/2} \cdot n\ell)$ time using a search tree algorithm. We use a more refined recursive step to obtain a better running time (see Algorithm 1). In particular we employ a trick used by Gramm et al. [11] in order to reduce the search space to $d + 1$ subcases. Note that for nontrivial instances clearly $d < \ell$.

► **Theorem 5.** *For $d := \max_{i \in [n]} d_i$, CONRMC can be solved in $O((d + 1)^\ell \cdot n\ell)$ time.*

Proof. We prove that Algorithm 1 is correct by induction on ℓ . More specifically, we show that it returns **Yes** if there exists a vector $v \in \Sigma^\ell$ that satisfies $\delta(\mathbf{S}[i], v) \leq d_i$ for all $i \in [n]$. It is easy to see that the algorithm is correct for the base case $\ell = 0$, because it returns

Yes if d_i is nonnegative for all $i \in [n]$ and **No** otherwise (Lines 1 and 2). Consider the case $\ell > 0$. The terminating conditions in Lines 1 and 2 are clearly correct. We show that branching on R is correct in Lines 4 and 5. If $v[j] \neq \mathbf{S}[i, j]$ holds for all $j \in R$, then we have a contradiction $\delta(v, \mathbf{S}[i]) \geq |R| > d_i$. Thus the branching on R leads to a correct output. Now the induction hypothesis ensures that the recursion on $\mathbf{S}[:, [\ell] \setminus \{j\}]$ (notice that it has exactly one column less) returns a desired output. This implies that our algorithm is correct.

As regards the time complexity, note that each node in the search tree has at most $d + 1$ children. Moreover, the depth of the search tree is at most ℓ because the number of columns decreases for each recursion. Since each recursion step only requires linear (that is, $O(n\ell)$) time, the overall running time is in $O((d + 1)^\ell \cdot n\ell)$. ◀

Since $d < \ell$ for nontrivial input instances, Theorem 5 yields “linear-time fixed-parameter tractability” with respect to ℓ , meaning an exponential speedup over the previous result due to Hermelin and Rozenberg [12].

► **Corollary 6.** *CONRMC can be solved in $O(n\ell^{\ell+1})$ time.*

Proof. For each $i \in [n]$, we construct an CONRMC-instance, where the input matrix is $\mathbf{S}_i = \mathbf{S}[:, P_*(\mathbf{S}[i])]$ and $d_{i,i'} = d - \delta(\mathbf{S}[i], \mathbf{S}[i'])$ for each $i' \in [n]$. We return **Yes** if and only if there is a **Yes**-instance $(\mathbf{S}_i, d_{i,1}, \dots, d_{i,n})$ of CONRMC. Each CONRMC-instance requires $O(n\ell)$ time to construct, and $O(nk^{k+1})$ time to solve, because \mathbf{S}_i contains at most k columns. ◀

We remark that this algorithm cannot be significantly improved assuming the ETH.¹ It is known that there is no $\ell^{o(\ell)} \cdot n^{O(1)}$ -time algorithm for the special case CLOSEST STRING unless the ETH fails [17]. The running time of our algorithm matches this lower bound (up to a constant in the exponent) and therefore there is presumably no substantially faster algorithm with respect to ℓ .

As a consequence of Corollary 6, we obtain a fixed-parameter algorithm for MINLRMC with respect to the maximum number k of missing entries per row in the input matrix.

► **Corollary 7.** *MINLRMC can be solved in time $O(n^2\ell + n^2k^{k+1})$.*

5 Combined parameter $d + k$

In this section we generalize two algorithms (one by Gramm et al. [11] and one by Ma and Sun [18]) for the special case of MINRMC in which the input matrix is complete (known as the CLOSEST STRING problem) to the case of incomplete matrices. We will describe both algorithms briefly. In fact, both algorithms solve the special case of CONRMC, referred to as NEIGHBORING STRING (generalizing CLOSEST STRING by allowing row-individual distances), where the input matrix is complete.

NEIGHBORING STRING

Input: A matrix $\mathbf{T} \in \Sigma^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$

Question: Is there a row vector $v \in \Sigma^\ell$ such that $\delta(v, \mathbf{T}[i]) \leq d_i$ for each $i \in [n]$?

The algorithm of Gramm et al. [11] is given in Algorithm 2. First, it determines whether the first row vector $\mathbf{T}[1]$ is a solution. If not, then it finds another row vector $\mathbf{T}[i]$ that differs from $\mathbf{T}[1]$ on more than d_i positions and branches on the column positions $Q(\mathbf{T}[1], \mathbf{T}[i])$ where $\mathbf{T}[1]$ and $\mathbf{T}[i]$ disagree.

¹ The Exponential Time Hypothesis asserts that 3-SAT cannot be solved in $O^*(2^{o(n+m)})$ time for a 3-CNF formula with n variables and m clauses.

■ **Algorithm 2** Algorithm for NEIGHBORING STRING by Gramm et al. [11].

Input: A matrix $\mathbf{T} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Task: Decide whether there exists a row vector $v \in \Sigma^\ell$ with $d(v, \mathbf{T}[i]) \leq d_i$ for all $i \in [n]$.

- 1: if $d_i < 0$ for some $i \in [n]$ then return **No**.
- 2: if $\delta(\mathbf{T}[1], \mathbf{T}[i]) \leq d_i$ for all $i \in [n]$ then return **Yes**.
- 3: Choose any $i \in [n]$ such that $\delta(\mathbf{T}[1], \mathbf{T}[i]) > d_i$.
- 4: Choose any $Q' \subseteq Q(\mathbf{T}[1], \mathbf{T}[i])$ with $|Q'| = d_i + 1$.
- 5: **for all** $j \in Q'$ **do**
- 6: Let $\mathbf{T}' = \mathbf{T}[:, [\ell] \setminus \{j\}]$ and $d'_{i'} = d_i - \delta(\mathbf{T}[i, j], \mathbf{T}[i', j])$ for each $i' \in [n]$.
- 7: **if** recursion on $(\mathbf{T}', d'_1, \dots, d'_n)$ returns **Yes** **then return Yes**.
- 8: **return No**.

Using a search variant of Algorithm 2, Hermelin and Rozenberg [12, Theorem 4] claimed that MINRMC is fixed-parameter tractable with respect to $d + k$. Here, we reveal that their algorithm is in fact not correct. The algorithm chooses an arbitrary row vector $\mathbf{S}[i]$ and calls the algorithm by Gramm et al. [11] with input matrix $\mathbf{S}' = \mathbf{S}[:, [\ell] \setminus P_*(\mathbf{S}[i])]$. This results in a set of row vectors v satisfying $\delta(v, \mathbf{S}') \leq d$. Then, the algorithm constructs an instance of CONRMC where the input matrix is $\mathbf{S}[P_*(\mathbf{S}[i])]$ and the distance bound is given by $d_{i'} = d - \delta(v, \mathbf{S}'[i'])$ for each $i' \in [n]$. The correctness proof was based on the erroneous assumption that the algorithm of Gramm et al. [11] finds *all* row vectors v satisfying $\delta(v, \mathbf{S}') \leq d$ in time $O((d+1)^d \cdot n\ell)$. Although Gramm et al. [11] noted that this is indeed the case when d is optimal, it is not always true. In fact, it is generally impossible to enumerate all solutions in time $O((d+1)^d \cdot n\ell)$ because there can be $\Omega(\ell^d)$ solutions. We use the following simple matrix to illustrate the error in the algorithm of Hermelin and Rozenberg [12]:

$$\mathbf{S} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ * & 0 & 0 \end{bmatrix}.$$

We show that the algorithm may output an incorrect answer for $d = 2$. If the algorithm chooses $i = 3$, then the algorithm by Gramm et al. [11] returns only one row vector 00. Then the algorithm of Hermelin and Rozenberg [12] constructs an instance of CONRMC with $\mathbf{S}' = [0 \ 1]^T$ and $d_1 = d_2 = 0$, resulting in **No**. However, the row vector $v = 001$ satisfies $\delta(v, \mathbf{S}) = 2$ and thus the correct output is **Yes**. To remedy this, we give a fixed-parameter algorithm for MINRMC, adapting the algorithm by Gramm et al. [11].

Before presenting our algorithm, let us give an observation noted by Gramm et al. [11] for the case of no missing entries. Suppose that the input matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ contains more than nd *dirty* columns (a column is said to be dirty if it contains at least two distinct symbols from the alphabet). Clearly, we can assume that every column is dirty. For any vector $v \in \Sigma^\ell$, there exists $i \in [n]$ with $\delta(v, \mathbf{S}[i]) \geq d$ by the pigeon hole principle and hence we can immediately conclude that it is a **No**-instance. It is easy to see that this argument also holds for MINRMC and thus CONRMC.

► **Lemma 8.** *Let $(\mathbf{S}, d_1, \dots, d_n)$ be a CONRMC instance, where $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$. If \mathbf{S} contains more than nd dirty columns for $d = \max_{i \in [n]} d_i$, then there is no row vector $v \in \Sigma^\ell$ with $\delta(v, \mathbf{S}[i]) \leq d_i$ for all $i \in [n]$.*

Our algorithm is given in Algorithm 3. It generalizes Algorithm 2 and finds the solution vector even if the input matrix is incomplete. In contrast to NEIGHBORING STRING, the output cannot be immediately determined even if $d_1 = 0$. We use Algorithm 1 to overcome

Algorithm 3 Algorithm for CONRMC (generalizing Algorithm 2).

Input: An incomplete matrix $\mathbf{S} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Task: Decide whether there exists a row vector $v \in \Sigma^\ell$ with $d(v, \mathbf{S}[i]) \leq d_i$ for all $i \in [n]$.

- 1: **if** $d_1 = 0$ **then**
 - 2: Let $\mathbf{S}' = \mathbf{S}[[2, n], P_*(\mathbf{S}[1])]$ and $d'_i = d_i - \delta(\mathbf{S}[1], \mathbf{S}[i])$ for each $i \in [2, n]$.
 - 3: **return** the output of Algorithm 1 on $(\mathbf{S}', d'_2, \dots, d'_n)$.
 - 4: Let $R_i = (P_*(\mathbf{S}[1]) \setminus P_*(\mathbf{S}[i])) \cup Q(\mathbf{S}[1], \mathbf{S}[i])$ for each $i \in [2, n]$.
 - 5: **if** $|R_i| \leq d_i$ for all $i \in [2, n]$ **then return Yes**.
 - 6: Choose any $i \in [n]$ with $|R_i| > d_i$.
 - 7: Choose any $R \subseteq R_i$ with $|R| = d_i + 1$.
 - 8: **for all** $j \in R$ **do**
 - 9: Let $\mathbf{S}' = \mathbf{S}[:, [\ell] \setminus \{j\}]$ and $d'_{i'} = d_i - \delta(\mathbf{S}[i, j], \mathbf{S}[i', j])$ for each $i' \in [n]$.
 - 10: **if** recursion on $(\mathbf{S}', d'_{i'}, \dots, d'_n)$ returns **Yes** **then return Yes**.
 - 11: **return No**.
-

this issue (Line 3). Algorithm 3 also considers the columns where the first row vector has missing entries (recall that $P_*(\mathbf{S}[1])$ denotes column indices j with $\mathbf{S}[1, j] = *$) in the branching step (Line 8), and not only the columns where $\mathbf{S}[1]$ and $\mathbf{S}[i]$ disagree. Again, we restrict the branching to $d_i + 1$ subcases (Line 7). This reduces the size of the search tree significantly. We show the correctness of Algorithm 3 and analyze its running time in the proof of the following theorem.

► **Theorem 9.** For $d = \max_{i \in [n]} d_i$, CONRMC can be solved in $O(n\ell + (d+1)^{d+k+1}n)$ time.

Proof. First, we prove that Algorithm 3 is correct by induction on $d_1 + |P_*(\mathbf{S}[1])|$. More specifically, we show that the algorithm returns **Yes** if and only if a vector $v \in \Sigma^\ell$ satisfying $\delta(\mathbf{S}[i], v) \leq d_i$ for all $i \in [n]$ exists.

Consider the base case $d_1 + |P_*(\mathbf{S}[1])| = 0$. Since $d_1 = 0$, the algorithm terminates in Line 3. When $d_1 = 0$, any solution vector must agree with $\mathbf{S}[1]$ on each entry unless the entry is missing in $\mathbf{S}[1]$. Hence, the output in Line 3 is correct by Theorem 5. Consider the case $d_1 + |P_*(\mathbf{S}[1])| > 0$. Let $R_i = (P_*(\mathbf{S}[1]) \setminus P_*(\mathbf{S}[i])) \cup Q(\mathbf{S}[1], \mathbf{S}[i])$ for each $i \in [2, n]$. If $|R_i| \leq d_i$ holds for all $i \in [2, n]$, then the vector $\mathbf{S}[1] \oplus \sigma^\ell$ (the vector obtained by filling each missing entry in $\mathbf{S}[1]$ with σ) is a solution for an arbitrary character $\sigma \in \Sigma$. Hence, Line 5 is correct. Suppose that there exists a solution vector $v \in \Sigma^\ell$ with $d(v, \mathbf{S}[i]) \leq d_i$ for all $i \in [n]$. We show that the branching in Line 8 is correct. Let R be as specified in Line 7. We claim that there exists a $j \in R$ with $v[j] = \mathbf{S}[i, j]$ for every choice of R . Otherwise, $v[j] \neq \mathbf{S}[i, j]$ and $\mathbf{S}[i, j] \neq *$ holds for all $j \in R$ and we have $\delta(v, \mathbf{S}[i]) > d_i$ (a contradiction). Note that $\mathbf{S}[:, [\ell] \setminus \{j\}]$ has exactly one less missing entry if $j \in P_*(\mathbf{S}[1])$ and that $d'_1 = d_1 - 1$ in case of $j \in Q(\mathbf{S}[1], \mathbf{S}[i])$. It follows that $d_1 + |P_*(\mathbf{S}[1])|$ is strictly smaller in the recursive call (Line 10). Hence, the induction hypothesis ensures that the algorithm returns **Yes** when $v[j] = \mathbf{S}[i, j]$ holds. On the contrary, it is not hard to see that the algorithm returns **No** if there is no solution vector. Thus, Algorithm 3 is correct.

We examine the time complexity. Assume without loss of generality that $k = |P_*(\mathbf{S}[1])|$ and $d = d_1$ hold initially. Consider the search tree where each node corresponds to a call on either Algorithm 1 or Algorithm 3. If $d_1 > 0$, then $d_1 + |P_*(\mathbf{S}[1])|$ decreases by 1 in each recursion and there are at most $d + 1$ recursive calls. Let u be some node in the search tree that invokes Algorithm 1 for the first time. We have seen in the proof of Theorem 5 that the subtree rooted at u is a tree of depth at most $|P_*(\mathbf{S}[1])|$, in which each node has at most

■ **Algorithm 4** Algorithm for NEIGHBORING STRING by Ma and Sun [18].

Input: A matrix $\mathbf{T} \in (\Sigma \cup \{*\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Task: Decide whether there exists a row vector $v \in \Sigma^\ell$ with $d(v, \mathbf{T}[i]) \leq d_i$ for all $i \in [n]$.

- 1: if $\delta(\mathbf{T}[1], \mathbf{T}[i]) \leq d_i$ for all $i \in [n]$ **then return Yes** .
- 2: Choose any $i \in [n]$ such that $\delta(\mathbf{T}[1], \mathbf{T}[i]) > d_i$.
- 3: Let $Q = Q(\mathbf{T}[1], \mathbf{T}[i])$.
- 4: **for all** $v \in \Sigma^{|Q|}$ such that $\delta(v, \mathbf{T}[1]) \leq d_1$ and $\delta(v, \mathbf{T}[i]) \leq d_i$ **do**
- 5: Let $\mathbf{T}' = \mathbf{T}[:, [\ell] \setminus Q]$ and $d'_1 = \min\{d_1 - \delta(v, \mathbf{T}[1, Q]), \lceil d_1/2 \rceil - 1\}$.
- 6: Let $d'_{i'} = d_{i'} - d(v, \mathbf{T}[i', Q])$ for each $i' \in [2, n]$.
- 7: **if** recursion on $(\mathbf{T}, d'_1, \dots, d'_n)$ returns **Yes** **then return Yes**.
- 8: **return No**.

$d_i - \delta(\mathbf{S}[1], \mathbf{S}[i]) + 1 \leq d + 1$ children. Note also that u lies at depth $d + k - |P_*(\mathbf{S}[1])|$. Thus, the depth of the search tree is at most $d + k$ and the search tree has size $O((d + 1)^{d+k})$. We can assume that $\ell \leq nd$ by Lemma 8 and hence each node requires $O(nd)$ time. This results in the overall running time of $O(n\ell + (d + 1)^{d+k+1}n)$. ◀

Now, we provide a more efficient fixed-parameter algorithm when the alphabet size is small, based on an algorithm by Ma and Sun [18] (see Algorithm 4). Whereas Algorithm 2 considers each position of (a subset of) $Q(\mathbf{T}[1], \mathbf{T}[i])$ one by one, their algorithm considers all vectors on $Q(\mathbf{T}[1], \mathbf{T}[i])$ in a single recursion. The following lemma justifies that d_1 can actually be halved in each iteration (the vectors u and w correspond to $\mathbf{T}[1]$ and $\mathbf{T}[i]$, respectively).

► **Lemma 10.** [18, Lemma 3.1] *Let $u, v, w \in \Sigma^\ell$ be row vectors satisfying $\delta(u, w) > \delta(v, w)$. Then, it holds that $\delta(u[Q'], v[Q']) < \delta(u, v)/2$ for $Q' = [\ell] \setminus Q(u, w)$.*

Proof. Assume that $\delta(u[Q'], v[Q']) \geq \delta(u, v)/2$. We can rewrite the value of $\delta(u, v) + \delta(v, w)$ as follows:

$$\delta(u, v) + \delta(v, w) = \delta(u[Q'], v[Q']) + \delta(v[Q'], w[Q']) + \delta(u[Q], v[Q]) + \delta(v[Q], w[Q]),$$

where Q is a shorthand for $Q = Q(v, w)$. It follows from the definition of Q' that $u[Q'] = w[Q']$ and hence

$$\delta(u[Q'], v[Q']) = \delta(v[Q'], w[Q']). \quad (1)$$

We also note that $\delta(u[Q] + v[Q]) + \delta(v[Q] + w[Q]) \geq |Q| = \delta(v, w)$ because it must hold that $u[j] \neq v[j]$ or $v[j] \neq w[j]$ for each $j \in Q$. Now, we obtain the following contradiction:

$$\delta(u, v) + \delta(v, w) \geq 2\delta(u[Q'], v[Q']) + \delta(u, w) > \delta(u, v) + \delta(v, w),$$

concluding the proof. ◀

Lemma 10 plays a crucial role in obtaining the running time $O(n\ell + (16|\Sigma|)^d nd)$ of Ma and Sun [18]. However, Lemma 10 may not hold in the presence of missing entries. To work around this issue, let us introduce a new variant of CLOSEST STRING which will be useful to derive a fixed-parameter algorithm for CONRMC (Theorem 12). We will use a special character “ \diamond ” to denote a “dummy” character.

NEIGHBORING STRING WITH DUMMIES (NSD)

Input: A matrix $\mathbf{T} \in (\Sigma \cup \{\diamond\})^{n \times \ell}$ and $d_1, \dots, d_n \in \mathbb{N}$.

Question: Is there a row vector $v \in \Sigma^\ell$ such that $\delta(v, \mathbf{T}[i]) \leq d_i$ for each $i \in [n]$?

Note that the definition of NSD forbids dummy characters in the solution vector v . Observe that Lemma 10 holds even if row vectors u and w contain dummy characters. We show (based on Lemma 10) that NSD can be solved using the algorithm of Ma and Sun [18] as a subroutine.

► **Lemma 11.** *NSD can be solved in $O(n\ell + |\Sigma|^k \cdot nk + 2^{4d-3k} \cdot |\Sigma|^d \cdot nd)$ time, where $d := \max_{i \in [n]} d_i$ and k is the minimum number of dummy characters in any row vector of \mathbf{T} .*

Proof. With Lemma 10, assuming $d = d_1$ one can prove by induction on d_1 that Algorithm 4 solves the NSD problem if the first row vector $\mathbf{T}[1]$ contains no dummy characters by induction on d_1 . Refer to [18, Theorem 3.2] for details. We describe how we use Algorithm 4 of Ma and Sun [18] to solve NSD. Let $I = (\mathbf{T}, d_1, \dots, d_n)$ be an instance of NSD. We assume that $|P_\diamond(\mathbf{T}[1])| = k$. For each row vector u of Σ^k , we invoke Algorithm 4 with the input matrix $\mathbf{T}' = \mathbf{T}[[\ell] \setminus P_\diamond(\mathbf{T}[1])]$ and the distance bounds $d_1 - k, d_2 - \delta(u, \mathbf{T}[2, P_\diamond(\mathbf{T}[1])]), \dots, d_n - \delta(u, \mathbf{T}[n, P_\diamond(\mathbf{T}[1])])$. Note that $\mathbf{T}'[1]$ contains no dummy character and thus the output of Algorithm 4 is correct. We return **Yes** if and only if Algorithm 4 returns **Yes** at least once. Let us prove that this solves NSD. If I is a **Yes**-instance with solution vector $v \in \Sigma^\ell$, then it is easy to verify that Algorithm 4 returns **Yes** when $u = v[P_\diamond(\mathbf{T}[1])]$. On the contrary, the distance bounds in the above procedure ensure that I is a **Yes**-instance if Algorithm 4 returns **Yes**.

Now we show that this procedure runs in the claimed time. Ma and Sun [18] proved that Algorithm 4 runs in

$$O\left(n\ell + \binom{d_{\max} + d_{\min}}{d_{\min}} \cdot (4|\Sigma|)^{d_{\min}} \cdot nd_{\max}\right)$$

time, where $d_{\max} = \max_{i \in [n]} d_i$ and $d_{\min} = \min_{i \in [n]} d_i$. In fact, they showed that each node in the search tree requires $O(nd_{\max})$ time by remembering previous distances, as it only concerns $O(d_{\max})$ columns. In the same spirit, one can compute distances from the first row vector for each NSD-instance under consideration in $O(nk)$ time, given the corresponding distances in the input matrix. Since we have $d_{\max} \leq d$ and $d_{\min} \leq d - k$ for each call of Algorithm 4, it remains to show that $\binom{2d-k}{d} \in O(2^{2d-k})$. Using Stirling's approximation $\sqrt{2\pi n}^{n+1/2} e^{-n} \leq n! \leq en^{n+1/2} e^{-n}$ which holds for all positive integers n , we obtain

$$\binom{2d-k}{d} = \frac{(2d-k)!}{d! \cdot (d-k)!} \leq c \cdot \frac{(2d-k)^{2d-k}}{d^d \cdot (d-k)^{d-k}}$$

for some constant c . We claim that the last term is upper-bounded by $c \cdot 2^{2d-k}$. We use the fact that the function $x \mapsto x \log x$ is convex over its domain $x > 0$ (note that the second derivative is given by $x \mapsto 1/x$). Since a convex function $f: D \rightarrow \mathbb{R}$ satisfies $(f(x) + f(y))/2 \geq f((x+y)/2)$ for any $x, y \in D$, we obtain

$$d \log d + (d-k) \log(d-k) \geq 2 \left(\frac{2d-k}{2} \right) \cdot \log \left(\frac{2d-k}{2} \right).$$

It follows that $d^d \cdot (d-k)^{d-k} = 2^{d \log d + (d-k) \log(d-k)} \geq 2^{(2d-k) \log(d-k/2)} = (d-k/2)^{2d-k}$. This shows that $\binom{2d-k}{d} \in O(2^{2d-k})$. ◀

Finally, to show our second main result in this section, we provide a polynomial-time reduction from CONRMC to NSD.

► **Theorem 12.** *CONRMC can be solved in $O(n\ell + 2^{4d+k} \cdot |\Sigma|^{d+k} \cdot n(d+k))$ time.*

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & * \\ * & * & 2 \end{bmatrix} \quad \mathbf{T} = \begin{bmatrix} 0 & 0 & 0 & \diamond & \diamond \\ 1 & 1 & \diamond & \sigma & \diamond \\ \diamond & \diamond & 2 & \sigma & \sigma \end{bmatrix}$$

■ **Figure 2** An illustration of the reduction in Theorem 12. Given the matrix \mathbf{S} with $k = 2$ (left), our reduction constructs the matrix \mathbf{T} with k additional columns (right). Note that every row vector in \mathbf{T} contains exactly two dummy characters. The MINRMC instance $(\mathbf{S}, d = 1)$ is a **Yes**-instance with a solution vector $v = 100$. The corresponding NSD-instance $(\mathbf{T}, d + k = 3)$ is also a **Yes**-instance with a solution vector $v' = 100\sigma\sigma$.

Proof. Let $I = (\mathbf{S}, d_1, \dots, d_n)$ be an instance of CONRMC. We construct an instance $I' = (\mathbf{T}, d_1 + k, \dots, d_n + k)$ of NSD where $\mathbf{T} \in (\Sigma \cup \{\diamond\})^{n \times (\ell + k)}$ and each row vector of \mathbf{T} contains exactly k dummy characters. Note that such a construction yields an algorithm running in $O(n(\ell + k) + |\Sigma|^k \cdot nk + 2^{4d+k} \cdot |\Sigma|^{d+k} \cdot n(d+k)) = O(n\ell + 2^{4d+k} \cdot |\Sigma|^{d+k} \cdot n(d+k))$ time using Lemma 11. Let $\sigma \in \Sigma$ be an arbitrary character. We define the row vector $\mathbf{T}[i]$ for each $i \in [n]$ as follows: Let $\mathbf{T}[i, [\ell]] = \mathbf{S}[i] \oplus \diamond^\ell$ (in other words, the row vector $\mathbf{T}[i, [\ell]]$ is obtained from $\mathbf{S}[i]$ by replacing $*$ by \diamond) for the leading ℓ entries. For the remainder, let

$$\mathbf{T}[i, \ell + j] = \begin{cases} \sigma & \text{if } j \leq |P_*(\mathbf{S}[i])|, \\ \diamond & \text{otherwise,} \end{cases}$$

for each $j \in [k]$ (Figure 2 shows an illustration). We claim that I is a **Yes** instance if and only if I' is a **Yes** instance.

(\Rightarrow) Let $v \in \Sigma^\ell$ be a solution of I . We claim that the vector $v' \in \Sigma^{\ell+k}$ with $v'[[\ell]] = v$ and $v'[\ell + 1, \ell + k] = \sigma^k$ is a solution of I' . For each $i \in [n]$, we have

$$\delta(v', \mathbf{T}[i]) = \delta(v'[[\ell]], \mathbf{T}[i, [\ell]]) + \delta(\sigma^k, \mathbf{T}[i, [\ell + 1, \ell + k]]).$$

It is easy to see that the first term is at most $d_i + |P_*(\mathbf{S}[i])|$ and that the second term equals $k - |P_*(\mathbf{S}[i])|$. Thus we have $\delta(v', \mathbf{T}) \leq d_i + k$.

(\Leftarrow) Let $v' \in \Sigma^\ell$ be a solution of I' . Since the row vector $\mathbf{T}[i, [\ell + 1, \ell + k]]$ contains $k - |P_*(\mathbf{S}[i])|$ dummy characters, we have $\delta(v'[[\ell]], \mathbf{T}[i, [\ell]]) \leq (d_i + k) - (k - |P_*(\mathbf{S}[i])|) = d_i + |P_*(\mathbf{S}[i])|$ for each $i \in [n]$. It follows that $\delta(v'[[\ell]], \mathbf{S}[i]) \leq d_i$ holds for each $i \in [n]$. ◀

Note that the algorithm of Theorem 12 is faster than the algorithm of Theorem 9 for $|\Sigma| < d/16$ and faster than the $O^*(|\Sigma|^k \cdot d^d)$ -time algorithm by Hermelin and Rozenberg [12] for $|\Sigma| < d/2^{4+d/k}$.

6 Conclusion

We studied problems appearing both in stringology and in the context of matrix completion. The goal in both settings is to find a consensus string (matrix row) that is close to all given input strings (rows). The special feature here now is the existence of wildcard letters (missing entries) appearing in the strings (rows). Thus, these problems naturally generalize the well-studied CLOSEST STRING and related string problems. Although with applications in the context of data mining, machine learning, and computational biology at least as well motivated as CLOSEST STRING, so far there is comparatively little work on these “wildcard problems”. This work is also meant to initiate further research in this direction.

We conclude with a list of challenges for future research:

- Can the running time of Theorem 12 be improved? Since Ma and Sun [18] proved that CLOSEST STRING can be solved in $O((16|\Sigma|)^d \cdot n\ell)$ time, a plethora of efforts have been made to reduce the base in the exponential dependence in the running time [6, 19, 4, 5, 22]. A natural question is whether these results can be translated to MINRMC and CONRMC as well.
- Another direction would be to consider the generalization of MINRMC with *outliers*. The task is to determine whether there is a set $I \subseteq [n]$ of row indices and a vector $v \in \Sigma^\ell$ such that $|I| \leq t$ and $\delta(v, \mathbf{S}[[n] \setminus I]) \leq d$. For complete input matrices, this problem is known as CLOSEST STRING WITH OUTLIERS and fixed-parameter tractability with respect to $d + t$ is known [2]. Hence, it is interesting to study whether the outlier variant of MINRMC (or CONRMC) is fixed-parameter tractable with respect to $d + k + t$.
- Finally, let us mention a maximization variant MAXRMC where the goal is to have a radius at least d . The case of complete input matrices is referred to as FARTHEST STRING [21] and fixed-tractability with respect to $|\Sigma| + d$ is known [10, 21]. Is MAXRMC also fixed-parameter tractable with respect to $|\Sigma| + d$?

References

- 1 Amihod Amir, Jessica Fidler, Liam Roditty, and Oren Sar Shalom. On the efficiency of the Hamming c -centerstring problems. In *25th Symposium on Combinatorial Pattern Matching (CPM '14)*, pages 1–10. Springer, 2014.
- 2 Christina Boucher and Bin Ma. Closest string with outliers. *BMC Bioinformatics*, 12(S-1):S55, 2011.
- 3 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, 114, 2014.
- 4 Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. A three-string approach to the closest string problem. *Journal of Computer and System Sciences*, 78(1):164–178, 2012.
- 5 Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. Randomized fixed-parameter algorithms for the closest string problem. *Algorithmica*, 74(1):466–484, 2016.
- 6 Zhi-Zhong Chen and Lusheng Wang. Fast exact algorithms for the closest string and substring problems with application to the planted (ℓ, d) -motif model. *IEEE/ACM Transactions Computational Biology and Bioinformatics*, 8(5):1400–1410, 2011.
- 7 Eduard Eiben, Robert Ganian, Iyad Kanj, Sebastian Ordyniak, and Stefan Szeider. On clustering incomplete data. *CoRR*, abs/1911.01465, 2019. URL: <http://arxiv.org/abs/1911.01465>.
- 8 Moti Frances and Ami Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997.
- 9 Robert Ganian, Iyad A. Kanj, Sebastian Ordyniak, and Stefan Szeider. Parameterized algorithms for the matrix completion problem. In *35th International Conference on Machine Learning (ICML '18)*, volume 80 of *Proceedings of Machine Learning Research*, pages 1642–1651. PMLR, 2018.
- 10 Jens Gramm, Jiong Guo, and Rolf Niedermeier. Parameterized intractability of distinguishing substring selection. *Theory of Computing Systems*, 39(4):545–560, 2006.
- 11 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.
- 12 Danny Hermelin and Liat Rozenberg. Parameterized complexity analysis for the closest string with wildcards problem. *Theoretical Computer Science*, 600:11–18, 2015. Preliminary version appeared at *CPM '14*.
- 13 Dušan Knop, Martin Koutecký, and Matthias Mnich. Combinatorial n -fold integer programming and applications. *Mathematical Programming*, 2019.

20:14 Parameterized Algorithms for Matrix Completion with Radius Constraints

- 14 Tomohiro Koana, Vincent Froese, and Rolf Niedermeier. Complexity of combinatorial matrix completion with diameter constraints. *CoRR*, abs/2002.05068, 2020. [arXiv:2002.05068](#).
- 15 Ming Li, Bin Ma, and Lusheng Wang. Finding similar regions in many sequences. *Journal of Computer and System Sciences*, 65(1):73–96, 2002.
- 16 Ming Li, Bin Ma, and Lusheng Wang. On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171, 2002.
- 17 Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Slightly superexponential parameterized problems. *SIAM Journal on Computing*, 47(3):675–702, 2018.
- 18 Bin Ma and Xiaoming Sun. More efficient algorithms for closest string and substring problems. *SIAM Journal on Computing*, 39(4):1432–1443, 2009.
- 19 Naomi Nishimura and Narges Simjour. Enumerating neighbour and closest strings. In *7th International Symposium on Parameterized and Exact Computation (IPEC '12)*, pages 252–263. Springer, 2012.
- 20 Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, pages 827–831. Springer, 2005.
- 21 Lusheng Wang and Binhai Zhu. Efficient algorithms for the closest string and distinguishing string selection problems. In *3rd International Frontiers in Algorithmics Workshop (FAW '09)*, pages 261–270. Springer, 2009.
- 22 Shota Yuasa, Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. Designing and implementing algorithms for the closest string problem. *Theoretical Computer Science*, 786:32–43, 2019.

In-Place Bijective Burrows-Wheeler Transforms

Dominik Köppl 

Department of Informatics, Kyushu University, Fukuoka, Japan
Japan Society for Promotion of Science (JSPS), Tokyo, Japan
<https://dkppl.de/>
dominik.koeppel@inf.kyushu-u.ac.jp

Daiki Hashimoto

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
daiki_hashimoto@shino.ecei.tohoku.ac.jp

Diptarama Hendrian 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Ayumi Shinohara 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

Abstract

One of the most well-known variants of the Burrows-Wheeler transform (BWT) [Burrows and Wheeler, 1994] is the bijective BWT (BBWT) [Gil and Scott, arXiv 2012], which applies the extended BWT (EBWT) [Mantaci et al., TCS 2007] to the multiset of Lyndon factors of a given text. Since the EBWT is invertible, the BBWT is a bijective transform in the sense that the inverse image of the EBWT restores this multiset of Lyndon factors such that the original text can be obtained by sorting these factors in non-increasing order.

In this paper, we present algorithms constructing or inverting the BBWT in-place using quadratic time. We also present conversions from the BBWT to the BWT, or vice versa, either (a) in-place using quadratic time, or (b) in the run-length compressed setting using $\mathcal{O}(n \lg r / \lg \lg r)$ time with $\mathcal{O}(r \lg n)$ bits of words, where r is the sum of character runs in the BWT and the BBWT.

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

Keywords and phrases In-Place Algorithms, Burrows-Wheeler transform, Lyndon words

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.21

Related Version <http://arxiv.org/abs/2004.12590>

Supplementary Material At https://github.com/daikihashimoto/BWT_to_BBWT, we have some preliminary implementations available giving empirical evidence of our conversions.

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number JP18F18120.

Diptarama Hendrian: JSPS KAKENHI Grant Number JP19K20208.

Ayumi Shinohara: JSPS KAKENHI Grant Number JP15H05706.

Acknowledgements We thank the anonymous reviewers for attracting our attention to the related work of Mantaci et al. [24].

1 Introduction

The *Burrows-Wheeler transform* (BWT) [6] is one of the most favored options both for (a) compressing and (b) indexing data sets. On the one hand, compression programs like **bzip2** apply the BWT to achieve high compression rates. For that, they leverage the effect that the BWT built on repetitive data tends to have long character runs, which can be



© Dominik Köppl, Daiki Hashimoto, Diptarama Hendrian, and Ayumi Shinohara;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 21; pp. 21:1–21:15



Leibniz International Proceedings in Informatics

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

compressed by run-length compression, i.e., representing a substring of ℓ a's by the tuple (a, ℓ) . On the other hand, self-indexing data structures like the FM-index [11] enhance the BWT to a full-text self-index. A combined approach of both compression and indexing is the run-length compressed FM-index [21], representing a BWT with r_{BWT} character runs, i.e., maximal repetitions of a character, run-length compressed in $\mathcal{O}(r_{\text{BWT}} \lg n)$ bits. This representation can be computed directly in run-length compressed space thanks to Policriti and Prezza [30]. The BWT and its run-length compressed representation have been intensively studied during the past decades (e.g., [12, 1, 14] and the references therein). Contrary to that, a variant, called the *bijective* BWT (BBWT) [16], is far from being well-studied despite its mathematically appealing characteristics¹. As a matter of fact, we are only aware of one index data structure based on the BBWT [3] and of two non-trivial construction algorithms [5, 2] of the (uncompressed) BBWT, both with the need of additional data structures.

In this article, we shed more light on the connection between the BWT and the BBWT by quadratic time in-place conversion algorithms in Sect. 5 constructing the BWT from the BBWT, or vice versa. We can also perform these conversions in the run-length compressed setting in $\mathcal{O}(n \lg r / \lg \lg r)$ time with space linear to the number of the character runs (cf. Sect. 4 and Thm. 3), where r is the sum of character runs in the BWT and the BBWT.

2 Related Work

Given a text T of length n , the BWT of T is the string obtained by assigning $\text{BWT}[i]$ to the character preceding the i -th lexicographically smallest suffix of T (or the last character of T if this suffix is the text itself). By this definition, we can construct the BWT with any suffix array [22] construction algorithm. However, storing the suffix array inherently needs $n \lg n$ bits of space. Crochemore et al. [9] tackled this space problem with an in-place algorithm constructing the BWT in $\mathcal{O}(n^2)$ online on the reversed text by simulating queries on a dynamic wavelet tree [17] that would be built on the (growing) BWT. They also gave an algorithm for restoring the text in-place in $\mathcal{O}(n^{2+\epsilon})$ time.

In the run-length compressed setting, Policriti and Prezza [30] can compute the run-length compressed BWT having r_{BWT} character runs in $\mathcal{O}(n \lg r_{\text{BWT}})$ time while using $\mathcal{O}(r_{\text{BWT}} \lg n)$ bits of space. They additionally presented an adaption of the wavelet tree on run-length compressed texts, yielding a representation using $\mathcal{O}(r_{\text{BWT}} \lg n)$ bits of space with $\mathcal{O}(\lg r_{\text{BWT}})$ query and update time. Finally, practical improvements of the run-length compressed BWT construction were considered by Ohno et al. [29].

The BBWT is the string obtained by assigning $\text{BBWT}[i]$ to the last character of the i -th smallest string in the list of all conjugates of the factors of the Lyndon factorization sorted with respect to the \prec_ω order [23, Def. 4]. Bannai et al. [2] recently revealed a connection between the bijective BWT and suffix sorting by presenting an $\mathcal{O}(n)$ time BBWT construction algorithm based on SAIS [28]. With dynamic data structures like a dynamic wavelet tree [27], Bonomo et al. [5] could devise an algorithm computing the BBWT in $\mathcal{O}(n \lg n / \lg \lg n)$ time. With nearly the same techniques, Mantaci et al. [24] presented an algorithm computing the BWT (and simultaneously the suffix array if needed) from the Lyndon factorization. All these construction algorithms need however data structures taking $\mathcal{O}(n \lg n)$ bits of space. However, the latter two (i.e., [5] and [24]) can work in-place by simulating the LF mapping (cf. Sects. 3.4 and 3.5), which we focus on in Sect. 5.1.

¹ The BBWT is a bijection between strings without the need of an artificial delimiter needed, e.g., to invert the BWT.

3 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. An algorithm is called *in-place* if it uses, besides a rewriteable input, only $\mathcal{O}(\lg n)$ bits of working space. We write $[b(I) \dots e(I)] = I$ for an interval I of natural numbers.

3.1 Strings

Let Σ denote an integer alphabet of size σ with $\sigma = n^{\mathcal{O}(1)}$. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer $j \in [1 \dots |T|]$, we access the j -th character of T with $T[j]$. Concatenating a string $T \in \Sigma^*$ k times is abbreviated by T^k . A string T is called *primitive* if there is no string $S \in \Sigma^+$ with $T = S^k$ for an integer k with $k \geq 2$.

When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X , Y and Z are called a *prefix*, *substring* and *suffix* of T , respectively; the prefix X , substring Y , or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. For two integers i, j with $1 \leq i \leq j \leq |T|$, let $T[i \dots j]$ denote the substring of T that begins at position i and ends at position j in T . If $i > j$, then $T[i \dots j]$ is the empty string. In particular, the suffix starting at position j of T is called the *j -th suffix* of T , and denoted with $T[j \dots]$. An occurrence of a substring S in T is treated as a sub-interval of $[1 \dots |T|]$ such that $S = T[b(S) \dots e(S)]$. The *longest common prefix (LCP)* of two strings S and T is the longest string that is a prefix of both S and T .

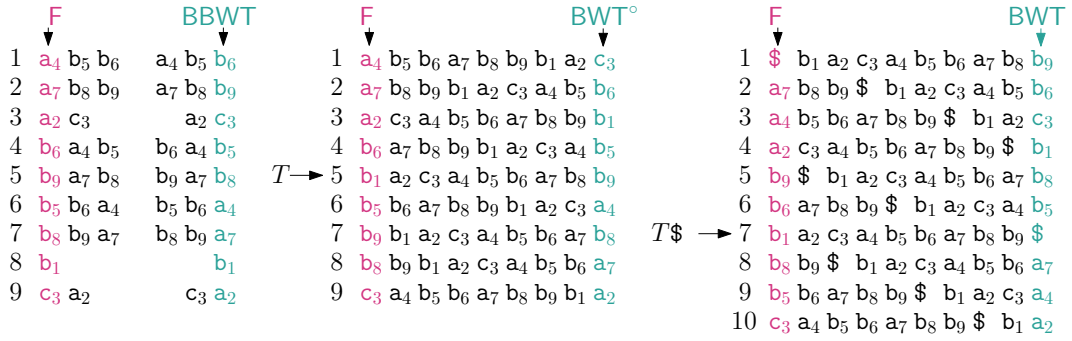
Orders on Strings. We denote the *lexicographic order* with \prec_{lex} . Given two strings S and T , $S \prec_{\text{lex}} T$ if S is a prefix of T or there exists an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1 \dots \ell - 1] = T[1 \dots \ell - 1]$ and $S[\ell] < T[\ell]$. Next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

Rank and Select Queries. Given a string $T \in \Sigma^*$, a character $c \in \Sigma$, and an integer j , the *rank* query $T.\text{rank}_c(j)$ counts the occurrences of c in $T[1 \dots j]$, and the *select* query $T.\text{select}_c(j)$ gives the position of the j -th c in T . We stipulate that $\text{rank}_c(0) = \text{select}_c(0) = 0$. A *wavelet tree* is a data structure supporting rank and select queries.

3.2 Lyndon Words

Given a string $T = T[1 \dots n]$, its i -th *conjugate* $\text{conj}_i(T)$ is defined as $T[i+1 \dots n]T[1 \dots i]$ for an integer $i \in [0 \dots n-1]$. We say that T and all of its conjugates belong to the *conjugate class* $\text{conj}(T) := \{\text{conj}_0(T), \dots, \text{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly* one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [20]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec_{\text{lex}} S$ for every proper suffix S of T [10, Prop. 1.2].

The *Lyndon factorization* [8] of $T \in \Sigma^+$ is the factorization of T into a sequence of lexicographically non-increasing Lyndon words $T_1 \dots T_t$, where (a) each $T_x \in \Sigma^+$ is a Lyndon word for $x \in [1 \dots t]$, and (b) $T_x \succeq_{\text{lex}} T_{x+1}$ for each $x \in [1 \dots t)$. Each Lyndon word T_x is called a *Lyndon factor*.



■ **Figure 1** All three BWT variants studied in this paper applied on our running example $T = \text{bacabbabb}$. *Left*: BBWT built on the last characters of the conjugates of all Lyndon words sorted in the \prec_ω order. *Middle and Right*: BWT[◦] and BWT built on the lexicographically sorted conjugates of T and of $T\$$, respectively. To ease understanding, each character is marked with its position in T in subscript. Reading these positions in F of BBWT and in F of BWT gives a circular suffix array (there are multiple possibilities with $T_3 = T_4 = \text{abb}$) and the suffix array (the position of $\$$ is uniquely defined as $|T\$| = 10$).

▶ **Lemma 1** ([10, Algo. 2.1]). *Given a string T of length n , there is an algorithm that outputs the Lyndon factors T_1, \dots, T_t one by one in increasing order in $\mathcal{O}(n)$ total time while keeping only a constant number of pointers to positions in T that (a) can move one position forward at one time or (b) can be set to the position of another pointer.*

For what follows, we fix a string $T[1..n]$ over an alphabet Σ with size σ . We use the string $T := \text{bacabbabb}$ as our running example. Its Lyndon factors are $T_1 = \text{b}$, $T_2 = \text{ac}$, $T_3 = \text{abb}$, and $T_4 = \text{abb}$.

3.3 Burrows-Wheeler Transforms

We denote the bijective BWT of T by BBWT, where $\text{BBWT}[i]$ is the last character of the i -th string in the list storing the conjugates of all Lyndon factors T_1, \dots, T_t of T sorted with respect to the \prec_ω order. A property of BBWT used in this paper as a starting point for an inversion algorithm is the following:

▶ **Lemma 2** ([5, Lemma 15]). $\text{BBWT}[1] = T[n]$.

Proof. There is no conjugate of a Lyndon factor that is smaller than the smallest Lyndon factor T_t since $T_t \preceq_{\text{lex}} T_x \prec_{\text{lex}} T_x[j..]$ for every $j \in [2..|T_x|]$ and every $x \in [1..t]$. Therefore, T_t is the smallest string among all conjugates of all Lyndon factors. Hence, $\text{BBWT}[1]$ is the last character of T_t , which is $T[n]$. ◀

The BWT of T , called in the following BWT, is the BBWT of $\$T$ for a delimiter $\$ \notin \Sigma$ smaller than all other characters in T (cf. [15, Lemma 12] since $\$T$ is a Lyndon word). Originally, the BWT is defined by reading the last characters of all cyclic rotations of T (without $\$$) sorted lexicographically [6]. Here, we call the resulting string BWT[◦]. BWT[◦] is equivalent to BWT if T contains the aforementioned unique delimiter $\$$. We further write BWT_P (and analogously BBWT_P or BWT_P°) to denote the BWT of P for a string P .

Since BWT (and analogously BBWT or BWT[◦]) is a permutation of T , it is natural to identify each entry of BWT with a text position: By construction $\text{BWT}[i] = T[j]$, where $T[j+1..]$ is the i -th lexicographically smallest suffix, i.e., $\text{SA}[i] = j+1$, where SA is the suffix array of T . A similar relation is given between BBWT and the circular suffix array [19, 2],

which is uniquely defined up to positions of equal Lyndon factors. Figure 1 gives an example for all three variants. In what follows, we review means to simulate a linear traversal of the text in forward or backward manner by BWT, and then translate this result to BBWT.

3.4 Backward and Forward Steps

Having the location of $T[i]$ in BWT, we can compute $T[i+1]$ (i.e., $T[1]$ for $i=1$) and $T[i-1]$ (i.e., $T[n]$ for $i=0$) by rank and select queries. To move from $T[i]$ to $T[i+1]$, which we call a *forward step*, we can use the FL mapping:

$$\text{FL}[i] := \text{BWT.select}_{\text{F}[i]}(\text{F.rank}_{\text{F}[i]}(i)), \quad (1)$$

where $\text{F}[i]$ is the i -th lexicographically smallest character in BWT. To move from $T[i]$ to $T[i-1]$, we can use the *backward step* of the FM-index [11], which is also called LF mapping, and is defined as follows:

$$\text{LF}[i] := \text{F.select}_{\text{BWT}[i]}(\text{BWT.rank}_{\text{BWT}[i]}(i)) = C[\text{BWT}[i]] + \text{BWT.rank}_{\text{BWT}[i]}(i), \quad (2)$$

where $C[c]$ is the number of occurrences of those characters in BWT that are smaller than c (for each character $c \in [1 \dots \sigma]$). We observe from the second equation of (2) that there is no need for F when having C . This is important, as we can compute $C[i]$ in $\mathcal{O}(n)$ time only having BWT available. Hence, we can compute $\text{LF}[i]$ in $\mathcal{O}(n)$ time in-place. However, the same trick does not work with $\text{FL}[i] = \text{BWT.select}_{\text{F}[i]}(i - C[\text{F}[i]])$. To lookup $\text{F}[i]$, we can use the selection algorithm of Chan et al. [7] using BWT and $\mathcal{O}(\lg n)$ bits as working space (the algorithm restores BWT after execution) to compute an entry of F in $\mathcal{O}(n)$ time.

In summary, we can compute both $\text{FL}[i]$ and $\text{LF}[i]$ in-place in $\mathcal{O}(n)$ time. The algorithm of Crochemore et al. [9, Thm. 2] inverting BWT in-place in $\mathcal{O}(n^{2+\epsilon})$ time uses the result of Munro and Raman [26] computing $\text{F}[i]$ in $\mathcal{O}(n^{1+\epsilon})$ time for a constant $\epsilon > 0$ in the comparison model. As noted by Chan et al. [7, Sect. 1], the time bound for the inversion can be improved to $\mathcal{O}(n^2)$ time in the RAM model under the assumption that BWT is rewritable.

If we allow more space, it is still advantageous to favor storing C instead of F if $\sigma = o(n)$ because storing F and C in their plain forms take $n \lg \sigma$ bits and $\sigma \lg n$ bits, respectively. To compute $\text{FL}[i]$, we can also compute FL without F by endowing C with a predecessor data structure (which we do in Sect. 4.3).

Finally, we also need LF and FL on BBWT for our conversion algorithms. We can define LF and FL similarly for BBWT with the following peculiarity:

3.5 Steps in the Bijective BWT

The major difference to the BWT is that the LF mapping of the BBWT can contain multiple cycles, meaning that LF (or FL) recursively applied to a BBWT position would result in searching circular (more precisely, the search stays within the same Lyndon factor). This is because BBWT is the extended BWT [23, Thm. 20 and Remark 12] applied to the multiset of Lyndon factors $\{T_1, \dots, T_t\}$. This fact was exploited for circular pattern matching [19], but is not of interest here.

Instead, we follow the analysis of the so-called *rewindings* [3, Sect. 3]: Remembering that we store the last character of all conjugates of all Lyndon factors in BBWT, we observe that the entries in BBWT representing the Lyndon factors (i.e., the last characters of the Lyndon factors) are in sorted order (starting with $T_t[[T_t]]$ and ending with $T_1[[T_1]]$). That is because the lexicographic order and the \prec_ω order are the same for Lyndon words [5,

Thm. 8]. Applying the backward step at such an entry results in a rewinding, i.e., we can move from the beginning of a Lyndon factor T_x (represented by $T_x[|T_x|]$ in BBWT) to the end of T_x (represented by $T_x[1]$ in BBWT) with one backward step. We use this property with Lemma 2 in the following sections to read the Lyndon factors from T individually in the order T_t, \dots, T_1 .

4 Run-Length Compressed Conversions

We now consider BWT and BBWT represented as run-length compressed strings taking $\mathcal{O}(r_{\text{BWT}} \lg n)$ and $\mathcal{O}(r_{\text{BBWT}} \lg n)$ bits of space, where r_{BWT} and r_{BBWT} are the number of character runs in BWT and BBWT, respectively. For $r := \max(r_{\text{BWT}}, r_{\text{BBWT}})$, the goal of this section is the following:

► **Theorem 3.** *We can convert RLBBWT to RLBWT in $\mathcal{O}(n \lg r / \lg \lg r)$ time using $\mathcal{O}(r \lg r)$ bits as working space, or vice versa.*

To prove this theorem, we need a data structure that works in the run-length compressed space while supporting rank and select queries as well as updates more efficiently than the $\mathcal{O}(n)$ time in-place approach described in Sects. 3.4 and 3.5:

4.1 Run-length Compressed Wavelet Trees

Given a run-length compressed string S of uncompressed length n with r character runs, there is an $\mathcal{O}(r \lg n)$ bits representation of S that supports access, rank, select, insertions, and deletions in $\mathcal{O}(\lg r)$ time [30, Lemma 1]. It consists of (1) a dynamic wavelet tree maintaining the starting characters of each character run and (2) a dynamic Fenwick tree maintaining the lengths of the runs. It can be accelerated to $\mathcal{O}(\lg r / \lg \lg r)$ time by using the following representations:

1. The dynamic wavelet tree of Navarro and Nekrich [27] on a text of length r uses $\mathcal{O}(r \lg r)$ bits, and supports both updates and queries in $\mathcal{O}(\lg r / \lg \lg r)$ time.
2. The dynamic Fenwick tree of Bille et al [4, Thm. 2] on r ($\lg n$)-bit numbers uses $\mathcal{O}(r \lg n)$ bits, and supports both updates and queries in constant time if updates are restricted to be in-/decremental.

The obtained time complexity of this data structure directly improves the construction of RLBWT:

► **Corollary 4** ([30, Thm. 2]). *We can construct the RLBWT in $\mathcal{O}(r_{\text{BWT}} \lg n)$ bits of space online on the reversed text in $\mathcal{O}(n \lg r_{\text{BWT}} / \lg \lg r_{\text{BWT}})$ time.*

In the run-length compressed wavelet tree representation, RLBWT and RLBBWT support an update operation and a backward step in $\mathcal{O}(\lg r / \lg \lg r)$ time with $r := \max(r_{\text{BWT}}, r_{\text{BBWT}})$. This helps us to devise the following two conversions:

4.2 From RLBBWT to RLBWT

We aim for directly outputting the characters of T in reversed order since we can then use the algorithm of Cor. 4 building RLBWT online on the reversed text. We start with the first entry of BBWT (corresponding to the last Lyndon factor T_t , i.e., storing $T_t[|T_t|] = T[n]$ according to Lemma 2) and do a backward step until we come back at this first entry (i.e., we have visited all characters of T_t). During that search, we copy the read characters to RLBWT and mark in an array R of length r_{BBWT} at entry i how often we visited the i -th

character run of RLBBWT. Finally, we remove the read cycle of RLBBWT by decreasing the run lengths of RLBBWT by the numbers stored in R . By doing so, we remove the last Lyndon factor T_t from RLBBWT and consequently know that the currently first entry of BBWT must correspond to T_{t-1} . This means that we can apply the algorithm recursively on the remaining RLBBWT to extract and delete the Lyndon factors in reversed order while building RLBBWT in the meantime. By removing T_t , BBWT is still a valid BBWT since BBWT becomes the BBWT of $T' := T_1 \cdots T_{t-1}$ whose Lyndon factors are the same as of T (but without T_t). Note that it is also possible to build RLBBWT in forward order, i.e., computing $\text{RLBBWT}_{T_1 \dots T_x}$ for increasing x by applying the algorithm of Mantaci et al. [24, Fig. 1] while omitting the suffix array construction.

4.3 From RLBBWT to RLBBWT

To build BBWT, we need to be aware of the Lyndon factors of T , which we compute with Lemma 1 by simulating a forward scan on T with FL on BWT. To this end, we store the entries of the C array in a Fusion tree [13] using $\mathcal{O}(\sigma \lg n)$ bits and supporting predecessor search in $\mathcal{O}(\lg \sigma / \lg \lg \sigma) = \mathcal{O}(\lg r / \lg \lg r)$ time.² This time complexity also covers a forward step in RLBBWT by simulating F with the Fusion tree on C . Hence, this fusion tree allows us to apply Lemma 1 computing the Lyndon factorization of T with a multiplicative $\mathcal{O}(\lg r / \lg \lg r)$ time penalty since this algorithm only needs to perform forward traversals. The starting point of such a traversal is the position i with $\text{BWT}[i] = \$$ because $\text{FL}[i]$ returns the first character of T . Whenever we detect a Lyndon factor T_x (starting with $x = 1$), we copy this factor to our dynamic RLBBWT. For that, we always maintain the first and the last position of T_x in memory. Having the last position of T_x , we perform backward steps on RLBBWT until returning at the first position of T_x to read the characters of T_x in reversed order. Then we continue with the algorithm of Lemma 1 at the position after T_x (for recursing on T_{x+1}). Inserting a Lyndon factor into RLBBWT works exactly as sketched by Bonomo et al. [5, Thm. 17] (we review this algorithm in detail in Sect. 5.1).

5 In-Place Conversions

We finally present our in-place conversions that work in quadratic time by computing LF or FL in $\mathcal{O}(n)$ time having only stored either BWT, BBWT, or BWT° . We note that the constructions from the text also work in the comparison model, while inverting a transform or converting two different transforms have a multiplicative $\mathcal{O}(n^\epsilon)$ time penalty as the fastest option to access F in the comparison model uses $\mathcal{O}(n^{1+\epsilon})$ time for a constant $\epsilon > 0$ [26]. We start with the construction and inversion of BWT° (Sects. 5.1 and 5.2), where we show (a) that we can construct BWT° from the text in the same manner as Bonomo et al. [5] construct BBWT, and (b) that the latter construction works also in-place. Next, we show in Sect. 5.3 how to invert BBWT with the BWT inversion algorithm of Crochemore et al. [9, Fig. 3], which allows us to also convert BBWT to BWT with the BWT construction algorithm of the same paper [9, Fig. 2]. Finally, we show a conversion from BWT to BBWT in Sect. 5.4. An overview is given in Table 1.

² We assume that the alphabet Σ is *effective*, i.e., that each character of Σ appears at least once in T . Otherwise, assume that T uses σ' characters. Then we build the static dictionary of Hagerup [18] in $\mathcal{O}(\sigma' \lg \sigma')$ time, supporting access to a character in $\mathcal{O}(\lg \lg \sigma') = \mathcal{O}(\lg \lg r)$ time, assigning each of the σ' characters an integer from $[1 \dots \sigma']$. We further map RLBBWT to the alphabet $[1 \dots \sigma']$, which can be done in $\mathcal{O}(r)$ time by using $\mathcal{O}(r \lg n)$ space for a linear-time integer sorting algorithm.

■ **Table 1** Overview of in-place conversions in focus of Sect. 5 working in quadratic time.

From \ To	T	BWT	BBWT	BWT [◦]	
To T		\	[9, Fig. 3]	Sect. 5.3	Sect. 5.2
To BWT	[9, Fig. 2]		\	Sect. 5.3	
To BBWT	Sect. 5.1	Sect. 5.4		\	
To BWT [◦]	Sect. 5.1				\

5.1 Constructing BWT[◦] and BBWT

We can compute BWT[◦] and BBWT from T with the algorithm of Bonomo et al. [5] computing the extended BWT [23]. The extended BWT is the BWT defined on a set of primitive strings. As stated in Sect. 3.5, the extended BWT coincides with BBWT if this set of primitive strings is the set of Lyndon factors of T [5, Thm. 14]. We briefly describe the algorithm of Bonomo et al. [5] for computing the BBWT (cf. Fig. 2): For each Lyndon factor T_x (starting with $x = 1$), prepend $T_x[|T_x|]$ to BBWT. To insert the remaining characters of the factor T_x , let $p \leftarrow 1$ be the position of the currently inserted character. Then perform, for each $j = |T_x| - 1$ down to 1, a backward step $p \leftarrow \text{LF}[p] + 1$, and insert $T_x[j]$ at $\text{BBWT}[p]$. To understand why this computes BBWT, we observe that the last character of the most recently inserted Lyndon factor T_x is always the first character in $\text{BBWT}_{T_1 \dots T_x}$ according to Lemma 2. By recursively inserting the preceding character at the place returned by a backward step, we precisely insert this character at the position where we would expect it (another backward step from the same position p would then return the inserted character). Using only n backward steps and n insertions, this algorithm works in-place in $\mathcal{O}(n^2)$ time by simulating LF as described in Sect. 3.4.

Consequently, we can build BWT[◦] if T is a Lyndon word since in this case BWT[◦] and BBWT coincide [15, Lemma 12]. That is because sorting the suffixes of T is equivalent to sorting the conjugates of T (if T is a Lyndon word, then its Lyndon factorization consists only of T itself).

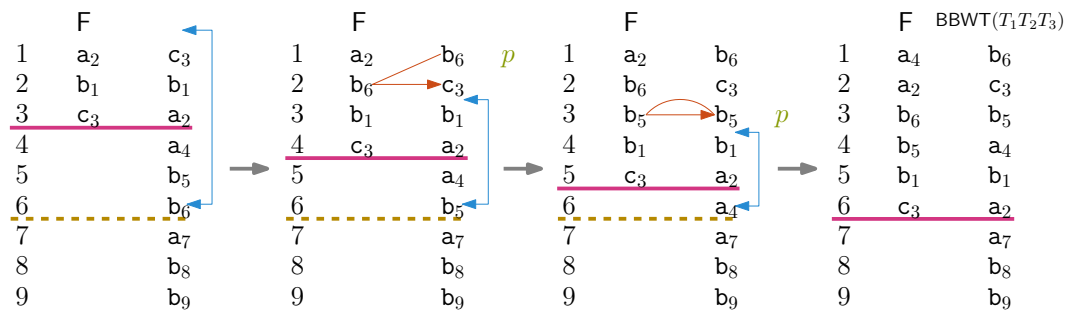
It is easy to generalize this to work for a general string T . First, if T is primitive, then we compute its so-called *Lyndon conjugate*, i.e., a conjugate of T that is a Lyndon word. (The Lyndon conjugate of T is uniquely defined if T is primitive.) We can find the Lyndon conjugate of T in $\mathcal{O}(n)$ time with the following two lemmata:

► **Lemma 5** ([10, Prop. 1.3]). *Given two Lyndon words S and T , ST is a Lyndon word if $S \prec_{\text{lex}} T$.*

► **Lemma 6.** *Given a primitive string T , we can find its Lyndon conjugate in $\mathcal{O}(n)$ time with $\mathcal{O}(\lg n)$ bits of space.*

Proof. We use Lemma 1 to detect the last Lyndon factor T_t of the Lyndon factorization $T_1 \dots T_t$ of T with $\mathcal{O}(\lg n)$ bits of working space. According to Lemma 5, $T_t T_1$ is a Lyndon word since $T_t \prec_{\text{lex}} T_1$, and so is $T_t T_1 \dots T_{t-1}$ a Lyndon word by a recursive argument. Hence, we have found T 's Lyndon conjugate. ◀

Let $\text{conj}_j(T)$ be the Lyndon conjugate of T for $j \in [0 \dots n - 1]$. Since BWT[◦] is identical to $\text{BBWT}_{\text{conj}_j(T)}$, we are done by running the algorithm of Bonomo et al. [5] on $\text{conj}_j(T)$. Finally, if T is not primitive, then there is a primitive string P such that $T = P^k$ for an integer $k \geq 2$. We can compute BWT_P° with the above considerations. For obtaining BWT[◦],



■ **Figure 2** Computing BBWT from our running example $T = \text{bacabbabb}$ in four steps (visualized by four columns separated by three arrows \rightarrow), cf. Sect. 5.1. In each column, the characters from the top to the solid horizontal line (—) form the currently built BBWT. The characters below that up to the dashed horizontal line (---) are under consideration of being merged into BBWT. This dashed line is always before the beginning of the next yet unread Lyndon factor. *First column:* We have already computed the BBWT of $T_1T_2 = \text{bac}$, which is cba . In the following we want to add the next Lyndon factor $T_3 = \text{abb}$ to it. For that, we prepend its last character to the currently constructed BBWT. *Second column:* We move the last character above the dashed line to the position $\text{LF}[p] + 1$ with $p = 1$, and update $p \leftarrow \text{LF}[p] + 1$. We recurse in the *third column*, and have produced the BBWT of $T_1T_2T_3 = \text{bacabb}$ in the *forth column*.

according to [25, Prop. 2], we only need to make each character in BWT_P° to a character run of length k , i.e., if $\text{BWT}_P^\circ[i] = c$, we append c^k to BWT° for increasing $i \in [1 \dots |P|]$ (cf. [15, Thm. 13]). Checking whether T is primitive can be done in $\mathcal{O}(n^2)$ time by checking for each pair of positions their longest common prefix.

5.2 Inverting BWT°

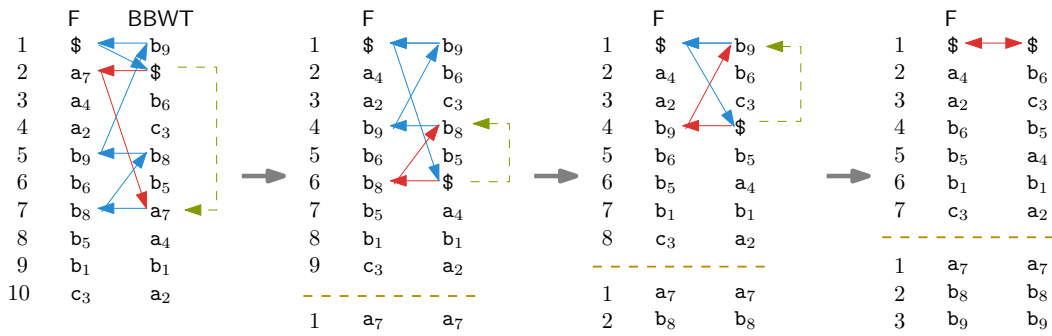
To invert BWT° , we use the techniques of Crochemore et al. [9, Fig. 3] inverting BWT in-place in $\mathcal{O}(n^2)$ time. An invariant is that the BWT entry, whose FL mapping corresponds to the next character to output, is marked with a unique delimiter $\$$. Given that $\text{BWT}[i] = \$$, the algorithm outputs $\text{BWT}[\text{FL}[i]]$, sets $\text{BWT}[\text{FL}[i]] \leftarrow \$$, removes $\text{BWT}[i]$, and recurses until $\$$ is the last character remaining in BWT. By doing so, it restores the text in text order.

To adapt this algorithm for inverting BWT° , we additionally need a pointer p storing the first symbol of the text (since there is no unique delimiter such as $\$$ in general). Given that p points to $\text{BWT}^\circ[i]$, we set $i \leftarrow \text{FL}[i]$ and subsequently output $\text{BWT}^\circ[i]$. From now on, the algorithm works exactly as [9, Fig. 3] if we set $\text{BWT}^\circ[i] \leftarrow \$$ after outputting $\text{BWT}^\circ[i]$. More involving is inverting BBWT or converting BBWT to BWT, which we tackle next.

5.3 Inverting BBWT

Similarly to Sect. 4.2, we read the Lyndon factors from BBWT in the order T_t, \dots, T_1 , and move each read Lyndon factor directly to a text buffer such that while reading the last Lyndon factor T_x for an $x \in [1 \dots t]$ from $\text{BBWT}_{T_1 \dots T_x}$, we move the characters of T_x to $T_{x+1} \dots T_t$, producing $\text{BBWT}_{T_1 \dots T_{x-1}}$ and $T_x \dots T_t$. This allows us to recurse by reading always the last Lyndon factor T_x stored in $\text{BBWT}_{T_1 \dots T_x}$.

Here, we want to apply the inversion algorithm for BWT° described in Sect. 5.3. For adapting this algorithm to work with BBWT, it suffices to insert $\$$ at $\text{BBWT}[2]$ (cf. Fig. 3). By doing so, we add $\$$ to the cycle of the currently last Lyndon factor T_x stored in BBWT, i.e., we enlarge the Lyndon factor T_x to $\$T_x$. That is because (a) $\text{BBWT}[1]$ corresponds to



■ **Figure 3** Inverting BBWT of our running example $T = \text{bacabbabb}$ (cf. Sect. 5.3). *First Column:* We prepend the $\$$ delimiter to the last Lyndon factor T_t by inserting $\$$ at $\text{BBWT}[2]$. A **forward step** symbolized by the dashed arrow (\dashrightarrow) leads us from $\$$ to the first character of T_t . *Second Column:* We output $\text{BBWT}[6] = T_t[1] = T[7]$, remove $\$$ and update $\text{BBWT}[6] \leftarrow \$$. The output is appended to the string shown below the dashed horizontal line (\dashrightarrow). We continue with a **forward step** to access $\text{BBWT}[4] = T_t[2] = T[8]$, and recurse in the *third column*. *Forth Column:* Since a **forward step** returns $\$$, we know that we have successfully extracted $T_t = \text{abb}$.

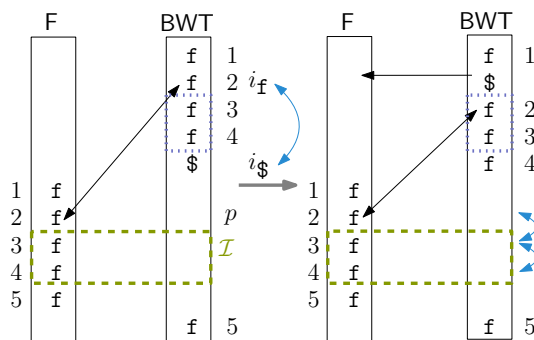
the last character $T_x[|T_x|]$ of T_x (cf. Lemma 2), and after inserting $\$$, $F[1] = \$$, $F[2] = T_x[1]$, hence $\text{FL}[1] = 2$ (a forward step on the last character of T_x gives $\$$) and $\text{FL}[2]$ gives the position in BBWT corresponding to $T_x[1]$. Moreover, inserting $\$$ makes BBWT the BBWT of $T' := T_1 \cdots T_{x_1} \T_x , where $\$T_x$ is the last Lyndon factor of T' . We now use the property that $\$T_x[i \dots |T_x|]$ is a Lyndon word for each $i \in [1 \dots |T_x|]$, allowing us to perform the inversion steps of Crochemore et al. [9, Fig. 3] on BBWT. By doing so, we can remove the entry of BBWT corresponding to $\text{conj}_j(T_x)$ for increasing $j \in [0 \dots |T_x| - 1]$ and prepend the extracted characters to the text buffer storing $T_{x+1} \cdots T_t$ within our working space while keeping BBWT a valid BBWT.

Instead of inverting BBWT, we can convert BBWT to BWT in-place by running the in-place BWT construction algorithm of Crochemore et al. [9, Fig. 2] on the text buffer after the extraction of each Lyndon factor. Unfortunately, this works not character-wise, but needs a Lyndon factor to be fully extracted before inserting its characters into BWT. Interestingly, for the other direction (from BWT to BBWT), we can propose a different kind of conversion that works directly on BWT without decoding it.

5.4 From BWT to BBWT on the Fly

Like in Sect. 4.3, we process the Lyndon factors of T individually to compute BBWT by scanning BWT in text order to simulate Lemma 1. Suppose that we have built BWT on $T\$ \neq \$$ with $\$$ being the $(t + 1)$ -th Lyndon factor of $T\$$, and suppose that we have detected the first Lyndon factor T_1 . Let \mathbf{f} denote the last character of T_1 . Further let $i_{\mathbf{f}}$ and $i_{\$}$ be the position of the last character of T_1 and the last character of T , respectively, such that $\text{BWT}[i_{\mathbf{f}}] = \mathbf{f}$ and $\text{BWT}[i_{\$}] = \$$. Let $p := \text{LF}[i_{\mathbf{f}}]$ such that $F[p] = \mathbf{f}$ and $\text{BWT}[p] = T_1[|T_1| - 1]$ if $|T_1| > 1$ or $\text{BWT}[p] = \$$ otherwise. Since T_1 and T_2 are Lyndon factors, $T_1 \succeq_{\text{lex}} T_2$. Consequently, the suffix $T[\mathbf{b}(T_2) \dots]$ (the context of $\text{BWT}[i_{\mathbf{f}}]$) is lexicographically smaller than the suffix $T[\mathbf{b}(T_1) \dots]$ (the context of $\text{BWT}[i_{\$}]$), i.e., $i_{\mathbf{f}} < i_{\$}$. Figure 4 gives an overview of the introduced setting.

Our aim is to change BWT such that a forward or backward step within the characters belonging to T_1 always results in a cycle. Informally, we want to cut T_1 out of BWT, which additionally allows us to recursively continue with the FL mapping to find the end of the



■ **Figure 4** Setting of Sect. 5.4 with focus on forming a cycle for a Lyndon factor ending with \mathbf{f} in BWT. *Left:* We exchange $\text{BWT}[i_{\mathbf{f}}]$ with $\text{BWT}[i_{\mathbf{\$}}]$ with the aim to form a cycle. *Right:* To obtain this cycle we additionally need to swap $\text{BWT}[p]$ with the elements of the dashed rectangle (--) corresponding to the interval \mathcal{I} having the same height as the dotted rectangle (···) covering $\text{BWT}[i_{\mathbf{f}} + 1 \dots i_{\mathbf{\$}} - 1]$.

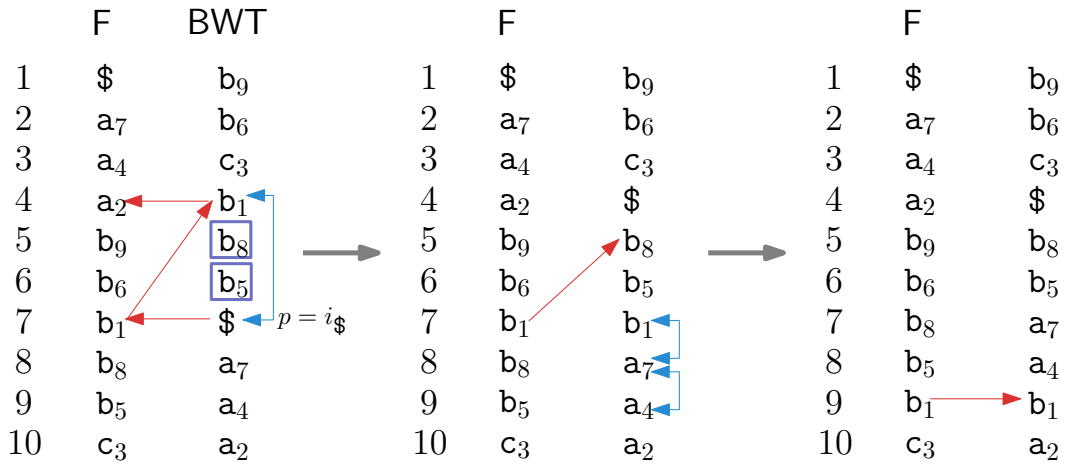
next Lyndon factor T_2 . For that, we exchange $\text{BWT}[i_{\mathbf{\$}}]$ with $\text{BWT}[i_{\mathbf{f}}]$ (cf. Fig. 5). Then the character $T[\mathbf{e}(T_1) + 1]$ (i.e., the first character of T_2) becomes the next character of $\mathbf{\$}$ in terms of the forward step ($\text{BWT}[\text{FL}[i_{\mathbf{f}}]] = T[\mathbf{b}(T_2)]$), while a backwards search on the first character of T_1 yields T_1 's last character (LF returns $i_{\mathbf{\$}}$, but now $\text{BWT}[i_{\mathbf{\$}}] = T_1[\lfloor T_1 \rfloor] = \mathbf{f}$). This is sufficient as long as $\text{BWT}[i] \neq \mathbf{f}$ for every $i \in (i_{\mathbf{f}} \dots i_{\mathbf{\$}}]$. Otherwise, it can happen that we change the mapping from the i -th \mathbf{f} of F to the i -th \mathbf{f} of BWT (or vice versa) unintentionally. In such a case, we swap some entries in BWT within the \mathbf{f} interval of F . In detail, we conduct the exchange ($\text{BWT}[i_{\mathbf{\$}}]$ with $\text{BWT}[i_{\mathbf{f}}]$), but continue with swapping $\text{BWT}[i]$ and $\text{BWT}[i + 1]$ unless $\text{BWT}[\text{FL}[i]]$ becomes that \mathbf{f} that corresponds to $T_1[\lfloor T_1 \rfloor]$ for increasing i starting with $i = p$ until $F[i] \neq \mathbf{f}$ or $\text{LF}[i] \notin [i_{\mathbf{f}} \dots i_{\mathbf{\$}}]$. This may not be sufficient if the characters we swap are identical (cf. Fig. 6). In such a case, we recurse on the $T_1[\lfloor T_1 \rfloor - 1]$ interval of F .

Instead of checking whether we have created a cycle after each swap, we want to compute the exact number of swaps needed for this task. For that we note that exchanging $\text{BWT}[i_{\mathbf{\$}}]$ with $\text{BWT}[i_{\mathbf{f}}]$ decrements the values of $\text{BWT}.\text{rank}_{\mathbf{f}}(j)$ for every $j \in [i_{\mathbf{f}} \dots i_{\mathbf{\$}}]$ by one. In particular, $\text{BWT}.\text{select}_{\mathbf{f}}$ changes for those \mathbf{f} 's in BWT that are between $i_{\mathbf{f}}$ and $i_{\mathbf{\$}}$. Hence, the number of swaps m is the number of positions $k \in [i_{\mathbf{f}} + 1 \dots i_{\mathbf{\$}} - 1]$ with $\text{BWT}[k] = \mathbf{f}$. The swaps are performed within the range \mathcal{I} starting with $p + 1$ and covering all positions i with $\text{LF}[i] \in [i_{\mathbf{f}} \dots i_{\mathbf{\$}}]$ and $F[i] = \mathbf{f}$ since \mathcal{I} covers all entries whose mapping has changed. However, if $\text{BWT}[p \dots]$ starts with a character run of $T[\mathbf{e}(T_1) - 1]$ (or of $T[\mathbf{b}(T_1)]$ if $|T_1| = 1$)³, swapping the identical characters does not change BWT, and therefore has no effect of changing LF. Instead, we search the end of this run within \mathcal{I} to swap the first entry i below this run with the first entry of this run, and recurse on swapping entry i with entries below of it.

Correctness. To see why the swaps restore the LF mapping for T_1 and the remaining part of the text $T_2 \dots T_t$, we examine those substrings of T that we might no longer find with the LF mapping after exchanging $\text{BWT}[i_{\mathbf{\$}}]$ with $\text{BWT}[i_{\mathbf{f}}]$.

In detail, we examine each substring $S_j := x_j y_j \mathbf{f} \in \Sigma^3$ with $j \in [1 \dots m]$ that is represented in BWT (before changing it) with $\text{BWT}[p + j] = y_j$, $\text{BWT}[\text{LF}[p + j]] = x_j$, $\text{BWT}[\text{FL}[p + j]] = \mathbf{f}$, and $i_j := \text{FL}[p + j] \in [i_{\mathbf{f}} + 1 \dots i_{\mathbf{\$}} - 1]$. Due to the LF-mapping, $\text{BWT}.\text{select}_{\mathbf{f}}(\text{BWT}.\text{rank}_{\mathbf{f}}(i_{\mathbf{f}}) + j) =$

³ For $|T_1| = 1$, $p = i_{\mathbf{\$}}$, and hence, $\text{BWT}[p]$ was $\mathbf{\$}$ but now is $\mathbf{f} = T_1[\lfloor T_1 \rfloor] = T_1[1]$.



■ **Figure 5** Computing BBWT from BWT (cf. Sect. 5.4) of our running example $T = \text{bacabbabb}\$$. In the *left* column, we find the first Lyndon factor $T_1 = \mathbf{b}$ of T by **forward steps** with FL. Since $|T_1| = 1, p = i_\$$. We obtain the *middle* column by **exchanging** $\text{BWT}[4]$ with $\text{BWT}[7] = \$$. Since there are two \mathbf{b} 's between \mathbf{b} at $\text{BWT}[4]$ and $\$$ in the *left* column, we need to **swap** $\text{BWT}[p]$ with the two elements below of it in the *middle* column. This gives a cycle in the *right* column. We can recurse since the FL mapping of $\$$ now yields the second character of T .

i_j , meaning that $\text{BWT}[i_j]$ is the j -th \mathbf{f} in $\text{BBWT}[i_\mathbf{f} + 1 .. i_\$ - 1]$, which stores m \mathbf{f} 's. After exchanging $\text{BWT}[i_\$]$ with $\text{BWT}[i_\mathbf{f}]$, $\text{FL}[p + j]$ becomes i_{j+1} for $j \in [0 .. m]$ with $i_{m+1} := i_\$$. However, for all $i > p + m$, $\text{FL}[i]$ did not change. Hence, we only have to focus on the range $\mathcal{I} = [p + 1 .. p + m]$.

First, suppose that $y_1 = \text{BWT}[p + 1] \neq \text{BWT}[p]$. If we swap $\text{BWT}[p]$ with $\text{BWT}[p + 1]$, then $\text{LF}[p]$ is still i_1 , but $\text{BWT}[\text{LF}[p]]$ becomes x_1 such that we have fixed the substring $x_1 y_1 \mathbf{f}$. This also works in a more general setting: If $y_j = \text{BWT}[p + j] \neq \text{BWT}[p]$ for every $j \in [1 .. m]$, we can perform m swaps like above for all m entries in $\text{BWT}[\mathcal{I}]$ to fix all substrings S_j .

Now suppose that $y_j = \text{BWT}[p + j] = \text{BWT}[p]$ for $j \in [1 .. \ell]$ with the largest possible $\ell \in [2 .. m]$. Let $k > p + \ell$ be the first entry with $\text{BWT}[k] \neq \text{BWT}[p]$. First, suppose that $k \in \mathcal{I}$. Then $\text{F}[k] = \mathbf{f}$, and swapping $\text{BWT}[k]$ with $\text{BWT}[p]$ restores the LF mapping for the substrings S_j with $j \in [1 .. \ell]$ since this swap decrements $\text{BWT.rank}_{y_j}[p + j]$ by one for every $j \in [1 .. \ell]$. We recurse on swapping $\text{BWT}[k]$ with the following BBWT entries in \mathcal{I} until all m substrings got restored. Finally, if $k \geq p + m$, then all y_i are equal such that we can find the x_i in BWT consecutively stored at positions with an F value of y_i . Thus, we can apply the swaps there recursively.

Time Complexity. Fixing a Lyndon factor T_x , we spend $\mathcal{O}(|\mathcal{I}|)$ time for the swaps in $\text{BWT}[\mathcal{I}]$, and perform the swaps recursively at most $|T_x|$ times, where we need additionally $\mathcal{O}(n)$ time per recursion step for computing $\text{LF}[p]$, summing up to $\mathcal{O}(|T_x|(|\mathcal{I}| + n)) = \mathcal{O}(n|T_x|)$ time. Since $\sum_{x=1}^t |T_x| = n$, we yield $\mathcal{O}(n^2)$ total time.

6 Open Problems

Our algorithm of Sect. 5.3 converts BBWT to BWT, Lyndon factor by Lyndon factor. It would be interesting to find another conversion that works character-wise. Here, our inversion algorithm extracts a Lyndon factor in text order from BBWT, while the used BWT construction algorithm parses the text in reverse text order.

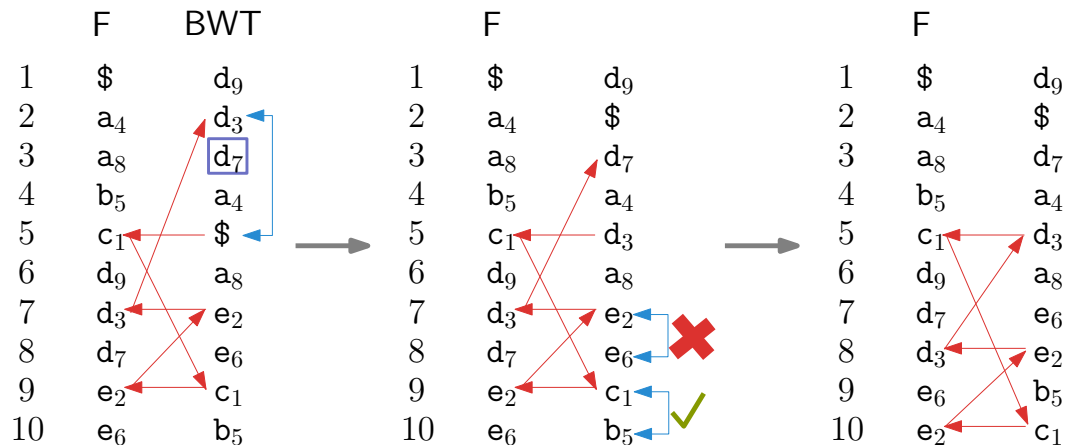


Figure 6 Special case for computing BWT from BBWT (cf. Sect. 5.4) with the different example string $T\$:= cedabedad\$$ having $T_1 = ced$ as its first Lyndon factor. *Left column:* We find the first Lyndon factor $T_1 = ced$ of T by forward steps with FL. Its last character is stored at $BWT[2]$. By exchanging $\$$ with the last character of T_1 in BWT, we obtain the middle column. *Middle column:* The LF mapping for the third d in F becomes invalid. However, there is only a character run of $T_1[|T_1| - 1] = e$ in BWT of the $T_1[|T_1|] = d$ interval $[7..8]$ in F starting with $p = 7$. So we recurse on $LF[p]$ to find characters different from $T_1[|T_1| - 2] = c$ to swap in the respective $T_1[|T_1| - 1] = e$ interval $[9..10]$. *Right Column:* We have created a cycle with the characters of the first Lyndon factor. A forward step at $\$$ gives the first character of the next Lyndon factor.

Crochemore et al. [9, Sect. 4] proposed a space and time trade-off algorithm based on their in-place techniques computing or inverting BWT. We are positive that it should be possible to adapt their techniques for computing or inverting BBWT or BWT° with a trade-off parameter.

From the combinatorial perspective, we question whether the number of distinct Lyndon words of T is bounded by the runs in BBWT. If we can affirm this question, it would be possible to adapt the BBWT based index data structure [3] for RLBBWT using $\mathcal{O}(r_{BBWT} \lg n)$ bits of space because this solution needs a bit vector with rank and select support marking the positions in BBWT corresponding to the distinct Lyndon factors. If this number is at most the number of runs r_{BBWT} , then we can store this bit vector entropy-compressed in $\mathcal{O}(r \lg n)$ bits when $r_{BBWT} = o(n)$ since $nH_0(r) = n \lg(n/(n-r)) + r \lg((n-r)/r) \leq n \lg r \Leftrightarrow r \lg((n-r)/r) \leq n \lg(r(n-r)/n)$ for $r = r_{BBWT}$.

Speaking of RLBBWT, we wonder whether we can construct RLBBWT online in run-length compressed space similar to Cor. 4. With the run-length compressed wavelet tree, the algorithm of Bonomo et al. [5, Thm. 17] works in $\mathcal{O}(n \lg r_{BBWT} / \lg \lg r_{BBWT})$ time with $\max_{x \in [1..t]} |T_x| + \mathcal{O}(r_{BBWT} \lg n)$ bits of space by reading each Lyndon factor of the text individually.

References

- 1 Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform.: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.
- 2 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the bijective BWT. *ArXiv 1911.06985*, 2019.
- 3 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPICs*, pages 17:1–17:14, 2019.

- 4 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.
- 5 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014.
- 6 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
- 7 Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the “restore” model. *ACM Trans. Algorithms*, 14(2):11:1–11:18, 2018.
- 8 Kuo Tsai Chen, Ralph H. Fox, and Roger C. Lyndon. Free differential calculus, IV. The quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.
- 9 Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau. Computing the Burrows-Wheeler transform in place and in small space. *J. Discrete Algorithms*, 32:44–52, 2015.
- 10 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
- 11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.
- 12 Paolo Ferragina, Giovanni Manzini, and S. Muthu Muthukrishnan. *The Burrows-Wheeler Transform: Ten Years Later*. DIMACS, 2004.
- 13 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.
- 14 Travis Gagie, Giovanni Manzini, Gonzalo Navarro, and Jens Stoye. 25 Years of the Burrows-Wheeler Transform (Dagstuhl Seminar 19241). *Dagstuhl Reports*, 9(6):55–68, 2019.
- 15 Raffaele Giancarlo, Antonio Restivo, and Marinella Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387(3):236–248, 2007.
- 16 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012.
- 17 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
- 18 Torben Hagerup. Fast deterministic construction of static dictionaries. In *Proc. SODA*, pages 414–418, 1999.
- 19 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient algorithm for circular Burrows-Wheeler transform. In *Proc. CPM*, volume 7354 of *LNCS*, pages 257–268, 2012.
- 20 R. C. Lyndon. On Burnside’s problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954.
- 21 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.
- 22 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- 23 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.
- 24 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014.
- 25 Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows-Wheeler transform and Sturmian words. *Inf. Process. Lett.*, 86(5):241–246, 2003.
- 26 J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996.

- 27 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.
- 28 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.
- 29 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages 409–419, 2017.
- 30 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

Genomic Problems Involving Copy Number Profiles: Complexity and Algorithms

Manuel Lafond

Department of Computer Science, Université de Sherbrooke, Quebec J1K 2R1, Canada
manuel.lafond@usherbrooke.ca

Binhai Zhu

Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA
bhzh@montana.edu

Peng Zou

Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA
peng.zou@student.montana.edu

Abstract

Recently, due to the genomic sequence analysis in several types of cancer, genomic data based on *copy number profiles* (CNP for short) are getting more and more popular. A CNP is a vector where each component is a non-negative integer representing the number of copies of a specific segment of interest. The motivation is that in the late stage of certain types of cancer, the genomes are progressing rapidly by segmental duplications and deletions, and hence obtaining the exact sequences becomes difficult. Instead, the number of copies of important segments can be predicted from expression analysis and carries important biological information. Therefore, significant research has recently been devoted to the analysis of genomic data represented as CNP's.

In this paper, we present two streams of results. The first is the negative results on two open problems regarding the computational complexity of the Minimum Copy Number Generation (MCNG) problem posed by Qingge et al. in 2018. The *Minimum Copy Number Generation* (MCNG) is defined as follows: given a string S in which each character represents a gene or segment, and a CNP C , compute a string T from S , with the minimum number of segmental duplications and deletions, such that $cnp(T) = C$. It was shown by Qingge et al. that the problem is NP-hard if the duplications are tandem and they left the open question of whether the problem remains NP-hard if arbitrary duplications and/or deletions are used. We answer this question affirmatively in this paper; in fact, we prove that it is NP-hard to even obtain a constant factor approximation. This is achieved through a general-purpose lemma on set-cover reductions that require an exact cover in one direction, but not the other, which might be of independent interest. We also prove that the corresponding parameterized version is W[1]-hard, answering another open question by Qingge et al.

The other result is positive and is based on a new (and more general) problem regarding CNP's. The *Copy Number Profile Conforming* (CNPC) problem is formally defined as follows: given two CNP's C_1 and C_2 , compute two strings S_1 and S_2 with $cnp(S_1) = C_1$ and $cnp(S_2) = C_2$ such that the distance between S_1 and S_2 , $d(S_1, S_2)$, is minimized. Here, $d(S_1, S_2)$ is a very general term, which means it could be any genome rearrangement distance (like reversal, transposition, and tandem duplication, etc). We make the first step by showing that if $d(S_1, S_2)$ is measured by the breakpoint distance then the problem is polynomially solvable. We expect that this will trigger some related research along the line in the near future.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Computational genomics, cancer genomics, copy number profiles, NP-hardness, approximation algorithms, FPT algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.22

Related Version A full version of this paper is available at <https://arxiv.org/abs/2002.04778>.

Funding *Manuel Lafond*: ML was supported by NSERC of Canada, and BZ was partially supported by NNSF of China under project 61628207.



© Manuel Lafond, Binhai Zhu, and Peng Zou;
licensed under Creative Commons License CC-BY
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 22; pp. 22:1–22:15

Leibniz International Proceedings in Informatics



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

1 Introduction

In cancer genomics research, intra-tumor genetic heterogeneity is one of the central problems [15, 16, 21]. Understanding the origins of cancer cell diversity could help cancer prognostics [4, 14] and also help explain drug resistance [3, 6]. It is known for some types of cancers, such as high-grade serous ovarian cancer (HGSOC), that heterogeneity is mainly acquired through genome rearrangements and endoreduplications, the replication of the genome without the usual mitosis reproduction cycle. These result in aberrant *copy number profiles (CNPs)* – nonnegative integer vectors representing the numbers of genes occurring in a genome [17]. To understand how the cancer progresses, an evolutionary tree is certainly desirable, and inferring such a tree based on these genomic data becomes a new problem. In [20], Schwarz et al. proposed a way to construct a phylogenetic tree directly from integer copy number profiles, the underlying problem being to convert CNPs into one another using the minimum number of duplications/deletions [22]. This was recently followed with several other distances measures between CNPs that can be used to reconstruct cancer phylogenies [26, 9, 5, 19, 25].

In [8], a more complex distance computation was used as a subroutine to compute an ancestor profile given a set of k profiles. The problem was shown to be NP-hard, though an ILP formulation was given. In fact, Chowdhury *et al.* considered copy number changes at different levels, from single gene, single chromosome to whole genome, to enhance the tumor phylogeny reconstruction [2]. In [18], another fundamental problem was proposed. The motivation is that in the early stages of cancer, when large numbers of endoreduplications are still rare, genome sequencing is still possible. However, in the later stage we might only be able to obtain genomic data in the form of CNPs. This leads to the problem of comparing a sequenced genome with a genome with only copy-number information.

Given a genome G represented as a string and a copy number profile \vec{c} , the *Minimum Copy Number Generation (MCNG)* problem asks for the minimum number of deletions and duplications needed to transform G into any genome in which each character occurs as many times as specified by \vec{c} . Qingge et al. proved that the problem is NP-hard when the duplications are restricted to be tandem and posed several open questions: (1) Is the problem NP-hard when the duplications are arbitrary and/or deletions are allowed? (2) Does the problem admit a decent approximation? (3) Is the problem fixed-parameter tractable (FPT)? In this paper, we answer all these three open questions. We show that MCNG is NP-hard to approximate within any constant factor, and that it is W[1]-hard when parameterized by the solution size. The inapproximability follows from a new general-purpose lemma on set-cover reductions that require an exact cover in one direction, but not the other. The W[1]-hardness uses a new set-cover variant in which every optimal solution is an exact cover. These set-cover extensions can make reductions easier, and may be of independent interest.

We also consider a new fundamental problem called *Copy Number Profile Conforming (CNPC)*, which is defined as follows. Given two CNP's \vec{c}_1 and \vec{c}_2 , compute two strings/genomes S_1 and S_2 with $cnp(S_1) = \vec{c}_1$ and $cnp(S_2) = \vec{c}_2$ such that the distance between S_1 and S_2 , $d(S_1, S_2)$, is minimized. The distance $d(S_1, S_2)$ could be general, which means it could be any genome rearrangement distance (such as reversal, transposition, and tandem duplication, etc). We make the first step by showing that if $d(S_1, S_2)$ is measured by the breakpoint distance then the problem is polynomially solvable.

2 Preliminaries

A genome G is a string, i.e. a sequence of characters, all of which belong to some alphabet Σ (the characters of G can be interpreted as genes or segments – in this paper we assume the latter, i.e., Σ is a set of segments). We use genome and string interchangeably in this paper,

when the context is clear. A *substring* of G is a sequence of contiguous characters that occur in G , and a *subsequence* is a string that can be obtained from G by deleting some characters. We write $G[p]$ to denote the character at position p of G (the first position being 1), and we write $G[i..j]$ for the substring of G from positions i to j , inclusively. For $s \in \Sigma$, we write $G - s$ to denote the subsequence of G obtained by removing all occurrences of s .

We represent an alphabet as an ordered list $\Sigma = (s_1, s_2, \dots, s_m)$ of distinct characters. Slightly abusing notation, we may write $s \in \Sigma$ if s is a member of this list. We write $n_s(G)$ to denote the number of occurrences of $s \in \Sigma$ in a genome G . A *Copy-Number Profile* (or *CNP*) on Σ is a vector $\vec{c} = \langle c_1, \dots, c_{|\Sigma|} \rangle$ that associates each character s_i of the alphabet with a non-negative integer $c_i \in \mathbb{N}^1$; formally,

$$cnp(G) = \langle n_{s_1}(G), n_{s_2}(G), \dots, n_{s_m}(G) \rangle.$$

We may write $\vec{c}(s)$ to denote the number associated with $s \in \Sigma$ in \vec{c} . We write $\vec{c} - s$ to denote the CNP obtained from \vec{c} by setting $\vec{c}(s) = 0$. An example, if $\Sigma = (a, b, c)$ and $G = abbcbcca$, then $cnp(G) = \langle 2, 4, 3 \rangle$ and $\vec{c}(a) = 2$.

Deletions and duplications on strings

We now describe the two string events of *deletion* and *duplication*. Both are illustrated in Figure 1.

Sequence	Operations
$G_1 = abc \cdot \underline{eab} \cdot cab$	$del(5, 7)$
$G_2 = a \cdot \underline{bbcc} \cdot ab$	$dup(2, 5, 6)$
$G_3 = abbcca \cdot \underline{bbcc} \cdot b$	

■ **Figure 1** Three strings (or toy genomes), G_1, G_2 and G_3 . From G_1 to G_2 , a deletion is applied to $G_1[5..7]$. From G_2 to G_3 , a duplication is applied to $G_2[2..5]$, with the copy inserted after position 6.

Given a genome G , a *deletion* on G takes a substring of G and removes it. Deletions are denoted by a pair (i, j) of the positions of the substring to remove. Applying deletion (i, j) to G transforms G into $G[1..i-1]G[j+1..n]$.

A *duplication* on G takes a substring of G , copies it and inserts the copy anywhere in G , except inside the copied substring. A duplication is defined by a triple (i, j, p) where $G[i..j]$ is the string to duplicate and $p \in \{0, 1, \dots, i-1, j, \dots, n\}$ is the position *after* which we insert (inserting after 0 prepends the copied substring to G). Applying duplication (i, j, p) to G transforms G into $G[1..p]G[i..j]G[p+1..n]$.

An *event* is either a deletion or a duplication. If G is a genome and e is an event, we write $G\langle e \rangle$ to denote the genome obtained by applying e on G . Given a sequence $E = (e_1, \dots, e_k)$ of events, we define $G\langle E \rangle = G\langle e_1 \rangle\langle e_2 \rangle \dots \langle e_k \rangle$ as the genome obtained by successively applying the events of E to G . We may also write $G\langle e_1, \dots, e_k \rangle$ instead of $G\langle (e_1, \dots, e_k) \rangle$.

The most natural application of the above events is to compare genomes.

► **Definition 1.** Let G and G' be two strings over alphabet Σ . The *Genome-to-Genome distance* between G and G' , denoted $d_{GG}(G, G')$, is the size of the smallest sequence of events E satisfying $G\langle E \rangle = G'$.

¹ Note that in the theory of formal languages, the CNP of a string is called the *Parikh vector*.

Note that d_{GG} has recently been considered in [11, 13], the latter in the case of tandem duplications only. We also define a distance between a genome G and a CNP \vec{c} , which is the minimum number of events to apply to G to obtain a genome with CNP \vec{c} .

► **Definition 2.** Let G be a genome and \vec{c} be a CNP, both over alphabet Σ . The Genome-to-CNP distance between G and \vec{c} , denoted $d_{GCNP}(G, \vec{c})$, is the size of the smallest sequence of events E satisfying $\text{cnp}(G\langle E \rangle) = \vec{c}$.

The above definition leads to the following problem, which was first studied in [18]. The Minimum Copy Number Generation (MCNG) problem:

Instance: a genome G and a CNP \vec{c} over alphabet Σ .

Task: compute $d_{GCNP}(G, \vec{c})$.

Qingge et al. proved that the MCNG problem is NP-hard when all the duplications are restricted to be tandem [18]. In the next section, we prove that this problem is not only NP-hard, but also NP-hard to approximate within any constant factor.

3 Hardness of Approximation for MCNG

In this section, we show that the d_{GCNP} distance is hard to approximate within any constant factor. This result actually holds if only deletions on G are allowed. This restriction makes the proof significantly simpler, so we first analyze the deletions-only case. We then extend this result to deletions *and* duplications.

Both proofs rely on a reduction from SET-COVER. Recall that in SET-COVER, we are given a collection of sets $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ over universe $U = \{u_1, u_2, \dots, u_m\} = \bigcup_{S_i \in \mathcal{S}} S_i$, and we are asked to find a set cover of \mathcal{S} having minimum cardinality (a set cover of \mathcal{S} is a subset $\mathcal{S}^* \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{S}^*} S = U$). If \mathcal{S}' is a set cover in which no two sets intersect, then \mathcal{S}' is called an *exact cover*.

There is one interesting feature (or constraint) of our reduction g , which transforms a SET-COVER instance \mathcal{S} into a MCNG instance $g(\mathcal{S})$. A set cover \mathcal{S}^* only works on $g(\mathcal{S})$ if \mathcal{S}^* is actually an exact cover, and a solution for $g(\mathcal{S})$ can be turned into a set cover for \mathcal{S}^* that is not necessarily exact. Thus we are unable to reduce directly from either SET-COVER nor its exact version. We provide a general-purpose lemma for such situations, and our reductions serve as an example of its usefulness.

The proof relies on a result on t -SET-COVER, the special case of SET-COVER in which every given set contains at most t elements. It is known that for any constant $t \geq 3$, the t -SET-COVER problem is hard to approximate within a factor $\ln t - c \ln \ln t$ for some constant c not depending on t [23].

► **Lemma 3.** Let \mathcal{B} be a minimization problem, and let g be a function that transforms any SET-COVER instance \mathcal{S} into an instance $g(\mathcal{S})$ of \mathcal{B} in polynomial time. Assume that both the following statements hold:

- any exact cover \mathcal{S}^* of \mathcal{S} of cardinality at most k can be transformed in polynomial time into a solution of value at most k for $g(\mathcal{S})$;
- any solution of value at most k for $g(\mathcal{S})$ can be transformed in polynomial time into a set cover of \mathcal{S} of cardinality at most k .

Then unless $P = NP$, there is no constant factor approximation algorithm for \mathcal{B} .

Proof. Suppose for contradiction that \mathcal{B} admits a factor b approximation for some constant b . Choose any constant t such that t -SET-COVER is hard to approximate within factor $\ln t - c \ln \ln t$, and such that $b < \ln t - c \ln \ln t$. Note that t might be exponentially larger than b , but is still a constant.

Now, let \mathcal{S} be an instance of t -SET-COVER over the universe $U = \{u_1, \dots, u_m\}$. Consider the intermediate reduction g' that transforms \mathcal{S} into another t -SET-COVER instance $g'(\mathcal{S}) = \{S' \subseteq S : S \in \mathcal{S}\}$. Since t is a constant, $g(\mathcal{S})$ has $O(|\mathcal{S}|)$ sets and this can be carried out in polynomial time.

Now define $\mathcal{S}' = g'(\mathcal{S})$ and consider the instance $B = g(\mathcal{S}') = g(g'(\mathcal{S}))$. By the assumptions of the lemma, a solution for B of value k yields a set cover \mathcal{S}^* for \mathcal{S}' . Clearly, \mathcal{S}^* can be transformed into a set cover for instance \mathcal{S} : for each $S' \in \mathcal{S}^*$, there exists $S \in \mathcal{S}$ such that $S' \subseteq S$, so we get a set cover for \mathcal{S} by adding this corresponding superset for each $S \in \mathcal{S}^*$. Thus B yields a set cover of \mathcal{S} with at most k sets.

In the other direction, consider a set cover $\mathcal{S}^* = \{S_1, \dots, S_k\}$ of \mathcal{S} with k sets. This easily translates into an *exact* cover of \mathcal{S}' with k sets by taking the collection

$$\{S_1, S_2 \setminus S_1, S_3 \setminus (S_1 \cup S_2), \dots, S_k \setminus \bigcup_{i=1}^{k-1} S_i\}.$$

By the assumptions of the lemma, this exact cover can then be transformed into a solution of value at most k for instance B .

Therefore, \mathcal{S} has a set cover of cardinality at most k if and only if B has a solution of value at most k . By this correspondence, a factor b approximation for \mathcal{B} would provide a factor $b < \ln t - c \ln \ln t$ approximation for t -SET-COVER. ◀

3.1 Constructing genomes and CNPs from SET-COVER instances

All of our hardness results rely on Lemma 3. We need to provide a reduction from SET-COVER to MCNG and prove that both assumptions of the lemma are satisfied.

This reduction is the same for deletions-only and deletions-and-duplications. Given \mathcal{S} and U , we construct a genome G and a CNP \vec{c} as follows (an example is illustrated in Figure 2). The alphabet is $\Sigma = \Sigma_{\mathcal{S}} \cup \Sigma_U$, where $\Sigma_{\mathcal{S}} := \{\langle \beta_{S_i} \rangle : S_i \in \mathcal{S}\}$ and $\Sigma_U := \{\alpha_{u_i} : u_i \in U\}$. Thus, there is one character for each set of \mathcal{S} and each element of U . Here, each $\langle \beta_{S_i} \rangle$ is a character that will serve as a separator between characters to delete. For a set $S_i \in \mathcal{S}$, define the string $q(S_i)$ as any string that contains each character of $\{\alpha_u : u \in S_i\}$ exactly once (in any fixed order, say by their indices). We put

$$G = \langle \beta_{S_1} \rangle q(S_1) \langle \beta_{S_2} \rangle q(S_2) \dots \langle \beta_{S_n} \rangle q(S_n),$$

i.e. G is the concatenation of the strings $\langle \beta_{S_i} \rangle q(S_i)$. As for the CNP \vec{c} , put

- $\vec{c}(\langle \beta_{S_i} \rangle) = 1$ for each $S_i \in \mathcal{S}$;
- $\vec{c}(\alpha_u) = f(u) - 1$ for each $u \in U$, where $f(u) = |\{S_i \in \mathcal{S} : u \in S_i\}|$ is the number of sets from \mathcal{S} that contain u .

Notice that in G , each $\langle \beta_S \rangle$ already has the correct copy-number, whereas each α_u needs exactly one less copy. Our goal is thus to reduce the number of each α_u by 1. This concludes the construction of MCNG instances from SET-COVER instances. We now focus on the hardness of the deletions-only case.

$S_1 = \{1, 2, 3\}$	$S_2 = \{1, 3, 4\}$	$S_3 = \{2, 3, 5\}$
$G = \langle \beta_{S_1} \rangle \alpha_1 \alpha_2 \alpha_3 \langle \beta_{S_2} \rangle \alpha_1 \alpha_3 \alpha_4 \langle \beta_{S_3} \rangle \alpha_2 \alpha_3 \alpha_5$		
$\vec{c}(\alpha_1) = \vec{c}(\alpha_2) = 1 \quad \vec{c}(\alpha_3) = 2 \quad \vec{c}(\alpha_4) = \vec{c}(\alpha_5) = 0$		

■ **Figure 2** An example of our construction, with $\mathcal{S} = \{S_1, S_2, S_3\}$ and $U = \{1, 2, 3, 4, 5\}$.

3.2 Warmup: the deletions-only case

Suppose that we are given a set cover instance \mathcal{S} and U , and let G and \vec{c} be the genome and CNP, respectively, as constructed above.

► **Lemma 4.** *Given an exact cover \mathcal{S}^* for \mathcal{S} of cardinality k , one can obtain a sequence of k deletions transforming G into a genome with CNP \vec{c} .*

Proof. Denote $\mathcal{S}^* = \{S_{i_1}, \dots, S_{i_k}\}$. Consider the sequence of k deletions that deletes the substrings $q(S_{i_1}), \dots, q(S_{i_k})$ (i.e. the sequence first deletes the substring $q(S_{i_1})$, then deletes $q(S_{i_2})$, and so on until $q(S_{i_k})$ is deleted). Since S_{i_1}, \dots, S_{i_k} is an exact cover, this sequence removes exactly one copy of each $\alpha_u \in \Sigma_U$ and does not affect the $\langle \beta_S \rangle$ characters. Thus the k deletions transform G into a genome with the desired CNP \vec{c} . ◀

► **Lemma 5.** *Given a sequence of k deletions transforming G into a genome with CNP \vec{c} , one can obtain a set cover for \mathcal{S} of cardinality at most k .*

Proof. Suppose that the deletion events $E = (e_1, \dots, e_k)$ transform G into a genome G^* with CNP \vec{c} . Note that no e_i deletion is allowed to delete a set-character $\langle \beta_{S_i} \rangle \in \Sigma_{\mathcal{S}}$, as there is only one occurrence of $\langle \beta_{S_i} \rangle$ in G and $\vec{c}(\langle \beta_{S_i} \rangle) = 1$. Thus all deletions remove only α_u characters. In other words, each e_j in E either deletes a substring of G between some $\langle \beta_{S_i} \rangle$ and $\langle \beta_{S_{i+1}} \rangle$ with $1 \leq i < n$, or e_j deletes a substring after $\langle \beta_{S_n} \rangle$. Moreover, exactly one of each α_u occurrences gets deleted from G .

Call $\langle \beta_{S_i} \rangle \in \Sigma_{\mathcal{S}}$ *affected* if there is some event of E that deletes at least one character between $\langle \beta_{S_i} \rangle$ and $\langle \beta_{S_{i+1}} \rangle$ with $1 \leq i < n$, and call $\langle \beta_{S_n} \rangle$ affected if some event of E deletes characters after $\langle \beta_{S_n} \rangle$. Let $\mathcal{S}^* := \{S_i \in \mathcal{S} : \langle \beta_{S_i} \rangle \text{ is affected}\}$. Then $|\mathcal{S}^*| \leq k$, since each deletion affects at most one $\langle \beta_{S_i} \rangle$ and there are k deletion events. Moreover, \mathcal{S}^* must be a set cover, because each $\alpha_u \in \Sigma_U$ has at least one occurrence that gets deleted and thus at least one set containing u that is included in \mathcal{S}^* . This concludes the proof. ◀

We have shown that all the assumptions required by Lemma 3 are satisfied. The inapproximability follows.

► **Theorem 6.** *Assuming $P \neq NP$, there is no polynomial-time constant factor approximation algorithm for MCNG when only deletions are allowed.*

We mention without proof that the reduction can be adaptable to the duplication-only case, by putting $\vec{c}(\alpha_u) = f(u) + 1$ for each $u \in U$.

The real deal: deletions and duplications

We now consider both deletions and duplications. The reduction uses the same construction as in Section 3.1. Thus we assume that we have a SET-COVER instance \mathcal{S} over U , and a corresponding instance of MCNG with genome G and CNP \vec{c} . In that case, we observe that Lemma 4 still holds whether we allow deletion only, or both deletions and duplications. Thus we only need to show that the second assumption of Lemma 3 holds.

Unfortunately, this is not as simple as in the deletions-only case. The problem is that some duplications may copy some α_u and $\langle \beta_{S_i} \rangle$ occurrences, and we lose control over what gets deleted, and over what $\langle \beta_{S_i} \rangle$ each α_u corresponds to (in particular, some $\langle \beta_{S_i} \rangle$ might now get deleted, which did not occur in the deletions-only case).

Nevertheless, the analogous result can be shown. That is, using the above reduction, our goal is to show that, given a sequence of k events (deletions and duplications) transforming G into a genome with CNP \vec{c} , one can obtain a set cover for \mathcal{S} of cardinality at most k .

We need some new notation and intermediate results beforehand. Let $E = (e_1, \dots, e_k)$ be a sequence of events transforming genome G into another genome G' . We would like to distinguish each position of G in order to know which specific character of G is at the origin of a character of G' .

We augment each individual character of G with a unique identifier, which is its position in G . That is, let $G = g_1g_2 \dots g_n$, define a new alphabet $\hat{\Sigma} = (g_1^1, g_2^2, \dots, g_n^n)$ and define the genome $\hat{G} = g_1^1g_2^2 \dots g_n^n$. Here, two characters g_i and g_j may be identical, but g_i^i and g_j^j are two distinct characters. We call $\hat{\Sigma}$ the *augmented alphabet* and \hat{G} the *augmented genome* of G . For instance if $G = abcb$ and $\Sigma = (a, b, c)$, then $\hat{\Sigma} = (a^1, a^2, b^3, c^4, b^5)$ and $\hat{G} = a^1a^2b^3c^4b^5$.

Since G and \hat{G} have the same length, we may apply the sequence E on \hat{G} , resulting in a genome $\hat{G}' := \hat{G}(E)$ on alphabet $\hat{\Sigma}$. Now \hat{G}' may contain some characters of $\hat{\Sigma}$ multiple times owing to duplications, but if we remove the superscript identifier from the characters of \hat{G}' , we obtain G' . The idea is that the identifiers on the characters of \hat{G}' tell us precisely where each character of \hat{G}' “comes from” in \hat{G} (and thus G).

► **Definition 7.** Let G and G' be genomes and let E an event sequence such that $G' = G(E)$. Let \hat{G} be the augmented genome of G and let $\hat{G}[i] = g^i$ be the character at position i .

If there is at least one occurrence of g^i in $\hat{G}(E)$, then position i is called *important with respect to E* . Otherwise, position i is called *unimportant with respect to E* .

Roughly speaking, position i is unimportant if it eventually gets deleted, and any character that was copied from position i from a duplication also gets deleted, as well as a copy of this copy, and so on – in other words, position i has no “descendant” in G' when applying E .

First, we prove some general properties that will be useful. Recall that $G - s$ removes all occurrences of s from G , and $\vec{c} - s$ puts $\vec{c}(s) = 0$.

► **Proposition 8.** Let G be a genome over alphabet Σ , let \vec{c} be a CNP and let $s \in \Sigma$. Then $d_{GCNP}(G - s, \vec{c} - s) \leq d_{GCNP}(G, \vec{c})$.

The next technical lemma states that if a genome alternates between positions to keep and positions to delete n times, then we need n events to remove the unimportant ones.

► **Lemma 9.** Let $\Sigma = X \cup Y$ be an alphabet defined by two disjoint sets $X = \{x_1, \dots, x_n\}$ and Y . Let $G = Y_0x_1Y_1x_2Y_2 \dots x_nY_n$ be a genome on Σ , where for all $i \in [n]$, Y_i is a non-empty string over alphabet Y and Y_0 is a possibly empty string on alphabet Y . Moreover let \vec{c} be a CNP such that $\vec{c}(x_i) = 1$ for all $x_i \in X$ and $\vec{c}(y) = 0$ for all $y \in Y$. Then $d_{GCNP}(G, \vec{c}) \geq n$, with equality when Y_0 is empty.

The proof is surprisingly technical and can be found in the full version. We may now prove the second assumption of Lemma 3.

► **Lemma 10.** Let \mathcal{S} be a SET-COVER instance, and let G and \vec{c} be the corresponding MCNG instance. Given a sequence of k events (deletions and duplications) transforming G into a genome with CNP \vec{c} , one can obtain a set cover for \mathcal{S} of cardinality at most k .

Proof. Suppose that the events $E = (e_1, \dots, e_k)$ transform G into a genome G^* with CNP \vec{c} . We construct a set cover for \mathcal{S} of cardinality k . For a position p with $G[p] = \alpha_u \in \Sigma_U$, define $\text{pred}(p)$ as the first Σ_S character to the left of position p . To be precise, if p' is the largest integer satisfying $G[p'] \in \Sigma_S$ and $p' < p$, then $\text{pred}(p) = G[p']$. Note that since $G[1] = \langle \beta_{S_1} \rangle$, $\text{pred}(p)$ is well-defined. Notice that by construction, if $G[p] = \alpha_u$ and $\langle \beta_S \rangle = \text{pred}(p)$, then $u \in S$. The set of $\text{pred}(p)$ of unimportant positions p will correspond to our set cover, which we now prove by separate claims.

▷ **Claim 11.** For each $u \in U$, there is at least one position p of G such that $G[p] = \alpha_u$ and such that p is unimportant w.r.t. E .

Proof. If we assume this is not the case, then each of the $f(u)$ positions p of G having $G[p] = \alpha_u$ has a descendant in G^* , implying that G^* has at least $f(u)$ copies of α_u and thereby contradicting that G^* complies with $\vec{c}(\alpha_u) = f(u) - 1$. ◁

Recall that $U = \{u_1, \dots, u_m\}$. Given that the claim holds, let $P = \{p_1, \dots, p_m\}$ be any set of positions of G such that for each $i \in [m]$, $G[p_i] = \alpha_{u_i}$ and p_i is unimportant w.r.t. E (choosing arbitrarily if there are multiple choices for p_i). Define $\Sigma_P = \{\text{pred}(p_i) : p_i \in P\}$ and $\mathcal{S}^* = \{S_i \in \mathcal{S} : \langle \beta_{S_i} \rangle \in \Sigma_P\}$.

▷ **Claim 12.** \mathcal{S}^* is a set cover.

Proof. For each $u_i \in U$, there is an unimportant position $p_i \in P$ such that $G[p_i] = \alpha_{u_i}$. Moreover, $\text{pred}(p_i)$ is some character $\langle \beta_S \rangle$ such that $\langle \beta_S \rangle \in \Sigma_P$ and such that $u_i \in S$. Since $S \in \mathcal{S}^*$, it follows that each u_i is covered. ◁

It remains to show that \mathcal{S}^* has at most k sets. Denote $P' = P \cup \{p : G[p] \in \Sigma_P\}$. Let \tilde{G} be the subsequence of G obtained by keeping only positions in P' (i.e. if we denote $P' = \{p'_1, \dots, p'_l\}$ with $p'_1 < p'_2 < \dots < p'_l$, then $\tilde{G} = G[p'_1]G[p'_2] \dots G[p'_l]$). Furthermore, define the CNP \vec{c}_0 such that $\vec{c}_0(\langle \beta_{S_i} \rangle) = 1$ for all $\langle \beta_{S_i} \rangle \in \Sigma_P$, $\vec{c}_0(\langle \beta_{S_i} \rangle) = 0$ for all $\langle \beta_{S_i} \rangle \in \Sigma_{\mathcal{S}} \setminus \Sigma_P$, and $\vec{c}_0(\alpha_u) = 0$ for all $\alpha_u \in \Sigma_U$. Note that \tilde{G} has the form $\langle \beta_{S_{i_1}} \rangle D_1 \langle \beta_{S_{i_2}} \rangle D_2 \dots \langle \beta_{S_{i_r}} \rangle D_r$ for some r , where the D_i 's are substrings over alphabet Σ_U . This is form of Lemma 9.

▷ **Claim 13.** $d_{GCNP}(\tilde{G}, \vec{c}_0) \leq k$.

Proof. Let G' be the genome obtained by replacing every position p of G by some dummy character λ , except for the positions of P' (thus if we remove all the λ occurrences we obtain \tilde{G}). Since G and G' have the same length, we can apply the E events on G' . Let $G'' := G'(E)$, and let l be the number of occurrences of λ in G'' . Recall that P' contains only positions p such that $G[p] \in \Sigma_P$, or such that p is unimportant w.r.t. E and $G[p] \in \Sigma_U$. It follows that if a position q is important w.r.t. E , then $G'[q] \in \Sigma_P \cup \{\lambda\}$. Moreover, for any $\langle \beta_S \rangle \in \Sigma_P$, G'' has as many occurrences of $\langle \beta_S \rangle$ as in $G'(E)$. In other words, G'' has one occurrence of each $\langle \beta_S \rangle \in \Sigma_P$ and the rest is filled with λ .

Let \vec{c}_1 be the CNP satisfying $\vec{c}_1(\lambda) = l$, $\vec{c}_1(\langle \beta_{S_i} \rangle) = \vec{c}_0(\langle \beta_{S_i} \rangle) = 1$ for every $\langle \beta_{S_i} \rangle \in \Sigma_P$, and $\vec{c}_1(x) = 0$ for any other character x . Then clearly, $\vec{c}_1 = \text{cnp}(G'')$, which implies $d_{GCNP}(G', \vec{c}_1) \leq k$ since E transforms G' into G'' . Moreover by Proposition 8, $d_{GCNP}(G' - \lambda, \vec{c}_1 - \lambda) \leq d_{GCNP}(G', \vec{c}_1) \leq k$. The claim follows from the observation that $\tilde{G} = G' - \lambda$ and $\vec{c}_0 = \vec{c}_1 - \lambda$. ◁

Observe that \tilde{G} and \vec{c}_0 have the required form for Lemma 9 (with $|\Sigma_P|$ important positions), and so $d_{GCNP}(\tilde{G}, \vec{c}_0) \geq |\Sigma_P|$. It follows from Claim 13 that $k \geq d_{GCNP}(\tilde{G}, \vec{c}_0) \geq |\Sigma_P| = |\mathcal{S}^*|$. We thus have a set cover \mathcal{S}^* for \mathcal{S} of cardinality at most k , completing the proof. ◀

We arrive to our main inapproximability result, which again follows from Lemma 3.

► **Theorem 14.** *Assuming $P \neq NP$, there is no polynomial-time constant factor approximation algorithm for MCNG.*

In the next section, we prove that the MCNG problem, parameterized by the solution size, is $W[1]$ -hard. This answers another open question in [18]. We refer readers for more details on FPT and $W[1]$ -hardness to the book by Downey and Fellows [7].

4 W[1]-hardness for MCNG

Since SET-COVER is W[2]-hard, naturally we would like to use the above reduction to prove the W[2]-hardness of MCNG. However, the fact that we use t -SET-COVER with constant t in the proof of Lemma 3 is crucial, and t -SET-COVER is in FPT. On the other hand, the property that is really needed in the instance of this proof, and in our MCNG reduction, is that we can transform any set cover instance into an exact cover. We capture this intuition and show that SET-COVER instances that have this property are W[1]-hard to solve.

An instance of SET-COVER-with-EXACT-COVER, or SET-COVER-EC for short, is a pair $I = (\mathcal{S}, k)$ where k is an integer and \mathcal{S} is a collection of sets forming a universe U . In this problem, we require that \mathcal{S} satisfies the property that *any* set cover for \mathcal{S} of size at most k is also an exact cover. We are asked whether there exists a set cover for \mathcal{S} of size at most k (in which case this set cover is also an exact cover).

► **Lemma 15.** *The SET-COVER-EC problem is W[1]-hard for parameter k .*

Proof. We show W[1]-hardness using the techniques introduced by Fellows et al. which is coined as MULTICOLORED-CLIQUE [10]. In the MULTICOLORED-CLIQUE problem, we are given a graph G , an integer k and a coloring $c : V(G) \rightarrow [k]$ such that no two vertices of the same color share an edge. We are asked whether G contains a clique of k vertices (noting that such a clique must have a vertex of each color). This problem is W[1]-hard w.r.t. k .

Given an instance (G, k, c) of MULTICOLORED-CLIQUE, we construct an instance $I = (\mathcal{S}, k')$ of SET-COVER-EC. We put $k' = k + \binom{k}{2}$. For $i \in [k]$, let $V_i = \{v \in V(G) : c(v) = i\}$ and for each pair $i < j \in [k]$, let $E_{ij} = \{uv \in E(G) : u \in V_i, v \in V_j\}$. The universe U of the SET-COVER-EC instance has one element for each color i , one element for each pair $\{i, j\}$ of distinct colors, and two elements for each edge, one for each direction of the edge. That is,

$$U = [k] \cup \binom{[k]}{2} \cup \{(u, v) \in V(G) \times V(G) : uv \in E(G)\}$$

Thus $|U| = k + \binom{k}{2} + 2|E(G)|$. For two colors $i < j \in [k]$, we will denote $U_{ij} = \{(u, v), (v, u) : u \in V_i, v \in V_j, uv \in E_{ij}\}$, i.e. we include in U_{ij} both elements corresponding to each $uv \in E_{ij}$. Now, for each color class $i \in [k]$ and each vertex $u \in V_i$, add to \mathcal{S} the set

$$S_u = \{i\} \cup \{(u, v) : v \in N(u)\}$$

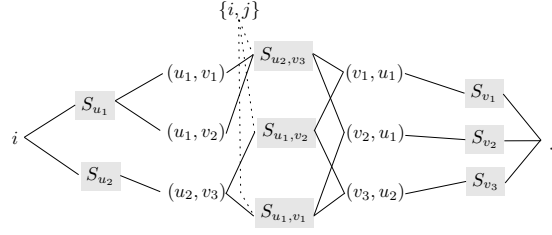
where $N(u)$ is the set of neighbors of u in G . Then for each $i < j \in [k]$, and for each edge $uv \in E_{ij}$, add to \mathcal{S} the set

$$S_{uv} = \{\{i, j\}\} \cup \{(x, y) \in U_{ij} : x \notin \{u, v\}\}$$

The idea is that S_{uv} can cover every element of U_{ij} , except those ordered pairs whose first element is u or v . Then if we do decide to include S_{uv} in a set cover, it turns out that we will need to include S_u and S_v to cover these missing ordered pairs. See Figure 3 for an example. For instance if we include S_{u_2, v_3} in a cover, the uncovered (u_2, v_3) and (v_3, u_2) can be covered with S_{u_2} and S_{v_3} . We show that G has a multicolored clique of size k if and only if \mathcal{S} admits a set cover of size k' . Note that we have not shown yet that (\mathcal{S}, k') is an instance of SET-COVER-EC, i.e. that any set cover of size at most k' is also an exact cover. This will be a later part of the proof.

First suppose that G has a multi-colored clique $C = \{v_1, \dots, v_k\}$, where $v_i \in V_i$ for each $i \in [k]$. Consider the collection

$$\mathcal{S}^* = \{S_{v_1}, \dots, S_{v_k}\} \cup \{S_{v_i v_j} : v_i, v_j \in C, 1 \leq i < j \leq k\},$$



■ **Figure 3** A graphical example of the constructed sets for the U_{ij} elements of a graph (not shown) with $E_{ij} = \{u_1v_1, u_1v_2, u_2v_3\}$, where the u_l 's are in V_i and the v_l 's in V_j (sets have a gray background, edges represent containment, the $\{i, j\}$ lines are dotted only for better visualization).

the cardinality of \mathcal{S}^* is $k + \binom{k}{2} = k'$. Each element $i \in U \cap [k]$ is covered since we include a set S_{v_i} for each color. Each element $\{i, j\} \in U \cap \binom{[k]}{2}$ is covered since we include a set $S_{v_iv_j}$ for each color pair i, j with $i < j$. Consider an element $(x_i, y_j) \in U \cap (V(G) \times V(G))$, where $x_i \in V_i$ and $y_j \in V_j$. Note that either $i < j$ or $j < i$ is possible, and that $v_iv_j \in E(G)$. If $x_i \notin \{v_i, v_j\}$, then $S_{v_iv_j}$ covers (x_i, y_j) . If $x_i = v_i$, then S_{v_i} covers (x_i, v_j) and if $x_i = v_j$, then S_{v_j} covers (x_i, v_j) . Thus \mathcal{S}^* is a set cover, and is of size at most k' .

For the converse direction, suppose that \mathcal{S}^* is a set cover for \mathcal{S} of size at most $k' = k + \binom{k}{2}$. Note that to cover the elements of $U \cap [k]$, \mathcal{S}^* must have at least one set S_u such that $u \in V_i$ for each color class $i \in [k]$. Moreover, to cover the elements of $U \cap \binom{[k]}{2}$, \mathcal{S}^* must have at least one set S_{uv} such that $u \in V_i, v \in V_j$ for each $i, j \in [k]$ pair. We deduce that \mathcal{S}^* has exactly $k + \binom{k}{2}$ sets. Hence for color $i \in [k]$, there is *exactly* one set S_u in \mathcal{S}^* for which $u \in V_i$, and for each $\{i, j\}$ pair, there is *exactly* one S_{uv} set in \mathcal{S}^* for which $u \in V_i, v \in V_j$.

We claim that $C = \{u : S_u \in \mathcal{S}^*\}$ is a multi-colored clique. We already know that C contains one vertex of each color. Now, suppose that some $u, v \in C$ do not share an edge, where $u \in V_i, v \in V_j$ and $i < j$. Let S_{xy} be the set of \mathcal{S}^* that covers $\{i, j\}$, with $x \in V_i, y \in V_j$. Since uv is not an edge but xy is, we know that $u \neq x$ or $v \neq y$ (or both). Moreover, S_{xy} does not cover the (x, y) and (y, x) elements of U_{ij} , and we know that at least one of these is not covered by S_u nor S_v (if $u \neq x$, then none covers (x, y) , if $v \neq y$, then none covers (y, x)). But $(x, y) \in U_{ij}$, and S_u, S_v and S_{xy} are the only sets of \mathcal{S}^* that have elements of U_{ij} , contradicting that \mathcal{S}^* is a set cover. This shows that C is a multi-colored clique.

It remains to show that \mathcal{S}^* is an exact cover. Observe that no two distinct S_u and S_v sets in \mathcal{S}^* can intersect because u and v must be of a different color, and no two distinct S_{uv} and S_{xy} sets in \mathcal{S}^* can intersect because $\{u, v\}$ and $\{x, y\}$ must be from two different color pairs. Suppose that $S_u, S_{xy} \in \mathcal{S}^*$ do intersect, and say that $x \in V_i, y \in V_j$ and $i < j$. Then all elements in $S_u \cap S_{xy}$ are of the form (u, v) for some v . Choose any such (u, v) . If u is of color i , then $u \neq x$ since otherwise by construction S_{xy} could not contain (u, v) . But when $u \neq x$, no set of \mathcal{S}^* covers the element (x, y) (it is not S_u nor S_{xy} , the only two possibilities). If u is of color j , then $u \neq y$ since again S_{xy} could not contain (u, v) . In this case, no set of \mathcal{S}^* covers (y, x) . We reach a contradiction and deduce that \mathcal{S}^* is an exact cover. ◀

It is now almost immediate that MCNG is W[1]-hard with respect to the natural parameter, namely the number of events to transform a genome G into a genome with a given profile \vec{c} (the detailed proof can be found in the full version).

▶ **Theorem 16.** *The MCNG problem is W[1]-hard.*

We do not know whether SET-COVER-EC or MCNG are also in W[1], i.e. whether they are W[1]-complete. We have finished presenting the negative results on MCNG. An

immediate question is whether we could obtain some positive result on a related problem. In the next section, we present some positive result for an interesting variation of MCNG.

5 The Copy Number Profile Conforming Problem

We define the more general *Copy Number Profile Conforming (CNPC)* problem as follows:

► **Definition 17.** *Given two CNP's $\vec{c}_1 = \langle u_1, u_2, \dots, u_n \rangle$ and $\vec{c}_2 = \langle v_1, v_2, \dots, v_n \rangle$, with $u_i, v_i \geq 0$ and $u_i, v_i \in \mathbb{N}$, the CNPC problem asks to compute two strings S_1 and S_2 with $\text{cnp}(S_1) = \vec{c}_1$ and $\text{cnp}(S_2) = \vec{c}_2$ such that the distance between S_1 and S_2 , $d(S_1, S_2)$, is minimized.*

Let $\sum_i u_i = m_1, \sum_i v_i = m_2$, we assume that m_1 and m_2 are bounded by a polynomial of n . (This assumption is needed as the solution of our algorithm could be of size $\max\{m_1, m_2\}$.) We simply say \vec{c}_1, \vec{c}_2 are polynomially bounded. Note that $d(S_1, S_2)$ is a very general distance measure, i.e., it could be any genome rearrangement distance (like reversal, transposition, and tandem duplication, etc, or their combinations, e.g. tandem duplication + deletion). In this paper, we use the breakpoint distance and the adjacency number. Our definitions of these notions are adapted from Angibaud et al. [1] and Jiang et al. [12], which generalize the corresponding concepts on permutations [24].

Given two sequences $A = a_1 a_2 \dots a_n$ and $B = b_1 b_2 \dots b_m$, if $\{a_i, a_{i+1}\} = \{b_j, b_{j+1}\}$ we say that $a_i a_{i+1}$ and $b_j b_{j+1}$ are matched to each other (in the graph theory terminology, they share an edge). Consider a maximum cardinality matching between length 2 substrings of A and B . A matched pair is called an *adjacency*, and an unmatched pair is called a *breakpoint* in A and B respectively. Then, the multiset of 2-substrings of A (resp. B) that belong to a breakpoint is denoted as $b_A(A, B)$ (resp. $b_B(A, B)$) and the corresponding number is $d_b(A, B)$ (resp. $d_b(B, A)$), and the number of common adjacencies between A and B is denoted as $a(A, B)$. Note that $d_b(A, B), d_b(B, A)$ and $a(A, B)$ do not depend on a particular choice of maximum matching. We illustrate the above definitions in Fig. 4.

sequence A	=	$\langle a c b d c b \rangle$
sequence B	=	$\langle a b c d a b c d \rangle$
matched pairs	:	$(cb \leftrightarrow bc), (dc \leftrightarrow cd), (cb \leftrightarrow bc)$
$a(A, B)$	=	$\{bc, bc, cd\}$
$b_A(A, B)$	=	$\{ac, bd\}$
$b_B(A, B)$	=	$\{ab, da, ab, cd\}$

■ **Figure 4** Example for adjacency and breakpoint definitions, with $d_b(A, B) = 2$ and $d_b(B, A) = 4$.

Coming back to our problem, we define $d(S_1, S_2) = d_b(S_1, S_2) + d_b(S_2, S_1)$. From the definitions, we have

$$d_b(S_1, S_2) + d_b(S_2, S_1) + 2 \cdot a(S_1, S_2) = (m_1 - 1) + (m_2 - 1),$$

or,

$$d_b(S_1, S_2) + d_b(S_2, S_1) = m_1 + m_2 - 2 \cdot a(S_1, S_2) - 2.$$

Hence, the problem is really to maximize $a(S_1, S_2)$.

22:12 Genomic Problems Involving Copy Number Profiles

► **Definition 18.** Given n -dimensional vectors $\vec{u} = \langle u_1, u_2, \dots, u_n \rangle$ and $\vec{w} = \langle w_1, w_2, \dots, w_n \rangle$, with $u_i, w_i \geq 0$, and $u_i, w_i \in \mathbb{N}$, we say \vec{w} is a sub-vector of \vec{u} if $w_i \leq u_i$ for $i = 1, \dots, n$, also denote this relation as $\vec{w} \leq \vec{u}$.

Henceforth, we simply call \vec{u}, \vec{w} integer vectors, with the understanding that no item in a vector is negative.

► **Definition 19.** Given two n -dimensional integer vectors $\vec{u} = \langle u_1, u_2, \dots, u_n \rangle$ and $\vec{v} = \langle v_1, v_2, \dots, v_n \rangle$, we say \vec{w} is a common sub-vector of \vec{u} and \vec{v} if \vec{w} is a sub-vector of \vec{u} and \vec{w} is also a sub-vector of \vec{v} (i.e., $\vec{w} \leq \vec{u}$ and $\vec{w} \leq \vec{v}$). Finally, \vec{w} is the maximum common sub-vector of \vec{u} and \vec{v} if there is no common sub-vector $\vec{w}' \neq \vec{w}$ of \vec{u} and \vec{v} which satisfies $\vec{w} \leq \vec{w}' \leq \vec{u}$ or $\vec{w} \leq \vec{w}' \leq \vec{v}$.

An example is illustrated as follows. We have $\vec{u} = \langle 3, 2, 1, 0, 5 \rangle$, $\vec{v} = \langle 2, 1, 3, 1, 4 \rangle$, $\vec{w}' = \langle 2, 1, 0, 0, 3 \rangle$ and $\vec{w} = \langle 2, 1, 1, 0, 4 \rangle$. Both \vec{w} and \vec{w}' are common sub-vectors for \vec{u} and \vec{v} , \vec{w}' is not the maximum common sub-vector of \vec{u} and \vec{v} (since $\vec{w}' \leq \vec{w}$) while \vec{w} is.

Given a CNP $\vec{u} = \langle u_1, u_2, \dots, u_n \rangle$ and alphabet $\Sigma = (x_1, x_2, \dots, x_n)$, for $i \in \{1, 2, \dots, n\}$, we use $S(\vec{u})$ to denote the multiset of letters (genes) corresponding to \vec{u} ; more precisely, u_i denotes the number of x_i 's in $S(\vec{u})$. Similarly, given a multiset of letters Z , we use $s(Z)$ to denote a string where all the letters in Z appear exactly once (counting multiplicity; i.e., $|Z| = |s(Z)|$). $s(Z)$ is similarly defined when Z is a CNP. We present **Algorithm 1**:

1. Compute the maximum common sub-vector \vec{v} of \vec{c}_1 and \vec{c}_2 .
2. Given the gene alphabet Σ , compute $S(\vec{v})$, $S(\vec{c}_1)$ and $S(\vec{c}_2)$. Let $X = S(\vec{c}_1) - S(\vec{v})$ and $Y = S(\vec{c}_2) - S(\vec{v})$.
3. If $S(\vec{v}) = \emptyset$, then return two arbitrary strings $s(\vec{c}_1)$ and $s(\vec{c}_2)$ as S_1 and S_2 , exit; otherwise, continue.
4. Find $\{x, y\}$, $x, y \in \Sigma$ and $x \neq y$, such that $x \in S(\vec{v})$ and $y \in S(\vec{v})$, and exactly one of x, y is in X (say $x \in X$), and the other is in Y (say $y \in Y$). If such an $\{x, y\}$ cannot be found then return two strings S_1 and S_2 by concatenating letters in X and Y arbitrarily at the ends of $s(\vec{v})$ respectively, exit; otherwise, continue.
5. Compute an arbitrary sequence $s(\vec{v})$ with the constraint that the first letter is x and the last letter is y . Then obtain $s_1 = s(\vec{v}) \circ x$ and $s_2 = y \circ s(\vec{v})$ (\circ denotes concatenation).
6. Finally, insert all the elements in $X - \{x\}$ arbitrarily at the two ends of s_1 to obtain S_1 , and insert all the elements in $Y - \{y\}$ arbitrarily at the two ends of s_2 to obtain S_2 .
7. Return S_1 and S_2 .

Let $\Sigma = \{a, b, c, d, e\}$. Also let $\vec{c}_1 = \langle 2, 2, 2, 4, 1 \rangle$ and $\vec{c}_2 = \langle 4, 4, 1, 1, 1 \rangle$. We walk through the algorithm using this input as follows.

1. The maximum common sub-vector \vec{v} of \vec{c}_1 and \vec{c}_2 is $\vec{v} = \langle 2, 2, 1, 1, 1 \rangle$.
2. Compute $S(\vec{v}) = \{a, a, b, b, c, d, e\}$, $S(\vec{c}_1) = \{a, a, b, b, c, c, d, d, d, e\}$ and $S(\vec{c}_2) = \{a, a, a, a, b, b, b, c, d, e\}$. Compute $X = \{c, d, d, d\}$ and $Y = \{a, a, b, b\}$.
3. Identify d and a such that $d \in S(\vec{v})$ and $a \in S(\vec{v})$, and $d \in X$ while $a \in Y$.
4. Compute $s(\vec{v}) = dabbcea$, $s_1 = dabbcea \cdot d$ and $s_2 = a \cdot dabbcea$.
5. Insert elements in $X - \{d\} = \{c, d, d\}$ arbitrarily at the right end of s_1 to obtain S_1 , and insert all the elements in $Y - \{a\} = \{a, b, b\}$ at the right end of s_2 to obtain S_2 .
6. Return $S_1 = dabbcea \cdot d \cdot cdd$ and $S_2 = a \cdot dabbcea \cdot abb$.

► **Theorem 20.** Let \vec{c}_1, \vec{c}_2 be polynomially bounded. The number of common adjacencies generated by Algorithm 1 is optimal with a value either n^* or $n^* - 1$, where $n^* = \sum_{i=1}^n v_i$ with the maximum common sub-vector of \vec{c}_1 and \vec{c}_2 being $\vec{v} = \langle v_1, v_2, \dots, v_n \rangle$.

Proof. First, note that if \vec{v} is a 0-vector (or $S(\vec{v}) = \emptyset$) then there will not be any adjacency in S_1 and S_2 . Henceforth we discuss $S(\vec{v}) \neq \emptyset$.

Notice that a common adjacency between S_1 and S_2 must come from two letters which are both in $S(\vec{v})$. That naturally gives us $n^* - 1$ adjacencies, where $n^* = |S(\vec{v})|$, which can be done by using the letters in $S(\vec{v})$ to form two arbitrary strings S_1 and S_2 (for which $s(\vec{v})$ is a common substring). If $\{x, y\}$ can be found such that $x, y \in S(\vec{v})$ and $x \neq y$, and one of them is in X (say $x \in X$), and the other is in Y (say $y \in Y$), then, obviously we could obtain $s_1 = s(\vec{v}) \circ x$ and $s_2 = y \circ s(\vec{v})$ which are substrings of S_1 and S_2 respectively. Clearly, there are $n^* = |S(\vec{v})|$ adjacencies between s_1 and s_2 (and also S_1 and S_2).

To see that this is optimal, first suppose that no $\{x, y\}$ pair as above can be found. This can only occur when there are no two components $i < j$ in $\vec{c}_1 = \langle c_{1,1}, \dots, c_{1,i}, \dots, c_{1,j}, \dots, c_{1,n} \rangle$, $\vec{c}_2 = \langle c_{2,1}, \dots, c_{2,i}, \dots, c_{2,j}, \dots, c_{2,n} \rangle$, and in the maximum common sub-vector $\vec{v} = \langle v_1, \dots, v_i, \dots, v_j, \dots, v_n \rangle$ of \vec{c}_1 and \vec{c}_2 which satisfy that $\min\{c_{1,i}, c_{2,i}\} = v_i \neq 0$ and $\max\{c_{1,i}, c_{2,i}\} \neq v_i$, and $\min\{c_{1,j}, c_{2,j}\} = v_j \neq 0$ and $\max\{c_{1,j}, c_{2,j}\} \neq v_j$. If this condition holds, then all the components i in $s(\vec{c}_1 - \vec{v})$ and $s(\vec{c}_2 - \vec{v})$, i.e., $c_{1,i} - v_i$ and $c_{2,i} - v_i$, have the property that at least one of the two is zero and $v_i = 0$. Therefore, except for the letters corresponding to \vec{v} , no other adjacency can be formed. As any string with CNP \vec{v} has n^* characters, at most $n^* - 1$ adjacencies can be formed. If an $\{x, y\}$ pair can be found, let $b \in \Sigma$, and let v_b be the minimum copy-number of b in \vec{c}_1 or \vec{c}_2 , i.e., $v_b = \min\{c_{1,b}, c_{2,b}\}$. Assume this minimum occurs in \vec{c}_1 , w.l.o.g. There can be at most $2v_b$ adjacencies involving b in \vec{c}_1 , and thus at most $2v_b$ adjacencies in common involving v_b . Summing over every $b \in \Sigma$, the sum of common adjacencies, counted for each character individually, is at most $\sum_{b \in \Sigma} 2v_b = 2n^*$. Since each adjacency is counted twice in this sum, the number of common adjacencies is at most n^* . ◀

Note that if we only want the breakpoint distance between S_1 and S_2 , then the polynomial boundness condition of \vec{c}_1 and \vec{c}_2 can be withdrawn as we can decide whether $\{x, y\}$ exists by searching directly in the CNPs (vectors).

6 Concluding Remarks

In this paper, we answered two recent open questions regarding the computational complexity of the Minimum Copy Number Generation problem. Our technique could be used for other optimization problems where the solution involves Set Cover whose solution must also be an exact cover. We also present a polynomial time algorithm for the Copy Number Profile Conforming (CNPC) problem when the distance is the classical breakpoint distance. The breakpoint distance is static, and we leave open the question for solving or approximating CNPC with dynamic rearrangement distance such as reversal, duplication+deletion, etc.

References

- 1 Sebastien Angibaud, Guillaume Fertin, Irena Rusu, Annelise Thevenin, and Stephane Vialette. On the approximability of comparing genomes with duplicates. *J. Graph Algorithms and Applications*, 13(1):19–53, 2009.
- 2 Salim Chowdhury, Stanley Shackney, Kerstin Heselmeyer-Haddad, Thomas Ried, Alejandro Shaeffer, and Russell Schwartz. Algorithms to model single gene, single chromosome, and whole genome copy number changes jointly in tumor phylogenetics. *PLOS Computational Biology*, 10(7), 2014.
- 3 SL Cooke, J Temple, S Macarthur, MA Zahra, LT Tan, RAF Crawford, CKY Ng, M Jimenez-Linan, E Sala, and JD Brenton. Intra-tumour genetic heterogeneity and poor chemoradiotherapy response in cervical cancer. *British Journal of Cancer*, 104(2):361, 2011.

- 4 Susanna L Cooke and James D Brenton. Evolution of platinum resistance in high-grade serous ovarian cancer. *The Lancet Oncology*, 12(12):1169–1174, 2011.
- 5 Garance Cordonnier and Manuel Lafond. Comparing copy-number profiles under multi-copy amplifications and deletions. *BMC genomics*, 21(2):1–12, 2020.
- 6 Prue A Cowin, Joshy George, Sian Fereday, Elizabeth Loehrer, Peter Van Loo, Carleen Cullinane, Dariush Etemadmoghadam, Sarah Ftouni, Laura Galletta, Michael S Anglesio, et al. Lrp1b deletion in high-grade serous ovarian cancers is associated with acquired chemotherapy resistance to liposomal doxorubicin. *Cancer Research*, 72(16):4060–4073, 2012.
- 7 Rodney Downey and Michael Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.
- 8 Mohammed El-Kebir, Benjamin Raphael, Ron Shamir, Roded Sharan, Simone Zaccaria, Meirav Zehavi, and Ron Zeira. Copy-number evolutions: complexity and algorithms. In *Proceedings of WABI'2016, LNCS*, volume 9838, pages 137–149. Springer, 2016.
- 9 Mohammed El-Kebir, Benjamin J Raphael, Ron Shamir, Roded Sharan, Simone Zaccaria, Meirav Zehavi, and Ron Zeira. Complexity and algorithms for copy-number evolution problems. *Algorithms for Molecular Biology*, 12(1):13, 2017.
- 10 Michael Fellows, Danny Hermelin, Frances Rosamond, and Stephane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009.
- 11 Patrick Holloway, Krister Swenson, David Ardell, and Nadia El-Mabrouk. Ancestral genome organization: an alignment approach. *Journal of Computational Biology*, 20(4):280–295, 2013.
- 12 Haitao Jiang, Chunfang Zheng, David Sankodd, and Binhai Zhu. Scaffold filling under the breakpoint and related distances. *IEEE/ACM Trans. Bioinformatics and Comput. Biology*, 9(4):1220–1229, 2012.
- 13 Manuel Lafond, Binhai Zhu, and Peng Zou. The tandem duplication distance is np-hard. In *Proceedings of STACS'2020, LIPIcs*, volume 154, pages 15:1–15:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.
- 14 Carlo C Maley, Patricia C Galipeau, Jennifer C Finley, V Jon Wongsurawat, Xiaohong Li, Carissa A Sanchez, Thomas G Paulson, Patricia L Blount, Rosa-Ana Risques, Peter S Rabinovitch, et al. Genetic clonal diversity predicts progression to esophageal adenocarcinoma. *Nature Genetics*, 38(4):468, 2006.
- 15 Andriy Marusyk, Vanessa Almendro, and Kornelia Polyak. Intra-tumour heterogeneity: a looking glass for cancer? *Nature Reviews Cancer*, 12(5):323, 2012.
- 16 Nicholas Navin, Alexander Krasnitz, Linda Rodgers, Kerry Cook, Jennifer Meth, Jude Kendall, Michael Riggs, Yvonne Eberling, Jennifer Troge, Vladimir Grubor, et al. Inferring tumor progression from genomic heterogeneity. *Genome Research*, 20(1):68–80, 2010.
- 17 Cancer Genome Atlas Research Network et al. Integrated genomic analyses of ovarian carcinoma. *Nature*, 474(7353):609, 2011.
- 18 Letu Qingge, Xiaozhou He, Zhihui Liu, and Binhai Zhu. On the minimum copy number generation problem in cancer genomics. In *Proceedings of ACM BCB'2018*, pages 260–269. ACM, 2018.
- 19 Gryte Satas, Simone Zaccaria, Geoffrey Mon, and Benjamin J Raphael. Scarlet: Single-cell tumor phylogeny inference with copy-number constrained mutation losses. *Cell Systems*, 10(4):323–332, 2020.
- 20 Roland F Schwarz, Anne Trinh, Botond Sipos, James D Brenton, Nick Goldman, and Florian Markowetz. Phylogenetic quantification of intra-tumour heterogeneity. *PLoS Computational Biology*, 10(4):e1003535, 2014.
- 21 Sohrab P Shah, Ryan D Morin, Jaswinder Khattra, Leah Prentice, Trevor Pugh, Angela Burleigh, Allen Delaney, Karen Gelmon, Ryan Guliany, Janine Senz, et al. Mutational evolution in a lobular breast tumour profiled at single nucleotide resolution. *Nature*, 461(7265):809, 2009.

- 22 Ron Shamir, Meirav Zehavi, and Ron Zeira. A linear-time algorithm for the copy number transformation problem. In *Proceedings of CPM'2016, LIPIcs*, volume 54, pages 16:1–16:13. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- 23 Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of 33rd ACM Symp. on Theory of Comput. (STOC'01)*, pages 453–461. ACM, 2001.
- 24 G.A. Watterson, W.J. Ewens, T.E. Hall, and A. Morgan. The chromosome inversion problem. *J. Theoretical Biology*, 99(1):1–7, 1982.
- 25 Ruofan Xia, Yu Lin, Jun Zhou, Tieming Geng, Bing Feng, and Jijun Tang. Phylogenetic reconstruction for copy-number evolution problems. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(2):694–699, 2018.
- 26 Simone Zaccaria, Mohammed El-Kebir, Gunnar W Klau, and Benjamin J Raphael. Phylogenetic copy-number factorization of multiple tumor samples. *Journal of Computational Biology*, 25(7):689–708, 2018.

Compressed Orthogonal Search on Suffix Arrays with Applications to Range LCP

Kotaro Matsuda

Graduate School of Information Science and Technology, The University of Tokyo, Japan
kotaro_matsuda@me2.mist.i.u-tokyo.ac.jp

Kunihiko Sadakane 

Graduate School of Information Science and Technology, The University of Tokyo, Japan
sada@mist.i.u-tokyo.ac.jp

Tatiana Starikovskaya

DIENS, École normale supérieure, PSL Research University, Paris, France
tat.starikovskaya@gmail.com

Masakazu Tateshita

Graduate School of Information Science and Technology, The University of Tokyo, Japan
masakazu_tateshita@me2.mist.i.u-tokyo.ac.jp

Abstract

We propose a space-efficient data structure for orthogonal range search on suffix arrays. For general two-dimensional orthogonal range search problem on a set of n points, there exists an $n \log n(1+o(1))$ -bit data structure supporting $O(\log n)$ -time counting queries [Mäkinen, Navarro 2007]. The space matches the information-theoretic lower bound. However, if we focus on a point set representing a suffix array, there is a chance to obtain a space efficient data structure. We answer this question affirmatively. Namely, we propose a data structure for orthogonal range search on suffix arrays which uses $O(\frac{1}{\varepsilon}n(H_0 + 1))$ bits where H_0 is the order-0 entropy of the string and answers a counting query in $O(n^\varepsilon)$ time for any constant $\varepsilon > 0$. As an application, we give an $O(\frac{1}{\varepsilon}n(H_0 + 1))$ -bit data structure for the range LCP problem.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Orthogonal Range Search, Succinct Data Structure, Suffix Array

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.23

Funding *Kunihiko Sadakane*: Supported by JST CREST Grant Number JPMJCR1402, Japan.

Acknowledgements The authors would like to thank anonymous referees for helpful comments.

1 Introduction

In this paper we consider the problem of orthogonal range search on suffix arrays (ORS on SA). The problem is defined as follows. We are given the suffix array $SA[1, n]$ of a string T of length n , which is a data structure for string matching [12]. Then we construct a two-dimensional point set $P = \{p_i = (i, SA[i]) \mid i = 1, \dots, n\}$. We consider two types of queries on P : a reporting query and a counting query. Given a query region $Q = [x_1, x_2] \times [y_1, y_2]$, the reporting query must output $Q \cap P$, and the counting query $|Q \cap P|$.

The problem is a special case of the general two-dimensional range search problem for which there exists $O(\log n)$ time solutions for the counting query using $O(n \log n)$ -bit space [4] or $n \log n(1 + o(1))$ -bit¹ space [11]. However there are no data structures using the fact that the point set represents a suffix array to reduce the data structure size.

¹ Throughout the paper $\log n$ denotes $\log_2 n$.

The orthogonal range search problem on suffix arrays has many applications in string algorithms. A direct application is the position-restricted substring search problem [11]. Given a pattern $P[1, m]$ and two integers $1 \leq \ell \leq r \leq n$, count all the occurrences of P in $T[\ell, r]$ or locate them. This corresponds to the counting and reporting problems on the point set P for the suffix array. Other problems are the interval LCP problem introduced by Cormode and Muthukrishnan [5] and the range LCP problem introduced by Amir et al. [2]. For a string $T[1, n]$, let $lcp(i, j)$ denote the length of the longest common prefix between $T[i, n]$ and $T[j, n]$. An interval LCP query $ilcp(i, \alpha, \beta)$ takes three integers $i, \alpha, \beta \in [1, n]$ and must return $\max_{j \in [\alpha, \beta]} lcp(i, j)$. A range LCP query $rlcp(\alpha, \beta)$ receives two integers $\alpha, \beta \in [1, n]$ and must return $\max_{i, j \in [\alpha, \beta]} lcp(i, j)$.

Although the orthogonal range search problem on suffix arrays plays an important role in string algorithms, a bottleneck is its space usage. For a string of length n on an alphabet of size σ , its suffix array uses $n \log n$ bits [12]. A data structure for orthogonal range search uses $n \log n + o(n \log n)$ bits, which can be used solely without the standard representation of the suffix array. If $\sigma < n$, this can be much more than the $n \log \sigma$ bits of space required to store the string itself.

It seems $n \log n$ bits are necessary for storing suffix arrays or ORS data structures because it can represent a permutation of $\{1, 2, \dots, n\}$ which requires $\Omega(n \log n)$ bits to represent. For suffix arrays, however, there are data structures for storing them in $O(n \log \sigma)$ bits [8] or $O(n(H_0(T) + 1))$ bits [18] where $H_0(T)$ is the order-0 entropy of T . This is not surprising, if we do not consider query time, because the suffix arrays is computed from the string that can be compressed in $O(n(H_0(T) + 1))$ bits. This should hold also for ORS data structures on suffix arrays. However there are no such data structures for ORS which have $o(n)$ query time.

In this paper, we propose a space-efficient data structure for orthogonal range search on suffix arrays. The main result is as follows.

► **Theorem 1.** *For a string T of length n , consider orthogonal range search on the suffix array of T . For any constant $\varepsilon > 0$, there exists a data structure using $O(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ bits which supports a counting query in $O(n^\varepsilon)$ time and a reporting query in $O((occ + 1) \cdot n^\varepsilon)$ time.*

This is the first data structure to achieve linear ($O(n \log \sigma)$ -bit) space and sub-linear query time.

As an application, we give space-efficient solutions for the interval LCP and the range LCP problems. For the interval LCP problem, [5, 9] introduced two different data structures that use $O(n \log n)$ bits of space and have query time $O(\log^{1+\varepsilon} n)$, where $\varepsilon > 0$ is an arbitrary constant (in fact, both works show a series of data structures with different space-time trade-offs, we only give the data structures with lowest space requirements here). In this work, we show a data structure that requires $O(\frac{1}{\varepsilon} n(H_0(T) + 1))$ bits of space and maintains interval LCP queries in $O(n^\varepsilon)$ time.

As for the range LCP queries, Amir et al. [2] gave two data structures; one uses $O(n \log^{2+\varepsilon} n)$ bits with query time $O(\log \log n)$ and the other uses $O(n \log n)$ bits with query time $O(\delta \log \log n)$, where $\delta = \beta - \alpha + 1$ is the length of the range. Patil et al. [15] gave a data structure of $O(n \log n)$ bits with query time $O(\sqrt{\delta} \log^\varepsilon \delta)$. Abedin et al. [1] gave a data structure of $O(n \log n)$ bits with query time $O(\log^{1+\varepsilon} n)$. In addition, Amir, Lewenstein, and Thankanchan [3] considered range LCP queries with a bounded number of mismatches.

In this work, we develop a new data structure for range LCP queries that requires $O(\frac{1}{\varepsilon} n(H_0(T) + 1))$ bits of space and achieves $O(n^\varepsilon)$ query time.

2 Preliminaries

2.1 Succinct data structures

A succinct data structure is a data structure whose size asymptotically achieves the information-theoretic lower bound for representing the data. In this paper, we use succinct data structures for bit-vectors. A bit-vector is a string $B[1, n]$ on the binary alphabet $\{0, 1\}$. Its information-theoretic lower bound is n bits, and there exists a succinct data structure using $n + o(n)$ bits supporting the following operations in constant time [16]:

- $rank_c(B, i)$: returns the number of c 's in $B[1, i]$ ($c = 0, 1$),
- $select_c(B, j)$: returns the position of the j -th c from the beginning in B ($c = 0, 1$).

2.2 Suffix arrays

Consider a string $T[1, n]$ of length n on an alphabet \mathcal{A} of size σ . We add the unique terminator $\$$ at the end of the string, that is, $T[n + 1] = \$$. We assume the terminator is alphabetically smaller than any letter in \mathcal{A} . The suffix array of T stores lexicographic order of suffixes of T . Let $SA[0, n]$ be the suffix array of T . Then $SA[i] = j$ means that the lexicographically i -th smallest suffix of T is the suffix $T[j, n + 1]$ starting at position j of T . It holds $SA[0] = n + 1$ and for $i = 1, \dots, n$, $1 \leq SA[i] \leq n$ and the values are a permutation of $\{1, 2, \dots, n\}$. A pattern search for pattern length m can be done in $O(m \log n)$ using the string T and its suffix array SA . The required space is $n \log \sigma$ bits for T and $n \log n$ bits for SA [12].

2.3 Compressed suffix arrays

Compressed suffix arrays [8] are data structures for compressing suffix arrays from $n \log n$ bits to $O(n \log \sigma)$ bits. This is further compressed into $O(n(H_0(T) + 1))$ bits [18]. There are several variants of the compressed suffix arrays and the most basic one takes $O(\log n)$ time to compute an entry $SA[i]$ of the suffix array.

Instead of SA , compressed suffix arrays stores the Ψ function defined as follows:

$$\Psi[i] = \begin{cases} SA^{-1}[SA[i] + 1], & \text{if } SA[i] \leq n; \\ SA^{-1}[1], & \text{otherwise.} \end{cases}$$

Figure 1 shows an example of the suffix array and the compressed suffix array. An important property of the Ψ function is that it is piece-wise monotone.

► **Lemma 2** (Prop. 4.1 in [18]). *For a string of length n on an alphabet of size σ , consider its suffix array SA and Ψ . For any $1 \leq i < j \leq n$, if $T[SA[i]] = T[SA[j]]$, it holds $\Psi[i] < \Psi[j]$.*

Proof. For any $1 \leq i \leq n$, $\Psi[i] = SA^{-1}[SA[i] + 1]$ is the lexicographic order of the suffix $T[SA[i] + 1, n + 1]$, which is obtained by removing the first character of the suffix $T[SA[i], n + 1]$. For any $1 \leq i < j \leq n$, if $T[SA[i]] = T[SA[j]]$, the suffixes $T[SA[i], n + 1]$ and $T[SA[j], n + 1]$ have the same first character and their relative order is determined by the suffixes made by removing the first characters. This means $\Psi[i] = SA^{-1}[SA[i] + 1] < SA^{-1}[SA[j] + 1] = \Psi[j]$. ◀

From this property, the Ψ function consists of at most σ increasing sequences and it can be compressed into $O(n(H_0(T) + 1))$ bits, and an entry of the suffix array can be computed from Ψ in $O(\log n)$ time. An entry of the inverse suffix array $ISA[j] = SA^{-1}[j]$ can be also computed in $O(\log n)$ time. For more details, see [18].

	Ψ	SA	$T[SA[i], n + 1]$		$\Psi_0 SA_0$	$T[SA_0[i], n + 1]$	
0	3	12	\$	0	3	1	<i>abraca dabra</i> \$
1	0	11	<i>a</i> \$	1	4	4	<i>aca dabra</i> \$
2	6	8	<i>abra</i> \$	2	5	6	<i>a dabra</i> \$
3	7	1	<i>abracadabra</i> \$	3	6	2	<i>braca dabra</i> \$
4	8	4	<i>acadabra</i> \$	4	2	5	<i>ca dabra</i> \$
5	9	6	<i>adabra</i> \$	5	0	7	<i>dabra</i> \$
6	10	9	<i>bra</i> \$	6	1	3	<i>raca dabra</i> \$
7	11	2	<i>bracadabra</i> \$				
8	5	5	<i>cadabra</i> \$	0	3	7	\$
9	2	7	<i>dabra</i> \$	1	4	1	<i>abraca</i> \$
10	1	10	<i>ra</i> \$	2	5	4	<i>aca</i> \$
11	4	3	<i>racadabra</i> \$	3	0	6	<i>a</i> \$
				4	6	2	<i>braca</i> \$
				5	3	5	<i>ca</i> \$
				6	2	3	<i>raca</i> \$

■ **Figure 1** An example of the suffix array and its sub-array.

2.4 Suffix trees and compressed suffix trees

Suffix tree [21] is a data structure for storing all the suffixes of a string T by a tree structure. Leaves of the suffix tree have one-to-one correspondence with all the suffixes. An internal node v of the suffix tree corresponds to a substring s of T so that the suffixes corresponding to the leaves in the subtree rooted at v share the same prefix s . The string depth of v is defined as the length of s .

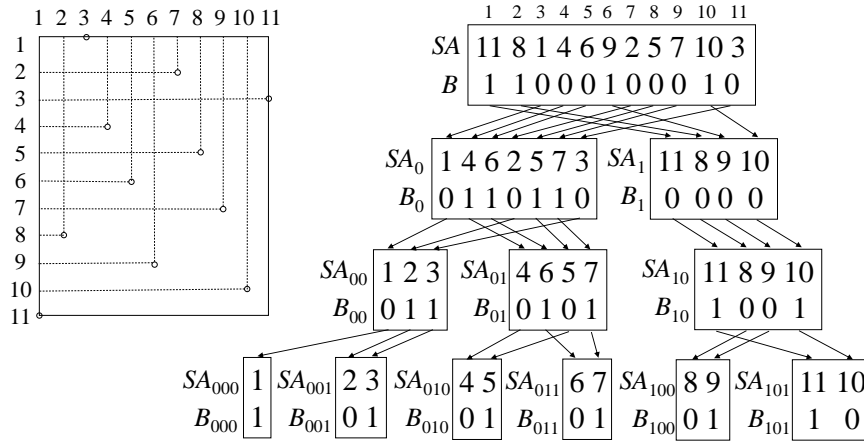
Compressed suffix tree [19] reduces the space of the suffix tree from $O(n \log n)$ bits to $6n + o(n) + \text{SIZE}_{SA}(n, \sigma)$ bits where $\text{SIZE}_{SA}(n, \sigma)$ is the size of a data structure for the suffix array. Let $\text{TIME}_{SA}(n, \sigma)$ denote the time to compute an entry of the suffix array or the inverse suffix array using a compressed suffix array. Then the string depth of a node is computed in $O(\text{TIME}_{SA}(n, \sigma))$ time. Also, given two positions i, j in the string, $\text{lcp}(i, j)$ is computed in $O(\text{TIME}_{SA}(n, \sigma))$ time. Note that there is a data structure [18] satisfying $\text{SIZE}_{SA}(n, \sigma) = O(n(H_0(T) + 1))$ bits and $\text{TIME}_{SA}(n, \sigma) = O(\log n)$.

By augmenting the compressed suffix tree with a constant time level ancestor query data structure [13], we can answer weighted level ancestor queries in $O(\text{TIME}_{SA}(n, \sigma) \log n)$ time. A weighted level ancestor query $WLA(u, d)$ on a suffix tree is to find the nearest ancestor of a node u in the suffix tree with string depth at most d . This is solved by simply binary searching nodes on the path from u to the root using string depths of nodes as keys.

2.5 Wavelet trees and orthogonal range search

Wavelet tree [7] is a data structure for computing *rank* and *select* on strings efficiently for large alphabets. The wavelet tree of a string T of length n on an alphabet of size σ occupies $(n + o(n)) \log \sigma$ bits of space and allows computing $\text{rank}_c(T, i)$ and $\text{select}_c(T, j)$ for any character c in the alphabet in $O(\log \sigma)$ time. Wavelet trees can be used to answer two-dimensional orthogonal range searches in $O(\log \sigma)$ time [11]: Given a two-dimensional rectangle query $Q = [x_1, x_2] \times [y_1, y_2]$, the orthogonal range reporting query must output $Q \cap P$, and the orthogonal range counting query $|Q \cap P|$.

Because in this paper we consider wavelet trees only for strings consisting of suffix array entries, we explain the data structure of the wavelet tree using suffix arrays.



■ **Figure 2** An example of a point set for the suffix array of Figure 1 (left), and its wavelet tree representation (right).

The wavelet tree W of the suffix array SA of length n is defined recursively. In the root node of W , we store a bit-vector $B[1, n]$, where $B[i]$ is the first bit of the binary encoding of $SA[i]$ for $i = 1, \dots, n$. We construct two arrays, SA_0 and SA_1 , where SA_0 stores all the entries of SA whose first bit is 0 in the same order as in SA , and SA_1 stores the rest (all the entries of SA whose first bit is 1) in the same order as in SA . We consider that the first bits of entries of SA_0 and SA_1 are removed. The left and the right children of the root node of W store the wavelet trees for SA_0 and SA_1 , respectively. We say that the root node is in level 0 and its children are in level 1, and so on. Let B_0 and B_1 denote the bit-vectors in the left child and the right child, respectively. Then the length of B_0 is $rank_0(B, n)$ and if $B[i] = 0$, $SA[i]$ corresponds to $SA_0[rank_0(B, i)]$. Similarly, the length of B_1 is $rank_1(B, n) = n - rank_0(B, n)$ and if $B[i] = 1$, $SA[i]$ corresponds to $SA_1[rank_1(B, i)]$. We add the auxiliary data structures for computing *rank* and *select* on the bit-vectors.

3 Compressed Orthogonal Range Search on Suffix Arrays

In this section, we prove Theorem 1 and give an $O(n(H_0(T) + 1))$ -bit space implementation of the orthogonal range search queries on the point set $P = \{p_i = (i, SA[i]) \mid i = 1, \dots, n\}$. The wavelet tree [11] on SA occupies $n \log n + o(n \log n)$ bits of space and maintains the orthogonal range queries in $O(\log n)$ time. We will show that for this special case of P one can achieve $O(n(H_0(T) + 1))$ space complexity (at an expense of higher query time).

If we store the bit-vectors of the nodes of the wavelet tree explicitly, we need $O(n \log n)$ bits. However, each bit in the bit-vectors is some bit of an entry of the suffix array, and the suffix array can be represented in $O(n(H_0(T) + 1))$ bits. We will use this fact to develop a data structure for orthogonal range queries by imitating the wavelet tree and decoding the required information on the fly from the compressed suffix array of the string.

3.1 Orthogonal range search

In the orthogonal range search algorithm using wavelet trees, we need to compute $rank(B_v, i)$ on the bit-vector B_v of node v of the wavelet tree. To compute it using the compressed suffix array, we need the relation between bits of the bit-vector and entries of the suffix array.

We divide B into blocks of length Δ , where Δ is a parameter to be determined later. Let $B^j = B[j\Delta + 1, (j + 1)\Delta]$ be the j -th block ($j = 0, 1, \dots, n/\Delta$). For each block j , we store $B^j.rank_0$: the number of 0's in the blocks to the left of B^j .

Similarly, for the bit-vector B_v of node v , we create $\lceil \frac{n}{\Delta} \rceil$ many blocks B_v^j . The block B_v^j contains the entries corresponding to the entries of SA such that they belong to B^j and their binary representation starts with v . For the j -th block B_v^j , we store

- $B_v^j.rank_0$: the number of 0's in the blocks to the left of B_v^j ,
- $B_v^j.blocknum$: the total number of bits in the blocks to the left of B_v^j .

To compute $rank_0(B_v, i)$, we first find the block j of the bit-vector in the root node that contains the i -th bit of B_v , extract the entries of the compressed suffix array for the block of length Δ , compute the number of 0's in the block up to position i , and add it to the number of 0's before the block in B_v , which is stored in $B_v^j.rank_0$.

However, this approach requires even more space than the wavelet tree: at depth $\log n$, there are n nodes and therefore $\lceil \frac{n}{\Delta} \rceil \cdot n$ blocks. To overcome this, we stop the recursion for blocks at every $k = \lceil \varepsilon \log n \rceil$ steps, and restart the data structure construction. Details are as follows.

Consider the nodes of the wavelet tree at depth dk ($d = 0, 1, \dots, \frac{1}{\varepsilon} - 1$). There are 2^{dk} such nodes, and for each such node v' , we divide the bit-vector $B_{v'}$ into $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil$ many blocks $B_{v'}^j$ ($j = 0, \dots, \lfloor \frac{n}{\Delta \cdot 2^{dk}} \rfloor$) of length Δ each. Similarly to the above data structure, we store, for each j , $B_{v'}^j.rank_0$: the number of 0's in the blocks to the left of block $B_{v'}^j$.

Then, each bit-vector $B_{v''}$ of a node v'' at depth $dk + 1$ to $(d + 1)k$ is divided into $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil$ many blocks $B_{v''}^j$, corresponding to the blocks of a bit-vector in a node at depth dk which is the ancestor of v'' with that level. For the j -th block $B_{v''}^j$, we store

- $B_{v''}^j.rank_0$: the number of 0's in the blocks to the left of $B_{v''}^j$,
- $B_{v''}^j.blocknum$: the total number of bits in the blocks to the left of $B_{v''}^j$.

The number of blocks between depths $dk + 1$ and $(d + 1)k$ is $\lceil \frac{n}{\Delta \cdot 2^{dk}} \rceil \cdot 2^k \cdot 2^{dk} = 2^k \cdot \lceil \frac{n}{\Delta} \rceil = n^\varepsilon \cdot \lceil \frac{n}{\Delta} \rceil$. Therefore, the total space is $O(\frac{1}{\varepsilon} \cdot n^\varepsilon \frac{n}{\Delta} \log n)$ bits.

At depth dk of the wavelet tree, there are 2^{dk} many bit-vectors $B_{00\dots 00}, B_{00\dots 01}, \dots, B_{11\dots 11}$. Each bit-vector corresponds to a sub-array SA_v of SA such that the first dk bits of the binary encodings of the suffix array entries are equal to v . That is, the bit-vector $B_{00\dots 00}$ corresponds to the sub-array $SA_{00\dots 00}$ containing entries in the range $[1, \frac{n}{2^{dk}}]$, $B_{00\dots 01}$ to $SA_{00\dots 01}$ containing entries in $[\frac{n}{2^{dk}} + 1, \frac{2n}{2^{dk}}]$, and $B_{11\dots 11}$ to $SA_{11\dots 11}$ containing entries in $[\frac{(2^{dk} - 1)n}{2^{dk}} + 1, n]$.

► **Lemma 3.** *The sub-array SA_v , which consists of the entries $SA[\ell(v - 1) + 1, \ell v]$ where $\ell = \frac{n}{2^{dk}}$, contains positions in the substring $T_v = T[\ell(v - 1) + 1, \ell v]$, and it can be compressed in $O(\ell(H_0(T_v) + 1))$ bits. Each entry of SA_v can be computed in $O(\log \ell)$ time.*

Proof. SA_v stores positions of suffixes between $\ell(v - 1) + 1$ and ℓv . From the construction of the wavelet tree, SA_v stores positions of T_v . We insert one entry $\ell v + 1$ to the sub-array, whose position in the sub-array is determined by the lexicographic order of the corresponding suffixes in the entire string. Let p be this position. We subtract $\ell(v - 1) + 1$ from each entry in SA_v so that they become integers from 0 to ℓ to obtain an array SA'_v . (See Figure 1 for an example.) For any $0 \leq i \leq v + 1$ except for p , we define $\Psi'_v[i] = SA'_v{}^{-1}[SA'_v[i] + 1]$. By Lemma 2, Ψ'_v consists of at most σ increasing sequences and the values are in $[0, \ell]$. Then we can compress Ψ'_v in $O(\ell(H_0(T_v) + 1))$ bits, and each entry of $SA'_v[i]$ can be computed in $O(\log \ell)$ time (Theorem 4.1 in [18]). ◀

■ **Algorithm 1** $\text{calcblock}(v, \text{level}, k)$: Given a node v with depth level , find the node v' that is the nearest ancestor of v whose depth is a multiple of k .

```

1:  $v' = v$ 
2: for  $i = 1, \dots, \text{level} - k$  do
3:    $v' = \lfloor \frac{v'-1}{2} \rfloor$ 
4: return  $v'$ 

```

■ **Algorithm 2** $\text{calcblockrank}(v, \text{level}, j, c, k)$: returns the rank_0 value at $B_v[c]$ in the bit-vector B_v by extracting Δ entries of the compressed suffix array in the block that contains $B_v[c]$.

```

1:  $\text{blockvalue} = c - B_v^j.\text{blocknum}$ 
2:  $v' = \text{calcblock}(v, \text{level}, k)$ 
3:  $\text{rank} = 0$ 
4:  $i = 1$ 
5:  $\text{cnt} = 0$ 
6: while  $\text{cnt} < \text{blockvalue}$  do
7:   if the highest  $(\text{level} - k \cdot \lfloor \frac{\text{level}}{k} \rfloor)$  bits of  $SA_{v'}[j \cdot \Delta + i]$  are equal to those of  $v$  then
8:      $\text{cnt} = \text{cnt} + 1$ 
9:     if the  $(\text{level} - k \cdot \lfloor \frac{\text{level}}{k} \rfloor + 1)$ -st bit of  $SA_{v'}[j \cdot \Delta + i]$  is 0 then
10:       $\text{rank} = \text{rank} + 1$ 
11:     $i = i + 1$ 
12: return  $\text{rank}$ 

```

For each node v of depth dk of the wavelet tree, we represent SA_v as in Lemma 3. Therefore, the total space for depth dk is $\sum_v O(\ell(H_0(T_v) + 1))$ bits. From concavity of logarithm, this is upper bounded by $O(n(H_0(T) + 1))$ bits. Then the total for all levels which are multiple of k is $O(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ bits. Levels that are not multiples of k require $O(\frac{1}{\varepsilon} \cdot n^\varepsilon \frac{n}{\Delta} \log n)$ bits of space.

Using this data structure for orthogonal range search, the counting query is solved by Algorithms 1–3. The rank operations take $O(\Delta \cdot \log n)$ time for nodes of depths which are multiples of k , and $O(\Delta)$ time for other nodes. Therefore, the total query time is $O(\Delta \cdot \log n \cdot \frac{1}{\varepsilon} + \Delta \cdot \log n) = O(\Delta \cdot \log n)$. The reporting query is solved analogously, and for output of size occ takes $O(\text{occ} \cdot \Delta \cdot \log n)$ time. By letting $\Delta = n^\varepsilon \log n$, we obtain a data structure of $O(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ bits and $O(n^\varepsilon \log^2 n)$ query time. To improve the query time to $O(n^\varepsilon)$, we use another parameter $\varepsilon' < \varepsilon$ such that $n^{\varepsilon'} \log^2 n = O(n^\varepsilon)$ and $\frac{1}{\varepsilon'} = O(\frac{1}{\varepsilon})$. Then we obtain Theorem 1.

3.2 Range successor/predecessor

As a simple extension, we consider 2-dimensional range successor and predecessor queries [14] defined as follows.

► **Definition 4.** Let P be a set of n points on the $[1, n] \times [1, n]$ grid. The two-dimensional range successor and predecessor queries are defined as:

$$\begin{aligned}
ORS_{xSucc}([x, +\infty], [y, y']) &= \operatorname{argmin}_i \{(i, j) \in P \cap [x, +\infty] \times [y, y']\}, \\
ORS_{xPred}([-\infty, x'], [y, y']) &= \operatorname{argmax}_i \{(i, j) \in P \cap [-\infty, x'] \times [y, y']\}, \\
ORS_{ySucc}([x, x'], [y, +\infty]) &= \operatorname{argmin}_j \{(i, j) \in P \cap [x, x'] \times [y, +\infty]\}, \\
ORS_{yPred}([x, x'], [-\infty, y']) &= \operatorname{argmax}_j \{(i, j) \in P \cap [x, x'] \times [-\infty, y']\}.
\end{aligned}$$

■ **Algorithm 3** $ORS_SA([x_1, x_2], [y_1, y_2])$: For the point set $p_i = (i, SA[i])$ ($i = 1, \dots, n$) where SA is the suffix array of a string of length n , return the number of points $p_i \in Q = [x_1, x_2] \times [y_1, y_2]$.

```

1: return  $count([x_1, x_2], [y_1, y_2], \varepsilon, 0, [0, 0], [1, n], k)$ 
2:
3: function  $count([x_1, x_2], [y_1, y_2], v, level, j_1, j_2, [a, b], k)$ 
4:  $\#leftchild(v)$ : returns the bit-vector of the left child of  $v$ , i.e., the bit-vector of  $v$  to
   which 0 is appended at the end.
5:  $\#rightchild(v)$ : returns the bit-vector of the right child of  $v$ , i.e., the bit-vector of  $v$  to
   which 1 is appended at the end.
6: if  $x_1 > x_2$  then
7:   return 0
8: if  $[a, b] \cap [y_1, y_2] = \emptyset$  then
9:   return 0
10: if  $[a, b] \subset [y_1, y_2]$  then
11:   return  $x_2 - x_1 + 1$ 
12:  $k = \lceil \varepsilon \log n \rceil$ 
13: if  $level \% k = 0$  then
14:    $j_1 = \lfloor \frac{x_1 - 1}{\Delta} \rfloor$ 
15:    $j_2 = \lfloor \frac{x_2}{\Delta} \rfloor$ 
16:  $x_1^l = calcblockrank(v, level, j_1, x_1 - 1, k) + B_v^{j_1}.rank_0 + 1$ 
17:  $x_2^l = calcblockrank(v, level, j_2, x_2, k) + B_v^{j_2}.rank_0$ 
18:  $x_1^r = x_1 - x_1^l + 1$ 
19:  $x_2^r = x_2 - x_2^l$ 
20:  $m = \lfloor \frac{a+b}{2} \rfloor$ 
21: return  $count([x_1^l, x_2^l], [y_1, y_2], leftchild(v), level + 1, j_1, j_2, [a, m], k) +$ 
        $count([x_1^r, x_2^r], [y_1, y_2], rightchild(v), level + 1, j_1, j_2, [m + 1, b], k)$ 

```

We focus on range successor and predecessor queries on suffix arrays. In addition to the data structure in Section 3.1, we store the following.

- $B_v^j.xSucc$: the index i such that $SA[i]$ attains the minimum value in blocks B_v^j to $B_v^{\lfloor \frac{n}{\Delta} \rfloor}$.
- $B_v^j.xPred$: the index i such that $SA[i]$ attains the maximum value in blocks B_v^0 to B_v^j .
- $B_v^j.ySucc$: $\min\{SA[x] \mid x \in [\frac{n}{\Delta} \cdot j, \frac{n}{\Delta} \cdot (j+1) - 1] \text{ and } SA[x] \in [\ell, n]\}$ where ℓ is the leftmost index corresponding to node v
- $B_v^j.yPred$: $\max\{SA[x] \mid x \in [\frac{n}{\Delta} \cdot j, \frac{n}{\Delta} \cdot (j+1) - 1] \text{ and } SA[x] \in [0, r]\}$ where r is the rightmost index corresponding to node v

Using these data structures, we can answer $ORS_{xSucc}([x, +\infty], [y, y'])$ ($ORS_{xPred}([-\infty, x'], [y, y'])$) as follows. First, we extract $SA[x, (\lfloor \frac{x}{\Delta} \rfloor + 1) \cdot \Delta - 1]$ ($SA[\lfloor \frac{x'}{\Delta} \rfloor \cdot \Delta, x']$), and if there is a value in $[y, y']$, the one with the smallest (largest) index is the answer. This is done in $O(\Delta \cdot \log n)$ time. Otherwise, we perform an ORS query. During the search every time we compute $rank(B_v, c)$, if $B_v[c]$ belongs to B_v^j and the range of characters $[s, t]$ for node v satisfies $[s, t] \subset [y, y']$, we store $B_v^{j-1}.xSucc$ ($B_v^{j-1}.xPred$) as a candidate of the answer and compare it with the minimum (maximum) index of the extracted suffix array values in $[y, y']$. If $\Delta = n^\varepsilon \log n$, the time complexity is $O(\Delta \cdot \log n + \Delta \cdot \log n) = O(n^\varepsilon \log^2 n)$. This can be easily reduced to $O(n^\varepsilon)$ time. $ORS_{ySucc}([x, x'], [y, +\infty])$ ($ORS_{yPred}([x, x'], [-\infty, y'])$) is solved similarly.

We obtain the following.

► **Theorem 5.** For a string T of length n , there exists an $O(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ -bit data structure supporting two-dimensional range successor/predecessor queries in $O(n^\varepsilon)$ time.

Keller et al. [10] showed the following.

► **Lemma 6** (cf. [10]). The interval LCP queries can be reduced to two-dimensional range successor/predecessor queries.

Proof. We reduce an interval LCP query $ilcp(p, \alpha, \beta)$ to the two-dimensional range successor/predecessor queries on a set $P = \{(i, SA[i]) \mid i \in [1, n]\}$ for the string. Let $q = SA^{-1}[p]$. We compute the largest lexicographic order $x < q$ such that $SA[x] \in [\alpha, \beta]$ and the smallest lexicographic order $y > q$ such that $SA[y] \in [\alpha, \beta]$. This is done by $x = ORS_{xPred}([-\infty, p - 1], [\alpha, \beta])$ and $y = ORS_{xSucc}([p + 1, +\infty], [\alpha, \beta])$. Then it holds $ilcp(p, \alpha, \beta) = \max\{lcp(p, x), lcp(p, y)\}$. ◀

From Theorem 5 and Lemma 6, we obtain:

► **Corollary 7.** For a string T of length n , there exists an $O(\frac{1}{\varepsilon} \cdot n(H_0(T) + 1))$ -bit data structure supporting interval LCP queries in $O(n^\varepsilon)$ time.

4 Range LCP queries

In this section, we will show a data structure that occupies $O(\frac{1}{\varepsilon}n(H_0(T) + 1))$ bits of space and supports $rlcp$ queries in $O(n^\varepsilon)$ time. We follow the outline of the data structure presented in [1], but improve the space complexity of the key components of the data structure.

► **Definition 8** (Bridges). Let i and j be two distinct positions in the string T with $i < j$ and let $h = lcp(i, j)$ and $h > 0$. Then, we call the tuple (i, j, h) a bridge. Moreover, we call h its height, i its left leg and j its right leg, and $lcp(T[i, n], T[j, n])$ its label.

The set of all bridges is denoted by \mathcal{B}_{all} . Next, we will define the set of special bridges \mathcal{B}_{spe} via the heavy-path decomposition of the suffix tree ST of the string T .

► **Definition 9** (Heavy Path Decomposition [20]). Let τ be a tree. The nodes in τ are categorised into light and heavy ones. The root node is light. Furthermore, for each node of τ , exactly one of its children is heavy and the others are light. The heavy child has the largest number of leaves in its subtree among the children (ties are broken arbitrarily). When all edges incoming to light nodes are deleted, τ is decomposed into (heavy) paths.

Recall that there is one-to-one correspondence between the suffixes of T and the leaves of ST. By ℓ_i we denote the leaf corresponding to $T[ISA[i], n]$.

► **Definition 10** (Special Bridges). Let u_i, u_j in ST be the children of the lowest common ancestor of ℓ_i and ℓ_j on the path to ℓ_i, ℓ_j , respectively. A bridge $(i, j, h) \in \mathcal{B}_{all}$ is special if it satisfies at least one of two following conditions.

1. u_i is a light node **and** $j = \min\{x \mid (i, x, h) \in \mathcal{B}_{all}\}$
2. u_j is a light node **and** $i = \max\{x \mid (x, j, h) \in \mathcal{B}_{all}\}$

The reason to introduce this definition is that we can express range LCP queries via the special bridges.

► **Lemma 11** (cf. [1]). Let \mathcal{B}_{spe} be the set of all special bridges. Then $|\mathcal{B}_{spe}| = O(n \log n)$ and for any α, β we have $rlcp(\alpha, \beta) = \max\{h \mid (i, j, h) \in \mathcal{B}_{spe} \text{ and } i, j \in [\alpha, \beta]\}$.

► **Definition 12** (cf. [1]). For $i, j \in \mathbb{N}$ and $h \in \mathbb{N}$, we define

$$\text{crightLeg}(i, h) = \begin{cases} \min\{x \mid (i, x, h) \in \mathcal{B}_{spe}\} & \text{if there exists a bridge } (i, \cdot, h) \in \mathcal{B}_{spe}; \\ +\infty, & \text{otherwise.} \end{cases}$$

$$\text{cleftLeg}(j, h) = \begin{cases} \max\{x \mid (x, j, h) \in \mathcal{B}_{spe}\} & \text{if there exists a bridge } (\cdot, j, h) \in \mathcal{B}_{spe}; \\ -\infty, & \text{otherwise.} \end{cases}$$

► **Lemma 13.** By maintaining a data structure of space $O(\frac{1}{\varepsilon}n(H_0(T) + 1))$ bits, we can answer $\text{crightLeg}(k, h)$ and $\text{cleftLeg}(k, h)$ queries in $O(n^\varepsilon)$ time.

Proof. By [1], we can reduce computing $\text{crightLeg}(k, h)$ and $\text{cleftLeg}(k, h)$ to four range successor/predecessor queries on the suffix array of T and standard operations on ST. The claim follows by Theorem 1. ◀

► **Lemma 14** (cf. [1]). Suppose that $(i, j, h) \in \mathcal{B}_{spe}$. Then, $\forall k \in [1, h - 1]$, there exists at least one $(i + k, \cdot, h - k) \in \mathcal{B}_{spe}$ such that $\text{crightLeg}(i + k, h - k) \in (i + k, j + k]$ and $(\cdot, h + k, h - k) \in \mathcal{B}_{spe}$ such that $\text{crightLeg}(i + k, h - k) \in (i + k, j + k]$.

Let S be a set of m weighted points in a $[1, n] \times [1, n]$ grid. A 2D-RMQ with input (a, b, a', b') asks to return the highest weighted point in S within the orthogonal region corresponding to $[a, b] \times [a', b']$. There is a data structure for this problem that uses $O(m \log n)$ bits of space and $O(\log^{1+\gamma} m)$ query time [4]. Let $\Delta = \log^2 n$ and $\mathcal{B}_{spe}^{x \pmod{\Delta}}$ be the set of special bridges of heights congruent to x modulo Δ . For some $\pi \in [0, \Delta - 1]$ we have $|\mathcal{B}_{spe}^{\pi \pmod{\Delta}}| = O(n/\log n)$ as a corollary of Lemma 11 and the Pigeonhole principle. We map each special bridge $(i, j, h) \in \mathcal{B}_{spe}^\pi$ into a 2D point (i, j) with weight h and maintain the 2D-RMQ data structure over these points.

Let (α^*, β^*, h^*) be the tallest special bridge, such that both $\alpha^*, \beta^* \in [\alpha, \beta]$. We query the 2D-RMQ structure and find the tallest bridge $(i', j', h') \in \mathcal{B}_{spe}^{\pi \pmod{\Delta}}$, such that $i', j' \in [\alpha, \beta]$.

▷ **Claim 15.** If $h^* \geq \pi$, then (i', j', h') is well-defined. Furthermore, we have $\tau \leq \text{rlcp}(\alpha, \beta) \leq \tau + \Delta$, where $\tau = \max\{\text{ilcp}(p, \alpha, \beta) \mid p \in (\beta - \Delta, \beta]\} \cup \{h'\}$.

Proof. If $\beta^* \in (\alpha, \beta - \Delta]$, then there is $h^* \in [h', h' + \Delta)$ by Lemma 14. Else, there is $h^* = \max\{\text{ilcp}(p, \alpha, \beta) : p \in (\beta - \Delta, \beta]\}$. ◀

Below, we consider two possible cases: $h^* < \pi$ and $\pi \geq h^*$. In the second case, by Claim 15, we can find τ such that $\tau \leq \text{rlcp}(\alpha, \beta) \leq \tau + \Delta$ in $O(\log^\gamma n + \Delta \cdot n^\varepsilon) = O(n^{\varepsilon'})$ time for some $\varepsilon' > \varepsilon$. We will now explain how to find the true value of h^* .

4.1 Case 1: $\pi \leq h^*$

By Claim 15, in this case we know the interval $[\pi + \Delta \cdot k, \pi + \Delta \cdot (k + 1))$ that contains all possible values for h^* . Therefore, to find the true value of h^* in this case it suffices to check, for each $h \in [\pi + \Delta \cdot k, \pi + \Delta \cdot (k + 1)]$ if there is a special bridge (i, j, h) with legs $i, j \in [\alpha, \beta]$.

Recall that \mathcal{B}_{spe}^h is the set of all special bridges with height h . For each $h \in [1, n]$, we maintain a separate structure that can answer whether there is a bridge of height h with legs in $[\alpha, \beta]$. For $k = 1, 2, \dots, |\mathcal{B}_{spe}^h|$, let $L_h[k]$ (resp., $R_h[k]$) denote the left leg (resp., right leg) of k -th bridge among all bridges in \mathcal{B}_{spe}^h in the ascending order of the left (resp., right) legs. Based on whether $h \equiv \pi \pmod{\Delta}$ or not, we have two subcases.

4.1.1 Subcase 1a: $h \equiv \pi \pmod{\Delta}$

Let $RL_h[k] = \text{crightLeg}(L_h[k], h)$ for all $k = 1, 2, \dots, |\mathcal{B}_{spe}^h|$. We maintain the y -fast trie [22] over L_h and a succinct range minimum query data structure [6, 17] over RL_h . The total space is $O(|\mathcal{B}_{spe}^{\pi \pmod{\Delta}}| \log n) = O(n)$ bits. We can then decide if there is a bridge (i, j, h) such that $i, j \in [\alpha, \beta]$ via the following steps:

- Find the smallest k such that $L_h[k] \geq \alpha$ using the y -fast trie;
- Find the index k corresponding to the smallest element in $RL_h[k, |\mathcal{B}_{spe}^h|]$ using a range minimum query;
- Find $\text{crightLeg}(L_h[k], h)$ and report YES if it is $\leq \beta$, and report NO otherwise.

The time complexity is $O(\log \log n + n^\varepsilon) = O(n^\varepsilon)$ by Lemma 13.

4.1.2 Subcase 1b: $h \not\equiv \pi \pmod{\Delta}$

Let q be the largest integer smaller than h that is congruent to $\pi \pmod{\Delta}$ and $z = (h - q)$. Note that for each special bridge (i, j, h) , there exists at least one special bridge $(i + z, j', h - z) = (i + z, j', q)$ and $(i', j + z, h - z) = (i', j + z, q)$ (see Lemma 14).

Now, define arrays RL_h and LR_h of length $|\mathcal{B}_{spe}^q|$, such that for any $k = 1, 2, \dots, \mathcal{B}_{spe}$, $RL_h[k] = \text{crightLeg}(L_q[k] - z, h)$ and $LR_h[k] = \text{cleftLeg}(R_q[k] - z, h)$. To handle Subcase 1b, we maintain y -fast tries over L_q and R_q , and a range minimum query data structure over RL_h and a range maximum query data structure over LR_h for each $h \not\equiv \pi \pmod{\Delta}$.

► **Lemma 16.** *The range minimum query data structures over RL_h and the range maximum query data structures over LR_h , for all $h \not\equiv \pi \pmod{\Delta}$, can be implemented in $O(1/\varepsilon \cdot n(H_0(T) + 1))$ bits of space with query time $O(n^\varepsilon)$.*

Proof. We show how to implement the data structures for the arrays RL_h , for the arrays LR_h they can be implemented analogously.

First, we store the data structure of Lemma 13. Second, for each h , we divide the array RL_h into non-overlapping blocks of length $\log n$. For each block, we compute the minimum value in it to obtain an array RL'_h of length $|\mathcal{B}_{spe}^q|/\log n$. On top of RL'_h , we maintain a succinct range minimum query data structure. In total, the data structures occupy $O(1/\varepsilon \cdot n(H_0(T) + 1) + \Delta \cdot |\mathcal{B}_{spe}^{\pi \pmod{\Delta}}| \log n) = O(1/\varepsilon \cdot n(H_0(T) + 1))$ bits of space.

To answer a range minimum query on RL'_h , we first find the index of a block that contains the minimum, and then compute the value of each entry in the block using the data structure of Lemma 13. ◀

We can now decide if there is a bridge (i, j, h) such that $i, j \in [\alpha, \beta]$ via the following steps:

- Find the smallest k such that $L_q[k] - z \geq \alpha$ using the y -fast trie over L_q .
- Find the index k' corresponding to the smallest element in $RL_h[k, |\mathcal{B}_{spe}^q|]$ using a range minimum query.
- Return YES if $\text{crightLeg}(L_q[k'] - z, h) \leq \beta$, otherwise continue to the next step.
- Find the largest l such that $R_q[l] - z \leq \beta$. We use the y -fast trie for R_q for this.
- Find the index l' corresponding to the largest element in $LR_h[1, l]$ using a range maximum query.
- Return YES if $\text{cleftLeg}(R_q[l'] - z, h) \geq \alpha$, and return NO otherwise.

The time complexity is $O(n^\varepsilon \text{polylog } n)$, and the space complexity is $O(\frac{1}{\varepsilon} n(H_0(T) + 1))$ bits.

4.2 Case 2: $h^* < \pi$

In this case, we must check, for every $0 < h < \pi$, if there is a special bridge (i, j, h) , where $i, j \in [\alpha, \beta]$. We will use the same reduction as in Case 2 with $q = 1$.

Note that \mathcal{B}_{spe}^q can be of size $\Theta(n)$, so we cannot store the y -fast trie for L_q and R_q , it could take $\Theta(n \log n)$ bits of space. Recall, however, that we only use the y -fast tries to answer predecessor (resp., successor) queries: given an integer x , find the smallest (resp., the largest) index k such that $L_q[k]$ (resp., $R_q[k]$) is larger or equal (resp., smaller or equal) to x .

► **Lemma 17.** *The predecessor (resp., successor) queries on L_q (resp., R_q) can be answered in $O(n)$ bits of space and $O(1)$ query time.*

Proof. We show a data structure for L_q , a data structure for R_q can be defined analogously. Let L'_q be the sequence obtained by encoding the number of special bridges (i, j, q) with $i = k$ for all $k (1 \leq k \leq n)$ in unary code and concatenating them in order. $|L'_q| \leq 3n$ because there are at most $2n$ special bridges with height q by the definition. Therefore, to answer a predecessor query for an integer x it suffices to answer *rank* and *select* queries on L'_q : we return $rank_1(L'_q, select_0(L'_q, x)) + 1$. Both *rank* and *select* queries can be answered in $O(n)$ bits of space and $O(1)$ time. ◀

By using the data structures above, we can use our solution for Case 1 to obtain similar complexities.

References

- 1 Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, and Sharma V. Thankachan. A linear-space data structure for range-lcp queries in poly-logarithmic time. In *COCOON'18*, pages 615–625, 2018. doi:10.1007/978-3-319-94776-1_51.
- 2 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.
- 3 Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In *SPIRE'15*, pages 350–361, 2015. doi:10.1007/978-3-319-23826-5_33.
- 4 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.
- 5 Graham Cormode and S. Muthukrishnan. Substring compression problems. In *SODA'05*, pages 321–330, 2005.
- 6 Johannes Fischer and Volker Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE'07*, pages 459–470, 2007. doi:10.1007/978-3-540-74450-4_41.
- 7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *SODA'03*, page 841–850, 2003.
- 8 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. on Computing*, 35(2):378–407, 2005.
- 9 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.
- 10 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.10.010.

- 11 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.
- 12 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 13 Gonzalo Navarro and Kunihiko Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):Article No. 16, 39 pages, 2014.
- 14 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *SWAT'12*, pages 271–282, 2012.
- 15 Manish Patil, Rahul Shah, and Sharma V. Thankachan. Faster range LCP queries. In *SPIRE'13*, pages 263–270, 2013. doi:10.1007/978-3-319-02432-5_29.
- 16 Rajeev Raman, Venkatesh Raman, and Srinavasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007. doi:10.1145/1290672.1290680.
- 17 Kunihiko Sadakane. Space-efficient data structures for flexible text retrieval systems. In *ISAAC'02*, pages 14–24, 2002. doi:10.1007/3-540-36136-7_2.
- 18 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms*, 48(2):294–313, 2003.
- 19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- 20 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.
- 21 Peter Weiner. Linear pattern matching algorithms. In *SWAT(FOCS)'73*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.
- 22 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Text Indexing and Searching in Sublinear Time

J. Ian Munro

Cheriton School of Computer Science, University of Waterloo, Canada
imunro@uwaterloo.ca

Gonzalo Navarro

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

Yakov Nekrich

Department of Computer Science, Michigan Technological University, Houghton, MI, USA
yakov.nekrich@gmail.com

Abstract

We introduce the first index that can be built in $o(n)$ time for a text of length n , and can also be queried in $o(q)$ time for a pattern of length q . On an alphabet of size σ , our index uses $O(n \log \sigma)$ bits, is built in $O(n \log \sigma / \sqrt{\log n})$ deterministic time, and computes the number of occurrences of the pattern in time $O(q / \log_\sigma n + \log n \log_\sigma n)$. Each such occurrence can then be found in $O(\log n)$ time. Other trade-offs between the space usage and the cost of reporting occurrences are also possible.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases data structures, string indexes

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.24

Funding *J. Ian Munro*: Funded with Canada Research Chairs Programme and NSERC Discovery Grant.

Gonzalo Navarro: Funded with Basal Funds FB0001, Conicyt, Chile.

1 Introduction

We address the problem of indexing a text $T[0..n-1]$, over alphabet $[0..\sigma-1]$, in *sublinear* time on a RAM machine of $w = \Theta(\log n)$ bits. This is not possible when we build a classical index (e.g., a suffix tree [42] or a suffix array [26]) that requires $\Theta(n \log n)$ bits, since just writing the output takes time $\Theta(n)$. It is also impossible when $\log \sigma = \Theta(\log n)$ and thus just reading the $n \log \sigma$ bits of the input text takes time $\Theta(n)$. On smaller alphabets (which arise frequently in practice, for example on DNA, protein, and letter sequences), sublinear-time indexing becomes possible when the text comes packed in words of $\log_\sigma n$ characters and we build a *compressed* index that uses $o(n \log n)$ bits. For example, there exist various indexes that use $O(n \log \sigma)$ bits [35] (which is asymptotically the best worst-case size we can expect for an index on T) and could be built, in principle, in time $O(n / \log_\sigma n)$. Still, only linear-time indexing in compressed space had been achieved [3, 6, 30, 32] until the very recent result of Kempa and Kociumaka [24].

When the alphabet is small, one may also aim at RAM-optimal pattern search, that is, count the number of occurrences of a (packed) string $Q[0..q-1]$ in T in time $O(q / \log_\sigma n)$. There exist some classical indexes using $O(n \log n)$ bits and counting in time $O(q / \log_\sigma n + \text{polylog}(n))$ [36, 11], as well as compressed ones [32].

In this paper we introduce the first index that can be *built and queried in sublinear time*. Our index, as explained, is compressed. It uses $O(n \log \sigma)$ bits and can be constructed in deterministic time $O(n \log \sigma / \sqrt{\log n})$. Thus the construction time is $O(n / \sqrt{\log n})$ when the



© J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 24; pp. 24:1–24:15

Leibniz International Proceedings in Informatics



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

■ **Table 1** Previous and our results for index construction on a text of length n and a search pattern of length q , over an alphabet of size σ , on a RAM machine of w bits, for any constant $\varepsilon > 0$. Grayed rows are superseded by a more recent result in all aspects we consider. Note that $O(n)$ -time randomized construction can be replaced by $O(n(\log \log n)^2)$ deterministic constructions [39].

Source	Construction time	Space (bits)	Query time (counting)
Classical [42, 27, 41, 19]	$O(n)$	$O(n \log n)$	$O(q \log \sigma)$
Cole et al. [17]	$O(n)$	$O(n \log n)$	$O(q + \log \sigma)$
Fischer & Gawrychowski [21]	$O(n)$	$O(n \log n)$	$O(q + \log \log \sigma)$
Bille et al. [11]	$O(n)$	$O(n \log n)$	$O(q/\log_\sigma n + \log q + \log \log \sigma)$
Classical + perfect hashing	$O(n)$ randomized	$O(n \log n)$	$O(q)$
Navarro & Nekrich [36]	$O(n)$ randomized	$O(n \log n)$	$O(q/\log_\sigma n + \log_\sigma^\varepsilon n)$
Barbay et al. [3]	$O(n)$	$O(n \log \sigma)$	$O(q \log \log \sigma)$
Belazzougui & Navarro [6]	$O(n)$	$O(n \log \sigma)$	$O(q(1 + \log_w \sigma))$
Munro et al. [30, 29]	$O(n)$	$O(n \log \sigma)$	$O(q + \log \log \sigma)$
Munro et al. [32]	$O(n)$	$O(n \log \sigma)$	$O(q + \log \log_w \sigma)$
Munro et al. [32]	$O(n)$	$O(n \log \sigma)$	$O(q/\log_\sigma n + \log_\sigma^\varepsilon n)$
Belazzougui & Navarro [6]	$O(n)$ randomized	$O(n \log \sigma)$	$O(q(1 + \log \log_w \sigma))$
Belazzougui & Navarro [5]	$O(n)$ randomized	$O(n \log \sigma)$	$O(q)$
Kempa and Kociumaka [24]	$O(n \log \sigma / \sqrt{\log n})$	$O(n \log \sigma)$	$O(q(1 + \log_w \sigma))$
Ours	$O(n \log \sigma / \sqrt{\log n})$	$O(n \log \sigma)$	$O(q/\log_\sigma n + \log n \cdot \log_\sigma n)$

alphabet size is a constant. Our index also supports counting queries in $o(q)$ time: it counts in optimal time plus an additive poly-logarithmic penalty, $O(q/\log_\sigma n + \log n \log_\sigma n)$. After counting the occurrences of Q , any such occurrence can be reported in $O(\log n)$ time.

A slightly larger and slower-to-build variant of our index uses $O(n(\sqrt{\log n \log \sigma} + \log \sigma \log^\varepsilon n))$ bits for any constant $0 < \varepsilon < 1/2$ and is built in time $O(n \log^{3/2} \sigma / \log^{1/2-\varepsilon} n)$. This index can report the occ pattern occurrences in time $O(q/\log_\sigma n + \sqrt{\log_\sigma n \log \log n} + \text{occ})$.

As a comparison (see Table 1), the other indexes that count in time $O(q/\log_\sigma n + \text{polylog}(n))$ use either more space ($O(n \log n)$ bits) and/or construction time ($O(n)$) [11, 36, 32]. The indexes using less space, on the other hand, use as little as $O(n \log \sigma)$ bits but are slower to build and/or to query [30, 29, 32, 3, 5, 6, 24]. A recent construction [24] is the only one able to build in sublinear time ($O(n \log \sigma / \sqrt{\log n})$) and to use compressed space ($O(n \log \sigma)$ bits), just like ours, but it is still unable to search in $o(q)$ time.

Those compressed indexes can then deliver each occurrence in $O(\log^\varepsilon n)$ time, or even in $O(1)$ time if a structure of $O(n \log^{1-\varepsilon} \sigma \log^\varepsilon n)$ further bits is added, though there is no sublinear-time construction for those extra structures either [38, 22].

Our technique is reminiscent to the Geometric BWT [15], where a text is sampled regularly, so that the sampled positions can be indexed with a suffix tree in sublinear space. In exchange, all the possible alignments of the pattern and the samples have to be checked in a two-dimensional range search data structure. To speed up the search, we use a data structure for LCE queries. An LCE data structure enables us to compute in constant time the longest common prefix of any two text positions. Using this information we can efficiently find the locus of each alignment from the previous one.

2 Preliminaries and LCE Queries

We denote by $|S|$ the number of symbols in a sequence S or the number of elements in a set S . For two strings X and Y , $LCP(X, Y)$ denotes the longest common prefix of X and Y . For a string X and a set of strings \mathcal{S} , $LCP(X, \mathcal{S}) = \max_{Y \in \mathcal{S}} LCP(X, Y)$, where we

compare lengths to take the maximum. We assume that the concepts associated with suffix trees [42] are known. The longest common extension (LCE) query on S asks for the length of the longest common prefix of suffixes $S[i..]$ and $S[j..]$, $LCE(i, j) = |LCP(S[i..], S[j..])|$. LCE queries were introduced by Landau and Vishkin [25]. Several recent publications demonstrate that LCE data structures can use $o(n)$ space and/or can be constructed in $o(n)$ time [40, 31, 24, 12]. The following result will play an important role in our construction.

► **Lemma 1.** [24] *Given a text T of length n over an alphabet of size σ , we can build an LCE data structure using $O(n \log \sigma)$ bits of space in $O(n / \log_\sigma n)$ time. This data structure supports LCE queries on T in $O(1)$ time.*

3 The General Approach

We divide the text $T[0..n-1]$, over alphabet $[0..\sigma-1]$, into *blocks* of $r = O(\log_\sigma n)$ consecutive symbols (to avoid tedious details, we assume that both r and $\log_\sigma n$ are integers and that n is divisible by both). The set \mathcal{S}' consists of all the suffixes starting at positions ir , for $i = 0, 1, \dots, n/r - 1$; these are called *selected* positions. Our data structure consists of the following three components.

1. The suffix tree \mathcal{T}' for the suffixes starting at the selected positions, using $O((n/r) \log n)$ bits. Thus \mathcal{T}' is a compacted trie for the suffixes in \mathcal{S}' . Suffixes are represented as strings of meta-symbols where every meta-symbol corresponds to a substring of $\log_\sigma n$ consecutive symbols. Deterministic dictionaries are used at the nodes to descend by the meta-symbols in constant time. Predecessor structures are also used at the nodes, to descend when less than a metasymbol of the pattern is left. Given a pattern Q , we can identify all selected suffixes starting with Q in $O(|Q| / \log_\sigma n)$ time, plus an $O(\log \log n)$ additive term coming from the predecessor operations at the deepest node.
2. A data structure on a set \mathcal{Q} of points. Each point of \mathcal{Q} corresponds to a pair (ind_i, rev_i) for $i = 1, \dots, (n/r) - 1$ where ind_i is the index of the i -th selected suffix of T in the lexicographically sorted set \mathcal{S}' and rev_i is an integer that corresponds to the reverse block preceding that i -th selected suffix in T . Our data structure supports two-dimensional range counting and reporting queries on \mathcal{Q} .
3. A data structure for *suffix jump* queries on \mathcal{T}' . Given a string $Q[0..q-1]$, its locus node u , and a positive integer $i \leq r - 1$, a (suffix) i -jump query returns the locus node of $Q[i..q-1]$, or it says that $Q[i..q-1]$ does not prefix any string in \mathcal{S}' . The suffix jump structure has essentially the same functionality as the suffix links, but we do not store suffix links explicitly in order to save space and improve the construction time.

As described, \mathcal{T}' is a compact trie over an alphabet of meta-symbols corresponding to strings of length $\log_\sigma n$. Therefore, whenever we speak of a *node* $u \in \mathcal{T}'$, we refer indistinctly to an explicit or an implicit node (i.e., in the middle of an edge, coming from compacting a unary path). Further, we cannot then properly speak of the “locus node” of a string Q , even if we identify meta-symbols with their forming strings, because $|Q|$ might not be a multiple of $\log_\sigma n$. Rather, the *locus of Q* will be denoted $u[l..s]$, where $u \in \mathcal{T}'$, called its *locus node*, is the deepest node whose string label is a prefix of Q and $[l..s]$ is the maximal interval such that the string labels of the children u_l, \dots, u_s of u are prefixed by Q .

Using our structure, we can find all the occurrences in T of a pattern $Q[0..q-1]$ whenever $q > r$. Occurrences of Q are classified according to their positions relative to selected symbols. An occurrence $T[f..f+q-1]$ of Q is an i -occurrence if $T[f+i]$ (corresponding to the i -th symbol of Q) is the leftmost selected symbol in $T[f..f+q-1]$.

First, we identify all 0-occurrences by looking for Q in \mathcal{T}' : We traverse the path corresponding to Q in \mathcal{T}' to find $Q_0 = LCP(Q, \mathcal{S}')$, the longest prefix of Q that is in \mathcal{T}' , with locus $u_0[l_0..s_0]$. Let $q_0 = |Q_0|$; if $q_0 = q$, then $u_0[l_0..s_0]$ is the locus of Q and we count or report all its 0-occurrences as the positions of suffixes in the subtrees of $u_0[l_0..s_0]$.¹ If $q_0 < q$, there are no 0-occurrences of Q .

Next, we compute a 1-jump from u_0 to find the locus of $Q_0[1..] = Q[1..q_0 - 1]$ in \mathcal{T}' . If the locus does not exist, then there are no 1-occurrences of Q . If it exists, we traverse the path in \mathcal{T}' for Q_1 starting from that locus, not redoing the path from the root. Let $Q_1 = Q[1..q_1 - 1] = LCP(Q[1..q - 1], \mathcal{S}')$ be the longest prefix of $Q[1..q - 1]$ found in \mathcal{T}' , with locus $u_1[l_1..s_1]$. If $q_1 < q$, then again there are no 1-occurrences of Q . If $q_1 = q$, then $u_1[l_1..s_1]$ is the locus of $Q[1..q - 1]$. In this case, every 1-occurrence of Q corresponds to an occurrence of Q_1 in T that is preceded by $Q[0]$. We can identify them by answering a two-dimensional range query $[ind_1, ind_2] \times [rev_1, rev_2]$ where ind_1 (ind_2) is the leftmost (rightmost) leaf in the subtrees of $u_1[l_1..s_1]$ and rev_1 (rev_2) is the smallest (largest) integer value of any reverse block that starts with $Q[0]$.

We proceed and consider i -occurrences for $i = 2, \dots, r-1$ using the same method. Suppose that we have already considered the possible j -occurrences of Q for $j = 0, \dots, i-1$, so we have computed all the loci $u_j[l_j..s_j]$ of $Q_j = Q[j..q_j - 1] = LCP(Q[j..q - 1], \mathcal{S}')$. Further, let $q'_j \leq q_j$ be j plus the string depth of u_j , measured in symbols. This is the maximum number of symbols we can read from Q_j so that we reach a node of \mathcal{T}' . Let t be such that $q'_t = \max(q'_0, \dots, q'_{i-1})$. We then compute the $(i-t)$ -jump from u_t . If $Q[i..q'_t - 1]$ is not found in \mathcal{T}' , then it is enough for us to know that $q_i < q'_t$ without actually finding the locus of Q_i . If $Q[i..q'_t - 1]$ is found with locus node u , we traverse from u downwards to complete the path for $Q[i..q - 1]$. We then find the locus $u_i[l_i..s_i]$ of $Q[i..q_i - 1] = LCP(Q[i..q - 1], \mathcal{S}')$. If $q_i = q$, then $Q[i..q - 1]$ is found, so we count or report all i -occurrences by answering a two-dimensional query as described above.

Analysis. The total query time is $O(q/\log_\sigma n + r(\log \log n + t_q + t_s))$, where t_q and t_s are the times to answer a range query and to compute a suffix jump, respectively.

All the downward steps in the suffix tree amortize to $O(q/\log_\sigma n + r)$: we advance q'_t by $\log_\sigma n$ units in each downward step, but q'_t can be $(\log_\sigma n) - 1$ units less than the maximum position q_t we have reached up to now on Q (i.e., we take the suffix jump from u_t , whereas the actual locus with string depth q_t is $u_t[l_t..s_t]$). In addition we perform a predecessor step to find the ranges $[l_j..s_j]$ of the locus of each Q_j , which adds $O(r \log \log n)$ time. As said, the suffix tree (point 1) uses $O((n/r) \log n)$ bits.

The data structure of point 2 is a wavelet tree [14, 23, 34] built on $t = O(n/r)$ points. Its height is the logarithm of the y -coordinate range, $h = \log(\sigma^r) = O(r \log \sigma)$, and it uses $O(t \cdot h) = O(n \log \sigma) \subseteq O((n/r) \log n)$ bits. Such structure answers range counting queries in time $t_q = O(h) = O(r \log \sigma)$, thus $r \cdot t_q = O(r^2 \log \sigma)$, and reports each point in the range in time $O(h) = O(r \log \sigma)$.

In Sections 4 and 5 we show how to implement all the r suffix jumps (point 3) in time $r \cdot t_s = O(q/\log_\sigma n + r \log \log n)$, with a structure that uses $O((n/r) \log n)$ further bits.

Section 6 shows that the deterministic construction time of the structures of point 1 is $O(n(\log \log n)^2/r)$ and of point 3 is $O(n/r)$. The wavelet tree of point 2 can be built in time $O(t \cdot h/\sqrt{\log t}) = O(n \log \sigma/\sqrt{\log n})$ [33, 2].

¹ For fast counting, each node may also store the cumulative sum of its preceding siblings.

Finally, since a pattern shorter than r may not cross a block boundary and thus we could miss occurrences, Section 7 describes a special index for small patterns. Its space and construction time is within those of point 3 for $r \leq (1/4) \log_\sigma n$. This yields our first result.

► **Theorem 2.** *Let $0 < r < (1/4) \log_\sigma n$ be a parameter. Given a text T of length n over an alphabet of size σ , we can build an index using $O((n/r) \log n)$ bits in deterministic time $O(n((\log \log n)^2/r + \log \sigma/\sqrt{\log n}))$, so that it can count the number of occurrences of a pattern of length q in time $O(q/\log_\sigma n + r^2 \log \sigma + r \log \log n)$, and then report each such occurrence in time $O(r \log \sigma)$.*

If we set $r = \Theta(\log_\sigma n)$, we obtain a data structure with optimal asymptotic space usage.

► **Corollary 3.** *Given a text T of length n over an alphabet of size σ , we can build an index using $O(n \log \sigma)$ bits in deterministic time $O(n \log \sigma/\sqrt{\log n})$, so that it can count the number of occurrences of a pattern of length q in time $O(q/\log_\sigma n + \log n \log_\sigma n)$, and then report each such occurrence in time $O(\log n)$.*

We can improve the time of reporting occurrences by slightly increasing the construction time. Appendix A shows how to construct a range reporting data structure (point 2) that, after $t_q = O(\log \log n)$ time, can report each occurrence in constant time. The space of this structure is $O(n \log \sigma \log^\varepsilon n)$ bits and its construction time is $O((n \cdot r \cdot \log^2 \sigma)/\log^{1-\varepsilon} n)$, for any constant $0 < \varepsilon < 1/2$. If we plug in this range reporting data structure into our index (i.e., replacing point 2 above), we obtain our second result.

► **Theorem 4.** *Let $0 < r < (1/4) \log_\sigma n$ be a parameter. Given a text T of length n over an alphabet of size σ , we can build an index using $O((n/r) \log n + n \log \sigma \log^\varepsilon n)$ bits in deterministic time $O(n((\log \log n)^2/r + (r \log^2 \sigma)/\log^{1-\varepsilon} n))$, for any constant $0 < \varepsilon < 1/2$, so that it can count the occurrences of a pattern of length q in time $O(q/\log_\sigma n + r \log \log n)$, and then report each in $O(1)$ time.*

One interesting trade-off is when $r = \sqrt{\log_\sigma n}$. In this case the index uses $O(n(\sqrt{\log n \log \sigma} + \log \sigma \log^\varepsilon n))$ bits, can be constructed in $O((n \log^{3/2} \sigma)/\log^{1/2-\varepsilon} n)$ time, and reports the occurrences of a pattern of length q in time $O(q/\log_\sigma n + \sqrt{\log_\sigma n} \log \log n + \text{occ})$.

4 Suffix Jumps

Now we show how suffix jumps can be implemented. The solution described in this section takes $O(\log n)$ time per jump $O((n/r) \log n)$ extra bits of space; it is used when $|Q| \geq \log^3 n$. This already provides us with an optimal solution because, in this case, the time of the r suffix jumps, $O(\log n \log_\sigma n)$, is subsumed by the time $O(q/\log_\sigma n)$ to traverse the pattern. In the next section we describe an appropriate method for short patterns.

Given a substring $Q_t[0..q_t - 1]$ of the original query Q , with known locus $u_t[l_t..s_t]$, we find the locus $v[l..s]$ of $Q_t[i..]$ or determine that it does not exist.

We compute the locus of $Q_t[i..]$ by applying Lemma 1 $O(\log n)$ times; note that we know the text position f_1 of an occurrence of Q_t because we know its locus $u_t[l_t..s_t]$ in \mathcal{T}' ; therefore $Q_t[i..] = T[f_1 + i..]$. By binary search among the sampled suffixes (i.e., leaves of \mathcal{T}'), we identify in $O(\log n)$ time the suffix S_m that maximizes $|LCP(Q_t[i..], S_m)|$, because this measure decreases monotonically in both directions from S_m . At each step of the binary search we compute $\ell = |LCP(Q_t[i..], S)|$ for some suffix $S \in \mathcal{S}'$ using Lemma 1 and compare their $(\ell + 1)$ th symbols to decide the direction of the binary search. Once S_m is obtained we find, again with binary search, the smallest and largest suffixes $S_1, S_2 \in \mathcal{S}'$ such that $|LCP(S_1, S_m)| = |LCP(S_2, S_m)| = |LCP(Q_t[i..], S_m)|$; note $S_1 \leq S_m \leq S_2$.

Finally let v be the lowest common ancestor of the leaves that hold S_1 and S_2 in \mathcal{T}' . It then holds that $LCP(Q_t[i..], S') = LCP(Q_t[i..], S_m)$, and v is its locus node. Further, the locus is $v[l..s]$, where S_1 and S_2 descend by the l th and s th children of v , respectively (we can find l and s in $O(1)$ time with level ancestor queries on \mathcal{T}'). If $|LCP(S_m, Q_t[i..])| = q_t - i = |Q_t[i..]|$, then $v[l..s]$ is also the locus of $Q_t[i..]$; otherwise $Q_t[i..]$ prefixes no string in S' .

► **Lemma 5.** *Suppose that we know $Q_t[0..q_t - 1]$ and its locus in \mathcal{T}' . We can then compute $LCP(Q_t[i..q_t - 1], S')$ and its locus in \mathcal{T}' in $O(\log n)$ time, for any $0 \leq i \leq q_t - 1$.*

5 Suffix Jumps for Short Patterns

In this section we show how r suffix jumps can be computed in $O(|Q|/\log_\sigma n + r \log \log n)$ time when $|Q| \leq \log^3 n$. Our basic idea is to construct a set \mathcal{X}_0 of selected substrings with length up to $\log^3 n$. These are sampled at polylogarithmic-sized intervals from the sorted set S' . We also create a superset $\mathcal{X} \supset \mathcal{X}_0$ that contains all the substrings that could be obtained by trimming the first $i \leq r - 1$ symbols from strings in \mathcal{X}_0 . Using lexicographic naming and special dictionaries on \mathcal{X} , we pre-compute answers to all suffix jump queries for strings from \mathcal{X}_0 . We start by reading the query string Q and trying to match Q , $Q[1..]$, $Q[2..]$ in \mathcal{X}_0 . That is, for every $Q[i..q - 1]$ we find $LCP(Q[i..q - 1], \mathcal{X}_0)$ and its locus in \mathcal{T}' . With this information we can finish the computation of a suffix jump in $O(\log \log n)$ time, because the information on LCP s in \mathcal{X}_0 will narrow down the search in \mathcal{T}' to a polylogarithmic sized interval, on which we can use the binary search of Section 4.

Data Structure. Let S'' be the set obtained by sorting suffixes in S' and selecting every $(\log^{10} n)$ th suffix. We denote by \mathcal{X} the set of all substrings $T[i + f_1..i + f_2]$ such that the suffix $T[i..]$ is in the set S'' and $0 \leq f_1 \leq f_2 \leq \log^3 n$. We denote by \mathcal{X}_0 the set of substrings $T[i..i + f]$ such that the suffix $T[i..]$ is in the set S'' and $0 \leq f \leq \log^3 n$. Thus \mathcal{X}_0 contains all prefixes of length up to $\log^3 n$ for all suffixes from S'' and \mathcal{X} contains all strings that could be obtained by suffix jumps from strings in \mathcal{X}_0 .

We assign unique integer names to all substrings in \mathcal{X} : we sort \mathcal{X} and then traverse the sorted list assigning a unique integer $\text{num}(S)$ to each substring $S \in \mathcal{X}$. Our goal is to store pre-computed solutions to suffix jump queries. To this end, we keep three dictionaries:

- Dictionary D_0 contains the names $\text{num}(S)$ for all $S \in \mathcal{X}_0$, as well as their loci in \mathcal{T}' .
- Dictionary D contains the names $\text{num}(S)$ for all substrings $S \in \mathcal{X}$. For every entry $x \in D$, with $x = \text{num}(S)$, we store (1) the length $\ell(S)$ of the string S , (2) the length $\ell(S')$ and the name $\text{num}(S')$ where S' is the longest prefix of S satisfying $S' \in \mathcal{X}_0$, (3) for each j , $1 \leq j \leq r - 1$, the name $\text{num}(S[j..])$ of the string obtained by trimming the first j leading symbols of S if $S[j..]$ is in \mathcal{X} .
- Dictionary D_p contains $\text{num}(S\alpha)$ for all pairs (x, α) , where x is an integer and α is a string, such that the length of α is at most $\log_\sigma n$, $x = \text{num}(S)$ for some $S \in \mathcal{X}$, and the concatenation $S\alpha$ is also in \mathcal{X} . D_p can be viewed as a (non-compressed) trie on \mathcal{X} .

Using D_p , we can navigate among the strings in \mathcal{X} : if we know $\text{num}(S)$ for some $S \in \mathcal{X}$, we can look up the concatenation $S\alpha$ in \mathcal{X} for any string α of length at most $\log_\sigma n$. The dictionary D enables us to compute suffix jumps between strings in \mathcal{X} : if we know $\text{num}(S[0..])$ for some $S \in \mathcal{X}$, we can look up $\text{num}(S[i..])$ in $O(1)$ time.

The set S'' contains $O(\frac{n}{r \log^{10} n})$ suffixes. The set \mathcal{X} contains $O(\log^6 n)$ substrings for every suffix in S'' . The space usage of dictionary D is $O(n/\log^4 n)$ words, dominated by item (3). The space of D_p is $O(n \log_\sigma n / (r \log^4 n))$ words, given by the number of strings in \mathcal{X} times $\log_\sigma n$. This dominates the total space of our data structure, $O(n/\log^3 n)$ bits.

Suffix Jumps. Using the dictionary D , we can compute suffix jumps within \mathcal{X}_0 .

► **Lemma 6.** *For any string Q with $r \leq |Q| \leq \log^3 n$, we can find the strings $P_i = LCP(Q[i..], \mathcal{X}_0)$, their lengths p_i and their loci in \mathcal{T}' , for all $1 \leq i \leq r - 1$, in time $O(|Q|/\log_\sigma n + r \log \log_\sigma n)$.*

Proof. We find $P_0 = LCP(Q[0..q-1], \mathcal{X}_0)$ in $O(|P_0|/\log_\sigma n + \log \log_\sigma n)$ time: suppose that $Q[0..x]$ occurs in \mathcal{X}_0 . We can check whether $Q[0..x + \log_\sigma n]$ also occurs in \mathcal{X}_0 using the dictionaries D_p and D_0 . If this is the case, we increment x by $\log_\sigma n$. Otherwise we find with binary search, in $O(\log \log_\sigma n)$ time, the largest $f \leq \log_\sigma n$ such that $Q[0..x + f]$ occurs in \mathcal{X}_0 . Then $P_0 = Q[0..x + f] \in \mathcal{X}_0$, and its locus in \mathcal{T}' is found in D_0 .

When P_0 , of length $p_0 = |P_0|$, and its name $\text{num}(P_0)$ are known, we find $P_1 = LCP(Q[1..], \mathcal{X}_0)$: first we look up $v = \text{num}(P_0[1..])$ in component (3) of D , then we look up in component (2) of D the longest prefix of the string with name v that is in \mathcal{X}_0 . This is the 1-jump of P_0 in \mathcal{X}_0 ; now we descend as much as possible from there using D_p and D_0 , as done to find P_0 from the root. We finally obtain $\text{num}(P_1)$; its length p_1 and locus in \mathcal{T}' are found in D (component (1)) and D_0 , respectively.

We proceed in the same way as in Section 3 and find $LCP(Q[i..], \mathcal{X}_0)$ for $i = 2, \dots, r - 1$. The traversals in D_p amortize analogously to $O(|Q|/\log_\sigma n + r)$, and we have $O(r \log \log_\sigma n)$ further time to complete the r traversals. ◀

With all $LCP(Q[i..], \mathcal{X}_0)$ and their loci in \mathcal{T}' , we can compute suffix jumps in \mathcal{S}' .

► **Lemma 7.** *Suppose that we know $P_i = LCP(Q[i..q-1], \mathcal{X}_0)$ and its locus in \mathcal{T}' for all $0 \leq i \leq r - 1$. Assume we also know that $Q_t[0..q_t-1] = Q[t..t + q_t - 1]$ prefixes a string in \mathcal{S}' and its locus node $u_t \in \mathcal{T}'$. Then, given $j \leq r - 1$, we can compute $LCP(Q_t[j..], \mathcal{S}')$ and its locus in \mathcal{T}' , in $O(\log \log n)$ time.*

Proof. Let $v'[l'..s']$ be the locus of $LCP(Q_t[j..], \mathcal{X}_0) = LCP(Q[t + j..], \mathcal{X}_0)$ in \mathcal{T}' and let $\ell = |LCP(Q_t[j..], \mathcal{X}_0)|$. If $\ell = q_t - j$, then $v'[l'..s']$ is the locus of $Q_t[j..]$ in \mathcal{T}' . Otherwise let v_+ denote the child of v' in \mathcal{T}' that descends by $Q[t + j + \ell..t + j + \ell + \log_\sigma n - 1]$. If v_+ does not exist, then v' is the locus node v of $LCP(Q_t[j..], \mathcal{S}')$. We only have to find its children interval $[l..s]$ (which could expand $[l'..s']$) by a predecessor search on its children.

If v_+ exists, then the locus of $LCP(Q_t[j..], \mathcal{S}')$ is in the subtree \mathcal{T}_{v_+} of \mathcal{T}' rooted at v_+ . By definition, \mathcal{T}_{v_+} does not contain suffixes from \mathcal{X}_0 . Hence \mathcal{T}_{v_+} has $O(\log^{10} n)$ leaves. We then find $LCP(Q_t[j..], \mathcal{S}')$ among suffixes in \mathcal{T}_{v_+} using the binary search method described in Section 4: we find S_1, S_m , and S_2 in time $O(\log \log^{10} n) = O(\log \log n)$. The locus $v[l..s]$ of $LCP(Q_t[j..], \mathcal{S}')$ is then the lowest common ancestor of the leaves that hold S_1 and S_2 ; l and s are the children S_1 and S_2 descend from. ◀

► **Lemma 8.** *Suppose that $|Q| \leq \log^3 n$. Then we can find all the existing loci of $Q[i..]$ in \mathcal{T}' , for $0 \leq i \leq r - 1$, in time $O(|Q|/\log_\sigma n + r \log \log n)$, using $O(n/\log^3 n)$ bits of space.*

6 Construction

Sampled suffix tree. We can view T as a string \bar{T} of length n/r over an alphabet of size σ^r . Since \bar{T} consists of $O(n/r)$ meta-symbols and each meta-symbol fits in a $\Theta(\log n)$ -bit word, we can sort all meta-symbols in $O(n/r)$ time using RadixSort [18]. Thus we can generate \bar{T} and construct its suffix tree \mathcal{T}' in $O(n/r)$ time [19]. Further, we need $O((n/r)(\log \log n)^2)$ time to build the deterministic dictionaries and the predecessor data structures storing the children of each node [39, 4].

Suffix jumps. The lowest common ancestor and level ancestor structures [10, 8], which are needed in Section 4, are built in time $O(|\mathcal{T}'|) = O(n/r)$.

The sets of substrings and dictionaries D , D_0 , and D_p described in Section 5 can be constructed as follows. Let $m = O(n/r)$ be the number of selected suffixes in \mathcal{S}' . The number of suffixes in \mathcal{S}'' is $O(m/\log^{10} n)$. The number of substrings associated with each suffix in \mathcal{S}'' is $O(\log^6 n)$ and their total length is $O(\log^9 n)$. The total number of strings in \mathcal{X}_0 is $O(m/\log^7 n)$ and their total length is $O(\frac{m}{\log^{10} n} \cdot \log^6 n) = O(m/\log^4 n)$. The number of strings in \mathcal{X} is $k = O((m/\log^{10} n) \cdot \log^6 n) = O(m/\log^4 n)$ and their total length is $t = O((m/\log^{10} n) \cdot \log^9 n) = O(m/\log n)$. We can then collect all the strings $S \in \mathcal{X}$ from $T[i + f_1..i + f_2]$ for every sampled leaf of \mathcal{T}' pointing to $T[i]$, sort them in $O(t) = o(m)$ time with RadixSort (the metasymbols fit in $O(\log n)$ bits [18]), remove repetitions, and finally assign them lexicographic names $\text{num}(S)$. We keep a pointer to S in T for each $S \in \mathcal{X}$.

Next, we construct the dictionary D_0 that contains the names $\text{num}(S)$ of those $S \in \mathcal{X}_0$. For every $x = \text{num}(S)$ in D_0 we compute its locus $v[l..s]$ in \mathcal{T}' . The locus can be found in $O(|S|/\log_\sigma n + \log \log n)$ time by traversing \mathcal{T}' from the root. This adds up to $O(|\mathcal{X}_0| \log^3 n) = o(m)$ time. Finally, D_0 is a deterministic dictionary on the keys $\text{num}(S)$, so it can be constructed in $O(|\mathcal{X}_0|(\log \log n)^2) = o(m)$ deterministic time [39].

Similarly, D is a deterministic dictionary on k keys, which can be built in $O(k(\log \log n)^2) = o(m)$ time [39]. Since \mathcal{X} is prefix-closed, we can use the pointers to the strings S and the dictionary D_0 to determine the longest prefix $S' \in \mathcal{X}_0$ of S by binary search on $\ell(S')$, in $O(k \log \log n)$ total time. When we generate strings of \mathcal{X} , we also record the information about suffix jumps (e.g., we store a pointer from each S to $S[1..]$ before sorting them, so later we can obtain $\text{num}(S[1..])$ from S , then $\text{num}(S[2..])$ from $S[1..]$, and so on). We can then easily traverse those suffixes to compute all relevant suffix jumps for each string $S \in \mathcal{X}$, in total time $O(kr) = o(m)$. We then have items (1)–(3) for all the elements of D .

Finally, we construct the dictionary D_p by inserting all strings in \mathcal{X} into a trie data structure; at every node of this trie we store the name $\text{num}(S)$ of the corresponding string S . Once \mathcal{X} is sorted, the trie is easily built in $O(k)$ total time. Later, along a depth-first trie traversal we collect, for each node representing name y , its ancestors x up to distance $\log_\sigma n$ and the strings α separating x from y . All the pairs $(x, \alpha) \rightarrow y$ are then stored in D_p . Since \mathcal{X} is prefix-closed, the trie contains $O(k)$ nodes, and we include $O(k \log_\sigma n)$ pairs in D_p . Since D_p is also a deterministic dictionary, it can be built in time $O(k \log_\sigma n (\log \log n)^2) = o(m)$.

The total time to build the data structures for suffix jumps is then $O(n/r + m) = O(n/r)$.

Range searches. As said, the wavelet tree can be built in time $O(n \log \sigma / \sqrt{\log n})$ [33, 2]. Appendix A shows that the time to build the data structure for faster reporting is $O(n \cdot r \cdot \log^2 \sigma / \log^{1-\varepsilon} n)$, for any constant $0 < \varepsilon < 1/2$.

7 Index for Small Patterns

The data structure for small query strings consists of two tables. Assume $r \leq (1/4) \log_\sigma n$. We regard the text as an array $A[0..n/r]$ of length- $2r$ (overlapping) strings, $A[i] = T[ir..ir+2r-1]$. We build a table Tbl whose entries correspond to all strings of length $2r$: $Tbl[\alpha]$ lists all the positions i where $A[i] = \alpha$. Further, we build tables Tbl_j , for $1 \leq j \leq r$, containing all the possible length- j strings. Each entry $Tbl_j[\beta]$, with $|\beta| = j$, contains the list of length- $2r$ strings α such that $Tbl[\alpha]$ is not empty and β is a substring of α beginning within its first r positions (i.e., $\beta = \alpha[i..i+j-1]$ for some $0 \leq i < r$).

Table Tbl has $\sigma^{2r} = O(\sqrt{n})$ entries, and overall contains n/r pointers to A , thus its total space is $O((n/r) \log n)$ bits. Tables Tbl_j add up to $O(\sigma^r) = O(n^{1/4})$ cells. Since each distinct string α of length $2r$ produces $O(r^2)$ distinct substrings, there can be only $O(\sigma^{2r} r^2) = O(\sqrt{n} \log_\sigma^2 n)$ pointers in all the tables Tbl_j , for a total space of $o(n/r)$ bits.

To report the occurrences of $Q[0..q-1]$, we examine $Tbl_q[Q]$. For each string α in $Tbl_q[Q]$, we visit the entry $Tbl[\alpha]$ and report all the positions of $Tbl[\alpha]$ in A (with their offset).

To build Tbl , we can traverse A and add each i to the list of $Tbl[A[i]]$, all in $O(n/r)$ time. We then visit the slots of Tbl . For every α such that $Tbl[\alpha]$ is not empty, we consider all the sub-strings β of α starting within its first half and add α to $Tbl_{|\beta|}[\beta]$, recording also the corresponding offset of β in α (we may add the same α several times with different offsets). The time of this step is, as seen for the space, $O(\sigma^{2r} r^2) = O(\sqrt{n} \log_\sigma^2 n) = o(n/r)$.

To support counting, $Tbl_q[Q]$ also stores the number of occurrences in T of each string Q .

► **Lemma 9.** *There exists a data structure that uses $O((n/r) \log n)$ bits and reports all occurrences of a query string Q in $O(\text{occ})$ time if $|Q| \leq r$, with $r \leq (1/4) \log_\sigma n$. The data structure also computes occ in $O(1)$ time and can be built in time $O(n/r)$.*

8 Conclusion

We have described the first text index that can be built and queried in sublinear time. On a text of length n and alphabet of size σ , the index is built in $O(n \log \sigma / \sqrt{\log n})$ time, on a RAM machine of $\Theta(\log n)$ bits. This is sublinear for $\log \sigma = o(\sqrt{\log n})$. An index that is built in sublinear time must naturally use $o(n \log n)$ bits, hence our index is also compressed: our data structure has the asymptotically optimal space usage, $O(n \log \sigma)$ bits. Indeed, our index is the first one that simultaneously achieves three goals: sublinear construction time, asymptotically optimal space usage, and substring counting in nearly optimal time $O(q / \log_\sigma n + \log n \log_\sigma n)$ where q is the substring length. Previously described data structures with optimal (or even $O(n \log n)$) space usage either require $\Omega(n)$ construction time or $\Omega(q)$ time to count the occurrences of a substring.

We know no lower bound that prevents us from aiming at an index using the least possible space, $O(n \log \sigma)$ bits, the least possible construction time for this space in the RAM model, $O(n / \log_\sigma n)$, and the least possible counting time, $O(q / \log_\sigma n)$. Our index is the first one in breaking the $\Theta(n)$ construction time and $\Theta(q)$ query time barriers simultaneously, but it is open how close we can get to the optimal space and construction time.

References

- 1 S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997.
- 2 M. Babenko, P. Gawrychowski, T. Kociumaka, and T. Starikovskaya. Wavelet trees meet suffix trees. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 572–591, 2015.
- 3 J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, !PUBLISHER = "Springer", 69(1):232–268, 2014.
- 4 D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 159–172, 2010.
- 5 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):article 23, 2014.
- 6 D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

- 7 D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.
- 8 M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.
- 9 M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 88–94, 2000. doi:10.1007/10719839_9.
- 10 M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
- 11 P. Bille, I. L. Gørtz, and F. R. Skjoldjensen. Deterministic indexing for packed strings. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 6:1–6:11, 2017.
- 12 O. Birenzweig, S. Golan, and E. Porat. Locally consistent parsing for text indexing in small space. In *Proc. 31st ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 607–626, 2020.
- 13 T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.
- 14 B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988. doi:10.1137/0217026.
- 15 Y.-F. Chien, W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Geometric BWT: Compressed text indexing via sparse suffixes and range searching. *Algorithmica*, 71(2):258–278, 2015.
- 16 D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.
- 17 R. Cole, T. Kopelowitz, and M. Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015.
- 18 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- 19 M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
- 20 G. Feigenblat, E. Porat, and A. Shiftan. Linear time succinct indexable dictionary construction with applications. In *Proc. 26th Data Compression Conference (DCC)*, pages 13–22, 2016.
- 21 J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 160–171, !series = "LNCS 9133", 2015.
- 22 R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.
- 23 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
- 24 D. Kempa and T. 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.
- 25 G. M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.
- 26 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- 27 E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- 28 J. I. Munro. Tables. In *Proc. 16th FSTTCS*, pages 37–42, 1996.
- 29 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. *CoRR*, abs/1607.04346, 2016.

- 30 J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 408–424, 2017.
- 31 J. I. Munro, G. Navarro, and Y. Nekrich. Text indexing and searching in sublinear time. *CoRR*, abs/1712.07431, 2017.
- 32 J. I. Munro, G. Navarro, and Y. Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020.
- 33 J. I. Munro, Y. Nekrich, and J. S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016", !note = "Preliminary version appeared in SPIRE'14.
- 34 G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
- 35 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
- 36 G. Navarro and Y. Nekrich. Time-optimal top- k document retrieval. *SIAM Journal on Computing*, 46(1):89–113, 2017.
- 37 Y. Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.
- 38 S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.
- 39 M. Ruzic. Constructing efficient dictionaries in close to sorting time. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP A)*, pages 84–95 (part I), 2008, !series = "LNCS 5125".
- 40 Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In *Proc. 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 10:1–10:15, 2017.
- 41 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- 42 P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Symposium on Foundations on Computer Science (FOCS)*, pages 1–11, 1973.

A Range Reporting

In this section we prove a result on two-dimensional orthogonal range reporting queries. Our method builds upon previous work on wavelet tree construction [33, 2], applications of wavelet trees to range predecessor queries [7], and compact range reporting [14, 13].

► **Theorem 10.** *For a set of $t = O(n/r)$ points on a $t \times \sigma^{O(r)}$ grid, where $r \leq (1/4) \log_{\sigma} n$, and for any constant $0 < \varepsilon < 1/2$, there is an $O(n \log \sigma \log^{\varepsilon} n)$ -bit data structure that can be built in $O(n \cdot r \cdot \log^2 \sigma / \log^{1-\varepsilon} n)$ time and supports orthogonal range reporting queries in time $O(\log \log t + \text{pocc})$ where pocc is the number of reported points.*

A.1 Base data structure

We are given a set \mathcal{Q} of $t = O(n/r)$ points in $[0..t-1] \times [0..\sigma^{O(r)}]$. First we sort the points by x -coordinates (this is easily done by scanning the leaves of \mathcal{T}' , which are already sorted lexicographically by the selected suffixes), and keep the y -coordinates of every point in a sequence Y . Each element of Y can be regarded as a string of length $O(r)$ over an alphabet of size σ , or equivalently, an h -bit number where $h = O(r \log \sigma)$. Next we construct the range tree for Y using a method similar to the wavelet tree [23] construction algorithm. Let $Y(u_o) = Y$ for the root node u_o . We classify the elements of $Y(u_o)$ according to their highest bit and generate the corresponding subsequences of $Y(u_o)$, $Y(u_l)$ (highest bit zero) and $Y(u_r)$ (highest bit one), that must be stored in the left and right children of u , u_l and u_r , respectively. Then nodes u_l and u_r are recursively processed in the same

manner. When we generate the sequence for a node u of depth d , we assign elements to $Y(u_l)$ and $Y(u_r)$ according to their d -th highest bit. We can exploit bit parallelism and pack $(\log n)/h$ y -coordinates into one word; therefore we can produce $Y(u_l)$ and $Y(u_r)$ from $Y(u)$ in $O(|Y(u)| \cdot h/\log n)$ time. The total time needed to generate all sequences $Y(u)$ is $O(t \cdot h \cdot (h/\log n)) = O((n \cdot r \cdot \log^2 \sigma)/\log n)$.

For every sequence $Y(u)$ we also construct an auxiliary data structure that supports three-sided queries. If u is a right child, we create a data structure that returns all elements in a range $[x_1, x_2] \times [0, h]$ stored in $Y(u)$. To this end, we divide $Y(u)$ into groups $G_i(u)$ of $g = (1/2) \log n$ consecutive elements (the last group may contain up to $2g$ elements). Let $\min_i(u)$ denote the smallest element in every group and let $Y'(u)$ denote the sequence of all $\min_i(u)$. We construct a data structure that supports three-sided queries on $Y'(u)$; it uses $O(|Y'(u)| \log n) = O((|Y(u)|/g) \log n) = O(|Y(u)|)$ bits and reports the k output points in $O(\log \log n + k)$ time; we can use any range minimum data structure for this purpose [9]. We can traverse $Y(u)$ and identify the smallest element in each group in $O(|Y(u)|h/\log n)$ time, by using small precomputed tables that process $(\log n)/2$ bits in constant time. This adds up to $O(t \cdot h^2/\log n) = O(n \cdot r \cdot \log^2 \sigma/\log n)$ time.

Since the number of points in $Y'(u)$ is $O(|Y(u)|/g)$, the data structure for $Y'(u)$ can be created in $O(|Y(u)|/g)$ time and uses $O((|Y(u)|/g) \log n) = O(|Y(u)|)$ bits, which adds up to $O((n \log \sigma)/\log n)$ construction time and $O(n \log \sigma)$ bits of space.

In order to save space, we do not store the y -coordinates of points in a group. The y -coordinate of each point in $G = G_i(u)$ is replaced with its rank, that is, with the number of points in G that have smaller y -coordinates. Each group G is divided into $(\log \sigma)/(2 \log \log n)$ subgroups, so that each subgroup contains $2r \log \log n$ consecutive points from G . We keep the rank of the smallest point from each subgroup of G in a sequence G^t . Since the ranks of points in a group are bounded by g and thus can be encoded with $\log g \leq \log \log n$ bits, each subgroup can be encoded with less than $2r(\log \log n)^2$ bits. Hence we can store precomputed answers to all possible range minimum queries on all possible subgroups in a universal table of size $O(2^{2r(\log \log n)^2} \log^2 g) = o(n)$ bits. We can also store pre-computed answers for range minima queries on G^t using another small universal table: G^t is of length $(\log \sigma)/(2 \log \log n)$ and the rank of each minimum is at most g , so G^t can be encoded in at most $(\log \sigma)/2$ bits. This second universal table is then of size $O(2^{(\log \sigma)/2} \log^2 g) = o(n)$ bits.

A three-sided query $[x_1, x_2] \times [0, y]$ on a group G can then be answered as follows. We identify the point of smallest rank in $[x_1, x_2]$. This can be achieved with $O(1)$ table look-ups because a query on G can be reduced to one query on G^t plus a constant number of queries on sub-groups. Let x' denote the position of this smallest-rank point in $Y(u)$. We obtain the real y -coordinate of $Y(u)[x']$ using the translation method that will be described below. If the real y -coordinate of $Y(u)[x']$ does not exceed y , we report it and recursively answer three-sided queries $[x_1, x' - 1] \times [0, y]$ and $[x' + 1, x_2] \times [0, y]$. The procedure continues until all points in $[x_1, x_2] \times [0, y]$ are reported.

If u is a left child, we use the same method to construct the data structure that returns all elements in a range $[x_1, x_2] \times [y, +\infty)$ from $Y(u)$.

An orthogonal range reporting query $[x_1, x_2] \times [y_1, y_2]$ is then answered by finding the lowest common ancestor v of the leaves that hold y_1 and y_2 . Then we visit the right child v_r of v , identify the range $[x'_1, x'_2]$ and report all points in $Y(v_r)[x'_1..x'_2]$ with y -coordinates that do not exceed y_2 ; here x'_1 is the index of the smallest x -coordinate in $Y(v_r)$ that is $\geq x_1$ and x'_2 is the index of the largest x -coordinate of $Y(v_r)$ that is $\leq x_2$. We also visit the left child v_l of v , and answer the symmetric three-sided query. Finding x'_1 and x'_2 requires predecessor and successor queries on x -coordinates of any $Y(v_r)$; the needed data structures are described in Section A.3.

In total, the basic part of the data structure requires $O(n \log \sigma)$ bits of space and is built in time $O((n \cdot r \log^2 \sigma) / \log n)$.

A.2 Translating the answers

An answer to our three-sided query returns positions in $Y(v_l)$ (resp. in $Y(v_r)$). We need an additional data structure to translate such local positions into the points to be reported. While our wavelet tree can be used for this purpose, the cost of decoding every point would be $O(h)$. A faster decoding method [14, 37, 13] enables us to decode each point in $O(1)$ time. Below we describe how this decoding structure can be built within the desired time bounds.

Let us choose a constant $0 < \varepsilon < 1/2$ and, to simplify the description, assume that $\log_\sigma^\varepsilon n$ and $\log \sigma$ are integers. We will say that a node u is an x -node if the height of u is divisible by x . For an integer x the x -ancestor of a node v is the lowest ancestor w of v , such that w is an x -node. Let $d_k = h^{k\varepsilon}$ for $k = 0, 1, \dots, \lceil 1/\varepsilon \rceil$. We construct sequences $UP(u)$ in all nodes u . $UP(u)$ enables us to move from a d_k -node to its d_{k+1} -ancestor: Let k be the largest integer such that u is a d_k -node and let v be the d_{k+1} -ancestor of u . We say that $Y(u)[i]$ corresponds to $Y(v)[j]$ if $Y(u)[i]$ and $Y(v)[j]$ represent the y -coordinates of the same point. Suppose that a three-sided query has returned position i in $Y(u)$. Using auxiliary structures, we find the corresponding position i_1 in the d_1 -ancestor u_1 of u . Then we find i_2 that corresponds to i_1 in the d_2 -ancestor u_2 of u_1 . We continue in the same manner, at the k -th step moving from a d_k -node to its d_{k+1} -ancestor. After $O(1/\varepsilon)$ steps we reach the root node of the range tree.

It remains to describe the auxiliary data structures. To navigate from a node v to its ancestor u , v stores for every i in $Y(v)$ the corresponding position i' in $Y(u)$ (i.e., $Y(v)[i]$ and $Y(u)[i']$ are y -coordinates of the same point). In order to speed up the construction time, we store this information in two sequences. The sequence $Y(u)$ is divided into chunks; if u is a d_k -node, then the size of the chunk is $\Theta(2^{d_k})$. For every element in $Y(v)$ we store information about the chunk of its corresponding position in $Y(u)$ using the binary sequence $C(v)$: $C(v)$ contains a 1 for every element $Y(v)[i]$ and a 0 for every chunk in $Y(u)$ (0 indicates the end of a chunk). We store in $UP(v)[i]$ the relative value of its corresponding position in $Y(u)$. That is, if the element of $Y(u)$ that corresponds to $Y(v)[i]$ is in the j th chunk of $Y(u)$, then it is at $Y(u)[j \cdot 2^{d_k} + UP(v)[i]]$. In order to move from $Y(v)[i]$ in a node v to the corresponding position $Y(u)[i_k]$ in its d_k -ancestor u , we compute the target chunk in $Y(u)$, $j = \text{select}_1(C(v), i) - i$, and set $i_k = j \cdot 2^{d_k} + UP(v)[i]$. Here select_1 finds the i th 1 in $C(v)$, and can be computed in constant time using $o(|C(v)|)$ bits on top of $C(v)$ [16, 28].

Since the tree contains h/d_{k-1} levels of t d_{k-1} -nodes, and the $UP(v)$ sequences of d_{k-1} -nodes v store numbers up to 2^{d_k} , the total space used by all $UP(v)$ sequences for all d_{k-1} -nodes v is $O(t \cdot (h/d_{k-1}) \cdot d_k) = O(t \cdot h^{1+\varepsilon})$ bits, because $d_k/d_{k-1} = h^\varepsilon$. For any such node v , with d_k -ancestor u , the total number of bits in $C(v)$ is $|Y(v)| + |Y(u)|/2^{d_k}$. There are at most 2^{d_k} nodes v with the same d_k -ancestor u . Hence, summing over all d_{k-1} -nodes v , all $C(v)$ s use $t(h/d_{k-1}) + t(h/d_k) = O(t(h/d_{k-1}))$ bits. These structures are stored for all values $k-1 \in \{0, \dots, \lceil 1/\varepsilon \rceil - 1\}$. Summing up, all sequences $C(v)$ use $O(t \cdot h)$ bits. The total space needed by auxiliary structures is then $O(t \cdot h^{1+\varepsilon}) = O(n \log^{1+\varepsilon/2} \sigma \log^{\varepsilon/2} n)$ bits, dominated by the sequences $UP(v)$. This can be written as $O(n \log \sigma \log^\varepsilon n)$ bits.

To produce the auxiliary structures, we need essentially that each d_k -node u distributes its positions in the corresponding $C(v)$ and $UP(v)$ structures in each of the next $h^\varepsilon - 1$ levels of d_{k-1} -nodes below u . Precisely, there are $2^{l \cdot d_{k-1}}$ d_{k-1} -nodes v at distance $l \cdot d_{k-1}$ from u , and we use $l \cdot d_{k-1}$ bits from the coordinates in $Y(u)[i]$ to choose the appropriate node v where $Y(u)[i]$ belongs. Doing this in sublinear time, however, requires some care.

24:14 Text Indexing and Searching in Sublinear Time

Let us first consider the root u , the only d_k -node for $k = \lceil 1/\varepsilon \rceil$. We consider all the d_{k-1} -nodes v (thus, u is their only d_k -ancestor). These are nodes of height $l \cdot d_{k-1}$ for $l = 1, 2, \dots, h^\varepsilon - 1$. In order to construct sequences $UP(v)$ in all nodes v on level $l \cdot d_{k-1}$ for a fixed l , we proceed as follows. The sequence $Y[u]$ is divided into chunks, so that each chunk contains 2^h consecutive elements. The elements $Y(u)[i]$ within each chunk are sorted with key pairs $(\text{bits}((h^\varepsilon - l) \cdot d_{k-1}, Y(u)[i]), \text{pos}(i, u))$ where $\text{pos}(i, u) = i \bmod 2^h$ is the relative position of $Y(u)[i]$ in its chunk and $\text{bits}(\ell, x)$ is the number that consists of the highest ℓ bits of x . We sort integer pairs in the chunk using a modification of the algorithm of Albers and Hagerup [1, Thm. 1] that runs in $O(2^h \frac{h^2}{\log n})$ time. Our modified algorithm works in the same way as the second phase of their algorithm, but we merge words in $O(1)$ time. Merging can be implemented using a universal look-up table that uses $O(\sqrt{n})$ words of space and can be initialized in $O(\sqrt{n} \log^3 n)$ time.

We then traverse the chunks and generate the sequences $UP(v)$ and $C(v)$ for all the nodes v on level $l \cdot d_{k-1}$. For each bit string of length $l \cdot d_{k-1}$, we say that v is the q -descendant of u if the path from u to v is labeled with q . The sorted list of pairs for each chunk of u is processed as follows. All the pairs $(q, \text{pos}(i, u))$ (i.e., $q = \text{bits}((h^\varepsilon - l)d_{k-1}, Y(u)[i])$) are consecutive after sorting, so we scan the list identifying the group for each value of q ; let $n(q)$ be its number of pairs. Precisely, the points with value q must be stored at the q -descendant v of u (the consecutive values of q correspond, left-to-right, to the nodes v on level $l \cdot d_{k-1}$). For each group q , then, we identify the q -descendant v of u and append $n(q)$ 1-bits and one 0-bit to $C(v)$. We also append $n(q)$ entries to $UP(v)$ with the contents $\text{pos}(i, u)$, in the same order as they appear in the chunk of u .

We need time $O(2^h \cdot h / \log n)$ to generate the pairs $(\text{bits}(\cdot), \text{pos}(\cdot))$ for the 2^h coordinates of each chunk, and to store the pairs in compact form, that is, $O(\log(n)/h)$ pairs per word. We can then sort the chunks in time $O(2^h \cdot h^2 / \log n)$. We can generate the parts of sequences $C(v)$ and $UP(v)$ that correspond to a chunk for all nodes v on level $l \cdot d_{k-1}$ in $O(2^h + 2^h \cdot h / \log n) = O(2^h)$. Thus the total time needed to generate $UP(v)$ and $C(v)$ for all nodes v on level $l \cdot d_{k-1}$ and some fixed l is $O(t \log \sigma)$, where we remind that t is the total number of elements in the root node. The total time needed to construct $UP(v)$ and $C(v)$ for all d_{k-1} -nodes v is then $O(th^{2+\varepsilon} / \log n)$.

Now let u be an arbitrary d_k -node. Using almost the same method as above, we can produce sequences $UP(v)$ and $C(v)$ for all (d_{k-1}) -nodes v , such that u is a d_k -ancestor of v . There are only two differences with the method above. First, we divide the sequence $Y(u)$ into chunks of size 2^{d_k} . Second, the sorting of elements in a chunk is not based on the highest bits, but on a less significant chunk of bits: the pairs are now $(\text{bitval}(Y(u)[i]), \text{pos}(i, u))$. If the bit representation of $Y(u)[i]$ is $b_1 b_2 \dots b_d$, then $\text{bitval}(Y(u)[i])$ is the integer with bit representation $b_{f+1} b_{f+2} \dots b_{f+d_k}$ where f is the depth of the node u in the range tree. The total time needed to produce $C(v)$ and $UP(v)$ is $O(|Y(u)|d_k / \log n + |Y(u)|d_k^2 / \log n)$, the first term to create the pairs and the second to sort the chunks and produce $C(v)$ and $UP(v)$. The number of different elements in all d_k -nodes is $O(t \cdot h / d_k)$, and each produces the sequences of h^ε levels of d_{k-1} -nodes. Hence the time needed to produce the sequences for all d_{k-1} -nodes is $O((t \cdot h) / d_k \cdot h^\varepsilon \cdot d_k^2 / \log n) = O(t \cdot h^{1+\varepsilon} \cdot d_k / \log n) = O(t(h^2 / \log n)h^\varepsilon)$. The complexity stays the same after adding up the $1/\varepsilon$ values of k : $O(t \cdot h^{2+\varepsilon} / \log n) = O((n/r)r^2 \log^2 \sigma \log^\varepsilon n / \log n) = O((n \cdot r \cdot \log^2 \sigma / \log^{1-\varepsilon} n))$.

The data structure supporting select queries on $C(v)$ can be built in $O(|C(v)| / \log n)$ time [33, Thm. 5]. This amounts to $O(th / \log n) = O(n / \log_\sigma n)$ further time.

A.3 Predecessors and successors of x -coordinates

Now we describe how predecessor and successor queries on x -coordinates of points in $Y(u)$ can be answered for any node u in time $O(\log \log n)$.

We divide the sequence $Y(u)$ into blocks, so that each block contains $\log n$ points. We keep the minimum x -coordinate from every block in a predecessor data structure $Y^b(u)$. In order to find the predecessor of x in $Y(u)$, we first find its predecessor x'' in $Y^b(u)$; then we search the block of x'' for the predecessor of x in $Y(u)$.

The predecessor data structure finds x'' in $O(\log \log n)$ time. We compute the x -coordinate of any point in $Y(u)$ in $O(1)$ time as shown above. Hence the predecessor of x in a block is found in $O(\log \log n)$ time too, using binary search. We find the successor analogously.

The sampled predecessor/successor data structures store $O((n/r)(r \log \sigma)/\log n) = O(n/\log_\sigma n)$ elements over all the levels. An appropriate construction [20, Thm. 4.1] builds them in linear time ($O(n/\log_\sigma n)$) and space ($O(n \log \sigma)$ bits), once they are sorted.

Chaining with Overlaps Revisited

Veli Mäkinen 

Department of Computer Science, University of Helsinki, Finland
veli.makinen@helsinki.fi

Kristoffer Sahlin 

Department of Mathematics, Science for Life Laboratory, Stockholm University, Sweden
ksahlin@math.su.se

Abstract

Chaining algorithms aim to form a semi-global alignment of two sequences based on a set of anchoring local alignments as input. Depending on the optimization criteria and the exact definition of a chain, there are several $O(n \log n)$ time algorithms to solve this problem optimally, where n is the number of input anchors.

In this paper, we focus on a formulation allowing the anchors to overlap in a chain. This formulation was studied by Shibuya and Kurochkin (WABI 2003), but their algorithm comes with no proof of correctness. We revisit and modify their algorithm to consider a strict definition of precedence relation on anchors, adding the required derivation to convince on the correctness of the resulting algorithm that runs in $O(n \log^2 n)$ time on anchors formed by exact matches. With the more relaxed definition of precedence relation considered by Shibuya and Kurochkin or when anchors are non-nested such as matches of uniform length (k -mers), the algorithm takes $O(n \log n)$ time.

We also establish a connection between chaining with overlaps and the widely studied *longest common subsequence* problem.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Dynamic programming; Applied computing → Genomics

Keywords and phrases Sparse Dynamic Programming, Chaining, Maximal Exact Matches, Longest Common Subsequence

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.25

Related Version A preprint of the paper is available at <https://arxiv.org/abs/2001.06864>.

Funding *Veli Mäkinen*: Partially supported by the Academy of Finland (grant 309048).
Kristoffer Sahlin: Work partially conducted when affiliated with the University of Helsinki.

Acknowledgements We wish to thank Manuel Cáceres for spotting a mistake in our original coverage definition regarding nested anchors and the anonymous reviewers for useful suggestions to improve the readability.

1 Introduction

As optimal alignment of two strings takes quadratic time (which has recently been shown to be conditionally hard to improve [3]), there have been several attempts to avoid this bottleneck. One such technique is *sparse dynamic programming* [5], where a sparse set of cells of the dynamic programming matrix is identified whose computation is sufficient in computing the optimal alignment. This does not avoid the quadratic dependency in the worst case, so a slightly more heuristic *chaining* approach has been introduced in the context of computational genomics: Given a precomputed set of plausible *anchoring local matches*, extract a chain of matches that forms a good (semi-global) alignment.



© Veli Mäkinen and Kristoffer Sahlin;
licensed under Creative Commons License CC-BY
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 25; pp. 25:1–25:12
Leibniz International Proceedings in Informatics



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

In this paper, we investigate a chaining formulation that takes properly the overlaps between anchors into account. Namely, if anchors are not allowed to overlap in the solution, there are already several $O(n \log n)$ time solutions for various formulations of the chaining problem [11, 6, 1, 2], where n is the number of anchors. Some of the solutions and extensions focus on asymmetric measures, where overlaps are allowed in one of the strings [9, 10], or add other features that make the problem even harder [13]. While these formulations are useful in different contexts, this is an undesirable consequence in, *e.g.*, string alignment, where the solution may be different depending on which string is used to traverse the ordered anchors, and specifically the solution may overcount the amount of aligned characters.

The fully symmetric chaining variant allows arbitrary overlaps, guarantees not to overcount the amount of aligned characters, and in addition, is particularly important for its connections to the *Longest Common Subsequence* problem (LCS): An optimal chain in this formulation corresponds to a LCS of the input strings, restricted to the matches included in the anchors. As far as we know, except for trivial $O(n^2)$ time solutions, only Shibuya and Kurochkin [12] have proposed a solution aiming to solve the fully symmetric case of allowing overlaps of anchors in both strings simultaneously.

We revisit the algorithm by Shibuya and Kurochkin [12] and propose a modification that takes into account a strict order for the anchors. This modified algorithm runs in $O(n \log^2 n)$ time on exact matches as input. When relaxing the precedence order or when the input consist of non-nested anchors such as k -mer matches, the algorithm can be simplified to take $O(n \log n)$ time. The resulting algorithms are slightly simpler than the original [12], requiring only a general data structure for semi-dynamic range maximum queries, while the original uses in addition a tailored structure. We also provide detailed derivation of the algorithms, while the original [12] comes with no proof of correctness. Finally, we show that the relaxed chaining problem also solves a restricted version of the LCS problem.

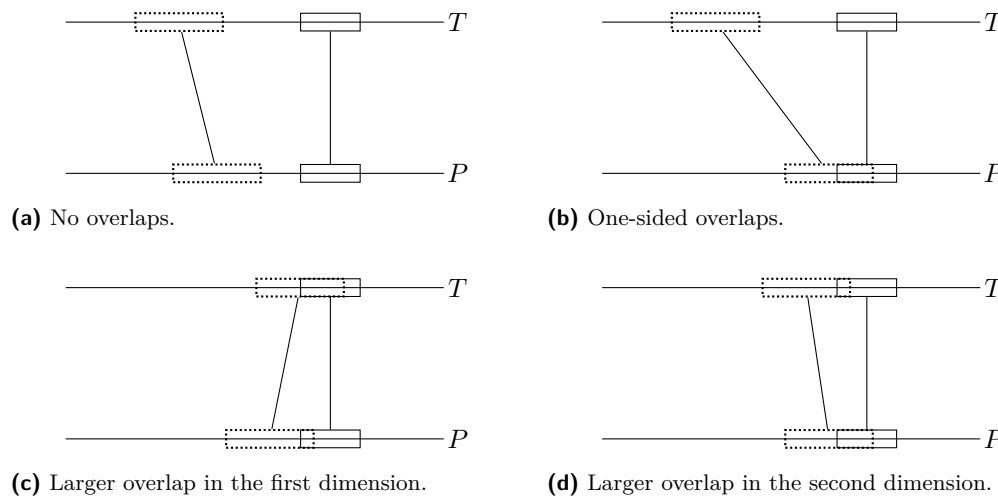
2 Chaining problems

Let T be a long text string and P short pattern string. An *anchor* interval pair $([a..b], [c..d])$ denotes a *match* between $T[a..b]$ and $P[c..d]$. For now, we assume these matches are precomputed, and they could be either full identities or close similarities. We often abstract out the original source of the anchors referring $[a..b]$ as an interval in the *first dimension* and $[c..d]$ as an interval in the *second dimension*. We denote the endpoints of the intervals in anchor I as $I.x$ for $x \in \{a, b, c, d\}$. We assume the endpoints to be positive integers.

Given two anchors I' and I we define two relations: *precedence* and *overlap*. The former is denoted $I' \prec I$ and this relation holds whenever $I'.a < I.a$, $I'.b < I.b$, $I'.c < I.c$, and $I'.d < I.d$. The latter is denoted $I' \cap I$ and holds whenever $[I'.a..I'.b] \cap [I.a..I.b] \neq \emptyset$ or $[I'.c..I'.d] \cap [I.c..I.d] \neq \emptyset$. The complement of the overlap relation is denoted $\neg I' \cap I$ (an empty intersection is interpreted as truth value False). We use the overlap relation only for $I' \prec I$. Figure 1 illustrates these concepts.

► **Problem 1** (Chaining with overlaps). *Let $A[1..N]$ be an array of anchor interval pairs $([a..b], [c..d])$. For each i , $1 \leq i \leq N$, compute the symmetric ordered coverage score $\max_{\text{chains}} S^i \text{ coverage}(S^i)$, where*

- $S^i[1..n]$ is an ordered subset (chain) of pairs from A ,
- $S^i[j-1] \prec S^i[j]$, for all $1 < j \leq n$,



■ **Figure 1** Different scenarios illustrating precedence and overlap of anchors. Dotted and solid rectangles denote anchors I' and I , respectively. In all these cases it holds $I' \prec I$. The separation into different cases based on the overlaps is determined by the properties of the chaining algorithms we study in the sequel. In (a) and (b) no overlaps are allowed in the first dimension, while in (c) and (d) anchors are assumed to overlap in the first dimension.

■ $S^i[n] = A[i]$, and

$$\text{coverage}(S^i) = \left(\sum_{j=1}^{n-1} \min \left(\min(S^i[j+1].a, S^i[j].b+1) - S^i[j].a, \right. \right. \\ \left. \left. \min(S^i[j+1].c, S^i[j].d+1) - S^i[j].c \right) \right) + \\ \min \left(S^i[n].b - S^i[n].a + 1, S^i[n].d - S^i[n].c + 1 \right).$$

Notice that for chains containing no overlaps, that is, $S^i[j].b < S^i[j+1].a$ and $S^i[j].d < S^i[j+1].c$, the measure $\text{coverage}(S^i)$ is just the sum of lengths of the anchors in it, where length is defined as the minimum of the interval lengths. For overlapping cases, only the segment before the overlap is added to the score. For example, let a chain S be $([1..5], [2..6]), ([3..8], [5..10])$. Then $\text{coverage}(S) = \min(8 - 3 + 1, 10 - 5 + 1) + \min(\min(3, 5 + 1) - 1, \min(5, 6 + 1) - 2) = 6 + 2 = 8$. That is, while the total length of the anchors in S is 10, their union covers only 8 units in the first dimension. The measure is clearly symmetric, but the term *ordered coverage* requires more insight: Notice that it is not sufficient to measure the size of the union of anchors in a chain independently and take their minimum. Instead, the proposed measure adds to the score, one anchor at the time, the minimum size of the newly covered region. Interpreted through the original source of anchors from string T and P , an optimal chain S under this measure induces an alignment between T and P with exactly $\text{coverage}(S)$ matching characters. Asymmetric formulations studied earlier can overestimate this amount. This proposed symmetric ordered coverage measure is thus important especially in various computational genomics applications, where optimal alignments are too expensive to be computed. We establish this alignment connection through the widely studied longest common subsequence problem: see Sect. 4.

25:4 Chaining with Overlaps

We develop an $O(N \log^2 N)$ time algorithm to solve this chaining with overlaps problem assuming one additional property of the input:

- *Equal Match Length property*: For each anchor I it holds $I.b - I.a = I.d - I.c$.

If the set of anchors is computed e.g. by *Maximal Exact Matches (MEMs)* [7], the input automatically satisfies the Equal Match Length property.

Our algorithm is based on techniques by Shibuya and Kurochkin [12], who solved a version of the problem with the definition of precedence relaxed to consider only start points of intervals: I' *weakly precedes* I if $I'.a < I.a$ and $I'.c < I.c$. Let us denote this relation $I' \prec^w I$.

► **Problem 2** (Chaining with overlaps and weak precedence). *Let $A[1..N]$ be an array of anchor interval pairs $([a..b], [c..d])$. For each i , $1 \leq i \leq N$, compute the symmetric weakly ordered coverage score $\max_{\text{weak chains } S^i} \text{coverage}(S^i)$, defined as in Problem 1, with the precedence condition relaxed to*

- $S^i[j-1] \prec^w S^i[j]$, for all $1 < j \leq n$.

To see the connection of the problems, consider a chain S for which $S[j-1] \prec^w S[j]$ holds but not $S[j-1] \prec S[j]$ for some j . That is, at least one of the intervals of $S[j]$ is nested inside (i.e. is subset of) the corresponding interval of $S[j-1]$. Say $[S[j].a..S[j].b]$ is nested in $[S[j-1].a..S[j-1].b]$ with $S[j-1].b - S[j].b \geq S[j-1].d - S[j].d$ (the other case is symmetric). Consider modifying $S[j-1]$ into $S[j-1]'$, where $S[j-1]'.a = S[j-1].a$, $S[j-1]'.b = S[j].b - 1$, $S[j-1]'.c = S[j-1].c$, and $S[j-1]'.d = S[j-1].d - (S[j-1].b - S[j].b) - 1$. Assuming Equal Match Length property such adjustment is possible and causes $S[j-1]' \prec S[j]$ without affecting the score. One can thus adjust any chain S for which the weak precedence relation holds into another chain S' , where the (strict) precedence relation holds, so that $\text{coverage}(S) = \text{coverage}(S')$.

As can be seen from the above construction, the two problems are identical when the input anchors are non-nested. This happens e.g. when anchors are matches of uniform length (k -mer matches). Even more importantly, if one is only interested in the overall maximum scoring chain, the two problems produce the same result.

► **Lemma 1**. *Assuming Equal Match Length property, the maximum of the solutions from Problem 1 and Problem 2 are the same, that is,*

$$\max_{1 \leq i \leq N} \max_{\text{chains } S^i} \text{coverage}(S^i) = \max_{1 \leq i \leq N} \max_{\text{weak chains } S^i} \text{coverage}(S^i).$$

Proof. Consider an optimal chain $S^i[1..n]$ for Problem 2. If $S^i[n-1] \prec S^i[n]$ does not hold, then one of the intervals of $S^i[n]$ is nested inside the corresponding interval of $S^i[n-1]$. This means that for $S^i[n-1] = A[i']$ it holds $\max_{\text{weak chains } S^{i'}} \text{coverage}(S^{i'}) \geq \max_{\text{weak chains } S^i} \text{coverage}(S^i)$. Continuing this induction, one observes that there is an overall maximum scoring chain, say S , that ends with strict precedence and moreover for all anchors I in this chain holds $I \prec S[n]$.

Consider now the construction given before this lemma that converts S into S' . Adjust the construction so that instead of modifying $S[j-1]$ into $S[j-1]'$, just remove $S[j]$. Repeat this from right to left until strict precedence holds in the whole modified chain S' . Such S' is an optimal solution to both problems, as its score remains unchanged during the process.

To see this, consider the same case as earlier with $[S[j].a..S[j].b]$ being nested in $[S[j-1].a..S[j-1].b]$ with $S[j-1].b - S[j].b \geq S[j-1].d - S[j].d$. By induction from the base case of $S[n-1] \prec S[n]$, we know that $S[j] \prec S[j+1]$ and $S[j-1] \prec S[j+1]$ if $S[j-1] \prec^w S[j]$ is the first nested case from the right.

To see that dropping $S[j]$ is safe, we need to consider cases a) $\neg S[j] \cap S[j+1]$ and $\neg S[j-1] \cap S[j+1]$ b) $S[j] \cap S[j+1]$ and $\neg S[j-1] \cap S[j+1]$, c) $\neg S[j] \cap S[j+1]$ and $S[j-1] \cap S[j+1]$, and d) $S[j] \cap S[j+1]$ and $S[j-1] \cap S[j+1]$. We cover here only case d), as all the other cases use similar or easier reasoning. We consider score induced by sub-chains $S[j-1], S[j], S[j+1]$ and $S[j-1], S[j+1]$, respectively, assuming that the chains continue, so that the coverage induced by $S[j+1]$ will not yet be added to the total score. In case d) the score induced by the sub-chain $S[j-1], S[j], S[j+1]$ is $S[j].a - S[j-1].a + \min(S[j+1].a - S[j].a, S[j+1].c - S[j].c) = \min(S[j+1].a - S[j-1].a, S[j].a - S[j-1].a + S[j+1].c - S[j].c)$. The score induced by the sub-chain $S[j-1], S[j+1]$ is $\min(S[j+1].a - S[j-1].a, S[j+1].c - S[j-1].c)$. Since $S[j].a - S[j-1].a \leq S[j].c - S[j-1].c$, the score induced by the sub-chain $S[j-1], S[j+1]$ is at least as high as the score induced by the sub-chain $S[j-1], S[j], S[j+1]$. ◀

Shibuya and Kurochkin [12] gave an $O(N \log N)$ time algorithm for Problem 2, but their algorithm comes with no proof of correctness. Our goal in this paper is to complement the original proposal with the required derivation steps to see that one can indeed solve the problem correctly in $O(N \log N)$ time. Instead of proving directly the correctness of the original proposal, we derive a simplified version of the algorithm, whose correctness is easier to verify.

We derive this algorithm in three steps: First we consider one-sided overlaps of anchors. Then we modify this algorithm to handle two-sided overlaps of anchors, solving Problem 1. Finally, we show that the use of strict precedence relation $I' \prec I$ can be relaxed to $I' \prec^w I$ in order to solve Problem 2.

3 Chaining algorithms

Our goal is here to study the variations of chaining algorithms under the symmetric ordered coverage. We will give chaining algorithms under the symmetric ordered coverage and equal-match property taking $O(n \log n)$ time. In order to do this we will structure the recurrence relations that solve Problems 1 and 2 such that one can factor out dependencies between anchors into different cases that are handled by evaluation order of the recurrences, range search, and special features of the scoring function. Assume now that the anchor interval pairs are stored in an array $A[1..N]$ in arbitrary order. We fill a table $C[1..N]$ so that $C[j]$ gives the maximum symmetric ordered coverage of using any subset of pairs that precede $A[j]$, before the effect of the pair $A[j]$ is added to the score: Hence, $\max_j C^+[j]$, where $C^+[j] = \min(A[j].b - A[j].a + 1, A[j].d - A[j].c + 1) + C[j]$, gives the total maximum symmetric ordered coverage.

After considering separately non-overlapping and overlapping cases (see Fig. 1), one observes that $C[j]$ can be computed by $\max(0, D[j], O[j])$, where

$$D[j] = \max_{\substack{j' : \\ A[j'] \prec A[j], \\ \neg A[j'] \cap A[j]}} C[j'] + \min \begin{cases} A[j'].b - A[j'].a + 1, \\ A[j'].d - A[j'].c + 1 \end{cases} \quad \text{and}$$

$$O[j] = \max_{\substack{j' : \\ A[j'] \prec A[j], \\ A[j'] \cap A[j]}} C[j'] + \min \begin{cases} \min(A[j].a, A[j'].b + 1) - A[j'].a, \\ \min(A[j].c, A[j'].d + 1) - A[j'].c \end{cases}.$$

These recurrences can be computed in $O(N^2)$ time: Sort A by values $A[i].b$ to handle one dimension of the precedence relation. Then compute each $C[j]$ in this order by scanning previously computed values $C[j']$ and check precedence in the other dimension. Add the coverage values (+ min part) depending on the overlap relation. Select the maximum among the options of $C[j']$ added with the coverage value.

By assuming Equal Match Length property, we can simplify the recurrence of $C[j] = \max(0, D[j], O[j])$ with

$$D[j] = \max_{\substack{j' : \\ A[j'] \prec A[j], \\ \neg A[j'] \cap A[j]}} C[j'] + A[j'].b - A[j'].a + 1 \quad \text{and}$$

$$O[j] = \max_{\substack{j' : \\ A[j'] \prec A[j], \\ A[j'] \cap A[j]}} C[j'] + \min \begin{cases} A[j].a - A[j'].a, \\ A[j].c - A[j'].c \end{cases} .$$

3.1 One-sided overlaps

We will now present an algorithm that works for *one-sided overlaps* (see Fig. 1): We restrict the chains so that no two anchors in the solution overlap in the first dimension (that is, in T). This lets us modify the recurrence of $O[j]$ into the form

$$O[j] = A[j].c + \max_{\substack{j' : \\ A[j'] \prec A[j], A[j'].b < A[j].a, \\ A[j'] \cap A[j]}} C[j'] - A[j'].c .$$

That is, we added the constraint on overlaps, removed the then obsolete $\min()$ and took out the value not affected by $\max()$. Now it is easy to see that the evaluation of the values can be done when visiting the starting points of the anchors in the first dimension, and the maximizations over range of values can be done using search trees, specified in the next lemma. We also specify a two-dimensional version of this structure, as we need it later.

► **Lemma 2.** *The following three operations can be supported with a one-dimensional range search tree \mathcal{T} in time $O(\log n)$, where n is the number of search keys inserted to the tree.*

- **Update(k, val):** Update value associated with key = k into val .
- **Upgrade(k, val):** Update value associated with key = k into $\max(\text{val}, \text{value})$.
- **RMaxQ(c, d):** Return maximum value, where $c \leq \text{key} \leq d$ (Range Maximum Query).
Moreover, the search tree can be built in $O(n)$ time, given the n pairs (key, value) sorted by component key.

► **Lemma 3.** *The following two operations can be supported with a two-dimensional range search tree \mathcal{T} in time $O(\log^2 n)$, where n is the number of search keys inserted to the tree.*

- **Update(p, s, val):** Update value associated with primary key = p and secondary key = s into val .
- **Upgrade(p, s, val):** Update value associated with primary key = p and secondary key = s into $\max(\text{val}, \text{value})$.
- **RMaxQ(a, b, c, d):** Return maximum value, where $a \leq \text{primary key} \leq b$ and $c \leq \text{secondary key} \leq d$ (2D Range Maximum Query).

Moreover, the search tree can be built in $O(n \log n)$ time, given the n triplets (primary key, secondary key, value) sorted first by primary key and then by secondary key.

These lemmas follow directly by maintaining maxima of values in each subtree for the corresponding standard range search structures [4] that support listing all the (key, value) pairs in a range. Such constructions are often used in sparse dynamic programming [5, 12, 8].

■ **Algorithm 1** Chaining allowing one-sided overlaps.

Input: A set of interval pairs $A[1..N]$ with all interval endpoints being distinct positive integers.

Output: Array $C^+[1..N]$ containing the symmetric ordered coverage values.

Initialize one-dimensional search trees \mathcal{T}^a and \mathcal{T}^b with keys $A[j].d$, $1 \leq j \leq N$, and with key 0, all keys associated with values $-\infty$;

$\mathcal{T}^a.\text{Upgrade}(0, 0)$;

$E = \{(A[j].a, j) \mid 1 \leq j \leq N\} \cup \{(A[j].b, j) \mid 1 \leq j \leq N\}$;

$E.\text{sort}()$;

for $i \leftarrow 1$ **to** $2N$ **do**

$j = E[i][2]$;

$I = A[j]$;

if $I.a == E[i][1]$ **then**

$C^a[j] = \mathcal{T}^a.\text{RMaxQ}(0, I.c - 1)$;

$C^b[j] = I.c + \mathcal{T}^b.\text{RMaxQ}(I.c, I.d)$;

$C[j] = \max(C^a[j], C^b[j])$;

$C^+[j] = C[j] + I.b - I.a + 1$;

end

else

$\mathcal{T}^a.\text{Upgrade}(I.d, C^+[j])$;

$\mathcal{T}^b.\text{Upgrade}(I.d, C[j] - I.c)$;

end

end

return $C^+[1..N]$;

We obtain Algorithm 1 to handle the one-sided overlaps case, where we have replaced arrays D and O with C^a and C^b , respectively, to reflect the cases shown in Fig. 1.

The pseudocode of Algorithm 1 assumes interval endpoints to be distinct. This assumption is only used for the ease of presentation. It can be relaxed by the standard method used in computational geometry: Replace each endpoint x by a pair $(x, j) = E[i]$ where $A[j]$ identifies the anchor in question. These pairs $E[i] = (x, j)$ are distinct, and can be used as the keys of the search trees (in place of just x). Range queries can be implemented to ignore the secondary key j .

► **Lemma 4.** *Problem 1 on N input pairs restricted to solutions that contain only one-sided overlaps can be solved in $O(N \log N)$ time, assuming the input satisfies Equal Match Length property.*

Proof. The evaluation order of Algorithm 1 guarantees that when computing the values $C^a[j]$ and $C^b[j]$, the data structures contain only anchors that precede the current anchor and do not overlap it in the first dimension. The range query on \mathcal{T}^a guarantees that we also consider only those anchors that precede and do not overlap in the second dimension for the computation of $C^a[j]$. The range query on \mathcal{T}^b guarantees that we also consider only those anchors that overlap in the second dimension for the computation of $C^b[j]$, but this is not enough to guarantee predecessor-relation to hold. That is, there can be an anchor I' stored in

\mathcal{T}^b with $I'.b < I.a$ and $I.c \leq I'.c \leq I'.d < I.d$ and thus the evaluation order and range query fail to guarantee $I'.c < I.c$ to make $I' \prec I$ (recall the definition). We need to show that if such I' is in an optimal chain to I , there is always another optimal chain to I not including I' . Consider the last anchor $A[j''] = I''$ in an optimal chain to $A[j'] = I'$ that overlaps and precedes I . Then we know that $C[j] \geq C[j''] + I.c - I''.c$ and $C[j'] \leq C[j''] + I'.c - I''.c$, so a chain where I'' directly precedes I does not decrease the score. If such I'' does not exist but an optimal chain to I includes I' , we have that $\max(C^a[j], C^b[j]) = C^a[j]$, as all anchors in an optimal chain to I' , excluding I' , are stored in \mathcal{T}^a , and including I' can only decrease the score as $I.c - I'.c \leq 0$. \blacktriangleleft

3.2 Two-sided overlaps

The trick by Shibuya and Kurochkin [12] to handle two-sided overlaps is to separate them to two cases (see Fig. 1): (c) overlaps in the first dimension are at least as long as in the second dimension and (d) overlaps are longer in the second dimension. Since our algorithm so far considers all anchors that do not overlap in the first dimension, it will be enough to consider how to enhance the algorithm to handle anchors that do overlap in the first dimension.

Consider case (c). That is, for any two pairs of anchors $I', I, I' \prec I$, it holds $I'.a < I.a \leq I'.b < I.b, I'.c < I.c, I'.d < I.d$ and $I'.d - I.c \leq I'.b - I.a$. The latter inequality can be written as $I.c - I.a \geq I'.c - I'.a$ (due to Equal Match Length property). Also, if I' precedes I in an optimal chain to I , the score calculated up to I' will increase by inclusion of I by $\min(I.a - I'.a, I.c - I'.c) = I.a - I'.a$ (due to Equal Match Length property). This means that once we first stop at anchor $I = A[j]$ in our algorithm, if we have inserted to a search tree T^c all anchors $A[j']$ that overlap I in the first dimension, using keys $A[j'].c - A[j'].a$ and values $C[j'] - A[j'].a$, we can query $T^c.\text{RMaxQ}(-\infty, A[j].c - A[j].a)$ and add $A[j].a$ to obtain the correct score for this case. However, in the order we evaluate the anchors we can only guarantee $A[j'].a < A[j].a \leq A[j'].b$ and thus $A[j'].c < A[j].c$ (property of case (c)), but not $A[j'].b < A[j].b$ or $A[j'].d < A[j].d$. To solve this, we add another dimension to the search tree, so we can add constraint $A[j'].b < A[j].b$ to the query, which also covers the remaining constraint $A[j'].d < A[j].d$ (property of case (c)).

Case (d) is almost symmetric to case (c): For any two pairs of anchors $I', I, I' \prec I$, it holds $I'.a < I.a \leq I'.b < I.b, I'.c < I.c, I'.d < I.d$ and $I'.d - I.c > I'.b - I.a$. The latter inequality can be written as $I'.c - I'.a > I.c - I.a$. Also, if I' precedes I in an optimal chain to I , the score calculated up to I' will increase by inclusion of I by $\min(I.a - I'.a, I.c - I'.c) = I.c - I'.c$ (due to Equal Match Length property). This means that once we first stop at anchor $I = A[j]$ in our algorithm, if we have inserted to a search tree T^d all anchors $A[j']$ that overlap I in the first dimension, using keys $A[j'].c - A[j'].a$ and values $C[j'] - A[j'].c$, we can query $T^d.\text{RMaxQ}(A[j].c - A[j].a + 1, \infty)$ and add $A[j].c$ to obtain the correct score for this case. As before, we need to add another dimension to the search tree to handle constraint $A[j'].d < A[j].d$, which also covers constraint $A[j'].b < A[j].b$ (property of case (d)). We are left with constraints $A[j'].a < A[j].a \leq A[j'].b$ and $A[j'].c < A[j].c$, where the first ones follow from the evaluation order, but now the latter is not automatically guaranteed to hold: Using arguments analogous to the proof of Lemma 4, we show that such nested case cannot change the optimal solution.

The resulting enhancement to handle two-sided overlaps is given as Algorithm 2.

The pseudocode of Algorithm 2 assumes interval endpoints to be distinct, but this can be relaxed as in the proof of Lemma 4. Using the data structure from Lemmas 2 and 3 we obtain the following result.

■ **Algorithm 2** Chaining with two-sided overlaps.

Input: A set of interval pairs $A[1..N]$ with all interval endpoints being distinct positive integers.

Output: Array $C^+[1..N]$ containing the symmetric ordered coverage values.

Initialize one-dimensional search trees \mathcal{T}^a and \mathcal{T}^b with keys $A[j].d$, $1 \leq j \leq N$, and with key 0, all keys associated with values $-\infty$;

Initialize two-dimensional search trees \mathcal{T}^c and \mathcal{T}^d with keys $(A[j].c - A[j].a, A[j].b)$ and $(A[j].c - A[j].a, A[j].d)$, respectively, for $1 \leq j \leq N$, associated with values $-\infty$;

$\mathcal{T}^a.\text{Upgrade}(0, 0)$;

$E = \{(A[j].a, j) \mid 1 \leq j \leq N\} \cup \{(A[j].b, j) \mid 1 \leq j \leq N\}$;

$E.\text{sort}()$;

for $i \leftarrow 1$ **to** $2N$ **do**

$j = E[i][2]$;

$I = A[j]$;

if $I.a == E[i][1]$ **then**

$C^a[j] = \mathcal{T}^a.\text{RMaxQ}(0, I.c - 1)$;

$C^b[j] = I.c + \mathcal{T}^b.\text{RMaxQ}(I.c, I.d)$;

$C^c[j] = I.a + \mathcal{T}^c.\text{RMaxQ}(-\infty, I.c - I.a, 0, I.b)$;

$C^d[j] = I.c + \mathcal{T}^d.\text{RMaxQ}(I.c - I.a + 1, \infty, 0, I.d)$;

$C[j] = \max(C^a, C^b, C^c, C^d)$;

$C^+[j] = C[j] + I.b - I.a + 1$;

$\mathcal{T}^c.\text{Upgrade}(I.c - I.a, I.b, C[j] - I.a)$;

$\mathcal{T}^d.\text{Upgrade}(I.c - I.a, I.b, C[j] - I.c)$;

end

else

$\mathcal{T}^a.\text{Upgrade}(I.d, C^+[j])$;

$\mathcal{T}^b.\text{Upgrade}(I.d, C[j] - I.c)$;

$\mathcal{T}^c.\text{Update}(I.c - I.a, I.b, -\infty)$;

$\mathcal{T}^d.\text{Update}(I.c - I.a, I.b, -\infty)$;

end

end

return $C^+[1..N]$;

► **Theorem 5.** *Problem 1 on N input pairs can be solved in $O(N \log^2 N)$ time (by Algorithm 2), assuming the input satisfies Equal Match Length property.*

Proof. As discussed earlier, it is sufficient to consider anchors $A[j']$ and $A[j]$ that satisfy the precedence and overlap relations except for $A[j'].c < A[j].c$ not holding, as all other constraints are properly covered by the combination of evaluation order and the queries. Such invalid anchors $A[j']$ can affect the query results from data structures \mathcal{T}^b and \mathcal{T}^d when computing the score for $A[j]$. Consider that an optimal chain to $A[j]$ has $A[j']$ as the previous anchor and thus $C[j] = C[j'] + A[j].c - A[j'].c$. Consider the last anchor $A[j'']$ in an optimal chain to $A[j']$ that precedes and overlaps $A[j]$. Assume the overlap is larger in the first dimension (the other case is considered already in the proof of Lemma 4). Then $C[j'] = C[j''] + A[j].a - A[j''].a$ as $A[j']$ must overlap $A[j]$ in the first dimension, $A[j'']$ must directly precede $A[j']$ for it being the last with this property, and the overlap between $A[j'']$ and $A[j']$ is larger in the first dimension due to transitivity. As $A[j].a - A[j''].a \geq A[j].c - A[j'].c$, the direct use of $A[j'']$ before $A[j]$ gives $C[j] \geq C[j''] + A[j].a - A[j''].a = C[j''] + A[j].a - A[j''].a + A[j'].a - A[j''].a \geq C[j'] + A[j].c - A[j'].c$. That is, $A[j']$ can be omitted from the optimal path. ◀

3.3 Overlaps with weak precedence

Let us now proceed to improve the running time of Algorithm 2 to $O(N \log N)$ by considering chains under the weak precedence relation (Problem 2). For this, we drop the second dimension of the data structures \mathcal{T}^c and \mathcal{T}^d , that were added to guarantee strict precedence. However, this is not sufficient for proving correctness as we used these constraints to indirectly guarantee precedence of start positions of anchors as well. Case (c) causes no problems, as the evaluation order guarantees that \mathcal{T}^c contains anchors $A[j']$ with $A[j'].a < A[j].a$ and the query restricts to cases $A[j'].c < A[j].c$. However, in case (d) the solution returned can have $A[j].c \leq A[j'].c$. We will consider this in the proof of the next theorem.

► **Theorem 6.** *Problem 2 on N input pairs can be solved in $O(N \log N)$ time (by Algorithm 2 with the operations on the second dimension of search trees \mathcal{T}^c and \mathcal{T}^d omitted), assuming the input satisfies Equal Match Length property.*

Proof. As discussed, it is sufficient to show that queries from \mathcal{T}^d correspond to proper solutions. For contradiction, assume that $C[j] = C^d[j]$, $C^d[j] > \max(C^a[j], C^b[j], C^c[j])$, and $C[j] = C[j'] + A[j].c - A[j'].c$ only for $A[j']$ s for which $A[j].c \leq A[j'].c$. Such solution is not proper (weak precedence not holding), so we need to show that there is an equivalently good proper solution.

First, if it also holds $A[j'].d \leq A[j].d$, we have the nested case handled already in the proof of Theorem 5. We continue with the case where $A[j].c \leq A[j'].c$ and $A[j].d < A[j'].d$ hold. This setting is illustrated in Fig. 2.

Consider an anchor $A[j'']$ in an optimal chain to $A[j']$ that overlaps position $A[j].c$ in the second dimension. It holds $C[j'] \leq C[j''] + A[j'].c - A[j''].c$, since any chain from $A[j'']$ to $A[j']$ can cover at most $A[j'].c - A[j''].c$ positions. But then there is an optimal chain to $A[j]$ avoiding $A[j']$ with score $C[j] \geq C[j''] + A[j].c - A[j''].c = C[j'].c - A[j'].c + A[j''].c + A[j].c - A[j''].c = C[j'] + A[j].c - A[j'].c$, which is a contradiction.

We are left with the case that there is no such $A[j'']$ in the optimal chain to $A[j']$ that overlaps position $A[j].c$ in the second dimension. Let then $A[j'']$ be the last anchor in an optimal chain to $A[j']$ that does not overlap $A[j].c$. We have three cases to consider: a) $\neg A[j''] \cap A[j']$, b) $A[j''] \cap A[j']$, and c) no such $A[j'']$ exist.

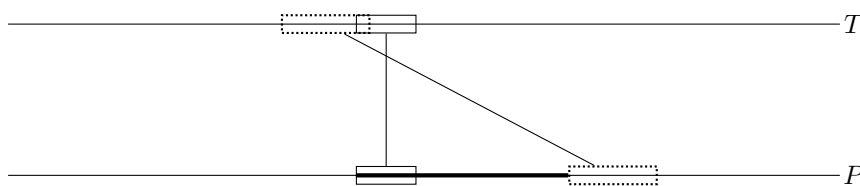
In case a) $C[j'] < C^+[j''] + A[j'].c - A[j].c$, assuming that only $A[j].c$ is left uncovered between anchors $A[j'']$ and $A[j']$. Using this we can write $C[j] = C[j'] + A[j].c - A[j'].c < C^+[j''] + A[j'].c - A[j].c + A[j].c - A[j'].c = C^+[j'']$. Since it holds $A[j''] \prec^w A[j]$ and $\neg A[j''] \cap A[j]$ we have that $C[j] \geq C^+[j'']$ using directly $A[j'']$ avoiding $A[j]$. This is contradiction.

In case b) $A[j'']$ can only overlap $A[j']$ in the first dimension. Then $C[j'] = C[j''] + A[j'].a - A[j''].a$. If $A[j'']$ also overlaps $A[j]$, it can do so only in the first dimension. Then $C[j] \geq C[j''] + A[j].a - A[j''].a \geq C[j''] + A[j'].a - A[j''].a$. That is, $A[j']$ can be avoided in an optimal chain to $A[j]$ by using $A[j'']$ instead. On the other hand, if $A[j'']$ does not overlap $A[j]$, we have $C[j] \geq C^+[j''] \geq C[j''] + A[j'].a - A[j''].a = C[j']$, in which case we also get a contradiction.

In case c) $C[j'] = 0$ and thus $C[j] = C[j'] + A[j].c - A[j'].c \leq 0$, which contradicts our assumption $C^d[j] > C^a[j] = 0$. ◀

4 Connection to LCS

String C is a *Longest Common Subsequence* (LCS) of strings T and P if it is a longest string that can be obtained by deleting 0 or more characters from both T and from P . Such $C[1..\ell]$ can be written as $T' := T[i_1]T[i_2] \cdots T[i_\ell]$ and as $P' := P[j_1]P[j_2] \cdots P[j_\ell]$, where



■ **Figure 2** Dotted and solid rectangles denote $A[j']$ and $A[j]$, respectively. Here the (weak) precedence is not holding as the interval of $A[j']$ in the second dimension succeeds the corresponding interval of $A[j]$. The thick line segment represents maximum coverage a chain ending at $A[j']$ (not including $A[j']$) can achieve after starting from $A[j]$. The algorithm subtracts this thick line segment length from the score, so that there is at least as good chain to $A[j]$ that avoids using $A[j']$.

$1 \leq i_1 < i_2 < \dots < i_\ell \leq |T|$ and $1 \leq j_1 < j_2 < \dots < j_\ell \leq |P|$. Consider the set of anchors A being exact matches between T and P . We say that C is an *anchor-restricted LCS* if it can be written as T' and as P' defined above such that for each (i_k, j_k) there is an anchor $([a..b], [c..d])$ in A with $a+x = i_k$ and $c+x = j_k$ for some x , $0 \leq x \leq b-a = d-c$. Informally, such C is a longest string with all characters appearing in increasing order in T and P where each such occurrence of a character is *supported* by at least one anchor. We show that an *anchor-restricted LCS* can be found by solving the problem of chaining under the weak precedence:

► **Theorem 7.** *Assume the anchors A are exact matches between input strings T and P . The score of a chain S such that $\text{coverage}(S) = \max_{1 \leq i \leq N} \text{coverage}(S^i)$ of Problem 2 equals the length of an anchor-restricted LCS of T and P .*

Proof. Due to Lemma 1, we can assume S is a chain under the strict precedence order. Each anchor in S contributes to the score by the minimum length of its intervals after the overlaps with the previous anchor intervals have been cut out. This minimum length equals the number of characters that can be included to the common subsequence. That is, we can extract an anchor-restricted subsequence of T and P of length $\text{coverage}(S)$ from the solution. We need to show that such subsequence is the longest among anchor-restricted subsequences. Assume, for contradiction, that there is an anchor-restricted LCS $C[1..\ell]$ longer than $\text{coverage}(S)$. Consider the chain of ℓ anchors formed by taking for each $C[k]$ an anchor containing match $T[i_k] = P[j_k]$. Assign a score 1 to each anchor included in the chain. Let us modify this chain into a chain where weak precedence holds such that the total score (number of matches induced by the solution) remains the same. First, we merge from left to right all runs of identical anchors; score of an anchor is then the length of the run in the original chain. Then we consider anchors from left to right. Consider the first pair of anchors I', I in the current chain for which $I.a \leq I'.a$ (case $I.c \leq I'.c$ is symmetric). Let the score of I' be x . By construction, we know that the left-most position possible for the first match of I included in C is $I'.a + x$. Therefore, we can remove I' from the chain and include x matches from I . The total score does not decrease by this change. This process can be repeated until the weak precedence relation holds up to I , and then continued similarly to the end of the chain, yielding a contradiction. ◀

5 Discussion

We studied symmetric chaining formulations starting from the motivation to avoid overcounting the matches of local anchors. This overcounting can also be avoided using the asymmetric formulation studied in Sect. 3.1, and this formulation is actually a special case

of the chaining between a sequence and a DAG [10]; it appears hard to extend the fully symmetric chaining formulations we studied here to work with DAGs. We are currently working on a practical alternative that uses a multiple alignment in place of a DAG, so that we can use our new methods for long read sequence alignment in transcript prediction.

References

- 1 Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. Multiple genome alignment: Chaining algorithms revisited. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michoacán, Mexico, June 25-27, 2003, Proceedings*, volume 2676 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2003. doi:10.1007/3-540-44888-8_1.
- 2 Mohamed Ibrahim Abouelhoda and Enno Ohlebusch. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms*, 3(2-4):321–341, 2005. doi:10.1016/j.jda.2004.08.011.
- 3 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 51–58. ACM, 2015. doi:10.1145/2746539.2746612.
- 4 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. URL: <http://www.worldcat.org/oclc/227584184>.
- 5 David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming I: linear cost functions. *J. ACM*, 39(3):519–545, 1992. doi:10.1145/146637.146650.
- 6 Stefan Felsner, Rudolf Müller, and Lorenz Wernisch. Trapezoid graphs and generalizations, geometry and algorithms. *Discrete Applied Mathematics*, 74(1):13–32, 1997. doi:10.1016/S0166-218X(96)00013-3.
- 7 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 8 Veli Mäkinen, Gonzalo Navarro, and Esko Ukkonen. Transposition invariant string matching. *J. Algorithms*, 56(2):124–153, 2005.
- 9 Veli Mäkinen, Leena Salmela, and Johannes Ylinen. Normalized N50 assembly metric using gap-restricted co-linear chaining. *BMC Bioinformatics*, 13:255, 2012. doi:10.1186/1471-2105-13-255.
- 10 Veli Mäkinen, Alexandru I. Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Trans. Algorithms*, 15(2):29:1–29:21, 2019. doi:10.1145/3301312.
- 11 Gene Myers and Webb Miller. Chaining multiple-alignment fragments in sub-quadratic time. In Kenneth L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA.*, pages 38–47. ACM/SIAM, 1995. URL: <http://dl.acm.org/citation.cfm?id=313651.313661>.
- 12 Tetsuo Shibuya and Igor Kurochkin. Match Chaining Algorithms for cDNA Mapping. In Gary Benson and Roderic D. M. Page, editors, *Algorithms in Bioinformatics*, pages 462–475, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- 13 Raluca Uricaru, Alban Mancheron, and Eric Rivals. Novel definition and algorithm for chaining fragments with proportional overlaps. *Journal of Computational Biology*, 18(9):1141–1154, 2011. doi:10.1089/cmb.2011.0126.


DAWGs for Parameterized Matching: Online Construction and Related Indexing Structures

Katsuhito Nakashima

Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
katsuhito_nakashima@shino.ecei.tohoku.ac.jp

Diptarama Hendrian 


Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Ryo Yoshinaka 

Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
ryoshinaka@tohoku.ac.jp

Hideo Bannai 

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

Masayuki Takeda 

Department of Informatics,
Kyushu University, Fukuoka, Japan
takeda@inf.kyushu-u.ac.jp

Noriki Fujisato

Department of Informatics,
Kyushu University, Fukuoka, Japan
noriki.fujisato@inf.kyushu-u.ac.jp

Yuto Nakashima 

Department of Informatics,
Kyushu University, Fukuoka, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Shunsuke Inenaga 

Department of Informatics,
Kyushu University, Fukuoka, Japan
PRESTO, Japan Science and Technology Agency,
Kawaguchi, Japan
inenaga@inf.kyushu-u.ac.jp

Ayumi Shinohara 

Graduate School of Information Sciences,
Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

Abstract

Two strings x and y over $\Sigma \cup \Pi$ of equal length are said to *parameterized match* (p -match) if there is a renaming bijection $f : \Sigma \cup \Pi \rightarrow \Sigma \cup \Pi$ that is identity on Σ and transforms x to y (or vice versa). The p -matching problem is to look for substrings in a text that p -match a given pattern. In this paper, we propose *parameterized suffix automata* (p -suffix automata) and *parameterized directed acyclic word graphs* (PDAWGs) which are the p -matching versions of suffix automata and DAWGs. While suffix automata and DAWGs are equivalent for standard strings, we show that p -suffix automata can have $\Theta(n^2)$ nodes and edges but PDAWGs have only $O(n)$ nodes and edges, where n is the length of an input string. We also give $O(n|\Pi| \log(|\Pi| + |\Sigma|))$ -time $O(n)$ -space algorithm that builds the PDAWG in a left-to-right online manner. As a byproduct, it is shown that the *parameterized suffix tree* for the reversed string can also be built in the same time and space, in a right-to-left online manner.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases parameterized matching, suffix trees, DAWGs, suffix automata

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.26

Related Version A full version of the paper is available at [17], <https://arxiv.org/abs/2002.06786>.

Funding *Diptarama Hendrian*: JSPS KAKENHI Grant Number JP19K20208.

Yuto Nakashima: JSPS KAKENHI Grant Number JP18K18002.

Ryo Yoshinaka: JSPS KAKENHI Grant Number JP18H04091.



© Katsuhito Nakashima, Noriki Fujisato, Diptarama Hendrian, Yuto Nakashima, Ryo Yoshinaka, Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, and Masayuki Takeda; licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 26; pp. 26:1–26:14



Leibniz International Proceedings in Informatics

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

Shunsuke Inenaga: JSPS KAKENHI Grant Number JP17H01697, JST PRESTO Grant Number JPMJPR1922.

Hideo Bannai: JSPS KAKENHI Grant Numbers JP16H02783, JP20H04141.

Ayumi Shinohara: JSPS KAKENHI Grant Number JP15H05706.

Masayuki Takeda: JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

The *parameterized matching problem* (*p-matching problem*) [2] is a class of pattern matching where the task is to locate substrings of a text that have “the same structure” as a given pattern. More formally, we consider a parameterized string (p-string) over a union of two disjoint alphabets Σ and Π for static characters and for parameter characters, respectively. Two equal length p-strings x and y are said to *parameterized match* (*p-match*) if x can be transformed to y (and vice versa) by a bijection which renames the parameter characters. The *p-matching problem* is, given a text p-string T and pattern p-string P , to report the occurrences of substrings of T that p-match P . P-matching is well-motivated by plagiarism detection, software maintenance, RNA structural pattern matching, and so on [2, 18, 15, 16].

The *parameterized suffix tree* (*p-suffix tree*) [1] is the fundamental indexing structure for p-matching, which supports p-matching queries in $O(m \log(|\Pi| + |\Sigma|) + pocc)$ time, where m is the length of pattern P , and $pocc$ is the number of occurrences to report. It is known that the p-suffix tree of a text w of length n can be built in $O(n \log(|\Pi| + |\Sigma|))$ time with $O(n)$ space in an offline manner [13] and in a *left-to-right online* manner [18]. A *randomized* $O(n)$ -time left-to-right online construction algorithm for p-suffix trees is also known [14]. Indexing p-strings has recently attracted much attention, and the p-matching versions of other indexing structures, such as *parameterized suffix arrays* [6, 12, 3, 9], *parameterized BWTs* [11], and *parameterized position heaps* [7, 8, 10], have also been proposed.

This paper fills in the missing pieces of indexing structures for p-matching, by proposing the parameterized version of the *directed acyclic word graphs* (DAWGs) [4, 5], which we call the *parameterized directed acyclic word graphs* (PDAWGs).

For any standard string T , the three following data structures are known to be equivalent:

- (1) The *suffix automaton* of T , which is the minimum DFA that is obtained by merging isomorphic subtrees of the suffix trie of T .
- (2) The DAWG, which is the edge-labeled DAG of which each node corresponds to a equivalence class of substrings of T defined by the set of ending positions in T .
- (3) The *Weiner-link graph*, which is the DAG consisting of the nodes of the suffix tree of the reversal \bar{T} of T and the reversed suffix links (a.k.a. soft and hard Weiner links).

The equality of (2) and (3) in turn implies symmetry of suffix trees and DAWGs, namely:

- (a) The suffix links of the DAWG for T form the suffix tree for \bar{T} .
- (b) Left-to-right online construction of the DAWG for T is equivalent to right-to-left online construction of the suffix tree for \bar{T} .

Firstly, we present (somewhat surprising) combinatorial results on the p-matching versions of data structures (1) and (2). We show that the *parameterized suffix automaton* (*p-suffix automaton*), which is obtained by merging isomorphic subtrees of the *parameterized suffix trie* of a p-string T of length n , can have $\Theta(n^2)$ nodes and edges in the worst case, while the PDAWG for any p-string has $O(n)$ nodes and edges. On the other hand, the p-matching versions of data structures (2) and (3) are equivalent: The *parameterized Weiner-link graph* of the p-suffix tree for \bar{T} is equivalent to the PDAWG for T . As a corollary to this, symmetry (a) also holds: The suffix links of the PDAWG for T form the p-suffix tree for \bar{T} .

Secondly, we present algorithmic results on PDAWG construction. We first propose left-to-right online construction of PDAWGs that works in $O(n|\Pi| \log(|\Pi| + |\Sigma|))$ time with $O(n)$ space. In addition, as a byproduct of this algorithm, we obtain a right-to-left online construction of the p-suffix tree in $O(n|\Pi| \log(|\Pi| + |\Sigma|))$ time with $O(n)$ space. This can be seen as the p-matching version of symmetry (b). We suspect that it is difficult to shave the $n|\Pi|$ term in the left-to-right online construction of PDAWGs, as well as in the right-to-left construction of p-suffix trees.

A full version of this work can be found in [17].

2 Preliminaries

We denote the set of all non-negative integers by \mathcal{N} . A linear order \prec over \mathcal{N} is identical to the ordinary linear order $<$ on integers except that 0 is always bigger than any other positive integers: $a \prec b$ if and only if $0 < a < b$ or $a \neq b = 0$. For a nonempty finite subset S of \mathcal{N} , $\max_{\prec} S$ and $\min_{\prec} S$ denote the maximum and minimum elements of S with respect to the order \prec , respectively.

We denote the set of strings over an alphabet A by A^* . For a string $w = xyz \in A^*$, x , y , and z are called *prefix*, *factor*, and *suffix* of w , respectively. The sets of the prefixes, factors, and suffixes of a string w are denoted by $\text{Prefix}(w)$, $\text{Factor}(w)$, and $\text{Suffix}(w)$, respectively. The length of w is denoted by $|w|$ and the i -th symbol of w is denoted by $w[i]$ for $1 \leq i \leq |w|$. The factor of w that begins at position i and ends at position j is $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, we abbreviate $w[1 : i]$ to $w[: i]$ and $w[i : |w|]$ to $w[i :]$ for $1 \leq i \leq |w|$. The empty string is denoted by ε , that is $|\varepsilon| = 0$. Moreover, let $w[i : j] = \varepsilon$ if $i > j$. The *reverse* \bar{w} of $w \in A^*$ is inductively defined by $\bar{\varepsilon} = \varepsilon$ and $\overline{a\bar{x}} = a\bar{x}$ for $a \in A$ and $x \in A^*$.

Throughout this paper, we fix two alphabets Σ and Π . We call elements of Σ *static* symbols and those of Π *parameter* symbols. Elements of Σ^* and $(\Sigma \cup \Pi)^*$ are called *static strings* and *parameterized strings* (or *p-strings* for short), respectively.

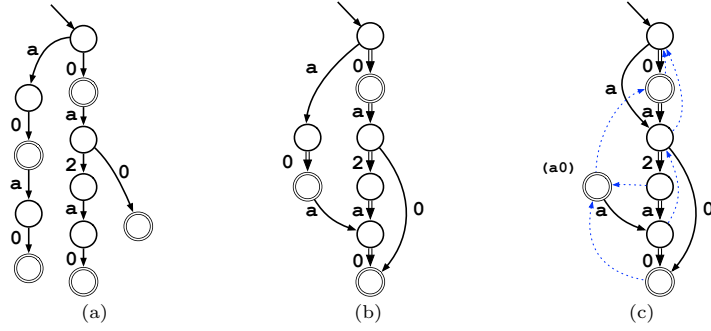
Given two p-strings S_1 and S_2 of length n , S_1 and S_2 are a *parameterized match* (*p-match*), denoted by $S_1 \approx S_2$, if there is a bijection f on $\Sigma \cup \Pi$ such that $f(a) = a$ for any $a \in \Sigma$ and $f(S_1[i]) = S_2[i]$ for all $1 \leq i \leq n$ [2]. The *prev-encoding* $\text{prev}(S)$ of a p-string S is the string over $\Sigma \cup \mathcal{N}$ of length $|S|$ defined by

$$\text{prev}(S)[i] = \begin{cases} S[i] & \text{if } S[i] \in \Sigma, \\ 0 & \text{if } S[i] \in \Pi \text{ and } S[i] \neq S[j] \text{ for } 1 \leq j < i, \\ i - j & \text{if } S[i] = S[j] \in \Pi, j < i \text{ and } S[i] \neq S[k] \text{ for any } j < k < i \end{cases}$$

for $i \in \{1, \dots, |S|\}$. We call a string $x \in (\Sigma \cup \mathcal{N})^*$ a *pv-string* if $x = \text{prev}(S)$ for some p-string S . For any p-strings S_1 and S_2 , $S_1 \approx S_2$ if and only if $\text{prev}(S_1) = \text{prev}(S_2)$ [2]. For example, given $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $\Pi = \{\mathbf{u}, \mathbf{v}, \mathbf{x}, \mathbf{y}\}$, $S_1 = \mathbf{uvvauvb}$ and $S_2 = \mathbf{xyyaxyb}$ are a p-match by f such that $f(\mathbf{u}) = \mathbf{x}$ and $f(\mathbf{v}) = \mathbf{y}$, where $\text{prev}(S_1) = \text{prev}(S_2) = 001\mathbf{a}43\mathbf{b}$. For a p-string T , let $\text{PFactor}(T) = \{\text{prev}(S) \mid S \in \text{Factor}(T)\}$ and $\text{PSuffix}(T) = \{\text{prev}(S) \mid S \in \text{Suffix}(T)\}$ be the sets of prev-encoded factors and suffixes of T , respectively. For a factor $x \in (\Sigma \cup \mathcal{N})^*$ of a pv-string, the *re-encoding* $\langle x \rangle$ of x is the pv-string of length $|x|$ defined by $\langle x \rangle[i] = Z(x[i], i - 1)$ for $i \in \{1, \dots, |x|\}$ where

$$Z(a, j) = \begin{cases} 0 & \text{if } a \in \mathcal{N} \text{ and } a > j, \\ a & \text{otherwise.} \end{cases}$$

We then have $\langle \text{prev}(T)[i : j] \rangle = \text{prev}(T[i : j])$ for any i, j . We apply PFactor etc. to pv-strings w so that $\text{PFactor}(w) = \{\langle x \rangle \mid x \in \text{Factor}(w)\}$.



■ **Figure 1** (a) The parameterized suffix trie $\text{PSTrie}(T)$, (b) the parameterized suffix automaton $\text{PSAuto}(T)$ and (c) the PDAWG $\text{PDAWG}(T)$ for $T = \text{xaxay}$ over $\Sigma = \{\mathbf{a}\}$ and $\Pi = \{\mathbf{x}, \mathbf{y}\}$, for which $\text{prev}(T) = 0\mathbf{a}2\mathbf{a}0$. Solid and broken arrows represent the edges and suffix links, respectively. Some nodes of $\text{PDAWG}(T)$ cannot be reached by following the edges from the source node.

Let $w, x, y \in (\Sigma \cup \mathcal{N})^*$. A symbol $a \in \Sigma \cup \mathcal{N}$ is said to be a *right extension* of x w.r.t. w if $xa \in \text{PFactor}(w)$. The set of the right extensions of x is denoted by $\text{REx}_w(x)$. The set of the *end positions* of x in a pv-string w is defined by $\text{RPos}_w(x) = \{i \in \{0, \dots, |w|\} \mid x = \langle w[i - |x| + 1 : i] \rangle\}$. Note that $0 \in \text{RPos}_w(x)$ iff $x = \varepsilon$. It is easy to see that $\text{RPos}_w(x) \subseteq \text{RPos}_w(y)$ if and only if $y \in \text{PSuffix}(x)$ or $\text{RPos}_w(x) = \text{RPos}_w(y)$. We write $x \equiv_w^R y$ iff $\text{RPos}_w(x) = \text{RPos}_w(y)$, and the equivalence class of pv-strings w.r.t. \equiv_w^R as $[x]_w^R$. Note that for any $x \notin \text{PFactor}(w)$, $[x]_w^R$ is the infinite set of all the pv-strings outside $\text{PFactor}(w)$. For a finite nonempty set X of strings which has no distinct elements of equal length, the shortest and longest elements of X are denoted by $\min X$ and $\max X$, respectively.

A basic indexing structure of a p-strings is a *parameterized suffix trie*. The parameterized suffix trie $\text{PSTrie}(T)$ is the trie for $\text{PSuffix}(T)$. That is, $\text{PSTrie}(T)$ is a tree (V, E) whose node set is $V = \text{PSuffix}(T)$ and edge set is $E = \{(x, a, xa) \in V \times (\Sigma \cup \mathcal{N}) \times V\}$. An example can be found in Figure 1 (a). Like the standard suffix tries for static strings, the size of $\text{PSTrie}(T)$ can be $\Theta(|T|^2)$. Obviously we can check whether T has a substring that p-matches P of length m in $O(m \log(|\Sigma| + |\Pi|))$ time using $\text{PSTrie}(T)$, assuming that finding the edge to traverse for a given character takes $O(\log(|\Sigma| + |\Pi|))$ time by, e.g., using balanced trees. We use the same assumption on other indexing structures considered in this paper.

3 Parameterized DAWG

3.1 Parameterized suffix automata

One natural idea to define the parameterized counterpart of DAWGs for p-strings, which we actually do not take, is to merge isomorphic subtrees of parameterized suffix tries. In other words, the parameterized suffix automaton of T , denoted by $\text{PSAuto}(T)$, is the minimal deterministic finite automaton that accepts $\text{PSuffix}(T)$. Figure 1 (b) shows an example of a parameterized suffix automaton. However, the size of $\text{PSAuto}(T)$ can be $\Theta(|T|^2)$, as witnessed by a p-string $T_k = \mathbf{x}_1 \mathbf{a}_1 \dots \mathbf{x}_k \mathbf{a}_k \mathbf{x}_1 \mathbf{a}_1 \dots \mathbf{x}_k \mathbf{a}_k$ over $\Sigma = \{\mathbf{a}_1, \dots, \mathbf{a}_k\}$ and $\Pi = \{\mathbf{x}_1, \dots, \mathbf{x}_k\}$.

3.2 Parameterized directed acyclic word graphs

In this section, we present a new indexing structure for parameterized strings, which we call *parameterized directed acyclic word graphs (PDAWGs)*. A PDAWG can be obtained from a parameterized suffix trie by merging nodes whose ending positions are the same.

In the example of Figure 1 (a), the subtrees rooted at \mathbf{a} and $0\mathbf{a}$ have different shapes but $\text{RPos}_w(\mathbf{a}) = \text{RPos}_w(0\mathbf{a}) = \{2, 4\}$. Particularly, the 0-edges of those two nodes point at nodes $\mathbf{a}0$ and $0\mathbf{a}0$ with different ending position sets, which shall not be merged. Our solution to this obvious conflict is to use only the edges of the “representative” node among ones with the same ending position sets. In the example, we take out-going edges of $0\mathbf{a}$ and do not care those of \mathbf{a} . The resultant PDAWG by our solution is shown in Figure 1 (c). This might first appear nonsense: by reading $\mathbf{a}0$, whose ending positions are 3 and 5, one will reach to the sink node, whose ending position is 5, and consequently one will reach no node by reading $\mathbf{a}0\mathbf{a} \in \text{PFactor}(T)$. We will argue in the next subsection that still we can correctly perform parameterized matching using our PDAWG by presenting a p-matching algorithm.

► **Definition 1** (Parameterized directed acyclic word graphs). *Let $w = \text{prev}(T)$ for a parameterized text $T \in (\Sigma \cup \Pi)^*$. The parameterized directed acyclic word graph (PDAWG) $\text{PDAWG}(T) = \text{PDAWG}(w)$ of T is the directed acyclic graph (V_w, E_w) where*

$$V_w = \{ [x]_w^R \mid x \in \text{PFactor}(w) \},$$

$$E_w = \{ ([x]_w^R, c, [y]_w^R) \in V \times (\Sigma \cup \mathcal{N}) \times V \mid y = \max[x]_w^R \cdot c \}$$

together with suffix links

$$\text{SL}_w([x]_w^R) = [\langle y[2 : |y|] \rangle_w^R \text{ where } y = \min[x]_w^R.$$

The nodes $[\varepsilon]_w^R$ and $[w]_w^R$ are called the source and the sink, respectively. Suffix links are defined on non-source nodes.

PDAWGs have the same size bound as DAWGs, shown by Blumer et al. [4].

► **Theorem 2.** *$\text{PDAWG}(T)$ has at most $2n - 1$ nodes and $3n - 4$ edges when $n = |T| \geq 3$. Those bounds are tight.*

By definition, a node u has an out-going edge labeled with a if and only if $a \in \text{REx}_w(\max u)$. For $a \in \text{REx}_w(\max u)$, by $\text{child}_w(u, a)$ we denote the unique element v such that $(u, a, v) \in E_w$. For $a \notin \text{REx}_w(\max u)$, we define $\text{child}_w(u, a) = \text{Null}$. For any $u \in V_w$, $\text{RPos}_w(\text{SL}_w(u))$ is the least proper superset of $\text{RPos}_w(u)$. The reversed suffix links form a tree with root $[\varepsilon]_w^R$. Actually, the tree is isomorphic to the parameterized suffix tree [2] for \bar{T} . We discuss the duality between PDAWGs and parameterized suffix tree in more detail in Subsection 3.5.

3.3 Parameterized pattern matching with PDAWGs

This subsection discusses how we can perform p-matching using our PDAWGs: We must reach a node $[x]_w^R \in V_w$ by reading $x \in \text{PFactor}(w)$ and reach no node if $x \notin \text{PFactor}(w)$. In DAWGs for static strings, by following the a -edge of $[x]_w^R$, we will arrive in $[xa]_w^R$, which is guaranteed by the fact that $x \equiv_w^R y$ implies $xa \equiv_w^R ya$. However, this does not hold for pv-strings. For instance, for $w = \text{prev}(\mathbf{xaxay}) = 0\mathbf{a}2\mathbf{a}0$ ($\mathbf{a} \in \Sigma$ and $\mathbf{x}, \mathbf{y} \in \Pi$), we see $\text{RPos}_w(\mathbf{a}) = \text{RPos}_w(0\mathbf{a}) = \{2, 4\}$ but $3 \in \text{RPos}_w(\mathbf{a}0) \setminus \text{RPos}_w(0\mathbf{a}0)$. Consequently $\mathbf{a}0\mathbf{a} \in \text{PFactor}(w)$ but $0\mathbf{a}0\mathbf{a} \notin \text{PFactor}(w)$. By definition, if we reach a node u by reading $\max u$, we can simply follow the a -edge by reading a symbol a , similarly to matching using a DAWG. We may behave differently after we have reached u by reading some other string in u . The following lemma suggests how we can perform p-matching using $\text{PDAWG}(T)$.

■ **Algorithm 1** Parameterized pattern matching algorithm based on PDAWG(T).

```

1  $p \leftarrow \text{prev}(P)$ ;
2 Let  $u \leftarrow [\varepsilon]_w^R$ ;
3 for  $i = 1$  to  $|P|$  do
4   Let  $u \leftarrow \text{Trans}(u, i - 1, p[i])$ ;
5   if  $u = \text{Null}$  then return False;
6 return True;
```

■ **Algorithm 2** Function $\text{Trans}(u, i, a)$.

```

1 if  $a \neq 0$  then return  $\text{child}(u, a)$ ;
2 else if there is no  $b \in \text{rex}(u)$  such that  $b \succ i$  then return Null;
3 else if there is only one  $b \in \text{rex}(u)$  such that  $b \succ i$  then return  $\text{child}(u, b)$ ;
4 else return  $\text{SL}(\text{child}(u, b))$  for the smallest (w.r.t.  $\prec$ )  $b \in \text{rex}(u)$  such that  $b \succ i$ ;
```

► **Lemma 3.** Suppose $x \in \text{PFactor}(w)$ and $a \in \Sigma \cup \mathcal{N}$. Then, for $y = \max[x]_w^R$,

$$[xa]_w^R = \begin{cases} [ya]_w^R & \text{if } a \neq 0 \text{ or } W = \emptyset, \\ [yk]_w^R & \text{if } a = 0 \text{ and } |W| = 1, \\ \text{SL}_w([yk]_w^R) & \text{if } a = 0 \text{ and } |W| \geq 2, \end{cases}$$

where $W = \{j \in \mathcal{N} \mid yj \in \text{PFactor}(w) \text{ and } j \succ |x|\}$ and $k = \min_{\prec} W$.

Proof. We first show for $x \in \text{PFactor}(w)$, $a \in \Sigma \cup \mathcal{N}$ and $y = \max[x]_w^R$,

$$\text{RPos}_w(xa) = \begin{cases} \bigcup_{k \in W} \text{RPos}_w(yk) & \text{if } a = 0, \\ \text{RPos}_w(ya) & \text{otherwise.} \end{cases} \quad (1)$$

where $W = \{k \in \mathcal{N} \mid yk \in \text{PFactor}(w) \text{ and } k \succ |x|\}$.

For $a \in \Sigma$, $i \in \text{RPos}_w(xa)$ iff both $i - 1 \in \text{RPos}_w(x) = \text{RPos}_w(y)$ and $T[i] = a$ hold iff $i \in \text{RPos}_w(ya)$. For $a \in \mathcal{N} \setminus \{0\}$, noting that $0 < a \leq |x|$, we have $i \in \text{RPos}_w(xa)$ iff

$$i - 1 \in \text{RPos}_w(x) = \text{RPos}_w(y), T[i] = T[i - a] \in \Pi \text{ and } T[i - b] \neq T[i] \text{ for all } 0 < b < a$$

iff $i \in \text{RPos}_w(ya)$. For $a = 0$, $i \in \text{RPos}_w(xa)$ iff

$$i - 1 \in \text{RPos}_w(x) = \text{RPos}_w(y), T[i] \in \Pi, \text{ and } T[i] \neq T[j] \text{ for all } i - |x| < j < i$$

iff $i \in \text{RPos}_w(yk)$ for some $k \succ |x|$. This proves Eq. (1).

If $a \neq 0$ or $|W| \leq 1$, we obtain the lemma immediately from Eq. (1). Suppose $a = 0$ and $|W| \geq 2$ and let $k = \min_{\prec} W$. By Eq. (1), we see that $\text{RPos}_w(yk) \subsetneq \text{RPos}_w(x0)$, where $k \succ |x|$. It is enough to show that for any z , $\text{RPos}_w(yk) \subseteq \text{RPos}_w(z)$ implies either $\text{RPos}_w(yk) = \text{RPos}_w(z)$ or $\text{RPos}_w(x0) \subseteq \text{RPos}_w(z)$. Since $\text{RPos}_w(yk) \subseteq \text{RPos}_w(z)$, $z \in \text{PSuffix}(yk)$. If $z = z'k$, by $|x| < k < |z'| \leq |y|$, $z \in [yk]_w^R$. Then, Eq. (1) implies $\text{RPos}_w(z'k) = \text{RPos}_w(yk)$. Suppose $z = z'0$ for some z' . If $|z| \leq |x0|$, $\text{RPos}_w(x0) \subseteq \text{RPos}_w(z)$. Otherwise, $|x0| < |z| < |yk|$ implies $z' \in [y]_w^R$. By Eq. (1), $|z'| < k$ implies $\text{RPos}_w(x0) \subseteq \text{RPos}_w(z)$ by the choice of k . ◀

The function Trans of Algorithm 2 is a straightforward realization of Lemma 3. By $\text{rex}(u)$ we denote the set of labels of the out-going edges of u , i.e., $\text{rex}(u) = \text{REx}_w(\max u)$. It takes a node $u \in V$, a natural number $i \in \mathcal{N}$, and a symbol $a \in \Sigma \cup \mathcal{N}$, and returns the node

where we should go by reading a from u assuming that we have read i symbols so far. That is, $\text{Trans}([x]_w^R, |x|, a) = [xa]_w^R$ for every $xa \in \text{PFactor}(w)$. Using Trans , Algorithm 1 performs p-matching. We can locate the node v of the PDAWG in $O(m \log(|\Sigma| + |\Pi|))$ for a given pattern P of length m if it has a p-matching occurrence, or can determine that P does not have such an occurrence. In case P has a p-matching occurrence, we can actually report all of its occurrences by traversing the subtree of the (reversed) suffix links that is rooted at the node v , since the reversed suffix link tree of $\text{PDAWG}(T)$ forms the p-suffix tree of \bar{T} (see Subsection 3.5). Thus we obtain the following:

► **Theorem 4.** *Using $\text{PDAWG}(T)$ enhanced with the suffix links, we can find all substrings of T that p-match a given pattern P in $O(m \log(|\Sigma| + |\Pi|) + \text{pocc})$ time, where m is the length of pattern P and pocc is the number of occurrences to report.*

3.4 Online algorithm for constructing PDAWGs

This subsection proposes an algorithm constructing the PDAWG online. Our algorithm is based on the one by Blumer et al. [4] for constructing DAWGs of static strings. We consider updating $\text{PDAWG}(w)$ to $\text{PDAWG}(wa)$ for a pv-string wa where $a \in \Sigma \cup \mathcal{N}$.

We first observe properties similar to the DAWG construction.

► **Definition 5.** *The longest repeated suffix (LRS) of a nonempty pv-string $wa \in (\Sigma \cup \mathcal{N})^+$ is defined to be $\text{LRS}(wa) = \max(\text{PSuffix}(wa) \cap \text{PFactor}(w))$. If $\text{LRS}(wa) \neq \varepsilon$, the string obtained from $\text{LRS}(wa)$ by removing the last symbol is called the pre-LRS w.r.t. wa and denoted as $\text{preLRS}(wa) = \text{LRS}(wa)[: |\text{LRS}(wa)| - 1]$.*

Note that the pre-LRS w.r.t. wa is a suffix of w and is defined only when $\text{LRS}(wa) \neq \varepsilon$. We have $\text{LRS}(wa) = \varepsilon$ if and only if a is new in the sense that $wa \in \Sigma^* \{0\} \cup (\Sigma \cup \mathcal{N} \setminus \{a\})^* \Sigma$.

The following lemma for node splits on PDAWGs is an analogue to that for DAWGs.

► **Lemma 6 (Node update).** *For $x = \text{LRS}(wa)$ and $y = \max[x]_w^R$,*

$$V_{wa} = V_w \setminus \{[x]_w^R\} \cup \{[x]_{wa}^R, [y]_{wa}^R, [wa]_{wa}^R\}.$$

If $x = y$, then $[x]_w^R = [x]_{wa}^R = [y]_{wa}^R$, i.e., $V_{wa} = V_w \cup \{[wa]_{wa}^R\}$. Otherwise, $[x]_w^R = [x]_{wa}^R \cup [y]_{wa}^R$ and $[x]_{wa}^R \neq [y]_{wa}^R$.

Proof. First remark that $\text{RPos}_{wa}(z) = \text{RPos}_w(z) \cup \{[wa]\}$ for all $z \in \text{PSuffix}(wa)$ and $\text{RPos}_{wa}(z) = \text{RPos}_w(z)$ for all $z \notin \text{PSuffix}(wa)$. For those $z \in \text{PSuffix}(wa) \setminus \text{PFactor}(w)$, we have $\text{RPos}_{wa}(z) = \{[wa]\}$ and $[wa]_{wa}^R = \text{PSuffix}(wa) \setminus \text{PFactor}(w) \in V_{wa} \setminus V_w$. For $z \in \text{PFactor}(w)$, if $[z]_w^R \neq [z]_{wa}^R$, some elements of $[z]_w^R$ are in $\text{PSuffix}(wa)$ and some are not. That is, $[z]_w^R$ is partitioned into two non-empty equivalence classes $\{z' \in [z]_w^R \mid z' \in \text{PSuffix}(wa)\}$ and $\{z' \in [z]_w^R \mid z' \notin \text{PSuffix}(wa)\}$. By definition, the longest of the former is $x = \text{LRS}(wa)$ and the longest of the latter is $y = \max[x]_w^R$. Otherwise, $[z]_w^R = [z]_{wa}^R \in V_w \cap V_{wa}$. ◀

► **Example 7.** Let $w = 0a2a$ and $a = 0$. Then $\text{LRS}(wa) = \langle w[2 : 3] \rangle = \langle wa[4 : 5] \rangle = a0$. We have $\text{LRS}(wa) \neq \max[\text{LRS}(wa)]_w^R = 0a2$, where $\text{RPos}_w(a0) = \text{RPos}_w(0a2) = \{3\}$. On the other hand, $\text{RPos}_{wa}(a0) = \{3, 5\} \neq \text{RPos}_{wa}(0a2) = \{3\}$. Therefore, $\text{PDAWG}(wa)$ has two more nodes than $\text{PDAWG}(w)$.

When updating $\text{PDAWG}(w)$ to $\text{PDAWG}(wa)$, all edges that do not involve the node $[\text{LRS}(wa)]_w^R$ are kept by definition. What we have to do is to manipulate in-coming edges for the new sink node $[wa]_{wa}^R$, and, if necessary, to split the LRS node $[\text{LRS}(wa)]_w^R$ into two

and to manipulate in-coming and out-going edges of them. Therefore, it is very important to identify the LRS node $[\text{LRS}(wa)]_w^R$ and to decide whether $\text{LRS}(wa) = \max[\text{LRS}(wa)]_w^R$. The special case where $\text{LRS}(wa) = \varepsilon$ is easy to handle, since the LRS node will never be split by $[\varepsilon]_w^R = \{\varepsilon\}$. Hereafter we assume that $\text{LRS}(wa) \neq \varepsilon$ and $\text{preLRS}(wa)$ is defined. The LRS node can be reached from the pre-LRS node $[\text{preLRS}(wa)]_w^R$, which can be found by following suffix links from the sink node $[w]_w^R$ of $\text{PDAWG}(w)$. This appears quite similar to online construction of DAWGs for static strings, but there are nontrivial differences. Main differences from the DAWG construction are in the following points:

- Our PDAWG construction uses $\text{Trans}_w(u, i, Z(a, i))$ with an appropriate i , when the original DAWG construction refers to $\text{child}_w(u, a)$,
- While $\text{preLRS}(wa)$ is the longest of its equivalence class for static strings in $\text{DAWG}(w)$, it is not necessarily the case for p-strings (like the one in Figure 1), which affects the procedure to find the node of $\text{LRS}(wa)$,
- When a node of $\text{PDAWG}(w)$ is split into two in $\text{PDAWG}(wa)$, the out-going edges of the two nodes are identical in the DAWG construction, while it is not necessarily the case any more in our PDAWG construction. Moreover, we do not always have an edge from the node of $\text{preLRS}(wa)$ to that of $\text{LRS}(wa)$ in $\text{PDAWG}(wa)$.

In DAWGs, the pre-LRS node is the first node with an a -edge that can be found by recursively following the suffix links from the old sink $[w]_w^R$. However, it is not necessarily the case for PDAWGs. The following lemma suggests how to find $[\text{LRS}(wa)]_w^R$ and $|\text{LRS}(wa)|$ and how to decide whether the node shall be split.

► **Lemma 8.** *Let $x' = \text{preLRS}(wa)$, $a' = Z(a, |x'|)$, i.e., $x'a' = \text{LRS}(wa)$, and $u_i = \text{SL}_w^i(w)$ for $i \geq 0$.*

1. *We have $x' \in u_i$ for the least i such that $\text{Trans}_w(u_i, |\min u_i|, Z(a, |\min u_i|)) \neq \text{Null}$,*
2. $|x'a'| = \begin{cases} |\max[x']_w^R| + 1 & \text{if } a \in \text{REx}_w(\max[x']_w^R), \\ \min_{\prec}\{a, \max(\text{REx}(\max[x']_w^R) \cap \mathcal{N})\} & \text{otherwise,} \end{cases}$
3. $[x'a']_w^R = \text{Trans}_w(u_i, |x'|, a')$,
4. $[x'a']_w^R \neq [x'a']_{wa}^R$ if and only if $|x'a'| \neq |\max[x'a']_w^R|$.

Proof. Suppose $x' \in u_i$.

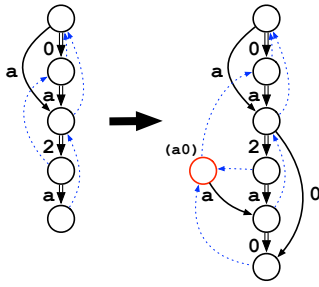
(1) Every string $z \in u_j$ with $j < i$ is properly longer than x' , so $z \cdot Z(a, |z|) \notin \text{PFactor}(w)$ by definition. On the other hand, for $z = \min u_i$, the fact $z \cdot Z(a, |z|) \in \text{PFactor}(x'a')$ implies $\text{Trans}_w(u_i, |z|, Z(a, |z|)) \neq \text{Null}$.

(2) If $a \in \text{REx}_w(y')$ for $y' = \max[x']_w^R$, we have $y'a \in \text{PFactor}(w)$ and thus $y'a = x'a'$. Suppose $a \notin \text{REx}_w(y')$. In this case, $[x']_w^R$ is not a singleton and thus not the source node, i.e., $|x'| \neq 0$. We have $\text{Trans}_w(u_i, |x'|, a') \neq \text{Trans}_w(u_i, |x'| + 1, Z(a, |x'| + 1)) = \text{Null}$ and thus $a \in \mathcal{N}$. Let $W = \text{REx}_w(\max u_i) \cap \mathcal{N}$ and $W_j = \{k \in W \mid k \succ j\}$. Lemma 3 implies that $a' = 0$ and $W_{|x'|} \neq \emptyset$ by $\text{Trans}_w(u_i, |x'|, a') \neq \text{Null}$. If $W_{|x'|+1} \neq \emptyset$, then $Z(a, |x'| + 1) = a \neq 0 = Z(a, |x'|)$, i.e., $|x'| = a - 1$. By $W_{|x'|} \neq \emptyset$, $|x'| = a - 1 \prec \max_{\prec} W_{|x'|}$. Therefore, $|x'| = \min_{\prec}\{a, \max_{\prec} W\} - 1$. If $W_{|x'|+1} = \emptyset \neq W_{|x'|}$, then $0 \notin W$ and $\max_{\prec} W = |x'| + 1$. By $Z(a, |x'|) = 0$, $|x'| \prec a$. Therefore, $|x'| = \min_{\prec}\{a, \max_{\prec} W\} - 1$.

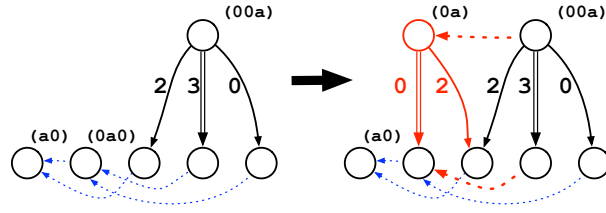
(3) By Lemma 3. (4) By Lemma 6. ◀

Edges are created or replaced in accordance with the definition of a PDAWG. The in-coming edges for the new sink node $[wa]_{wa}^R$ of $\text{PDAWG}(wa)$ are given as follows.

► **Lemma 9 (In-coming edges of the new sink).** *If $\text{LRS}(wa) \neq \varepsilon$, the in-coming edges for the new sink $[wa]_{wa}^R$ are exactly those $(u, Z(a, |\max u|), [wa]_{wa}^R)$ such that $u = \text{SL}_w^i([w]_w^R)$ for some $i \geq 0$ and $\text{child}(u, Z(a, |\max u|)) = \text{Null}$, i.e., $|\max u| > |\text{preLRS}(wa)|$. If $\text{LRS}(wa) = \varepsilon$, the in-coming edges for $[wa]_{wa}^R$ are exactly those $(\text{SL}_w^i([w]_w^R), a, [wa]_{wa}^R)$ for all $i \geq 0$.*



■ **Figure 2** PDAWG(w), PDAWG(wa) for $w = 0a2a$, $a = 0$. $\text{LRS}(wa) = x = a0$, $\text{preLRS}(wa) = x' = a$ and $y = \max[x]_w^R = 0a2$.



■ **Figure 3** Parts of PDAWG(w) and PDAWG(wa) for $w = 00a30a20a0$, $a = a$. $[0a]_{wa}^R$ does not inherit the out-going edges of $[0a]_w^R$ labeled with 3 and 0. Instead, the 3-edge and 0-edge are bundled into a single 0-edge which points at $\text{Trans}_w([0a]_w^R, 2, 0) = \text{SL}([00a3]_w^R) = [0a0]_w^R$.

This is not much different from DAWG update, except that the pre-LRS node has an edge towards the new sink when the pre-LRS is not the longest in the pre-LRS node (see Figure 2, where the pre-LRS node $[x']_{wa}^R$ has got a 0-edge towards the sink). If the LRS node $[\text{LRS}(wa)]_{wa}^R$ is not split, we have nothing more to do on edges.

Hereafter, we suppose that the LRS node must be split. That is, $x \neq y$ for $x = \text{LRS}(wa)$ and $y = \max[\text{LRS}(wa)]_w^R$. By definition, all edges of PDAWG(w) that do not involve the LRS node $[\text{LRS}(wa)]_w^R$ will be inherited to PDAWG(wa). The nodes $[x]_{wa}^R$ and $[y]_{wa}^R$ in PDAWG(wa) will have the following in-coming and out-going edges.

► **Lemma 10** (In-coming edges of the LRS node). *We have*

- $(u, b, [y]_{wa}^R) \in E_{wa}$ if and only if $(u, b, [y]_w^R) \in E_w$ and $|\max u| + 1 > |x|$,
- $(u, b, [x]_{wa}^R) \in E_{wa}$ if and only if $b = Z(a, |\max u|)$, $(u, b, [y]_w^R) \in E_w$ and $|\max u| + 1 \leq |x|$.

Lemma 10 is no more than a direct implication of the definition of edges of PDAWGs. An important fact is that $(u, b, [y]_w^R) \in E_w$ only if $u = \text{SL}_w^i([x']_w^R)$ with $x' = \text{preLRS}(wa)$ for some $i \geq 0$, which is essentially no difference from the DAWG case. Therefore, one can find all in-coming edges that may need to manipulate by following suffix links from the pre-LRS node. Note that in the on-line construction of a DAWG, the edge from the pre-LRS node $[x']_w^R$ to the LRS node $[y]_w^R$ in the old DAWG will be inherited to the new node $[x]_{wa}^R$ in the new DAWG. However, it is not necessarily the case in the PDAWG construction, as demonstrated in Figure 2, where the 2-edge from $[x']_w^R$ to $[y]_w^R$ in PDAWG(w) is kept as the 2-edge from $[x']_{wa}^R$ to $[y]_{wa}^R$ in PDAWG(wa) and, as a result, the new node $[x]_{wa}^R$ has no in-coming edges.

► **Lemma 11** (Out-going edges of the LRS node). *We have*

- $([y]_{wa}^R, b, u) \in E_{wa}$ if and only if $([y]_w^R, b, u) \in E_w$,
- $([x]_{wa}^R, b, u) \in E_{wa}$ if and only if $\text{Trans}([y]_w^R, |x|, b) = u$ if and only if either $([y]_w^R, b, u) \in E_w$ and $Z(b, |x|) \neq 0$ or $\text{Trans}([y]_w^R, |x|, 0) = u$ and $Z(b, |x|) = 0$.

Lemma 11 is also an immediate consequence of the definition of PDAWG edges. In the DAWG construction, those two nodes $[x]_{wa}^R$ and $[y]_{wa}^R$ simply inherit the out-going edges of the LRS node $[x]_w^R = [y]_w^R$. However, in the PDAWG construction, due to the prev-encoding rule on variable symbols, the node $[x]_{wa}^R$ will lose edges whose labels are integers greater than $|x|$, as demonstrated in Figure 3. Those edges are “bundled” into a single 0-edge which points at $\text{Trans}([y]_w^R, 0, |x|)$.

Updates of suffix links simply follow the definition.

■ **Algorithm 3** Constructing PDAWG(T).

```

1 Let  $V \leftarrow \{\top, \rho\}$ ,  $E \leftarrow \{(\top, a, \rho) \mid a \in \Sigma \cup \{0\}\}$ ,  $\text{SL}(\rho) = \top$ ,  $\text{len}(\top) = -1$ ,  $\text{len}(\rho) = 0$ ,
    $\text{sink} \leftarrow \rho$ , and  $t \leftarrow \text{prev}(T)$ ;
2 for  $i \leftarrow 1$  to  $|t|$  do
3   Let  $a \leftarrow t[i]$  and  $u \leftarrow \text{sink}$ ;
4   Create a new node and let  $\text{sink}$  be that node with  $\text{len}(\text{sink}) = i$ ;
5   while  $\text{Trans}(u, \text{len}(\text{SL}(u)) + 1, Z(a, \text{len}(\text{SL}(u)) + 1)) = \text{Null}$  do
6     Let  $\text{child}(u, Z(a, \text{len}(u))) \leftarrow \text{sink}$  and  $u \leftarrow \text{SL}(u)$ ;
   //  $u$  corresponds to  $[\text{preLRS}(t[:i])]_{t[:i-1]}^R$ 
7   if  $Z(a, \text{len}(u)) \in \text{rex}(u)$  then //  $\text{preLRS}(t[:i]) = \max[\text{preLRS}(t[:i])]_{t[:i-1]}^R$ 
8     Let  $k \leftarrow \text{len}(u) + 1$  and  $v \leftarrow \text{child}(u, Z(a, \text{len}(u)))$ 
9   else //  $\text{preLRS}(t[:i]) \neq \max[\text{preLRS}(t[:i])]_{t[:i-1]}^R$ 
10    Let  $k \leftarrow \min_{\prec} \{a, \max(\text{rex}(u) \cap \mathcal{N})\}$ ,  $v \leftarrow \text{Trans}(u, k - 1, 0)$ ,
     $\text{child}(u, Z(a, \text{len}(u))) \leftarrow \text{sink}$ , and  $u \leftarrow \text{SL}(u)$ ;
   //  $v$  corresponds to  $[\text{LRS}(t[:i])]_{t[:i-1]}^R$  and  $k = |\text{LRS}(t[:i])|$ 
11  if  $\text{len}(v) = k$  then Let  $\text{SL}(\text{sink}) \leftarrow v$ ; // No node split
12  else // Node split
13    Create a new node  $v'$ ; //  $v'$  corresponds to  $[\text{LRS}(t[:i])]_{t[:i]}^R$ 
14    Let  $\text{len}(v') \leftarrow k$ ;
    // In-coming edges of the new node
15    while  $\text{child}(u, Z(a, \text{len}(u))) = v$  do
16      Let  $\text{child}(u, Z(a, \text{len}(u))) \leftarrow v'$  and  $u \leftarrow \text{SL}(u)$ ;
    // Out-going edges of the new node
17    for each  $b \in \{b \in \text{rex}(v) \mid Z(b, k) \neq 0\}$  do
18      Let  $\text{child}(v', b) \leftarrow \text{child}(v, b)$ ;
19    if  $\text{Trans}(v, k, 0) \neq \text{Null}$  then Let  $\text{child}(v', 0) \leftarrow \text{Trans}(v, k, 0)$ ;
    // Suffix links
20    Let  $\text{SL}(v') \leftarrow \text{SL}(v)$ ,  $\text{SL}(v) \leftarrow v'$  and  $\text{SL}(\text{sink}) \leftarrow v'$ ;
21 return  $(V, E, \text{SL})$ ;
```

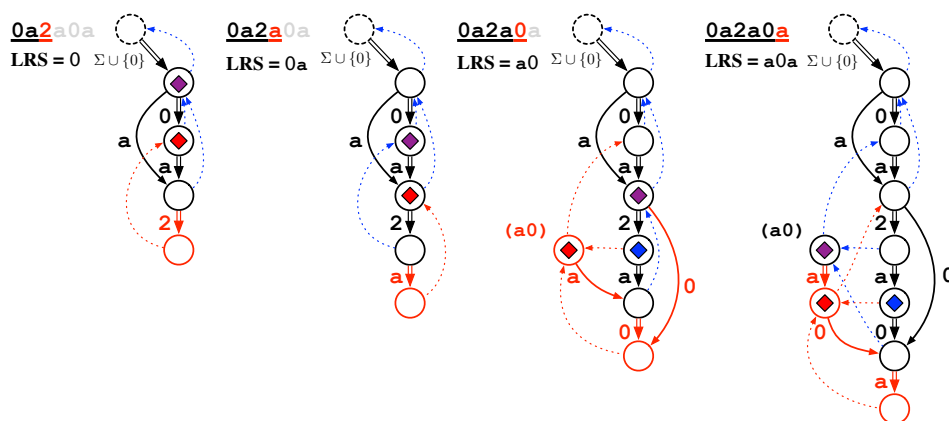
► **Lemma 12** (Suffix link update). Suppose $V_{wa} = V_w \cup \{[wa]_{wa}^R\}$. Then, for each $u \in V_{wa}$,

$$\text{SL}_{wa}(u) = \begin{cases} [\text{LRS}(wa)]_{wa}^R & \text{if } u = [wa]_{wa}^R, \\ \text{SL}_w(u) & \text{otherwise.} \end{cases}$$

Suppose $[x]_w^R \neq [x]_{wa}^R$ for $x = \text{LRS}(wa)$, i.e., $V_{wa} = V_w \setminus \{[y]_w^R\} \cup \{[wa]_{wa}^R, [x]_{wa}^R, [y]_{wa}^R\}$, where $y = \max[x]_w^R$. Then, for each $u \in V_{wa}$,

$$\text{SL}_{wa}(u) = \begin{cases} [x]_{wa}^R & \text{if } u \in \{[wa]_{wa}^R, [y]_{wa}^R\}, \\ \text{SL}_w([y]_w^R) & \text{if } u = [x]_{wa}^R, \\ \text{SL}_w(u) & \text{otherwise.} \end{cases}$$

Algorithm 3 constructs PDAWGs based on the above lemmas. An example of online construction of a PDAWG can be found in Figure 4. For technical convenience, like the standard DAWG construction algorithm, we add a dummy node \top to the PDAWG that has edges to the source node, denoted as ρ in Algorithm 3, labeled with all elements of



■ **Figure 4** A snapshot of left-to-right online construction of $\text{PDAWG}(T)$ with $T = xaxaya$ by Algorithm 3. Each figure shows $\text{PDAWG}(wa)$ for a prefix wa of $\text{prev}(T) = 0a2a0a$. Double arrows show primary edges. The new nodes, edges and suffix links are colored red. The purple, red and blue diamonds represent $[x']_{wa}^R$, $[x]_{wa}^R$ and $[y]_{wa}^R$, respectively, where $x' = \text{preLRS}(wa)$, $x = \text{LRS}(wa)$ and $y = \max[x]_{wa}^R$. When $x = \varepsilon$, the purple diamond is put on the dummy node \top .

$\Sigma \cup \{0\}$. This trick allows us to uniformly treat the special case where the LRS node is ρ , in which case the pre-LRS node is defined to be \top . In addition, we let $\text{SL}(\rho) = \top$. Each node u does not remember the elements of u but we remember $\text{len}(u) = |\max u|$. Note that $|\min u| = |\text{len}(\text{SL}(u))| + 1$. Hereafter we use functions SL , child , Trans , etc. without a subscript specifying a text, to refer to the data structure that the algorithm is manipulating, rather than the mathematical notion relative to the text. Of course, we design our algorithm so that those functions coincide with the corresponding mathematical notions.

Suppose we have constructed $\text{PDAWG}(w)$ and want to obtain $\text{PDAWG}(wa)$. The sink node of $\text{PDAWG}(w)$, denoted as *oldsink*, corresponds to $[w]_w^R$. We first make a new sink node *newsink* = $[wa]_{wa}^R$. Then we visit $u_i = \text{SL}^i(\text{oldsink})$ for $i = 1, 2, \dots, j$, until we find the pre-LRS node $u_j = [\text{preLRS}(wa)]_w^R$. By Lemma 8, we can identify u_j and $k' = |\text{preLRS}(wa)|$. For each node u_i with $i < j$, we make an edge labeled with $Z(a, \text{len}(u_i))$ pointing at *newsink* and, moreover, u_j also has an edge pointing at *newsink* if $k' < \text{len}(u_j)$ by Lemma 9. We then reach the LRS node $v = [\text{LRS}(wa)]_w^R = \text{Trans}(u_j, k', Z(a, k'))$. We compare $k = k' + 1 = |\text{LRS}(wa)|$ and $\text{len}(v)$ to decide whether the LRS node shall be split based on Lemma 6. If $|\text{LRS}(wa)| = \text{len}(v)$, the node v will not be split, in which case we obtain $\text{PDAWG}(wa)$ by making $\text{SL}(\text{newsink}) = v$ (Lemma 12).

Suppose $k < \text{len}(v)$. In this case, the LRS node v must be split. We reuse the old node v , which used to correspond to $[\text{LRS}(wa)]_w^R$, as a new node corresponding to $[\max[\text{LRS}(wa)]_w^R]_w^R$, and create another new node v' for $[\text{LRS}(wa)]_{wa}^R$ with $\text{len}(v') = k$. Edges are determined in accordance with Lemmas 10 and 11. All in-coming edges from $\text{SL}^i([\text{preLRS}(wa)]_w^R)$ to v in $\text{PDAWG}(w)$ are redirected to v' , except when $\text{preLRS}(wa) \neq \max[\text{preLRS}(wa)]_w^R$ for $i = 0$. The out-going edges from v will be kept. We create out-going edges of v' referring to the corresponding transitions from v . If $(v, b, u) \in E$ with $Z(b, k) \neq 0$, then we add (v', b, u) to E . In addition, we add $(v', 0, \text{Trans}_w(v, 0, k))$ to E if $\text{Trans}(v, 0, k) \neq \text{Null}$. At last, suffix links among *newsink*, v, v' are determined in accordance with Lemma 12.

We conclude the subsection with the complexity of Algorithm 3. Let us call an edge (u, a, v) *primary* if $\max v = \max u \cdot a$, and *secondary* otherwise. The following lemma is an adaptation of the corresponding one for DAWGs by Blumer et al. [4].

► **Lemma 13.** *Let $\text{SC}_w(u) = \{\text{SL}_w^i(u) \mid i \geq 0\}$ for a node u . If $\text{PDAWG}(w)$ has a primary edge from u to v , then the total number of secondary edges from nodes in $\text{SC}_w(u)$ to nodes in $\text{SC}_w(v)$ is bounded by $|\text{SC}_w(u)| - |\text{SC}_w(v)| + |\Pi| + 1$.*

Proof. Let us count the number of edges from nodes in $\text{SC}_w(u)$ to $\text{SC}_w(v)$. Baker [2, Lemma 1] showed that in a parameterized suffix tree, each path from the root to a leaf has at most $|\Pi|$ nodes with *bad suffix links*. Through the duality of PDAWGs and parameterized suffix trees stated in Lemma 16, this means that $\text{SC}_w(v)$ contains at most $|\Pi| + 1$ nodes which has no in-coming primary edges, where the additional one node is the root of the PDAWG. Since each node has at most one in-coming primary edge, the number of primary edges in concern is at least $|\text{SC}_w(v)| - |\Pi| - 1$ in total. Since each node in $\text{SC}_w(u)$ has just one out-going edge to $\text{SC}_w(v)$, we obtain the lemma. ◀

► **Theorem 14.** *Given a string T of length n , Algorithm 3 constructs $\text{PDAWG}(T)$ in $O(n|\Pi| \log(|\Sigma| + |\Pi|))$ time and $O(n)$ space online, by reading T from left to right.*

Proof. Since the size of a PDAWG is bounded by $O(n)$ (Theorem 2) and nodes are monotonically added, it is enough to bound the number of edges and suffix links that are deleted. In each iteration of the **for** loop, at most one suffix link is deleted. So at most n suffix links are deleted in total. We count the number of edges whose target is altered from $v = [\text{LSR}(wa)]_w^R$ to $v' = [\text{LSR}(wa)]_{wa}^R$ on Line 15 when updating $\text{PDAWG}(w)$ to $\text{PDAWG}(wa)$. Let k_i be the number of such edges at the i -th iteration of the **for** loop. Note that those are all secondary edges from a node in $\text{SC}_w(u_0)$ for the pre-LRS node u_0 . By Lemma 13,

$$\begin{aligned} \sum_{i=1}^n k_i &\leq \sum_{i=1}^n (|\text{SC}_{wa}(w)| - |\text{SC}_{wa}(wa)| + |\Pi| + 1) \\ &\leq \sum_{i=1}^n (|\text{SC}_w(w)| - |\text{SC}_{wa}(wa)| + |\Pi| + 1) \\ &= |\text{SC}_\varepsilon(\varepsilon)| - |\text{SC}_t(t)| + (|\Pi| + 1)n \in O(|\Pi|n). \end{aligned}$$

Since the suffix links of $\text{PDAWG}(T)$ forms the p -suffix tree of \bar{T} (see Subsection 3.5), the following corollary is immediate from Theorem 14.

► **Corollary 15.** *The p -suffix tree of a string S of length n can be constructed in $O(n|\Pi| \log(|\Sigma| + |\Pi|))$ time and $O(n)$ space online, by reading S from right to left.*

Differently from the online DAWG construction algorithm [4], we have the factor $|\Pi|$ in our algorithm complexity analysis. Actually our algorithm takes time proportional to the difference of the old and new PDAWGs modulo logarithmic factors, as long as the difference is defined so that the split node $[\text{LRS}(wa)]_w^R$ automatically becomes $[\max[\text{LRS}(wa)]_w^R]_{wa}^R$ rather than $[\text{LRS}(wa)]_{wa}^R$. In this sense, our algorithm is optimal. It is open whether we could improve the analysis.

3.5 Duality of PDAWGs and p -suffix trees

This subsection establishes the duality between parameterized suffix trees and PDAWGs. An example can be found in Figure 5. For this sake, we introduce the reverse of a pv -string and *Weiner links (reversed suffix links)* for parameterized suffix trees.

The “reverse” \tilde{x} of a pv -string x must satisfy that $\tilde{x} = \text{prev}(\bar{S})$ iff $x = \text{prev}(S)$ for any p -string $S \in (\Sigma \cup \Pi)^*$. For the empty string $\tilde{\varepsilon} = \varepsilon$. For $x \in (\Sigma \cup \mathcal{N})^*$ and $a \in \Sigma \cup \mathcal{N}$,

$$\tilde{xa} = \begin{cases} a\tilde{x} & \text{if } a \in \Sigma \cup \{0\}, \\ 0y & \text{otherwise,} \end{cases}$$

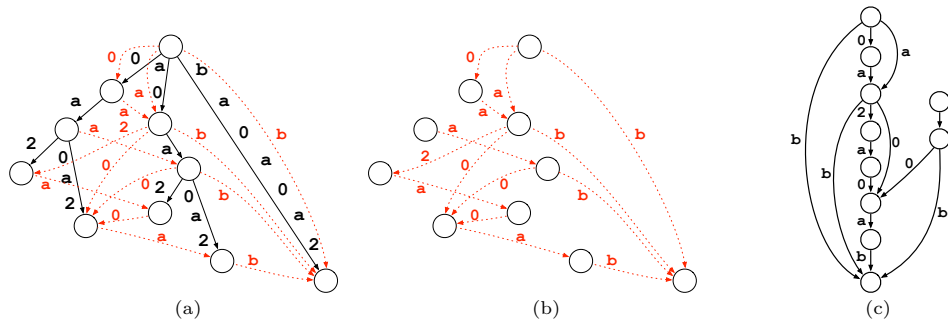


Figure 5 (a) The parameterized suffix tree $\text{PSTree}(S)$ for $S = \text{baxayay}$ over $\Sigma = \{a, b\}$ and $\Pi = \{x, y\}$, augmented with the Weiner links (dashed red arcs). (b) The DAG consisting of the p-suffix tree nodes and the Weiner-links. (c) The PDAWG $\text{PDAWG}(T)$ for $T = \bar{S} = \text{yayaxab}$. The graphs (b) and (c) are isomorphic.

where y is obtained from \tilde{x} by replacing the a -th element by a , i.e., $y = \tilde{x}[: a - 1] \cdot a \cdot \tilde{x}[a + 1 :]$. This is well-defined if $x a$ is a pv-string. For example, for $T = \text{xaxy}$ with $a \in \Sigma$ and $x, y \in \Pi$, we have $\text{prev}(\bar{T}) = \overline{0a20} = 00a2 = \text{prev}(y x a x) = \text{prev}(\bar{T})$.

The parameterized suffix tree $\text{PSTree}(T)$ of a p-string T is the path-compacted (or Patricia) tree for $\text{PSuffix}(T)$. For any $z \in (\Sigma \cup \mathcal{N})^*$, For $\text{PSTree}(T)$, the Weiner links are defined as follows. Let v be a node in $\text{PSTree}(T)$ such that $v = \text{prev}(S)$ for some substring S of T , and $a \in \Sigma \cup \mathcal{N}$. Let $\alpha(a, v)$ be the pv-string such that

$$\alpha(a, v) = \begin{cases} av & \text{if } av \in \text{PFactor}(T) \text{ and } a \in \Sigma \cup \{0\}, \\ \text{prev}(S[a] \cdot S) & \text{if } \text{prev}(S[a] \cdot x) \in \text{PFactor}(T) \text{ and } a \in \mathcal{N} \setminus \{0\}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Then a Weiner link is a triple (v, a, u) such that $u = \alpha(a, v)y$, where $y \in (\Sigma \cup \mathcal{N})^*$ is the shortest string such that $\alpha(a, v)y$ is a node of $\text{PSTree}(T)$. The Weiner link (v, a, u) is said to be *explicit* if $u = \alpha(a, v)$, and *implicit* otherwise¹.

To establish the correspondence between $\text{PDAWG}(T)$ and $\text{PSTree}(\bar{T})$ easily, here we rename the nodes $[x]_w^R$ of $\text{PDAWG}(T)$ to be $\max[x]_w^R$ where $w = \text{prev}(T)$.

► **Theorem 16.** *The following correspondence between $\text{PDAWG}(T) = (V_D, E_D)$ and $\text{PSTree}(\bar{T}) = (V_T, E_T)$ holds.*

- (1) $\text{PDAWG}(T)$ has a node $x \in V_D$ iff $\text{PSTree}(\bar{T})$ has a node $\tilde{x} \in V_T$.
- (2) $\text{PDAWG}(T)$ has a primary edge $(x, a, y) \in E_D$ iff $\text{PSTree}(\bar{T})$ has an explicit Weiner link $(\tilde{x}, a, \tilde{y})$.
- (3) $\text{PDAWG}(T)$ has a secondary edge $(x, a, y) \in E_D$ iff $\text{PSTree}(\bar{T})$ has an implicit Weiner link $(\tilde{x}, a, \tilde{y})$.
- (4) $\text{PDAWG}(T)$ has a suffix link from $\tilde{x}y$ to \tilde{x} iff $\text{PSTree}(\bar{T})$ has an edge $(x, y, xy) \in E_T$.

¹ Explicit Weiner links are essentially the same as the reversed suffix links used for right-to-left online construction of parameterized position heaps [8].

References

- 1 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC 1993*, pages 71–80, 1993.
- 2 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.
- 3 Richard Beal and Donald A. Adjeroh. p-suffix sorting as arithmetic coding. *J. Discrete Algorithms*, 16:151–169, 2012.
- 4 Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, Mu-Tian Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical computer science*, 40:31–55, 1985.
- 5 Maxime Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
- 6 Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Parameterized suffix arrays for binary strings. In *PSC 2008*, pages 84–94, 2008.
- 7 Diptarama, Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara. Position heaps for parameterized strings. In *CPM 2017*, pages 8:1–8:13, 2017. doi:10.4230/LIPIcs.CPM.2017.8.
- 8 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Right-to-left online construction of parameterized position heaps. In *PSC 2018*, pages 91–102, 2018.
- 9 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Direct Linear Time Construction of Parameterized Suffix and LCP Arrays for Constant Alphabets. In *SPIRE 2019*, pages 382–391. Springer International Publishing, 2019.
- 10 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The Parameterized Position Heap of a Trie. In *CIAC 2019*, pages 237–248. Springer International Publishing, 2019.
- 11 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *SODA 2017*, pages 397–407, 2017.
- 12 Tomohiro I, Satoshi Deguchi, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Light-weight parameterized suffix array construction. In *IWOCA 2009*, pages 312–323, 2009.
- 13 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *FOCS 1995*, pages 631–637, 1995.
- 14 Taehyung Lee, Joong Chae Na, and Kunsoo Park. On-line construction of parameterized suffix trees for large alphabets. *Inf. Process. Lett.*, 111(5):201–207, 2011.
- 15 Juan Mendivelso and Yoan Pinzón. Parameterized matching: Solutions and extensions. In *Proc. PSC 2015*, pages 118–131, 2015.
- 16 Juan Mendivelso, Sharma V. Thankachan, and Yoan Pinzón. A brief history of parameterized matching problems. *Discrete Applied Mathematics*, 2018. Available online. doi:10.1016/j.dam.2018.07.017.
- 17 Katsuhito Nakashima, Noriki Fujisato, Diptarama Hendrian, Yuto Nakashima, Ryo Yoshinaka, Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, and Masayuki Takeda. DAWGs for parameterized matching: online construction and related indexing structures. *CoRR*, abs/2002.06786, 2020. URL: <https://arxiv.org/abs/2002.06786>.
- 18 Tetsuo Shibuya. Generalization of a suffix tree for RNA structural pattern matching. *Algorithmica*, 39(1):1–19, 2004.

On Extensions of Maximal Repeats in Compressed Strings

Julian Pape-Lange 

Technische Universität Chemnitz, Straße der Nationen 62, 09111 Chemnitz, Germany
julian.pape-lange@informatik.tu-chemnitz.de

Abstract

This paper provides upper bounds for several subsets of maximal repeats and maximal pairs in compressed strings and also presents a formerly unknown relationship between maximal pairs and the run-length Burrows-Wheeler transform.

This relationship is used to obtain a different proof for the Burrows-Wheeler conjecture which has recently been proven by Kempa and Kociumaka in “Resolution of the Burrows-Wheeler Transform Conjecture”.

More formally, this paper proves that the run-length Burrows-Wheeler transform of a string S with z_S LZ77-factors has at most $73(\log_2 |S|)(z_S + 2)^2$ runs, and if S does not contain q -th powers, the number of arcs in the compacted directed acyclic word graph of S is bounded from above by $18q(1 + \log_q |S|)(z_S + 2)^2$.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatoric problems

Keywords and phrases Maximal repeats, Extensions of maximal repeats, Combinatorics on compressed strings, LZ77, Burrows-Wheeler transform, Burrows-Wheeler transform conjecture, Compact suffix automata, CDAWGs

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.27

Related Version The preprint of this paper can be found at <https://arxiv.org/abs/2002.06265>.

Acknowledgements Fabio Cunial suggested that my previous work might be extendable from counting maximal repeats to counting extensions of maximal repeats. He also pointed out that such a result would be more interesting since it is more closely linked to the size of the compacted directed acyclic word graph. Nicola Prezza noted that my previous work also resulted in a non-trivial upper bound for the number of runs in the run-length Burrows-Wheeler transform and that a more careful investigation of the extensions of maximal repeats might result in a better bound for the Burrows-Wheeler conjecture which was unsolved at that time. I also thank Djamel Belazzougui for notifying me of the “Resolution of the Burrows-Wheeler Conjecture” by Kempa and Kociumaka.

1 Introduction

A maximal repeat P of a string is a substring such that there are two occurrences of P in the string which are preceded by different characters and succeeded by different characters. Such a pair of occurrences is called a maximal pair.

Raffinot proves in [10] that there is a natural bijection from the internal nodes in a Compacted Directed Acyclic Word Graph (CDAWG) to the maximal repeats, which is given by the labels of the paths. Also, Furuya et al. present in [7] a relation between maximal repeats and the grammar compression algorithm RePair, and they use this relation to design MR-RePair, an improved variant of RePair.

Sometimes, maximal repeats are not sufficient, since they do not contain any information about the surrounding string. Therefore, in [1], Belazzougui et al. introduce the number of right extensions of maximal repeats as a measure for the repetitiveness of strings. They further



© Julian Pape-Lange;

licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 27; pp. 27:1–27:13

Leibniz International Proceedings in Informatics



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

prove that the number of arcs in the CDAWG is equal to the number of right extensions of maximal repeats and that the number of runs in the run-length Burrows-Wheeler transform (RLBWT) is bounded from above by the number of right extensions of maximal repeats.

In earlier work, I proved in [9] that the number of maximal repeats in a string S with z_S (self-referential) LZ77-factors and without q -th powers is bounded from above by $3q(z_S+1)^3-2$ and that this upper bound is tight up to a constant factor. This result implies that for a string S over an alphabet Σ , the number of arcs in the CDAWG, and thereby the number r_S of runs in the RLBWT, is bounded from above by $3|\Sigma|q(z_S+1)^3$.

We should expect that of all the $\mathcal{O}(q(z_S)^3)$ maximal repeats some provide less information than others. For example in the string

$$ba^{10}ba^{20}b\$ = baaaaaaaaabaaaaaaaaaaaaaaaaaaaaab$,$$

we can derive all maximal pairs of the maximal repeats of a^i from the maximal pairs of a^9 and a^{19} . In this way, highly-periodic maximal repeats with exponent close to the exponent of the corresponding runs are more important than other maximal repeats which are powers of the same base.

Blumer et al. have already shown in 1987 in [2] that the CDAWG cannot compress high powers and that the CDAWG of $a^n\$$ has size $\Theta(n)$. Contrary to the CDAWG, the RLBWT does not suffer from high powers and we should expect that there are many right extensions of maximal repeats which do not increase the number of runs. And in fact, if the string is very structured, we expect that the output consists of few runs of single characters. For example Christodoulakis et al. show in [4] that the Burrows-Wheeler transform of the n -th Fibonacci string F_n is given by $b^{f_{n-2}}a^{f_{n-1}}$.

Yet, until recently, it remained an open question whether there is an upper bound for the number of runs in the RLBWT which is polynomial in the number of LZ77-factors and the logarithm of the length of the string only. This Burrows-Wheeler transform conjecture was resolved in October 2019 by Kempa and Kociumaka who prove in the first version of their arXiv-article [8] that $r_S \in \mathcal{O}(z_S(\log n)^2)$ holds and promised that they will show $r_S \in \mathcal{O}\left(\delta_S \log \delta_S \max\left(1, \log \frac{n}{\delta_S \log \delta_S}\right)\right)$ for a complexity measure $\delta_S \leq z_S$ in an extended version. In April 2020 they uploaded the extended second version to their arXiv-article [8]. In this extended version they do not only prove this tighter upper bound $r_S \in \mathcal{O}\left(\delta_S \log \delta_S \max\left(1, \log \frac{n}{\delta_S \log \delta_S}\right)\right)$, but they also prove that this upper bound is asymptotically tight for all values of n and δ_S .

This paper provides a different approach to the Burrows-Wheeler transform conjecture and shows by using maximal repeats and their extensions that $r_S \leq 73(\log_2 |S|)(z_S + 2)^2$ holds.

On the way, this paper also shows that the number of arcs in the CDAWG is bounded from above by $18q(1 + \log_q |S|)(z_S + 2)^2$ and gives new insights into the combinatorial properties of extensions of maximal repeats which are either non-highly-periodic or cannot be extended by more than a period length.

2 Definitions

Let Σ be an *alphabet*. A *string* with *length* denoted by $|S|$ is the concatenation of *characters* $S[1]S[2] \cdots S[|S|]$ of Σ . Since it will be useful to have a predecessor and a successor for every character of the string, we also define $S[0] = \$$ and $S[|S| + 1] = \$$ with $\$ \notin \Sigma$. The *substring*

$S[i..j]$ with $0 \leq i \leq j \leq |S| + 1$ is the concatenation $S[i]S[i+1] \cdots S[j]$. For $i > j$ the substring $S[i..j]$ is defined to be the empty string with length 0. A *prefix* is a substring of the form $S[1..j]$ and a *suffix* is a substring of the form $S[i..|S|]$.

In this paper, we are not only interested in the substrings themselves but we are also interested in their relationship to the underlying string. We therefore use *positioned substrings*. Formally, a positioned substring is a pair (l, r) of indices and the content of the positioned substring is the substring $S[l..r]$. In order to use positioned substrings as substrings, we slightly abuse the notation in this paper and denote the positioned substrings like normal substrings with $S[l..r]$. Therefore, the term “positioned” only indicates that we are not allowed to forget the underlying indices.

An *occurrence* of a substring P is a positioned substring $S[l..r]$ such that $S[l..r] = P$ holds for the underlying substrings.

For example in the string $S = ababab$, the positioned substrings $S[1..3] = aba$ and $S[2..4] = bab$ overlap on the positioned substring $S[2..3] = ba$, but the positioned substrings $S[1..3] = aba$ and $S[4..6] = bab$ don't have a non-empty overlap. Also, in this string S , the substring $P = S[2..4] = bab$ has exactly two occurrences given by the positioned substrings $S[2..4] = bab = P$ and $S[4..6] = bab = P$.

The string S is *lexicographically strictly smaller/larger* than the string S' if S is lexicographically smaller/larger than S' and there is a mismatch $S[m] \neq S'[m]$.

A *maximal pair* of S is a triple $(n, m, l) \in \mathbb{N}^3$ with $l \geq 1$ such that $S[n..n+l-1]$ is equal to $S[m..m+l-1]$ and this property can not be extended to any side. More formally:

- $\forall i \in \mathbb{N}$ with $0 \leq i < l : S[n+i] = S[m+i]$ but
- $S[n-1] \neq S[m-1]$ and
- $S[n+l] \neq S[m+l]$.

With this notation, the string $S[n..n+l-1] = S[m..m+l-1]$ is the *corresponding maximal repeat*.

Since for a maximal pair (n, m, l) the inequality $S[n-1] \neq S[m-1]$ holds, the indices n and m cannot be equal. Also, by construction, $S[n..n+l-1]$ and $S[m..m+l-1]$ are contained in S and $S[n..n+l]$ and $S[m..m+l]$ are contained in $S\$$.

For a positioned maximal repeat $S[n..n+l-1]$, the *right-extension* of this maximal repeat is the substring $S[n..n+l]$ which is obtained by extending the maximal repeat by its successor. Similarly, the *double-sided extension* is $S[n-1..n+l]$.

Since maximal pairs are easier to handle than maximal repeats and their extensions, this paper introduces the notion of “substantially different maximal pairs” which allows to give an upper bound for the number of double-sided extensions:

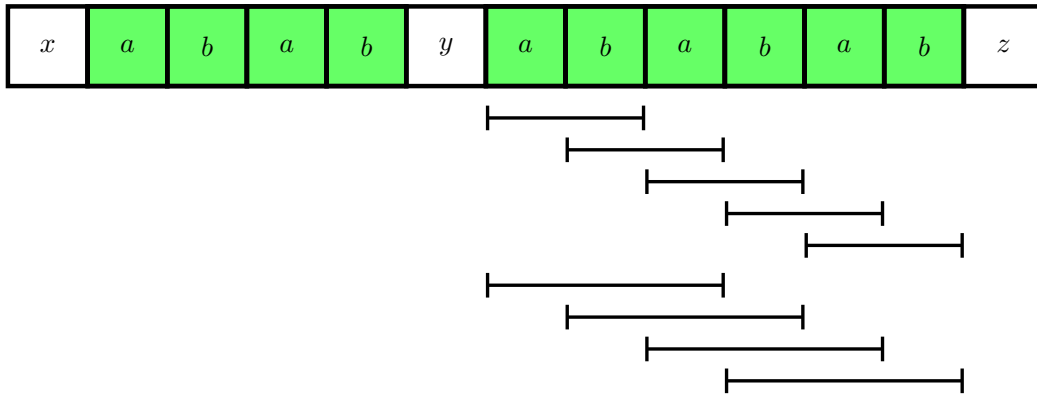
Two maximal pairs (n, m, l) and (n', m', l') are *copies of each other* if the two strings $S[n-1..n+l]$ and $S[m-1..m+l]$ are equal to the two strings $S[n'-1..n'+l']$ and $S[m'-1..m'+l']$. In particular, the two maximal pairs (n, m, l) and (m, n, l) are always copies of each other. However, it is not sufficient for two maximal pairs to have identical corresponding maximal repeats in order to be copies of each other.

If two maximal pairs are not copies of each other, they are *substantially different*.

For each of the substantially different maximal pairs there can be at most two double-sided extensions of the corresponding maximal repeat. Therefore, the number of double-sided repeats is at most twice the number of substantially different maximal pairs.

A string S which is not of the form P^q for an integer $q \in \mathbb{N}_{\geq 2}$ is *primitive*, and a square S^2 with a primitive root S is a *primitively rooted square*.

A *period* of a string S is an integer p such that all characters in S with distance p are equal. A string with period length p is called *p-periodic*.



■ **Figure 1** The string $S = xababyabababz$ with two maximal periodic extensions of the substring ab and nine extendable maximal substrings, all of them with root ab . The maximal periodic extensions are the two green substrings and each extendable substring is represented by a line indicating the underlying positions.

A string S is $\frac{1}{\geq q}$ -highly-periodic, if it has a period with length $\frac{1}{q}|S|$ or smaller. A maximal pair is $\frac{1}{\geq q}$ -highly-periodic if the corresponding maximal repeat is $\frac{1}{\geq q}$ -highly-periodic.

For example, the strings $aaaa = a^4$, $aaaaa = a^5$ and $ababababa = (ab)^4a = (ab)^{4.5}$ are $\frac{1}{\geq 4}$ -highly-periodic, but $aaaac = a^4c$ and $abababa = (ab)^{3.5}$ are not $\frac{1}{\geq 4}$ -highly-periodic.

Let $S[l..r]$ be a positioned p -periodic substring with $|S[l..r]| \geq p$. The *maximal p -periodic extension* of this occurrence is the positioned substring $S[l', r']$ such that

- $l' \leq l \leq r \leq r'$,
- $S[l'..r']$ is p -periodic,
- $S[l' - 1..r']$ is not p -periodic and
- $S[l'..r' + 1]$ is not p -periodic.

With this notation, the pair $S[l' - 1, r' + 1]$ is the *padded maximal p -periodic extension*.

If p is the minimal period length of $S[l..r]$, we will omit the p and simply write *maximal periodic extension* and *padded maximal periodic extension*.

Similar to maximal pairs, two padded maximal periodic extensions $S[l - 1, r + 1]$ and $S[l' - 1, r' + 1]$ are *copies of each other* if the corresponding strings are equal. If the two padded maximal periodic extensions are not copies of each other, they are *substantially different*.

A positioned substring $S[l..r]$ with minimal period length p is *extendable* if the maximal p -periodic extension is at least $p + 1$ characters longer than $S[l..r]$. A maximal pair is *extendable*, if both occurrences of the corresponding maximal repeat are extendable.

For example, in Figure 1, we have the string $S = xababyabababz$. The positioned substrings $S[2..3] = ab$, $S[3..4] = ba$ and $S[8..11] = baba$, each with minimal period length 2, are not extendable, since their maximal periodic extensions $S[2..5] = abab$ (for both $S[2..3]$ and $S[3..4]$) and $S[7..12] = ababab$ (for $S[8..11]$) are only p characters longer. The positioned substring $S[8..8] = b$ has minimal period length 1 and therefore its maximal periodic extension is $S[8..8]$. On the other hand, the positioned substring $S[9..10] = ab$ with minimal period length 2 has the maximal periodic extension $S[7..12] = ababab$ which is 4 characters longer. Hence, the positioned substring $S[9..10]$ is extendable.

Checking all substrings, one can see that the extendable substrings of S are exactly the 9 2-periodic positioned substrings of the positioned substring $S[7..12]$ with length less than 4.

Also, the maximal pair (7, 11, 2) is extendable even though both maximal periodic extensions are the same positioned substring. Also, this is the only extendable maximal pair of this string.

The (self-referential) *LZ77-decomposition* of a string S is a factorization $S = F_1F_2 \dots F_{z_S}$ in LZ77-factors, such that for all $i \in \{1, 2, \dots, z_S\}$ either

- F_i is a character which does not occur in $F_1F_2 \dots F_{i-1}$ or
- F_i is the longest possible prefix of $S[[F_1F_2 \dots F_{i-1}] + 1..|S|]$ which occurs at least twice in $F_1F_2 \dots F_i$.

Let $\pi_i \in \{0, 1, 2, \dots, |S|\}$ be given by the lexicographic order of the cyclic permutations $S[\pi_i + 1..|S| + 1]S[1..\pi_i]$ of $S\$$. The *Burrows-Wheeler transform* defined in [3] is given by the last characters of those strings, and, since $S[0] = \$ = S[|S| + 1]$ hold by definition, these characters are given by $S[\pi_i]$.

3 Non-Highly-Periodic Maximal Pairs

The main goal of this section is to prove that in a string S the number of substantially different non- $\frac{1}{\geq 6}$ -highly-periodic maximal pairs is bounded from above by $41(\log_2 |S|)(z_S + 1)(z_S + 2)$.

Along the way, this section will also prove that if S does not contain q -th powers, its CDAWG has at most $18q(1 + \log_q |S|)(z_S + 2)^2$ arcs.

In Theorem 8 of [9], I counted the number of maximal pairs around the boundaries of LZ77-factors which neither begin nor end with a power of a given exponent:

► **Theorem 1** (Theorem 8 of [9]). *Let S be a string. Let $F_1F_2 \dots F_z F_{z+1} = S\$$ be the LZ77-decomposition of $S\$$. Let $s_1, s_2, \dots, s_z, s_{z+1}$ be the starting indices of the LZ77-factors in $S\$$. Let $q \in \mathbb{N}_{\geq 2}$ and $i, j \in \{1, 2, \dots, z, z + 1\}$ be natural numbers.*

Then the number of different maximal pairs (n_k, m_k, l_k) such that for all k

- *the substring $S[n_k..s_i - 1]$ is not a fractional power with exponent greater than or equal to q ,*
- *the substring $S[s_i..n_k + l_k - 1]$ is not a fractional power with exponent greater than or equal to q ,*
- *the starting index s_i is contained in the interval $[n_k, n_k + l_k]$,*
- *the starting index s_{i+1} is not contained in the interval $[n_k, n_k + l_k]$ and*
- *the starting index s_j is contained in the interval $[m_k, m_k + l_k]$*

is bounded from above by $18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil$.

This can be slightly simplified by ignoring the underlying LZ77-structure which is not used in the proof:

► **Corollary 2.** *Let S be a string. Let $q \in \mathbb{N}_{\geq 2}$ be a natural number and i, j be indices of two characters in $S\$$.*

Then there are at most $18q \cdot \lceil \log_q(|S\$|) \rceil$ different maximal pairs (n_k, m_k, l_k) such that for all k

- *neither the substring $S[n_k..i - 1]$ nor the substring $S[i..n_k + l_k - 1]$ is $\frac{1}{\geq q}$ -highly-periodic and*
- *the indices i and j are contained in the intervals $[n_k, n_k + l_k]$ and $[m_k, m_k + l_k]$, respectively.*

Following the proof of Theorem 8 in [9], the substring $S[i..n_k + l_k - 1]$ naturally splits into $S[n_k..i - 1]$ and $S[i..n_k + l_k - 1]$ and we can even require that the longer part(s) is/are not a high power(s). In order to have a unique longer part, we define the string $S[n_k..i - 1]$ to be longer than $S[i..n_k + l_k - 1]$, if both of these substrings have the same length.

27:6 On Extensions of Maximal Repeats in Compressed Strings

► **Lemma 3.** *Let S be a string. Let $q \in \mathbb{N}_{\geq 2}$ be a natural number and i, j be indices of two characters in $S\$$.*

Then there are at most $18q(1 + \log_q |S|)$ different maximal pairs (n_k, m_k, l_k) such that for all k

- *the longer string of the substrings $S[n_k..i-1]$ and $S[i..n_k+l_k-1]$ is not $\frac{1}{\geq q}$ -highly-periodic and*
- *the indices i and j are contained in the intervals $[n_k, n_k+l_k]$ and $[m_k, m_k+l_k]$, respectively.*

As proven in Lemma 4 of [9], each maximal pair has a copy such that both double-sided extensions of the corresponding maximal repeats cross LZ77-boundaries. Also, each maximal pair introduces at most two new right extensions of maximal repeats. Therefore, we can deduce a bound similar to Theorem 1 of [9] for the right extensions of maximal repeats and the arcs of the CDAWG:

► **Theorem 4.** *Let S be a string. Let q be further a natural number such that S does not contain q -th powers.*

Then the number of right extensions of maximal repeats in S is bounded from above by $18q(1 + \log_q |S|)(z_S + 2)^2 - (z_S + 1)$. Also, the CDAWG of S has at most $18q(1 + \log_q |S|)(z_S + 2)^2$ arcs.

Proof. Summing up over the first indices $i \leq j$ of the $z_S + 1$ LZ77-factors of $S\$$ yields that there are at most

$$\sum_{i=1}^{z_S+1} \sum_{j=i}^{z_S+1} 18q(1 + \log_q |S|) = 9q(1 + \log_q |S|)(z_S + 1)(z_S + 2) \leq 9q(1 + \log_q |S|)(z_S + 2)^2 - (z_S + 1)$$

substantially different maximal pairs. And since each new substantially different maximal pair introduces at most two new right extensions of maximal repeats, there are at most $18q(1 + \log_q |S|)(z_S + 2)^2 - (z_S + 1)$ different right extensions of maximal repeats.

Since the number of right extensions of (non-empty) maximal repeats is equal to the number of arcs in the CDAWG which start at internal nodes and since there are exactly $|\Sigma \cup \{\$\}| \leq z_S + 1$ arcs starting at the root, there are at most $18q(1 + \log_q |S|)(z_S + 2)^2$ arcs in the CDAWG. ◀

Additionally, there might be maximal pairs, in which the longer part(s) is/are high power(s) but the corresponding periodicity does not extend to the whole maximal repeat. In order to find a good upper bound for those maximal pairs, we need an additional lemma to limit the number of possible period lengths of prefixes and suffixes with high powers.

► **Lemma 5.** *Let S be a string. Let further P_1, P_2 be two substrings of S such that*

- *P_1 and P_2 are both either prefixes or suffixes of S ,*
- *the length of P_2 fulfills the inequality $|P_1| \leq |P_2| \leq 2|P_1|$ and*
- *both P_1 and P_2 are $\frac{1}{\geq 3}$ -highly-periodic.*

Then P_1 and P_2 have the same minimal period length.

Proof. Without loss of generality assume that P_1 and P_2 are both prefixes of S . Let p_1 and p_2 be the minimal period lengths of P_1 and P_2 , respectively.

Since the inequalities $p_1 \leq \frac{1}{3}|P_1|$ and $p_2 \leq \frac{1}{3}|P_2| \leq \frac{2}{3}|P_1|$ hold, the periodicity lemma from [6] of Fine and Wilf proves, that P_1 is $\gcd(p_1, p_2)$ -periodic. Since p_1 is the minimal period length of P_1 , this implies that p_2 is a multiple of p_1 .

However, since $P_1 \subset P_2$ and $p_2 \leq \frac{2}{3}|P_1|$ hold, a p_2 -periodic base of P_2 is also p_1 -periodic.

Therefore $p_1 = p_2$ holds. ◀

- **Theorem 6.** *Let S be a string. Let i, j be indices of two characters in $S\$$.*
- Then there are at most $12 \log_2 |S|$ different maximal pairs (n_k, m_k, l_k) such that for all k
 - the longer string of the substrings $S[n_k..i-1]$ and $S[i..n_k+l_k-1]$ is $\frac{1}{\geq 3}$ -highly-periodic with period length p , but
 - the substring $S[n_k..n_k+l_k-1]$ is not p -periodic and
 - the indices i and j are contained in the intervals $[n_k, n_k+l_k]$ and $[m_k, m_k+l_k]$, respectively.

Proof. By contradiction:

It is sufficient to prove that there are at most $6 \log_2 |S|$ different maximal pairs with the restrictions given by the prerequisites which fulfill $|S[n_k..i-1]| \geq |S[i..n_k+l_k-1]|$. By symmetry, the maximal pairs which fulfill the inequality $|S[n_k..i-1]| < |S[i..n_k+l_k-1]|$ can be bounded with an identical argument.

Assume there are at least $\lceil 6 \log_2(|S|) \rceil + 1$ different maximal pairs (n_k, m_k, l_k) with $|S[n_k..i-1]| \geq |S[i..n_k+l_k-1]|$ and the restrictions given by the prerequisites.

Since for all maximal pairs $1 \leq n_k$ holds, the inequality $i - n_k \leq |S\$| - 1$ holds as well. On the other hand, since $S[n_k..i-1]$ is $\frac{1}{\geq 3}$ -highly-periodic, this substring has to contain at least three characters. Therefore, the inequality $3 \leq i - n_k$ holds.

Taking the logarithm yields

$$1 < \log_2(3) \leq \log_2(i - n_k) \leq \log_2(|S\$| - 1) \leq \lceil \log_2(|S|) \rceil.$$

For each maximal pair, the number $\log_2(i - n_k)$ lies in at least one of the $\lceil \log_2(|S|) \rceil - 1$ intervals $[h, h + 1]$ with $1 \leq h < \lceil \log_2(|S|) \rceil$.

Using $\lceil \log_2(|S|) \rceil - 1 \leq \lfloor \log_2(|S|) \rfloor$, the pigeonhole principle now yields that there has to be a natural number L' such that

$$\left\lceil \frac{\lceil 6 \log_2(|S|) \rceil + 1}{\lfloor \log_2(|S|) \rfloor} \right\rceil = 7$$

of these maximal pairs have a starting index with $L' \leq \log_2(i - n_k) \leq 1 + L'$.

Therefore, for $L = 2^{L'}$ this gives a natural number L such that $L \leq i - n_k \leq 2L$ holds for each of these 7 maximal pairs.

Since the index i is contained in the interval $[n_k, n_k+l_k]$ and $|S[n_k..i-1]| \geq |S[i..n_k+l_k-1]|$ holds, the index i is also contained in the interval $[n_k + \frac{l_k}{2}, n_k + l_k]$. Hence, the inequalities $n_k + \frac{l_k}{2} \leq i$ and thereby $\frac{l_k}{2} \leq i - n_k \leq 2L$ hold. Therefore, the length l_k is at most $4L$.

Since the index j is contained in the interval $[m_k, m_k+l_k]$, this implies that the inequality $m_k \geq j - l_k \geq j - 4L$ holds. On the other hand $m_k \leq j$ so the m_k are in an interval of length $4L$.

Using the pigeonhole principle again, there are

$$\left\lceil \frac{7}{6} \right\rceil = 2$$

of these maximal pairs $(n_a, m_a, l_a), (n_b, m_b, l_b)$ such that the distance $|m_a - m_b|$ is at most $\frac{2}{3}L$.

According to Lemma 5, both $S[n_a..i-1]$ and $S[n_b..i-1]$ have the same minimal period length. Hence, the corresponding maximal repeats are of the form $p_a P^3 s_a r_a$ and $p_b P^3 s_b r_b$ where $p_a P^3$ and $p_b P^3$ are the $|P|$ -periodic parts left of i , the substrings s_a and s_b are the maximal $|P|$ -periodic extensions of $p_a P^3$ and $p_b P^3$ to the right and r_a and r_b are the remaining characters of the maximal repeats.

Since the two $|P|$ -periodic strings $p_a P^3 s_a$ and $p_b P^3 s_b$ starting at n_a and n_b overlap at least by $3|P|$ and since s_a and s_b are the maximal $|P|$ -periodic extensions of $p_a P^3$ and $p_b P^3$, respectively, this implies that $s_a = s_b$. Therefore, the maximal repeats are of the form $p_a P^3 s r_a$ and $p_b P^3 s r_b$.

27:8 On Extensions of Maximal Repeats in Compressed Strings

Since $|m_a - m_b| \leq \frac{2}{3}L$ holds, we can show that the $|P|$ -periodic strings $p_a P^3 s$ and $p_b P^3 s$ starting at the indices m_a and m_b have at least an overlap of length $|P|$:

The strings $p_a P^3 s$ and $p_b P^3 s$ have at least the length $3|P|$. Therefore, if $P \geq \frac{L}{3}$ holds, the overlap is at least $3|P| - \frac{2}{3}L \geq |P|$.

The strings $p_a P^3 s$ and $p_b P^3 s$ also have at least the length L . Therefore, if $P \leq \frac{L}{3}$ holds, the overlap is at least $L - \frac{2}{3}L = \frac{L}{3} \geq |P|$.

In either case, the overlap is at least as long as P .

Therefore, the union of the occurrences of $p_a P^3 s$ and $p_b P^3 s$ starting at m_a and m_b is $|P|$ -periodic. This implies that these two occurrences end with the same character.

If the lengths of $p_a P^3 s$ and $p_b P^3 s$ are different, this implies that both occurrences of the smaller string starting at the indices n_a and m_a or at the indices n_b and m_b are preceded by the same character which is given by the $|P|$ -periodic extension to the left. This, however, implies that either (n_a, m_a, l_a) or (n_b, m_b, l_b) is not a maximal pair.

If, on the other hand, the lengths of $p_a P^3 s$ and $p_b P^3 s$ are equal, the starting indices n_a and n_b are equal and the starting indices m_a and m_b are equal as well. This, however, is only possible if either (n_a, m_a, l_a) or (n_b, m_b, l_b) is not a maximal pair or if both maximal pairs are identical.

Since both cases contradict the assumption, the assumption is wrong and the theorem is therefore true. \blacktriangleleft

► **Corollary 7.** *Let S be a string. Let $q \in \mathbb{N}_{\geq 3}$ be a natural number and i, j be indices of two characters in $S\$$.*

Then there are at most $12 \left(1 + 3 \frac{q}{\log_2 q}\right) \log_2 |S|$ different maximal pairs (n_k, m_k, l_k) such that for all k

- *the corresponding maximal repeat $S[n_k..n_k + l_k - 1]$ is not $\frac{1}{\geq 2q}$ -highly-periodic and*
- *the indices i and j are contained in the intervals $[n_k, n_k + l_k]$ and $[m_k, m_k + l_k]$, respectively.*

Also, there are at most $12 \left(1 + 3 \frac{q}{\log_2 q}\right) (\log_2 |S|)(z_S + 1)(z_S + 2)$ different double-sided extensions of non- $\frac{1}{\geq 2q}$ -highly-periodic maximal repeats.

Proof. Without loss of generality, the inequality $q \leq |S|$ holds.

If a maximal repeat $S[n_k..n_k + l_k]$ is not $\frac{1}{\geq 2q}$ -highly-periodic, then either the longer of the parts $S[n_k..i - 1]$ and $S[i..n_k + l_k - 1]$ is

- not $\frac{1}{\geq q}$ -highly-periodic or
- $\frac{1}{\geq q}$ -highly-periodic but the corresponding periodicity does not extend to the whole maximal repeat $S[n_k..n_k + l_k]$.

Therefore, the number of different maximal pairs which fulfill the prerequisites can be bound by Lemma 3 and Theorem 6 and there are at most

$$18q(1 + \log_q |S|) + 12(\log_2 |S|) \leq 36q(\log_q |S|) + 12(\log_2 |S|) = 12 \left(1 + 3 \frac{q}{\log_2 q}\right) \log_2 |S|$$

of those maximal pairs.

Summing up over the first indices $i \leq j$ of the $z_S + 1$ LZ77-factors of $S\$$ yields that there are at most

$$\sum_{i=1}^{z_S+1} \sum_{j=i}^{z_S+1} 12 \left(1 + 3 \frac{q}{\log_2 q}\right) \log_2 |S| = 6 \left(1 + 3 \frac{q}{\log_2 q}\right) \log_2 |S| (z_S + 1)(z_S + 2)$$

substantially different non- $\frac{1}{\geq 2q}$ -highly-periodic maximal pairs.

Hence, there are at most $12 \left(1 + 3 \frac{q}{\log_2 q}\right) (\log_2 |S|)(z_S + 1)(z_S + 2)$ different double-sided extensions of maximal repeats that are not $\frac{1}{\geq 2q}$ -highly-periodic. \blacktriangleleft

For $q = 3$ this proves that the number of substantially different non- $\frac{1}{\geq 6}$ -highly-periodic maximal pairs is bounded from above by $41(\log_2 |S|)(z_S + 1)(z_S + 2)$.

4 Highly-Periodic Maximal Pairs

The goal of this section is to prove that in a string S the number of substantially different non-extendable $\frac{1}{\geq 4}$ -highly-periodic maximal pairs bounded from above by $32(\log_2 |S|)(z_S + 1)^2$.

Both occurrences of those maximal pairs, including the corresponding maximal repeat as well as the preceding and succeeding characters, are inside of the two padded maximal periodic extensions of the corresponding positioned maximal repeats.

Therefore, we will first count the number of substantially different padded maximal periodic extensions of fourth powers and the number of substantially different padded maximal periodic extensions of a given fourth power. Afterwards, we will show that each pair of padded maximal periodic extensions gives rise to at most 4 substantially different non-extendable $\frac{1}{\geq 4}$ -highly-periodic maximal pairs.

We will need the “Three Squares Lemma” of Crochemore and Rytter presented in [5].

► **Lemma 8.** *Let u , v and w be primitive and let u^2 , v^2 and w^2 be prefixes/suffixes of S with $|u| < |v| < |w|$.*

Then $|w| > |u| + |v|$ holds.

► **Lemma 9.** *Let S be a string and i be an index of a character in S .*

Then there are at most $4 \lfloor \log_2 |S| \rfloor$ substantially different padded maximal periodic extensions $S[l-1..r+1]$ of fourth powers such that $l-1 < i \leq r+1$.

Proof. In this proof we will only count the number of padded maximal periodic extensions $S[l-1..r+1]$ of fourth powers such that at least half of the interval $[l, r]$ is smaller than i , i.e. $l + \frac{r-l+1}{2} \leq i$. The other case $l + \frac{r-l+1}{2} \geq i$ is symmetrical.

Since $S[l..r]$ is at least a fourth power, the string $S[l..i-1]$ is at least a square. Therefore, two maximal periodic extensions $S[l, r]$ and $S[l', r']$ of fourth powers have an overlap of least twice the smaller minimal period length. Therefore, if their minimal period lengths are equal, the padded maximal periodic extensions $S[l-1, r+1]$ and $S[l'-1, r'+1]$ are copies of each other. Conversely, if $S[l-1, r+1]$ and $S[l'-1, r'+1]$ are substantially different, then they have different minimal period lengths as well.

This implies that the number of substantially different padded maximal periodic extensions $S[l-1, r+1]$ of fourth powers such that at least half of the interval $[l, r]$ is smaller than i is bounded from above by the number of different primitively rooted squares that are suffixes of $S[1..i-1]$.

The three squares lemma implies that for three primitively rooted squares which are suffixes of each other, the largest square is more than twice as long as the smallest square.

Since the smallest square has at least two characters and the largest square has at most $|S|$ characters, there are at most $2 \lfloor \log_2 |S| \rfloor$ primitively rooted squares which are suffixes of $S[1..i-1]$.

Therefore, there are at most $2 \lfloor \log_2 |S| \rfloor$ padded maximal periodic extensions $S[l-1, r+1]$ of fourth powers such that at least half of the interval $[l, r]$ is smaller than i , i.e. $l + \frac{r-l+1}{2} \leq i$.

This implies that the number of padded maximal periodic extensions of fourth powers $S[l-1, r+1]$ such that $l-1 < i \leq r+1$ is bounded from above by $4 \lfloor \log_2 |S| \rfloor$. ◀

The proof also allows another useful conclusion.

27:10 On Extensions of Maximal Repeats in Compressed Strings

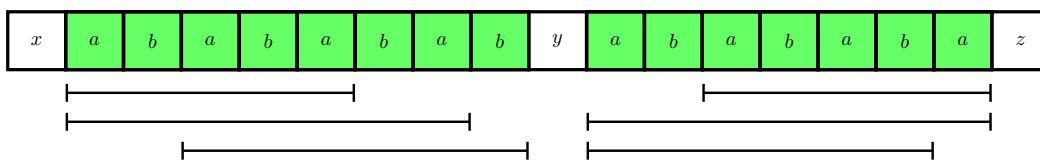


Figure 2 The string $S = xababababyabababaz$ with two maximal periodic extensions of the substring ab and the three non-extendable maximal pairs with the root ab . The maximal periodic extensions are the two green substrings and each maximal pair is represented by the two occurrences of its maximal repeat.

► **Corollary 10.** Let S be a string and i be an index of a character in $S\$$. Furthermore, let P be a $\frac{1}{\geq 4}$ -highly-periodic substring of S .

Then there are at most 2 substantially different padded maximal periodic extensions $S[l-1, r+1]$ of cyclic permutations of P such that $l-1 < i \leq r+1$.

Combining the previous corollary with the lemma before gives rise to an upper bound of the pairs of corresponding maximal periodic extensions.

► **Lemma 11.** Each pair of padded maximal periodic extensions of fourth powers which are up to cyclic rotation identical gives rise to at most 4 substantially different non-extendable $\frac{1}{\geq 4}$ -highly-periodic maximal pairs.

Proof. Each maximal pair has to be a prefix of the one padded maximal periodic extension and a suffix of the other padded maximal periodic extension, otherwise both corresponding positioned maximal repeats would be preceded or succeeded by the same character. There are two choices of which padded maximal periodic extension the corresponding positioned maximal repeat is a prefix.

For a fixed choice, the length of the maximal repeat is fixed, up to a multiple of the period length. Therefore there are only two possible lengths such that at least one of the positioned maximal repeat is not extendable.

Figure 2 shows a string with two maximal periodic extension of the substring ab and the 3 different non-extendable maximal pairs which arise from these extensions. ◀

Multiplying these upper bounds leads to the wanted upper bound:

► **Corollary 12.** Let S be a string.

Then there are at most $8(\log_2 |S|)(z_S + 1)^2$ substantially different pairs of padded maximal periodic extensions of fourth powers which are up to cyclic rotation identical.

Also, there are at most $32(\log_2 |S|)(z_S + 1)^2$ substantially different non-extendable $\frac{1}{\geq 4}$ -highly-periodic maximal pairs.

5 RLBWT and Maximal Pairs

The goal of this section is to prove that the runs of the RLBWT of a string S correspond to a subset of the maximal pairs, whose size can be bound from above by $73(\log_2 |S|)(z_S + 2)^2$.

Since we are interested in the number of runs, it is useful to observe the indices i where a new run starts. These are exactly the index 1 and the indices i with $S[\pi_{i-1}] \neq S[\pi_i]$.

Since $\$$ occurs exactly once in $S\$$, the strings $S[\pi_{i-1} + 1..|S| + 1]$ and $S[\pi_i + 1..|S| + 1]$ have a mismatch. Also, since $S[\pi_{i-1} + 1..|S| + 1]S[1..\pi_{i-1}]$ is lexicographically smaller than $S[\pi_i + 1..|S| + 1]S[1..\pi_i]$, the string $S[\pi_{i-1} + 1..|S| + 1]$ is lexicographically strictly smaller than $S[\pi_i + 1..|S| + 1]$.

Let m be the index of the first mismatch of these two strings. With this notation, the strings $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ and $S[\pi_i + 1.. \pi_i + m - 1]$ are equal and their predecessors as well as their successors are different. Therefore, if $m > 0$, they form a maximal pair. If $m = 0$, then $S[\pi_{i-1} + 1] < S[\pi_i + 1]$. This, however can only occur $|\Sigma|$ times.

On the other hand, since $S[\pi_{i-1} + 1.. \pi_{i-1} + m]$ is smaller than $S[\pi_i + 1.. |S| + 1]S[1.. \pi_i]$ and $S[\pi_i + 1.. \pi_i + m]$ is larger than $S[\pi_{i-1} + 1.. |S| + 1]S[1.. \pi_{i-1}]$, this maximal pair can only correspond to this pair (π_{i-1}, π_i) of lexicographically neighbored cyclic permutations and the maximal pairs corresponding to different pairs (π_{j-1}, π_j) of lexicographically neighbored cyclic permutations are substantially different.

► **Remark 13.** Belazzougui et al. show in Theorem 1 of [1] that the number of runs in the Burrows-Wheeler transform is even bounded in the number of right extensions of the maximal repeats. However, maximal pairs are easier to handle than right extensions of maximal repeats and we only lose a factor Σ in the worst-case by not using the right extensions.

However, while the number of maximal repeats and thereby the number of nodes in the CDAWG can be $\Theta(qz^3)$ for a suitable set of strings, the Burrows-Wheeler transform does not suffer from high powers as the CDAWG does:

► **Lemma 14.** *Let S be a string and let i be an index at which a new run in the Burrows-Wheeler transform starts.*

Then the maximal pair corresponding to the pair (π_{i-1}, π_i) of lexicographically neighbored cyclic permutations is not extendable.

Proof. Since a maximal pair is not extendable if at least one of its corresponding positioned maximal repeats is not extendable, we have to prove that at least one of the positioned maximal repeats $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ and $S[\pi_i + 1.. \pi_i + m - 1]$ is not extendable. Let p be the minimal period length of this maximal repeat.

Assume that the maximal p -periodic extensions of both occurrences $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ and $S[\pi_i + 1.. \pi_i + m - 1]$ contain at least $p + 1$ additional characters. In this proof, we will show that under this assumption that there is a cyclic permutation $S[w + 1.. |S| + 1]S[1.. w]$ of $S\$$ which is lexicographically between $S[\pi_{i-1} + 1.. |S| + 1]S[1.. \pi_{i-1}]$ and $S[\pi_i + 1.. |S| + 1]S[1.. \pi_i]$.

If the maximal p -periodic extension of $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ extends this occurrence to the left, the equation $S[\pi_{i-1}] = S[\pi_{i-1} + p]$ and thereby

$$S[\pi_i] \neq S[\pi_{i-1}] = S[\pi_{i-1} + p] = S[\pi_i + p]$$

holds. Therefore, the maximal p -periodic extension of $S[\pi_i + 1.. \pi_i + m - 1]$ does not extend this string to the left. This implies that at most one of the two maximal p -periodic extensions of the occurrences $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ and $S[\pi_i + 1.. \pi_i + m - 1]$ does extend the occurrence to the left.

Similarly, at most one of those occurrences is extended to the right by the maximal p -periodic extension.

Since, by assumption, both occurrences are p -periodically extendable, exactly one occurrence has to be p -periodically extendable to the left and exactly one occurrence has to be p -periodically extendable to the right. By symmetry we can assume without loss of generality that $S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1]$ is p -periodically extendable to the left and that $S[\pi_i + 1.. \pi_i + m - 1]$ is p -periodically extendable to the right.

Hence, $S[\pi_{i-1} - p.. \pi_{i-1} + m - 1]$ and $S[\pi_i + 1.. \pi_i + m + p]$ are p -periodic. Also, by definition of the Burrows-Wheeler transform, the inequality $S[\pi_{i-1} + m] < S[\pi_i + m]$ holds.

Combining the periodicity with this inequality yields

$$S[\pi_{i-1} + 1.. \pi_{i-1} + m - 1] = S[\pi_{i-1} + 1 - p.. \pi_{i-1} + m - 1 - p]$$

27:12 On Extensions of Maximal Repeats in Compressed Strings

and

$$S[\pi_{i-1} + m] < S[\pi_i + m] = S[\pi_i + m - p] = S[\pi_{i-1} + m - p]$$

which imply

$$S[\pi_{i-1} + 1..|S| + 1]S[1..\pi_{i-1}] < S[\pi_{i-1} + 1 - p..|S| + 1]S[1..\pi_{i-1} - p].$$

Similarly, we get

$$S[\pi_{i-1} + 1 - p..\pi_{i-1} + m - 1] = S[\pi_i + 1..\pi_i + m - 1 + p]$$

and

$$S[\pi_{i-1} + m] < S[\pi_i + m] = S[\pi_i + m + p]$$

which imply

$$S[\pi_{i-1} + 1 - p..|S| + 1]S[1..\pi_{i-1} - p] < S[\pi_i + 1..|S| + 1]S[1..\pi_i].$$

Since $S[\pi_{i-1} + 1 - p..|S| + 1]S[1..\pi_{i-1} - p]$ is lexicographically between the cyclic permutations $S[\pi_{i-1} + 1..|S| + 1]S[1..\pi_{i-1}]$ and $S[\pi_i + 1..|S| + 1]S[1..\pi_i]$, these two strings are not neighbors with regard to the Burrows-Wheeler transform. This contradicts the assumption and thereby concludes the proof. ◀

Therefore, the positioned maximal repeats of the associated maximal pairs corresponding to the RLBWT are either not highly-periodic or, if they are highly-periodic, the period cannot be extended by more than a period length. This implies the following corollary and thereby leads to another proof of the Burrows-Wheeler conjecture:

► **Corollary 15.** *Let S be a string.*

Then, there are at most $73(\log_2 |S|)(z_S + 2)^2$ runs in the RLBWT.

Proof. We count the number of index-pairs (π_{i-1}, π_i) where a new run starts.

Either (π_{i-1}, π_i) corresponds to

- the empty maximal pair (there are at most $|\Sigma|$ of such (π_{i-1}, π_i)),
- a non- $\frac{1}{\geq 6}$ -highly-periodic maximal pair (there are at most $41(\log_2 |S|)(z_S + 1)(z_S + 2)$ of such (π_{i-1}, π_i)) or
- a $\frac{1}{\geq 4}$ -highly-periodic non-extendable maximal pair (there are at most $32(\log_2 |S|)(z_S + 1)^2$ of such (π_{i-1}, π_i)).

We also have to count one additional run for $i = 0$.

Summing up shows that there are at most $73(\log_2 |S|)(z_S + 2)^2$ runs in the RLBWT. ◀

6 Conclusion

This paper proved that of the potentially $\mathcal{O}(qz^3)$ substantially different maximal pairs in a string, it is sufficient to understand a subset containing at most $73(\log_2 |S|)(z_S + 2)^2$ maximal pairs.

It seems therefore likely that it is possible to merge the nodes of the CDAWG which correspond maximal repeats of the same base and get a new data structure which is almost as universal and intuitive as the CDAWG but only contains $\mathcal{O}((\log |S|)(z_S)^2)$ arcs.

Also, the proofs presented in this paper do not use the underlying structure of the string. If the substrings of S and the reversed string S_{rev} are also highly compressible and have less than z' LZ77-factors each, it should be possible to prove that the number of runs in the RLBWT is bounded from above by $\mathcal{O}(z'(z_S)^2)$.

Thereby, it might be possible to derive an upper bound for the runs in the RLBWT which is only dependent on the number of LZ77-factors. Since the strings used to prove the asymptotic tightness for the upper bound $r_S \in \mathcal{O}\left(\delta_S \log \delta_S \max\left(1, \log \frac{n}{\delta_S \log \delta_S}\right)\right)$ in [8] have $z_S \in \Omega\left(\delta_S \log_2 \frac{n}{\delta_S}\right)$ LZ77-factors, such a result does not violate the asymptotic tightness.

References

- 1 Djamel Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.
- 2 Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.
- 3 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, DEC Systems Research Center, 1994.
- 4 Manolis Christodoulakis, Costas S. Iliopoulos, and Yoan José Pinzón Ardila. Simple algorithm for sorting the fibonacci string rotations. In Jiří Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Július Štuller, editors, *SOFSEM 2006: Theory and Practice of Computer Science*, pages 218–225, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 5 M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, May 1995. doi:10.1007/BF01190846.
- 6 N. J. Fine and H. 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>.
- 7 I. Furuya, T. Takagi, Y. Nakashima, S. Inenaga, H. Bannai, and T. Kida. Mr-repair: Grammar compression based on maximal repeats. In *2019 Data Compression Conference (DCC)*, pages 508–517, March 2019. doi:10.1109/DCC.2019.00059.
- 8 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. *CoRR*, abs/1910.10631, 2019. arXiv:1910.10631.
- 9 Julian Pape-Lange. On Maximal Repeats in Compressed Strings. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*, volume 128 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2019.18.
- 10 Mathieu Raffinot. On maximal repeats in strings. *Inf. Process. Lett.*, 80(3):165–169, 2001. doi:10.1016/S0020-0190(01)00152-1.

Faster Binary Mean Computation Under Dynamic Time Warping

Nathan Schaar

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
n.schaar@campus.tu-berlin.de

Vincent Froese

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
vincent.froese@tu-berlin.de

Rolf Niedermeier

Technische Universität Berlin, Faculty IV, Algorithmics and Computational Complexity, Germany
rolf.niedermeier@tu-berlin.de

Abstract

Many consensus string problems are based on Hamming distance. We replace Hamming distance by the more flexible (e.g., easily coping with different input string lengths) dynamic time warping distance, best known from applications in time series mining. Doing so, we study the problem of finding a mean string that minimizes the sum of (squared) dynamic time warping distances to a given set of input strings. While this problem is known to be NP-hard (even for strings over a three-element alphabet), we address the binary alphabet case which is known to be polynomial-time solvable. We significantly improve on a previously known algorithm in terms of worst-case running time. Moreover, we also show the practical usefulness of one of our algorithms in experiments with real-world and synthetic data. Finally, we identify special cases solvable in linear time (e.g., finding a mean of only two binary input strings) and report some empirical findings concerning combinatorial properties of optimal means.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Theory of computation → Pattern matching

Keywords and phrases consensus string problems, time series averaging, minimum 1-separated sum, sparse strings

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.28

Supplementary Material Python source code available at <https://www.akt.tu-berlin.de/menu/software/>.

Funding *Nathan Schaar*: Partially supported by DFG NI 369/19.

1 Introduction

Consensus problems are an integral part of stringology. For instance, in the frequently studied CLOSEST STRING problem one is given k strings of equal length and the task is to find a center string that minimizes the maximum Hamming distance to all k input strings. CLOSEST STRING is NP-hard even for binary alphabet [11] and has been extensively studied in context of approximation and parameterized algorithmics [6, 9, 7, 8, 13, 15, 17, 20]. Notably, when one wants to minimize the sum of distances instead of the maximum distance, the problem is easily solvable in linear time by taking at each position a letter that appears most frequently in the input strings.

Hamming distance, however, is quite limited in many application contexts; for instance, how to define a center string in case of input strings that do not all have the same length? In context of analyzing time series (basically strings where the alphabet consists of rational numbers), the “more flexible” *dynamic time warping distance* [18] enjoys high popularity and



© Nathan Schaar, Vincent Froese, and Rolf Niedermeier;
licensed under Creative Commons License CC-BY

31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).

Editors: Inge Li Gørtz and Oren Weimann; Article No. 28; pp. 28:1–28:13

Leibniz International Proceedings in Informatics



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

can be computed for two input strings in subquadratic time [12, 14], essentially matching corresponding conditional lower bounds [1, 3]. Roughly speaking (see Section 2 for formal definitions and an example), measuring the dynamic time warping distance (dtw for short) can be seen as a two-step process: First, one aligns one time series with the other (by stretching them via duplication of elements) such that both time series end up with the same length. Second, one then calculates the Euclidean distance of the aligned time series (recall that here the alphabet consists of numbers). Importantly, restricting to the binary case, the dtw distance of two time series can be computed in $O(n^{1.87})$ time [1], where n is the maximum time series length (a result that will also be relevant for our work).

With the dtw distance at hand, the most fundamental consensus problem in this (time series) context is, given k input “strings” (over rational numbers), compute a mean string that minimizes the sum of (squared) dtw distances to all input strings. This problem is known as DTW-MEAN in the literature and only recently has been shown to be NP-hard [4, 5]. For the most basic case, namely binary alphabet (that is, input and output are binary), however, the problem is known to be solvable in $O(kn^3)$ time [2]. By way of contrast, if one allows the mean to contain any rational numbers, then the problem is NP-hard even for binary inputs [5]. Moreover, the problem is also NP-hard for ternary input and output [4].

Formally, in this work we study the following problem:

BINARY DTW-MEAN (BDTW-MEAN)

Input: Binary strings s_1, \dots, s_k of length at most n and $c \in \mathbb{Q}$.

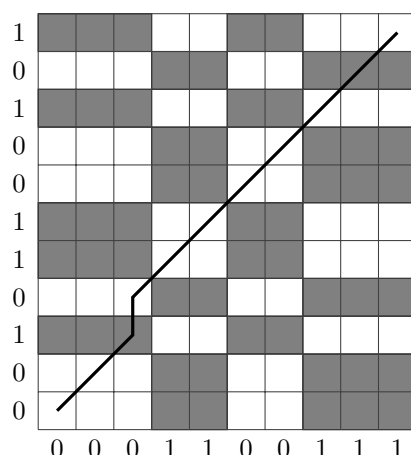
Question: Is there a binary string z such that $F(z) := \sum_{i=1}^k \text{dtw}(s_i, z)^2 \leq c$?

Herein, the dtw function is formally defined in Section 2. The study of the special case of binary data may occur when one deals with binary states (e.g., switching between the active and the inactive mode of a sensor); binary data were recently studied in the dynamic time warping context [16, 19]. Clearly, binary data can always be generated from more general data by “thresholding”.

Our main theoretical result is to show that BDTW-MEAN can be solved in $O(kn^{1.87})$ and $O(k(n + m(m - \mu)))$ time, respectively, where m is the maximum and μ is the median condensation length of the input strings (the condensation of a string is obtained by repeatedly removing one of two identical consecutive elements). While the first algorithm, relies on an intricate “blackbox-algorithm” for a certain number problem from the literature (which so far was never implemented), the second algorithm (which we implemented) is more directly based on combinatorial arguments. Anyway, our new bounds improve on the standard $O(kn^3)$ -time bound [2]. Moreover, we also experimentally tested our second algorithm and compared it to the standard one, clearly outperforming it (typically by orders of magnitude) on real-world and on synthetic instances. Further theoretical results comprise linear-time algorithms for special cases (two input strings or three input strings with some additional constraints). Further empirical results relate to the typical shape of a mean.

2 Preliminaries

For $n \in \mathbb{N}$, let $[n] := \{1, \dots, n\}$. We consider binary strings $x = x[1]x[2] \dots x[n] \in \{0, 1\}^n$. We denote the length of x by $|x|$ and we also denote the last symbol $x[n]$ of x by $x[-1]$. For $1 \leq i \leq j \leq |x|$, we define the substring $x[i, j] := x[i] \dots x[j]$. A maximal substring of consecutive 1’s (0’s) in x is called a *1-block* (*0-block*). The i -th block of x (from left to right) is denoted $x^{(i)}$. A string x is called *condensed* if no two consecutive elements are equal, that is, every block is of size 1. The *condensation* of x is denoted \tilde{x} and is defined as the



■ **Figure 1** An optimal warping path for the strings $x = 00101100101$ (vertical axis) and $y = 0001100111$ (horizontal axis). Black cells have local cost 1. The string x consists of eight blocks with sizes 2,1,1,2,2,1,1,1 and y consists of four blocks with sizes 3,2,2,3. An optimal warping path has to pass through $(8 - 4)/2 = 2$ non-neighboring blocks of the six inner blocks of x .

condensed string obtained by removing one of two equal consecutive elements of x until the remaining series is condensed. Note that the condensation length $|\tilde{x}|$ equals the number of blocks in x .

The dynamic time warping distance measures the similarity of two strings using non-linear alignments defined via so-called warping paths.

► **Definition 1.** A warping path of order $m \times n$ is a sequence $p = (p_1, \dots, p_L)$, $L \in \mathbb{N}$, of index pairs $p_\ell = (i_\ell, j_\ell) \in [m] \times [n]$, $1 \leq \ell \leq L$, such that

- (i) $p_1 = (1, 1)$,
- (ii) $p_L = (m, n)$, and
- (iii) $(i_{\ell+1} - i_\ell, j_{\ell+1} - j_\ell) \in \{(1, 0), (0, 1), (1, 1)\}$ for each $\ell \in [L - 1]$.

A warping path can be visualized within an $m \times n$ “warping matrix” (see Figure 1). The set of all warping paths of order $m \times n$ is denoted by $\mathcal{P}_{m,n}$. A warping path $p \in \mathcal{P}_{m,n}$ defines an alignment between two strings $x \in \mathbb{Q}^m$ and $y \in \mathbb{Q}^n$ in the following way: A pair $(i, j) \in p$ aligns element x_i with y_j with a local cost of $(x_i - y_j)^2$. The dtw distance between two strings x and y is defined as

$$\text{dtw}(x, y) := \min_{p \in \mathcal{P}_{m,n}} \sqrt{\sum_{(i,j) \in p} (x_i - y_j)^2}.$$

It is computable via standard dynamic programming in $O(mn)$ time¹ [18], with recent theoretical improvements to subquadratic time [12, 14].

3 DTW on Binary Strings

We briefly discuss some known results about the dtw distance between binary strings since these will be crucial for our algorithms for BDTW-MEAN.

¹ Throughout this work, we assume that all arithmetic operations can be carried out in constant time.

Abboud et al. [1, Section 5] showed that the dtw distance of two binary strings of length at most n can be computed in $O(n^{1.87})$ time. They obtained this result by reducing the dtw distance computation to the following integer problem.

MIN 1-SEPARATED SUM (MSS)

Input: A sequence (b_1, \dots, b_m) of m positive integers and an integer $r \geq 0$.

Task: Select r integers b_{i_1}, \dots, b_{i_r} with $1 \leq i_1 < i_2 < \dots < i_r \leq m$ and $i_j < i_{j+1} - 1$ for all $1 \leq j < r$ such that $\sum_{j=1}^r b_{i_j}$ is minimized.

The integers of the MSS instance correspond to the block sizes of the input string which contains more blocks.

► **Theorem 2** ([1, Theorem 8]). *Let $x \in \{0, 1\}^m$ and $y \in \{0, 1\}^n$ be two binary strings such that $x[1] = y[1]$, $x[m] = y[n]$, and $|\tilde{x}| \geq |\tilde{y}|$. Then, $\text{dtw}(x, y)^2$ equals the sum of a solution for the MSS instance $((|x^{(2)}|, \dots, |x^{(|\tilde{x}|-1)}|), (|\tilde{x}| - |\tilde{y}|)/2)$.*

The idea behind Theorem 2 is that exactly $(|\tilde{x}| - |\tilde{y}|)/2$ non-neighboring blocks of x are misaligned in any warping path (note that $|\tilde{x}| - |\tilde{y}|$ is even since x and y start and end with the same symbol). An optimal warping path can thus be obtained from minimizing the sum of block sizes of these misaligned blocks. For example, in Figure 1 the dtw distance corresponds to a solution of the MSS instance $((1, 1, 2, 2, 1, 1), 2)$.

Abboud et al. [1, Theorem 10] showed how to solve MSS in $O(n^{1.87})$ time, where $n = \sum_{i=1}^m b_i$. They gave a recursive algorithm that, on input $((b_1, \dots, b_m), r)$, outputs four lists C^{00}, C^{0*}, C^{*0} , and C^{**} , where, for $t \in \{0, \dots, r\}$,

- $C^{**}[t]$ is the sum of a solution for the MSS instance $((b_1, \dots, b_m), t)$,
- $C^{0*}[t]$ is the sum of a solution for the MSS instance $((b_2, \dots, b_m), t)$,
- $C^{*0}[t]$ is the sum of a solution for the MSS instance $((b_1, \dots, b_{m-1}), t)$, and
- $C^{00}[t]$ is the sum of a solution for the MSS instance $((b_2, \dots, b_{m-2}), t)$.

Note that $C^{**}[r]$ yields the solution. We will make use of their algorithm when solving BDTW-MEAN. We will also use the following simple dynamic programming algorithm for MSS which is faster for large input integers.

► **Lemma 3.** *MIN 1-SEPARATED SUM is solvable in $O(mr)$ time.*

Proof. Let $((b_1, \dots, b_m), r)$ be an MSS instance. We define a dynamic programming table M as follows: For each $i \in [m]$ and each $j \in \{0, \dots, \min(r, \lceil i/2 \rceil)\}$, $M[i, j]$ is the sum of a solution of the subinstance $((b_1, \dots, b_i), j)$. Clearly, it holds $M[i, 0] = 0$ and $M[i, 1] = \min\{b_1, \dots, b_i\}$ for all i . Further, it holds $M[3, 2] = b_1 + b_3$. For all $i \in \{4, \dots, m\}$ and $j \in \{2, \dots, \min(r, \lceil i/2 \rceil)\}$, the following recursion holds

$$M[i, j] = \min(b_i + M[i - 2, j - 1], M[i - 1, j]).$$

Hence, the table M can be computed in $O(mr)$ time. ◀

Note that the above algorithms only compute the dtw distance between binary strings with equal starting and ending symbol. However, it is an easy observation that the dtw distance of arbitrary binary strings can recursively be obtained from this via case distinction on which first and/or which last block to misalign.

► **Observation 4** ([1, Claim 6]). *Let $x \in \{0, 1\}^m$, $y \in \{0, 1\}^n$ with $m' := |\tilde{x}| \geq n' := |\tilde{y}|$. Further, let $a := |x^{(1)}|$, $a' := |x^{(m')}|$, $b := |y^{(1)}|$, and $b' := |y^{(n')}|$. The following holds:*

- If $x[1] \neq y[1]$, then

$$\text{dtw}(x, y)^2 = \begin{cases} \max(a, b), & m' = n' = 1 \\ a + \text{dtw}(x[a + 1, m], y)^2, & m' > n' = 1. \\ \min(a + \text{dtw}(x[a + 1, m], y)^2, b + \text{dtw}(x, y[b + 1, n])^2), & n' > 1 \end{cases}$$

- If $x[1] = y[1]$ and $x[m] \neq y[n]$, then

$$\text{dtw}(x, y)^2 = \begin{cases} a' + \text{dtw}(x[1, m - a'], y)^2, & n' = 1 \\ \min(a' + \text{dtw}(x[1, m - a'], y)^2, b' + \text{dtw}(x, y[1, n - b'])^2), & n' > 1 \end{cases}$$

For condensed strings, Brill et al. [2] derived the following useful closed form for the dtw distance (which basically follows from Observation 4 and Theorem 2).

► **Lemma 5** ([2, Lemma 1 and 2]). *For a condensed binary string x and a binary string y with $|\tilde{y}| \leq |x|$, it holds that*

$$\text{dtw}(x, y)^2 = \begin{cases} \lceil (|x| - |\tilde{y}|)/2 \rceil, & x_1 = y_1 \\ 2, & x_1 \neq y_1 \wedge |x| = |\tilde{y}| \\ 1 + \lfloor (|x| - |\tilde{y}|)/2 \rfloor, & x_1 \neq y_1 \wedge |x| > |\tilde{y}| \end{cases}$$

Note that Lemma 5 implies that one can compute the dtw distance in constant time when the condensation lengths of the inputs are known and the string with longer condensation length is condensed.

Our key lemma now states that the dtw distances between an arbitrary fixed string and all condensed strings of shorter condensation length can also be computed efficiently.

► **Lemma 6.** *Let $s \in \{0, 1\}^n$ with $\ell := |\tilde{s}|$. Given ℓ and the block sizes b_1, \dots, b_ℓ of s , the dtw distances between s and all condensed strings of lengths ℓ', \dots, ℓ for some given $\ell' \leq \ell$ can be computed in*

- (i) $O(n^{1.87})$ time and in
- (ii) $O(\ell(\ell - \ell'))$ time, respectively.

Proof. Let x be a condensed string of length $i \in \{\ell', \dots, \ell\}$. Observation 4 and Theorem 2 imply that we essentially have to solve MSS on four different subsequences of block sizes of s (depending on the first and last symbol of x) in order to compute $\text{dtw}(s, x)$. Namely, the four cases are $(b_2, \dots, b_{\ell-1})$, $(b_3, \dots, b_{\ell-1})$, $(b_2, \dots, b_{\ell-2})$, and $(b_3, \dots, b_{\ell-2})$. Let $r := \lceil (\ell - \ell')/2 \rceil$

(i) We run the algorithm of Abboud et al. [1, Theorem 10] on the instance $((b_2, \dots, b_{\ell-1}), r)$ to obtain in $O(n^{1.87})$ time the four lists $C^{\alpha\beta}$, for $\alpha, \beta \in \{0, 1\}$, where $C^{\alpha\beta}$ contains the solutions of $((b_{2+\alpha}, \dots, b_{\ell-1-\beta}), r')$ for all $r' \in \{0, \dots, r\}$. From these four lists, we can compute the requested dtw distances (using Observation 4) in $O(\ell)$ time.

(ii) We compute the solutions of the four above MSS instances using Lemma 3. For each $\alpha, \beta \in \{0, 1\}$, let $M^{\alpha\beta}$ be the dynamic programming table computed in $O(\ell(\ell - \ell'))$ time for the instance $((b_{2+\alpha}, \dots, b_{\ell-1-\beta}), r)$. Again, we can compute the requested dtw distances from these four tables in $O(\ell)$ time (using Observation 4). ◀

4 More Efficient Solution of BDTW-Mean

Brill et al. [2] gave an $O(kn^3)$ -time algorithm for BDTW-MEAN. The result is based on showing that there always exists a condensed mean of length at most $n + 1$. Thus, there are $2(n + 1)$ candidate strings to check. For each candidate, one can compute the dtw distance

to every input string in $O(kn^2)$ time. It is actually enough to only compute the dtw distance for the two length- $(n+1)$ candidates to all k input strings since the resulting dynamic programming tables also yield all the distances to shorter candidates. That is, the running time can actually be bounded in $O(kn^2)$.

We now give an improved algorithm. To this end, we first show the following improved bounds on the (condensation) length of a mean.

► **Lemma 7.** *Let s_1, \dots, s_k be binary strings with $|\tilde{s}_1| \leq \dots \leq |\tilde{s}_k|$ and let z be a mean of these k strings, that is,*

$$z \in \arg \min_{x \in \{0,1\}^*} \sum_{i=1}^k \text{dtw}(s_i, x)^2.$$

Then, it holds $\mu - 2 \leq |\tilde{z}| \leq m + 1$, where $\mu := |\tilde{s}_{\lceil k/2 \rceil}|$ is the median condensation length and $m := |\tilde{s}_k|$ is the maximum condensation length.

Proof. It suffices to show the claimed bounds for condensed means. Since $\text{dtw}(\tilde{x}, y) \leq \text{dtw}(x, y)$ holds for all strings x, y [2, Proposition 1], the bounds also hold for arbitrary means.

The upper bound $m + 1$ can be derived from Lemma 5. Let x be a condensed string of length $|x| \geq m + 2$ and let $x' := x[1, m]$. If $|x| > m + 2$, then $\text{dtw}(x', s_i)^2 < \text{dtw}(x, s_i)^2$ holds for every $i \in [k]$, which implies $F(x') = \sum_{i=1}^k \text{dtw}(s_i, x')^2 < \sum_{i=1}^k \text{dtw}(s_i, x)^2 = F(x)$. Hence, x is not a mean. If $|x| = m + 2$, then $\text{dtw}(x', s_i)^2 \leq \text{dtw}(x, s_i)^2$ holds for every $i \in [k]$, that is, $F(x') \leq F(x)$. If $F(x') < F(x)$, then x is clearly not a mean. If $F(x') = F(x)$, then $\text{dtw}(x', s_i)^2 = \text{dtw}(x, s_i)^2$ holds for all $i \in [k]$. In fact, $\text{dtw}(x', s_i)^2 = \text{dtw}(x, s_i)^2$ only holds if $|\tilde{s}_i| = m$ and $s_i[1] \neq x[1]$, in which case $\text{dtw}(x, s_i)^2 = 2$. Thus, we have $F(x) = 2k$ and $\tilde{s}_1 = \tilde{s}_2 = \dots = \tilde{s}_k$. But then \tilde{s}_1 is clearly the unique mean (with $F(\tilde{s}_1) = 0$).

For the lower bound, let x be a condensed string of length $\ell < \mu - 2$ and let $x' := x[1] \dots x[\ell]x[\ell - 1]x[\ell]$. Then, for every s_i with $|\tilde{s}_i| \leq \ell$ (of which there are less than $\lceil k/2 \rceil$ since $\ell < \mu$), it holds $\text{dtw}(x', s_i)^2 \leq \text{dtw}(x, s_i)^2 + 1$ (by Lemma 5).

Now, for every s_i with $|\tilde{s}_i| > \ell + 2$ (of which there are at least $\lceil k/2 \rceil$ since $\ell + 2 < \mu$), it holds $\text{dtw}(x', s_i)^2 \leq \text{dtw}(x, s_i)^2 - 1$. This is easy to see from Theorem 2 for the case that $s_i[1] = x'[1]$ and $s_i[-1] = x'[-1]$ holds since the number of misaligned blocks of s_i decreases by at least one. From this, Observation 4 yields the other three possible cases of starting and ending symbols since the sizes of the first and last block of x and of x' are clearly all the same (one).

It remains to consider input strings s_i with $\ell < |\tilde{s}_i| \leq \ell + 2$. We show that in this case $\text{dtw}(x', s_i)^2 \leq \text{dtw}(x, s_i)^2$ holds. Let $|\tilde{s}_i| = \ell + 2$. Note that then either $x'[1] = s_i[1]$ and $x'[-1] = s_i[-1]$ holds or $x'[1] \neq s_i[1]$ and $x'[-1] \neq s_i[-1]$ holds. In the former case, it clearly holds $\text{dtw}(x', s_i)^2 = 0$ by Lemma 5. In the latter case, we clearly have $\text{dtw}(x, s_i)^2 \geq 2$, and, by Lemma 5, we have $\text{dtw}(x', s_i)^2 = 2$. Finally, let $|\tilde{s}_i| = \ell + 1$ and note that then either $x'[1] = s_i[1]$ and $x'[-1] \neq s_i[-1]$ holds or $x'[1] \neq s_i[1]$ and $x'[-1] = s_i[-1]$ holds. Thus, we clearly have $\text{dtw}(x, s_i)^2 \geq 1$. By Lemma 5, we have $\text{dtw}(x', s_i)^2 = 1$.

Summing up, we obtain $F(x') \leq F(x) + a - b$, where $a = |\{i \in [k] \mid |\tilde{s}_i| < \ell\}| < \lceil k/2 \rceil$ and $b = |\{i \in [k] \mid |\tilde{s}_i| > \ell + 2\}| \geq \lceil k/2 \rceil$. That is, $F(x') < F(x)$ and x is not a mean. ◀

Note that the length bounds in Lemma 7 are tight. For the upper bound, consider the two strings 000 and 111 having the two means 01 and 10. For the lower bound, consider the seven strings 0, 0, 0, 101, 101, 010, 010 with the unique mean 0.

Lemma 7 upper-bounds the number of mean candidates we have to consider in terms of the condensation lengths of the inputs. In order to compute the dtw distances between mean candidates and input strings, we can now use Lemma 6. We arrive at the following result.

► **Theorem 8.** *Let s_1, \dots, s_k be binary strings with $|\tilde{s}_1| \leq \dots \leq |\tilde{s}_k|$ and $n := \max_{j=1, \dots, k} |s_j|$, $\mu := |\tilde{s}_{\lceil k/2 \rceil}|$, and $m := |\tilde{s}_k|$. The condensed means of s_1, \dots, s_k can be computed in*

- (i) $O(kn^{1.87})$ time and in
- (ii) $O(k(n + m(m - \mu)))$ time.

Proof. From Lemma 7, we know that there are $O(m - \mu)$ many candidate strings to check. First, in linear time, we determine the block lengths for each s_j . Now, let x be a candidate string, that is, x is a condensed binary string with $\mu - 2 \leq |x| \leq m + 1$. We need to compute $\text{dtw}(x, s_j)^2$ for each $j = 1, \dots, k$. Consider a fixed string s_j . For all candidates x with $|x| \geq |\tilde{s}_j|$, we can simply compute $\text{dtw}(x, s_j)^2$ in constant time using Lemma 5. For all x with $|x| < |\tilde{s}_j|$, we can use Lemma 6. Thus, overall, we can compute the dtw distances between all candidates and all input strings in $O(kn^{1.87})$ time, or alternatively in $O(km(m - \mu))$ time. Finally, we determine the candidates with the minimum sum of dtw distances in $O(k(m - \mu))$ time. ◀

We remark that similar results also hold for the related problems WEIGHTED BINARY DTW-MEAN, where the objective is to minimize $F(z) := \sum_{i=1}^k w_i \text{dtw}(s_i, z)^2$ for some $w_i \geq 0$, and BINARY DTW-CENTER with $F(z) := \max_{i=1, \dots, k} \text{dtw}(s_i, z)^2$ (that is, the dtw version of CLOSEST STRING). It is easy to see that also in these cases there exists a condensed solution. Moreover, the length is clearly bounded between the minimum and the maximum condensation length of the inputs. Hence, analogously to Theorem 8, we obtain the following.

► **Corollary 9.** *WEIGHTED BINARY DTW-MEAN and BINARY DTW-CENTER can be solved in $O(kn^{1.87})$ time and in $O(k(n + m(m - \nu)))$ time, where m is the maximum condensation length and ν is the minimum condensation length.*

5 Linear-Time Solvable Special Cases

Notably, Theorem 8 (ii) yields a linear-time algorithm when $m - \mu$ is constant and also when all input strings have the same length n and $m(m - \mu) \in O(n)$. Now, we show two more linear-time solvable cases.

► **Theorem 10.** *A condensed mean of two binary strings can be computed in linear time.*

Proof. Let $s_1, s_2 \in \{0, 1\}^*$ be two input strings. We first determine the condensations and block sizes of s_1 and s_2 in linear time. Let $\ell_i := |\tilde{s}_i|$, for $i \in [2]$, and assume that $\ell_1 \leq \ell_2$. In the following, all claimed relations between dtw distances can easily be verified using Observation 4 (together with Theorem 2) and Lemma 5.

If $\ell_1 = \ell_2$, then, by Theorem 8 (with $\mu = m = \ell_1$), all condensed means can be computed in $O(\ell_1)$ time.

If $\ell_1 < \ell_2$, then \tilde{s}_2 is a mean. To see this, note first that $F(\tilde{s}_2) = \text{dtw}(s_1, \tilde{s}_2)^2$. Let x be a condensed string. If $|x| < \ell_1$, then $\text{dtw}(s_1, x)^2 > 0$ and $\text{dtw}(s_2, x)^2 \geq \text{dtw}(s_2, \tilde{s}_1)^2 \geq \text{dtw}(\tilde{s}_2, \tilde{s}_1)^2 = \text{dtw}(\tilde{s}_2, s_1)^2$. Thus, $F(x) > F(\tilde{s}_2)$. Similarly, if $|x| > \ell_2$, then $\text{dtw}(s_1, x)^2 \geq \text{dtw}(s_1, \tilde{s}_2)^2$, $\text{dtw}(s_2, x)^2 > 0$, and $F(x) > F(\tilde{s}_2)$. If $\ell_1 \leq |x| < \ell_2$, then $\text{dtw}(s_1, x)^2 + \text{dtw}(s_2, x)^2 \geq \text{dtw}(s_1, \tilde{s}_2)^2$, and thus $F(x) \geq F(\tilde{s}_2)$. ◀

For three input strings, we show linear-time solvability if all strings begin with the same symbol and end with the same symbol.

► **Theorem 11.** *Let s_1, s_2, s_3 be binary strings with $s_1[1] = s_2[1] = s_3[1]$ and $s_1[-1] = s_2[-1] = s_3[-1]$. A condensed mean of s_1, s_2, s_3 can be computed in linear time.*

Proof. We first determine the condensations and block sizes of s_1, s_2 , and s_3 in linear time. Let $\ell_i := |\tilde{s}_i|$, for $i \in [3]$, and assume $\ell_1 \leq \ell_2 \leq \ell_3$. Note that every mean starts with $s_1[1]$ and ends with $s_1[-1]$. To see this, consider any string x with $x[1] \neq s_1[1]$ (or $x[-1] \neq s_1[-1]$) and observe that either removing the first (or last) symbol or adding $s_1[1]$ to the front (or $s_1[-1]$ to the end) yields a better F -value. Moreover, it is easy to see that every condensed mean has length at least ℓ_2 since increasing the length of any shorter condensed string by two increases the dtw distance to s_1 by at most one (Lemma 5) and decreases the dtw distances to s_2 and s_3 by at least one (Theorem 2).

Note that a mean could be even longer than ℓ_2 since further increasing the length by two increases the dtw distance to s_1 and s_2 by at most one and could possibly decrease the dtw distance to s_3 by at least two (if a misaligned block of size at least two can be saved). In fact, we can determine an optimal mean length in $O(\ell_3)$ time by greedily computing the maximum number ρ of 1-separated (that is, non-neighboring) blocks of size one among $s_3^{(2)}, \dots, s_3^{\ell_3-1}$. Then there is a mean of length $\ell_3 - 2\rho$ (that is, exactly ρ size-1 blocks of s_3 are misaligned). Clearly, any longer condensed string has a larger F -value and every shorter condensed string has at least the same F -value. ◀

We strongly conjecture that similar but more technical arguments can be used to obtain a linear-time algorithm for three arbitrary input strings. For more than three strings, however, it is not clear how to achieve linear time, since the mean length cannot be greedily determined.

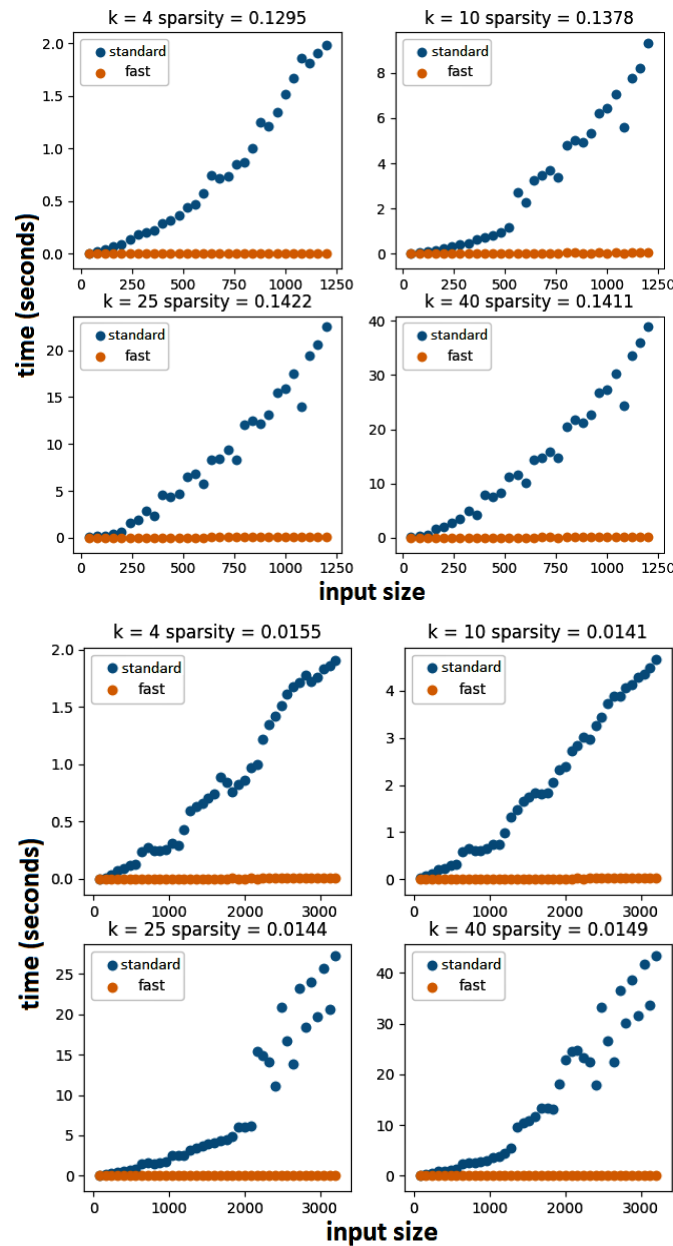
6 Empirical Evaluation

We conducted some experiments to empirically evaluate our algorithms and to observe structural characteristics of binary means. In Section 6.1 we compare the running times of our $O(k(n + m(m - \mu)))$ -time algorithm (Theorem 8 (ii)) with the standard $O(kn^2)$ -time dynamic programming approach [2] described in the beginning of Section 4. We implemented both algorithms in Python.² Note that we did not test the $O(kn^{1.87})$ -time algorithm since it uses another blackbox algorithm (which has not been implemented so far) in order to solve MSS. However, we believe that it is anyway slower in practice. In Section 6.2, we empirically investigate structural properties of binary condensed means such as the length and the starting symbol (note that these two characteristics completely define the mean). All computations have been done on an Intel i7 QuadCore (4.0 GHz).

For our experiments we used the CASAS human activity datasets³ [10] as well as some randomly generated data. The data in the CASAS datasets are generated from sensors which detect (timestamped) changes in the environment (for example, a door being opened/closed) and have previously been used in the context of binary dtw computation [16]. We used the datasets HH101–HH130 and sampled from them to obtain input strings of different lengths and sparsities (for a binary string s , we define the *sparsity* as $|\tilde{s}|/|s|$). For the random data, the sparsity value was used as the probability that the next symbol in the string will be different from the last one (hence, the actual sparsities are not necessarily exactly the sparsities given but very close to those).

² Source code available at <https://www.akt.tu-berlin.de/menue/software/>.

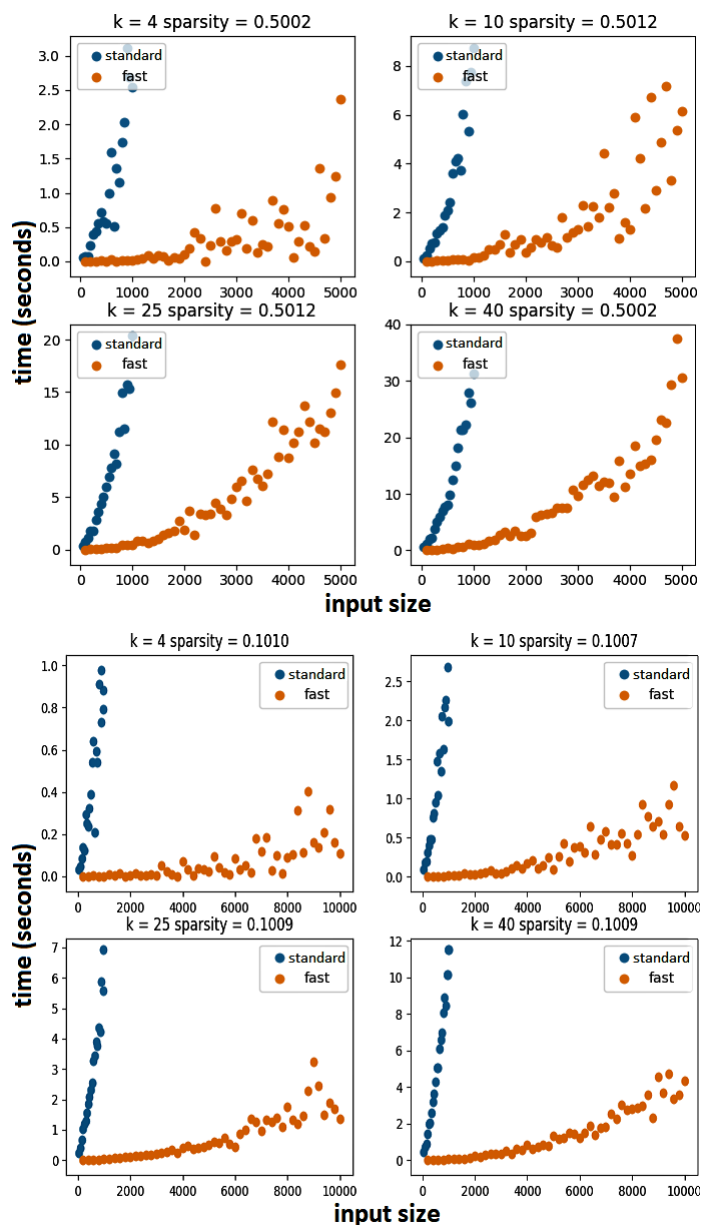
³ Available at <http://casas.wsu.edu/datasets/>.



■ **Figure 2** Running times of the standard and the fast algorithm on sparse data (sensor D002 in dataset HH101) in 10-minute intervals (top) and 1-minute intervals (bottom).

6.1 Running Time Comparison

To examine the speedup provided by our algorithm, we compare it with the standard $O(kn^2)$ -time dynamic programming algorithm on (very) sparse real-world data (sparsity ≈ 0.1 and ≈ 0.01) and on sparse (sparsity ≈ 0.1) and dense (sparsity ≈ 0.5) random data, both for various values of k . Figure 2 shows the running times on real-world data. For sparsity ≈ 0.1 , our algorithm is around 250 times faster than the standard algorithm and for sparsity ≈ 0.01 it is around 350 times faster. Figure 3 shows the running times of the algorithms on larger

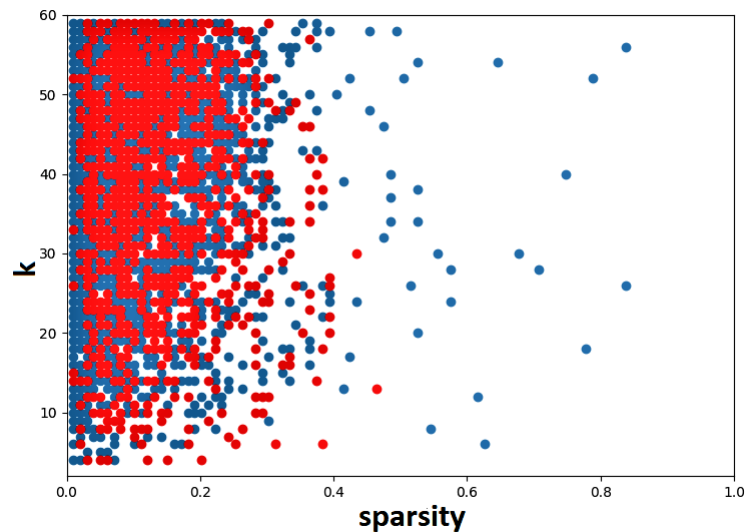


■ **Figure 3** Running times of the standard algorithm for $n \leq 1000$ and of the fast algorithm for $n \leq 5000$ on dense random data (top) and for $n \leq 10,000$ on sparse random data (bottom).

random data. For sparsity ≈ 0.1 , our algorithm is still twice as fast for $n = 10,000$ as the standard algorithm for $n = 1000$. These results clearly show that our algorithm is valuable in practice.

6.2 Structural Observations

We also studied the typical shape of binary condensed means. The questions of interest are “What is the typical length of a condensed mean?” and “What is the first symbol of a condensed mean?”. Since the answers to these two questions completely determine a condensed mean, we investigated whether they can be easily determined from the inputs.



■ **Figure 4** Difference between median condensed length and calculated mean length depending on sparsity and number of input strings. For every pair $(\sigma, k) \in \{0.01, \dots, 1.0\} \times [60]$, we calculated one mean for k strings with sparsity σ . No dot means that the median condensed length and the mean length did not differ by more than one. A blue (dark gray) dot means they differed slightly (difference between two and four) and a red (light gray) dot means they differed by at least five.

To answer the question regarding the mean length, we tested how much the actual mean length differs from the median condensed length. Recall that by Lemma 7 we know that every condensed mean has length at least $\mu - 2$, where μ is the median input condensation length. We call this lower bound the median condensed length. We used our algorithm (Theorem 8 (ii)) to compute condensed means on random data with sparsities $0.01, \dots, 1.0$, $k = 1, \dots, 60$ and $n \leq 400$. Figure 4 clearly shows that on dense data (sparsity > 0.5), the difference between the mean length and the median condensed length is rarely more than one. This can be explained by the fact that for dense strings all blocks are usually small such that there is no gain in making the mean longer than the median condensed length. We remark that a difference of one appears quite often which might be caused by different starting or ending symbols of the inputs. In general, for dense data the mean length almost always is at most the median condensed length plus one, whereas for sparser data the mean can become longer. As regards the dependency of the mean length on the number k of inputs, it can be observed that, for sparse data (sparsity < 0.5), the mean length differs even more for larger k . This may be possible because more input strings increase the probability that there is one input string with long condensation length and large block sizes. For dense inputs, there seems to be no real dependence on k .

To answer the question regarding the first symbol of a mean, we tested on random data with different k values and different sparsities ($n \leq 500$), how the starting symbol of the mean depends on the starting symbols or blocks of the input strings. First, we tested how often the starting symbol of the mean equals the majority of starting symbols of the input strings (see Table 1). Then, we also summed up the lengths of all starting 1-blocks and the lengths of all starting 0-blocks and checked how often the mean starts with the symbol corresponding to the larger of those two sums (see Table 2). Overall, the starting symbol of the mean matches the majority of starting symbols or blocks of the input strings in most

■ **Table 1** Frequency (over 1000 runs) of the first symbol of the mean also being the first symbol in the majority of input strings.

k /sparsity	0.05	0.1	0.2	0.5	0.8	1
5	76%	79%	82%	82%	82%	80%
15	75%	81%	82%	83%	85%	85%
40	82%	84%	88%	87%	91%	97%

■ **Table 2** Frequency (over 1000 runs) of the first symbol of the mean also being the majority of symbols throughout the first blocks of input strings.

k /sparsity	0.05	0.1	0.2	0.5	0.8	1
5	69%	73%	75%	83%	85%	80%
15	67%	73%	75%	82%	88%	85%
40	66%	70%	74%	81%	91%	97%

cases (≈ 70 – 90% , increasing with higher sparsity). For low sparsities, however, taking the length of starting blocks into account seems to yield less matches. This might be due to large outlier starting blocks (note that this effect is even worse for larger k).

To sum up the above empirical observations, we conclude that a condensed binary mean typically has a length close to the median condensed length and starts with the majority symbol among the starting symbols in the inputs.

7 Conclusion

In this work we made progress in understanding and efficiently computing binary means of binary strings with respect to the dtw distance. First, we proved tight lower and upper bounds on the length of a binary (condensed) mean which we then used to obtain fast polynomial-time algorithms to compute binary means by solving a certain number problem efficiently. We also obtained linear-time algorithms for $k \leq 3$ input strings. Moreover, we empirically showed that the actual mean length is often very close to the proven lower bound.

As regards future research challenges, it would be interesting to further improve the running time with respect to the maximum input string length n . This could be achieved by finding faster algorithms for our “helper problem” MIN 1-SEPARATED SUM (MSS). Can one solve BDTW-MEAN in linear time for every constant k (that is, $f(k) \cdot n$ time for some function f)? Also, finding improved algorithms for the weighted version or the center version (see Section 4) might be of interest.

References

- 1 A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS '15)*, pages 59–78, 2015.
- 2 Markus Brill, Till Fluschnik, Vincent Froese, Brijnesh J. Jain, Rolf Niedermeier, and David Schultz. Exact mean computation in dynamic time warping spaces. *Data Mining and Knowledge Discovery*, 33(1):252–291, 2019.
- 3 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS '15)*, pages 79–97, 2015.

- 4 Kevin Buchin, Anne Driemel, and Martijn Struijs. On the hardness of computing an average curve. *CoRR*, abs/1902.08053, 2019. Preprint appeared at the *35th European Workshop on Computational Geometry (EuroCG '19)*.
- 5 Laurent Bulteau, Vincent Froese, and Rolf Niedermeier. Tight hardness results for consensus problems on circular strings and time series. *CoRR*, abs/1804.02854, 2018.
- 6 Laurent Bulteau, Falk Hüffner, Christian Komusiewicz, and Rolf Niedermeier. Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, 114, 2014.
- 7 Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. A three-string approach to the closest string problem. *Journal of Computer and System Sciences*, 78(1):164–178, 2012.
- 8 Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. Randomized fixed-parameter algorithms for the closest string problem. *Algorithmica*, 74(1):466–484, 2016.
- 9 Zhi-Zhong Chen and Lusheng Wang. Fast exact algorithms for the closest string and substring problems with application to the planted (ℓ, d) -motif model. *IEEE/ACM Transactions Computational Biology and Bioinformatics*, 8(5):1400–1410, 2011.
- 10 Diane Cook, Aaron Crandall, Brian Thomas, and Narayanan Krishnan. Casas: A smart home in a box. *Computer*, 46, 2013.
- 11 Moti Frances and Ami Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997.
- 12 Omer Gold and Micha Sharir. Dynamic time warping and geometric edit distance: Breaking the quadratic barrier. *ACM Transactions on Algorithms*, 14(4):50:1–50:17, 2018.
- 13 Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.
- 14 William Kuszmaul. Dynamic time warping in strongly subquadratic time: Algorithms for the low-distance regime and approximate evaluation. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP '19)*, pages 80:1–80:15, 2019.
- 15 Ming Li, Bin Ma, and Lusheng Wang. On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171, 2002.
- 16 A. Mueen, N. Chavoshi, N. Abu-El-Rub, H. Hamooni, and A. Minnich. AWarp: Fast warping distance for sparse time series. In *2016 IEEE 16th International Conference on Data Mining (ICDM '16)*, pages 350–359, 2016.
- 17 Naomi Nishimura and Narges Simjour. Enumerating neighbour and closest strings. In *7th International Symposium on Parameterized and Exact Computation, (IPEC '12)*, pages 252–263. Springer, 2012.
- 18 H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):43–49, 1978.
- 19 Anooshiravan Sharabiani, Houshang Darabi, Samuel Harford, Elnaz Douzali, Fazle Karim, Hereford Johnson, and Shun Chen. Asymptotic dynamic time warping calculation with utilizing value repetition. *Knowledge and Information Systems*, 57(2):359–388, 2018.
- 20 Shota Yuasa, Zhi-Zhong Chen, Bin Ma, and Lusheng Wang. Designing and implementing algorithms for the closest string problem. *Theoretical Computer Science*, 786:32–43, 2019.

Approximating Text-To-Pattern Distance via Dimensionality Reduction

Przemysław Uznański 

Institute of Computer Science, University of Wrocław, Poland
puznanski@cs.uni.wroc.pl

Abstract

Text-to-pattern distance is a fundamental problem in string matching, where given a pattern of length m and a text of length n , over an integer alphabet, we are asked to compute the distance between pattern and the text at every location. The distance function can be e.g. Hamming distance or ℓ_p distance for some parameter $p > 0$. Almost all state-of-the-art exact and approximate algorithms developed in the past ~ 40 years were using FFT as a black-box. In this work we present $\tilde{O}(n/\varepsilon^2)$ time algorithms for $(1 \pm \varepsilon)$ -approximation of ℓ_2 distances, and $\tilde{O}(n/\varepsilon^3)$ algorithm for approximation of Hamming and ℓ_1 distances, all without use of FFT. This is independent to the very recent development by Chan et al. [STOC 2020], where $\mathcal{O}(n/\varepsilon^2)$ algorithm for Hamming distances not using FFT was presented – although their algorithm is much more “combinatorial”, our techniques apply to other norms than Hamming.

2012 ACM Subject Classification Theory of computation \rightarrow Sketching and sampling; Theory of computation \rightarrow Approximation algorithms analysis

Keywords and phrases Approximate Pattern Matching, ℓ_2 Distance, ℓ_1 Distance, Hamming Distance, Approximation Algorithms, Combinatorial Algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.29

Related Version A full version of the paper is available at [30], <https://arxiv.org/abs/2002.03459>.

Funding Supported by Polish National Science Centre grant 2019/33/B/ST6/00298.

1 Introduction

Text-to-pattern distance is a generalization of a classical pattern matching by incorporating the notion of similarity (or dissimilarity) between pattern and locations of text. The problem is defined in a following way: for a particular distance function between words (interpreted as vectors), given a pattern of length m and a text of length n , we are asked to output distance between the pattern and every m -substring of the text. Taking e.g. distance to be Hamming distance, we are essentially outputting number of mismatches in a classical pattern matching question (that is, not only detecting exact matches, but also counting how far pattern is to from being located in a text, at every position). Such a formulation, for a constant-size alphabet, was first considered by Fischer and Paterson in [12]. The algorithm of [12] uses $\mathcal{O}(n \log n)$ time and in substance computes the Boolean convolution of two vectors a constant number of times. This was later extended to poly(n) size alphabets by Abrahamson in [1, 21] with $\mathcal{O}(n\sqrt{m \log m})$ run-time.

The lack of progress in Hamming text-to-pattern distance complexity sparked interest in searching for relaxations of the problem, with a hope for reaching linear (or almost linear) run-time. There are essentially two takes on this. First consists of approximation algorithms. Until very recently, the fastest known $(1 \pm \varepsilon)$ -approximation algorithm for computing the Hamming distances was by Karloff [18]. The algorithm uses random projections from an arbitrary alphabet to the binary one and Boolean convolution to solve the problem in $\mathcal{O}(\varepsilon^{-2}n \log^3 n)$



© Przemysław Uznański;
licensed under Creative Commons License CC-BY
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).
Editors: Inge Li Gørtz and Oren Weimann; Article No. 29; pp. 29:1–29:11



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

time. Later Kopelowitz and Porat [19] gave a new approximation algorithm improving the time complexity to $\mathcal{O}(\varepsilon^{-1}n \log^3 n \log \varepsilon^{-1})$, which was later significantly simplified in Kopelowitz and Porat [20], with alternative formulation by Uznański and Studený [28].

Second widely considered way of relaxing exact text-to-pattern distance is to report exactly only the values not exceeding certain threshold value k , the so-called k -mismatch problem. The very first solution to the k -mismatch problem was shown by Landau and Vishkin in [23] working in $\mathcal{O}(nk)$ time, using essentially a very combinatorial approach of taking $\mathcal{O}(1)$ time per mismatch per alignment using LCP queries. This initiated a series of improvements to the complexity, with algorithms of complexity $\mathcal{O}(n\sqrt{k \log k})$ and $\mathcal{O}((k^3 \log k + m) \cdot n/m)$ by Amir et al. in [3], later improved to $\mathcal{O}((k^2 \log k + m \text{ poly log } m) \cdot n/m)$ by Clifford et al. [8] and finally $\mathcal{O}((m \log^2 m \log |\Sigma| + k\sqrt{m \log m}) \cdot n/m)$ by Gawrychowski and Uznański [13] (and following poly-log improvements by Chan et al. in [5]).

Moving beyond counting mismatches, we consider ℓ_1 distances, where we consider text and pattern over integer alphabet, and distance is sum of position-wise absolute differences. Using techniques similar to Hamming distances, the $\mathcal{O}(n\sqrt{m \log m})$ complexity algorithms were developed by Clifford et al. and Amir et al. in [6, 4] for reporting all ℓ_1 distances. It is a major open problem whether near-linear time algorithm, or even $\mathcal{O}(n^{3/2-\varepsilon})$ time algorithms, are possible for such problems. A conditional lower bound was shown by Clifford in [7], via a reduction from matrix multiplication. This means that existence of combinatorial algorithm with $\mathcal{O}(n^{3/2-\varepsilon})$ run-time solving the problem for Hamming distances implies combinatorial algorithms for Boolean matrix multiplication with $\mathcal{O}(n^{3-\delta})$ run-time, which existence is unlikely. Looking for unconditional bounds, we can state this as a lower-bound of $\Omega(n^{\omega/2})$ for Hamming distances pattern matching, where $2 \leq \omega < 2.373$ is the matrix multiplication exponent. Later, complexity of pattern matching under Hamming distance and under ℓ_1 distance was proven to be identical (up to poly-logarithmic terms), see Labib et al. and Lipsky et al. [22, 24].

Once again, existence of such lower-bound spurs interest in approximation algorithm for ℓ_1 distances. Lipsky and Porat [25] gave a deterministic algorithm with a run time of $\mathcal{O}(\frac{n}{\varepsilon^2} \log m \log U)$, while later Gawrychowski and Uznański [13] have improved the complexity to a (randomized) $\mathcal{O}(\frac{n}{\varepsilon} \log^2 n \log m \log U)$, where U is the maximal integer value on the input. Later [28] has shown that such complexity is in fact achievable (up to poly-log factors) with a deterministic solution.

Considering other norms, we mention following results. First, that for any $p > 0$ there is ℓ_p distance $(1 \pm \varepsilon)$ -approximated algorithm running in $\tilde{\mathcal{O}}(n/\varepsilon)$ time by [28]. More importantly, for specific case of $p = 2$ (or more generally, constant, positive even integer values of p) the exact problem reduces to computation of convolution, as observed by [25].

Text-to-pattern distance via convolution

Consider the case of computing ℓ_2 distances. We are computing output array $O[1 .. n - m + 1]$ such that $O[i] = \sum_j (T[i + j] - P[j])^2$. However, this is equivalent to computing, for every i simultaneously, the value of $\sum_j T[i + j]^2 + \sum_j P[j]^2 - 2 \sum_j T[i + j]P[j]$. While the terms $\sum_j T[i + j]^2$ and $\sum_j P[j]^2$ can be easily precomputed in $\mathcal{O}(n)$ time, we observe (following [25]) that $\sum_j T[i + j]P[j]$ is essentially a convolution. Indeed, let P^R denote reverse string to P . Then

$$\sum_j T[i + j]P[j] = \sum_j T[i + j]P^R[m + 1 - j] = \sum_{j+k=m+1+i} T[j]P[k] = (T \circ P^R)[m + 1 + i].$$

Since $T \circ P^R$ can be computed efficiently this provides a very strong tool in constructing text-to-pattern distance algorithms. Almost all of the discussed results use convolution as a black-box. For example, by appropriate binary encoding we can compute using a single convolution the number of Hamming mismatches generated by a single letter $c \in \Sigma$, which is a crucial observation leading to computation of exact Hamming distances in $\mathcal{O}(n\sqrt{n \log n})$ time. Other results rely on projecting large alphabets into smaller ones, e.g. [18, 20, 28].

Convolution over integers is computed by FFT in $\mathcal{O}(n \log n)$ time. This requires actual embedding of integers into field, e.g. \mathbb{F}_p or \mathbb{C} . This comes at a cost, if e.g. we were to consider text-to-pattern distance over (non-integer) alphabets that admit only field operations, e.g. matrices or geometric points. Convolution can be computed using “simpler” set of operations, that is just with ring operations in e.g. \mathbb{Z}_p using Toom-Cook multiplication [29], which is a generalization of famous divide-and-conquer Karatsuba’s algorithm [17]. This however comes at a cost, with Toom-Cook algorithm taking $\mathcal{O}(n2^{\sqrt{2 \log n}} \log n)$ time, and increased complexity of the algorithm.

Computing convolution comes with another string attached – it is inefficient to compute/sketch in the streaming setting. All of the efficient streaming text-to-pattern distance algorithms [5, 8, 9, 10, 26, 14, 27] use some form of sketching and are actually avoiding convolution computation. The reason for this is that convolution does not admit efficient sketching schemes other than with additive error, that is any algorithm based on convolution is supposed to make the same error of estimation in small and large distance regime.

Our results

We present approximation algorithm for computing the ℓ_2 text-to-pattern distance in $\tilde{\mathcal{O}}(n/\varepsilon^2)$ time, where $\tilde{\mathcal{O}}$ hides poly $\log n$ terms. Our algorithm is convolution-avoiding, and in fact it uses mostly additions and subtractions in its core part (some non-ring operations are necessary for output-scaling and hashing). We thus claim our algorithm to be more “combinatorial”, in the sense that it does not rely on field embedding and FFT computation. Our algorithm is also first non-trivial algorithm for text-to-pattern distance computation with other norms (than Hamming, which was presented recently in [5]).

► **Theorem 1.** *Text-to-pattern ℓ_2 distances can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^3 n}{\varepsilon^2})$ time.*

This mirrors the recent development of [5] where a combinatorial algorithm for Hamming distances was presented with $\mathcal{O}(n/\varepsilon^2)$ run-time. However, our techniques are general enough so that we can construct algorithm for ℓ_1 norm (and Hamming), however with $\tilde{\mathcal{O}}(n/\varepsilon^3)$ run-time.

► **Theorem 2.** *Text-to-pattern Hamming distances can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^4 n}{\varepsilon^3})$ time.*

► **Theorem 3.** *Text-to-pattern ℓ_1 distances over alphabet $[u]$ for some constant $u = \text{poly}(n)$ can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^2 n (\log^2 n + \log^4 u)}{\varepsilon^3})$ time.*

We present two novel techniques, to our knowledge never used previously in this setting. First, we show that a “mild” dimensionality reduction (linear map reducing from dimension $2d$ to d , while preserving ℓ_2 norm) can be used to repeatedly compress word, and produce

29:4 Approximating Text-To-Pattern Distance via Dimensionality Reduction

sketches for its every m -subword. Second, we show an approximate embedding of ℓ_1 space into ℓ_2^2 , that can be efficiently computed. We believe our techniques are of independent interest, both to stringology and general algorithmic communities.

2 Definitions and preliminaries

Distance between strings

Let $X = x_1x_2 \dots x_n$ and $Y = y_1y_2 \dots y_n$ be two strings. We define their ℓ_2 distance as

$$\|X - Y\| = \left(\sum_i |x_i - y_i|^2 \right)^{1/2}.$$

More generally, for any $p > 0$, we define their ℓ_p distance as

$$\|X - Y\|_p = \left(\sum_i |x_i - y_i|^p \right)^{1/p}.$$

Particularly, the ℓ_1 distance is known as the *Manhattan distance*. By a slight abuse of notation, we define the ℓ_0 (Hamming distance) to be

$$\|X - Y\|_0 = \sum_i |x_i - y_i|^0 = |\{i : x_i \neq y_i\}|,$$

where $x^0 = 1$ when $x \neq 0$ and $0^0 = 0$.

Text-to-pattern distance

For text $T = t_1t_2 \dots t_n$ and pattern $P = p_1p_2 \dots p_m$, the text-to-pattern d -distance is defined as an array S_d such that, for every i , $S_d[i] = d(T[i+1 .. i+m], P)$. Thus, for ℓ_p distance $S_{\ell_p}[i] = \left(\sum_{j=1}^m |t_{i+j} - p_j|^p \right)^{1/p}$, while for Hamming distance $S_{\text{HAM}}[i] = |\{j : t_{i+j} \neq p_j\}|$. Then $(1 \pm \varepsilon)$ -approximated distance is defined as an array S_ε such that, for every i , $(1 - \varepsilon) \cdot S_d[i] \leq S_\varepsilon[i] \leq (1 + \varepsilon) \cdot S_d[i]$.

3 Sketching via dimensionality reduction

Sketching is a tool in algorithm design, where a large object is summarized succinctly, so that some particular property is approximately preserved and some predefined operations/queries are still supported. Our interest lies on sketches that preserve ℓ_2 distances, for which we use the standard tools from dimensionality reduction.

► **Theorem 4** (Johnson-Lindenstrauss [15]). *Let $P \subseteq \mathbb{R}^m$ be of size m . Then for some $d = \mathcal{O}\left(\frac{\log m}{\varepsilon^2}\right)$ there is linear map $A \in \mathbb{R}^{d \times m}$ such that*

$$\forall_{x,y \in P} \|Ax - Ay\| = (1 \pm \varepsilon)\|x - y\|.$$

A map that preserves ℓ_2 distances is useful. Our goal is to construct a linear map such that we can apply the map to P and to every m -substring of T simultaneously and computationally efficiently. For this, we need to actually use constructive version of Johnson-Lindenstrauss lemma.

► **Theorem 5** (Achlioptas [2]). Consider a probability distribution \mathcal{D} over matrices $\mathbb{R}^{m \times d}$ defined as follow so that each matrix entry is either -1 or 1 independently and uniformly at random. Then for any $x \in \mathbb{R}^m$ there is

$$\Pr_{A \sim \mathcal{D}} \left(\frac{1}{\sqrt{d}} \|Ax\| = (1 \pm \varepsilon) \|x\| \right) \geq 1 - \delta$$

if only $d = \mathcal{O}\left(\frac{\log \delta^{-1}}{\varepsilon^2}\right)$ is large enough.

Computing such dimension-reduction naively takes $\mathcal{O}(md)$ time. However better constructions are possible.

► **Theorem 6** (Sparse JL, c.f. [11, 16]). There is probability distribution \mathcal{S} over matrices of dimension $d \times m$ with elements from $\{-1, 0, 1\}$, for large enough $d = \mathcal{O}\left(\frac{\log \delta^{-1}}{\varepsilon^2}\right)$, such that each column has only $s = \mathcal{O}(d\varepsilon)$ non-zero elements and for any vector $x \in \mathbb{R}^m$ there is

$$\Pr_{A \sim \mathcal{S}} \left(\frac{1}{\sqrt{s}} \|Ax\| = (1 \pm \varepsilon) \|x\| \right) \geq 1 - \delta.$$

Such matrices can be easily drawn from the distribution by selecting the s positions in each column independently at random and then filling them uniformly at random with $\{-1, 1\}$. The advantage of this is that single dimensionality reduction operation is computed in $\mathcal{O}(sm)$ time which is ε^{-1} factor faster than for dense matrices.

We now state the take-away from this section, which is our main technical tool to be used in the following.

► **Corollary 7.** For $d = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ large enough there is a probability distribution \mathcal{F} of linear maps $\varphi : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^d$ such that:

1. $\varphi(x, y) = A_0x + A_1y$ can be evaluated in $\mathcal{O}(d^2\varepsilon) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^3}\right)$ time,
2. $\Pr_{\varphi \sim \mathcal{F}} (\|\varphi(x, y)\|^2 = (1 \pm \varepsilon)(\|x\|^2 + \|y\|^2)) = 1 - n^{-\Omega(1)}$,
3. both A_0 and A_1 are $\{-\frac{1}{\sqrt{s}}, 0, \frac{1}{\sqrt{s}}\}$ -matrices where $s = \mathcal{O}(d\varepsilon)$ is the sparsity of each column of A_0 and A_1 .

4 Algorithm for ℓ_2 distances.

We first use Corollary 7 to construct dimensionality reduction with guarantees similar to Johnson-Lindenstrauss (reducing dimension n to dimension $\tilde{\mathcal{O}}(\varepsilon^{-2})$). In the following we assume that $d = \mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ is large enough. We show a procedure which assumes that m is divisible by d , and denote $s = \frac{m}{d}$. We assume s is a power of two, and if the case is otherwise, we can always pad input with enough zeroes at the end (we can do this, since extra zeroes have no effect on the output of linear map). We also denote $k = \log_2 s$.

We then have the following

► **Theorem 8.** Given input $x \in \mathbb{R}^m$, and $\varepsilon \leq \frac{1}{k}$, procedure `SINGLESKETCH` outputs $v \in \mathbb{R}^d$ such that

$$\|v\| = (1 \pm \mathcal{O}(k\varepsilon)) \|x\|$$

with high probability, in time $\mathcal{O}\left(\frac{m \log n}{\varepsilon}\right)$. The map $x \rightarrow v$ is linear.

Proof. We first bound the stretch. Denote by

$$\alpha_i = \sum_j \|v_j^{(i)}\|^2.$$

29:6 Approximating Text-To-Pattern Distance via Dimensionality Reduction

■ **Algorithm 1** At each level i , we partition its vectors into 2^{k-i} pairs, and compress each pair using φ_i producing vectors for level $i+1$.

```

1: Input:  $x \in \mathbb{R}^m$ .
2: Output:  $v \in \mathbb{R}^d$ .
3: procedure SINGLESKETCH( $x$ )
4:   Pick  $k$  fully independent maps  $\varphi_1, \dots, \varphi_k$  as in Corollary 7.
5:   Partition input  $x = (x_1, \dots, x_m)$  into  $s$  vectors  $v_1^{(0)}, \dots, v_s^{(0)}$  where  $v_i^{(0)} \leftarrow (x_{d \cdot (i-1)+1}, \dots, x_{d \cdot i})$ .
6:   for  $i \leftarrow 1 \dots k$  do
7:     for  $j \leftarrow 1 \dots 2^{k-i}$  do
8:        $v_j^{(i)} \leftarrow \varphi_i(v_{2j-1}^{(i-1)}, v_{2j}^{(i-1)})$ 
9:   return  $v = v_1^{(k)}$ .

```

Naturally,

$$\alpha_0 = \sum_j \|v_j^{(0)}\|^2 = \sum_{j=1}^s (x_{d \cdot (j-1)+1}^2 + \dots + x_{d \cdot j}^2) = \sum_{j=1}^n x_j^2 = \|x\|^2.$$

Moreover, by Corollary 7

$$\begin{aligned} \alpha_i &= \sum_{j=1}^{2^{k-i}} \|v_j^{(i)}\|^2 = \sum_{j=1}^{2^{k-i}} (1 \pm \varepsilon) (\|v_{2j-1}^{(i-1)}\|^2 + \|v_{2j}^{(i-1)}\|^2) \\ &= (1 \pm \varepsilon) \sum_{j=1}^{2^{k-i+1}} \|v_j^{(i-1)}\|^2 = (1 \pm \varepsilon) \alpha_{i-1} \end{aligned}$$

We could apply Corollary 7 at this step since for any usage of map φ_i , its inputs are independent from actual choice of φ_i (e.g. are result of processing x and $\varphi_1, \dots, \varphi_{i-1}$). Then we have $\|v\|^2 = \alpha_k = (1 \pm \varepsilon)^k \alpha_0 = (1 \pm \varepsilon)^k \|x\|^2$. Since $\varepsilon \leq \frac{1}{k}$, the claimed bound follows.

We then observe that the map is linear, since every building step of the map is linear. The total number of times we apply one of $\varphi_1, \dots, \varphi_k$ is $\mathcal{O}(m/d)$, so the total run-time is $\mathcal{O}(\frac{m}{d} d^2 \varepsilon)$. ◀

We then extend the algorithm to a scenario where for an input word (vector) $x \in \mathbb{R}^n$ we compute the same dimensionality reduction for all m -subwords of x that start at all the positions divisible by d . In the following we assume that d divides n , and denote $t = \frac{n-m}{d} + 1$ to be the number of such m -subwords. If its not the case, input can be padded with enough zeroes at the end.

► **Theorem 9.** *Given input $x \in \mathbb{R}^n$, denote by $y_1, \dots, y_t \in \mathbb{R}^m$ vectors such that $y_i = (x_{1+(i-1)d}, \dots, x_{m+(i-1)d})$. For $\varepsilon \leq \frac{1}{k}$ procedure ALLSKETCH outputs $v_1, \dots, v_t \in \mathbb{R}^d$ such that*

$$\|v_j\| = (1 \pm \mathcal{O}(k\varepsilon)) \|y_j\|$$

with high probability, in time $\mathcal{O}(\frac{n \log^2 n}{\varepsilon})$. Moreover, the map $y_i \rightarrow v_i$ is linear and identical to map from Theorem 8.

Algorithm 2

```

1: Input:  $x \in \mathbb{R}^n$ .
2: Output:  $v_1, \dots, v_t \in \mathbb{R}^d$  for  $t = \frac{n-m}{d} + 1$ .
3: procedure ALLSKETCH( $x$ )
4:   Let  $\varphi_1, \dots, \varphi_k$  be  $k$  fully independent maps used in procedure SINGLESKETCH.
5:   Partition input  $x = (x_1, \dots, x_n)$  into  $n/d$  vectors  $v_1^{(0)}, \dots, v_{\frac{n}{d}}^{(0)}$  where  $v_i^{(0)} \leftarrow (x_{d \cdot (i-1)+1}, \dots, x_{d \cdot i})$ .
6:   for  $i \leftarrow 1 \dots k$  do
7:     for  $j \leftarrow 1 \dots (\frac{n}{d} - 2^i + 1)$  do
8:        $v_j^{(i)} \leftarrow \varphi_i(v_j^{(i-1)}, v_{j+2^{i-1}}^{(i-1)})$ 
9:   return  $v_1^{(k)}, \dots, v_t^{(k)}$ .

```

Proof. The proof follows from inductive observation that $\|v_j^{(i)}\|^2 = (1 \pm \varepsilon)^i (\|v_j^{(0)}\|^2 + \dots + \|v_{j+2^{i-1}}^{(0)}\|^2)$, which results in

$$\begin{aligned}
\|v_j\|^2 &= (1 \pm \varepsilon)^k \sum_{i=1}^s \|v_{j+i}^{(0)}\|^2 \\
&= (1 \pm \varepsilon)^k \sum_{i=1}^m \|x_{i+(j-1)d}\|^2 \\
&= (1 \pm \varepsilon)^k \|y_j\|^2.
\end{aligned}$$

The rest of the proof follows reasoning from Theorem 8. ◀

► **Theorem 1.** *Text-to-pattern ℓ_2 distances can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^3 n}{\varepsilon^2})$ time.*

Proof. First, we note that for simplicity we compute $(\ell_2)^2$ distances since they are additive when taken under concatenation of inputs (unlike ℓ_2), that is $\|x \circ y - u \circ v\|^2 = \|x - u\|^2 + \|y - v\|^2$ for equal length x, u and equal length y, v .

We then assume w.l.o.g. that n is divisible by d . We then observe that contribution of any fragment of pattern to distance at every text location can be computed naively in $\mathcal{O}(c \cdot n)$ time where c is fragment length. We are thus safe to discard any suffix of pattern of length $\mathcal{O}(d)$ as this time is absorbed in total computation time. So we fix $h = \mathcal{O}(\log n / \varepsilon)$ and assume w.l.o.g. that $m' = m - 2h$ is divisible by d .

We denote by $\varepsilon' = \Omega(\varepsilon / \log n)$ such value that guarantees $(1 \pm \varepsilon)$ -approximation in Theorem 8 and Theorem 9. First, assume for simplicity that $\frac{m'}{d}$ is a power of two. We then consider P_0, \dots, P_h , the $(h+1)$ distinct m' -substrings of P , and for each we run procedure SINGLESKETCH on each of them, so by Theorem 8 we compute their sketches in total $\mathcal{O}(\frac{m \log n}{\varepsilon'} h)$ time. Similarly, for text T we run ALLSKETCH $\frac{d}{h}$ times to compute sketches of all m' -substrings of T starting at positions $1, h+1, 2h+1, \dots$. By Theorem 9 this takes $\mathcal{O}(\frac{n \log^2 n}{\varepsilon'} \cdot \frac{d}{h})$ time. Both steps take thus $\mathcal{O}(\frac{n \log^3 n}{\varepsilon^2})$ time, and maps used to compute sketches in both steps are linear.

We now observe that for any starting position t , the substring $T[t \dots (t + m' - 1)]$ can be partitioned into $T_1 = T[t \dots t_1]$, $T_2 = T[t_1 + 1 \dots t_2]$ and $T_3 = T[t_2 + 1 \dots (t + m' - 1)]$, where length of T_1 and T_3 is at most $2h$, length of T_2 is m' and t_1 and t_2 are multiples of h . We then compute the distances between corresponding fragments of T and P as follows (where

we consider corresponding partitioning of P into P_1, P_2 and P_3): computing $\|T_1 - P_1\|^2$ and $\|T_3 - P_3\|^2$ takes $\mathcal{O}(h)$ each ($\mathcal{O}(nh)$ in total for all alignments), while $(1 \pm \varepsilon)$ approximating $\|T_2 - P_2\|^2$ follows from pre-computed sketches.

We now discuss the general case when $\frac{m'}{d}$ is not a power of two. However we then observe that m' can be represented as $m' = d(2^{i_1} + \dots + 2^{i_s})$ where $s \leq \log n$. And so the necessary computation require actually querying s different sketches for fragments of length $d \cdot 2^{i_\ell}$. To avoid unnecessary $\mathcal{O}(\log n)$ overhead in time (and repeating running the preprocessing steps $\log n$ times for many various lengths of fragments) we observe that all the necessary sketches are already computed as temporary values in procedures `SINGLESKETCH` and `ALLSKETCH`. ◀

5 Hamming and ℓ_1 distances.

We now briefly discuss how to use our framework for approximating other norms. We first recall the classical result by [18].

► **Lemma 10** ([18]). *Let $d = \mathcal{O}(\log n/\varepsilon^2)$ be large enough. Consider $\mu : \Sigma \rightarrow \{0, 1\}^d$ where each $\varphi(c)$ is chosen uniformly and independently at random. Then*

$$\forall_{c_1 \neq c_2} \|\mu(c_1) - \mu(c_2)\|^2 = (1 \pm \varepsilon) \cdot \frac{d}{2}$$

with high probability.

We note that we assumed that the dimension $\mathcal{O}(\log n/\varepsilon^2)$ of map μ matches value of $d = \mathcal{O}(\log n/\varepsilon^2)$ from dimensionality-reductions in previous section. This can be easily ensured w.l.o.g. as we can always either pad with extra zeroes each image of μ mapping, or add extra null coordinates to dimensionality reduction. Extending the mapping from letters to words, that is for $w = c_1 \dots c_k \in \Sigma^*$ denote $\mu(w) = \mu(c_1) \dots \mu(c_k)$, we have a corollary:

► **Corollary 11.** *For μ as in Lemma 10, and any two words $u, v \in \Sigma^n$, there is*

$$\|\mu(u) - \mu(v)\|^2 = (1 \pm \varepsilon) \cdot \frac{d}{2} \|u - v\|_0$$

with high probability.

This allows us to estimate Hamming distance between words from ℓ_2^2 distance between the respective embeddings, which are of length $\mathcal{O}(\frac{n \log n}{\varepsilon^2})$.

► **Theorem 2.** *Text-to-pattern Hamming distances can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^4 n}{\varepsilon^3})$ time.*

Proof. By Corollary 11 it is enough to estimate the ℓ_2^2 text-to-pattern distance between embedded words $\mu(P)$ and $\mu(T)$ at starting positions $1, d + 1, 2d + 1, \dots$. We use procedure `SINGLESKETCH` to compute sketch of $\mu(P)$, and procedure `ALLSKETCH` to compute sketch of every (dm) -substring of $\varphi(T)$ starting at positions $1, d + 1, 2d + 1, \dots$. Former takes $\mathcal{O}(\frac{n \log^2 n}{\varepsilon^2 \varepsilon'})$ time, and latter takes $\mathcal{O}(\frac{n \log^3 n}{\varepsilon^2 \varepsilon'})$ time, where we set $\varepsilon' = \Omega(\varepsilon/k)$ so that error from sketching accumulates to $1 \pm \mathcal{O}(\varepsilon)$ in total. All in all this gives $\mathcal{O}(\frac{n \log^4 n}{\varepsilon^3})$ time algorithm. ◀

We now proceed to ℓ_1 distances. Our goal is to construct a mapping $f : [u] \rightarrow \{0, 1\}^d$ that embeds ℓ_1 into ℓ_2^2 approximately. That is, we require $\forall_{a, b \in [u]} |a - b| \sim (1 \pm \varepsilon) \|f(a) - f(b)\|^2$ where \sim hides constant factors. The existence of such map can be easily shown: (i) Take exact

map $f_1 : [u] \rightarrow \{0, 1\}^u$ defined as $f_1(a) = 1^a 0^{u-a}$, (ii) Take any ℓ_2 dimensionality-reduction map $f_2 : \{0, 1\}^u \rightarrow \{0, 1\}^d$, (iii) set $f = f_2 \circ f_1$. However, our goal is to compute such f faster than in time proportional to universe size u . We do it by running first a preprocessing phase, and then a fast computation procedure.

■ **Algorithm 3**

```

1: procedure PREPROCESS( $u$ )
2:   Pick  $\log(u/d)$  fully independent maps  $\varphi'_1, \dots, \varphi'_{\log(u/d)}$  as in Corollary 7.
3:    $s_0 \leftarrow (1, 1, \dots, 1) \in \mathbb{R}^d$ .
4:   for  $i \leftarrow 1 \dots \log(u/d)$  do
5:      $s_i \leftarrow \varphi'_i(s_{i-1}, s_{i-1})$ 
6: procedure PROJECT( $x \in [u], c$ )
7:   if  $c = 0$  then
8:     return  $(\underbrace{1, 1, \dots, 1}_x, \underbrace{0, \dots, 0}_{d-x})$ 
9:   else if  $x < \frac{1}{2}d \cdot 2^c$  then
10:    return  $\varphi'_c(\text{PROJECT}(x, c-1), (0, \dots, 0))$ 
11:   else
12:    return  $\varphi'_c(s_{c-1}, \text{PROJECT}(x - \frac{1}{2}d \cdot 2^c, c-1))$ 

```

► **Lemma 12.** $\psi : x \rightarrow \text{PROJECT}(x, \log(u/d))$ represents a linear map $[u] \rightarrow \mathbb{R}^d$ that embeds approximately ℓ_1 to ℓ_2^2 , that is

$$|x - y| = (1 \pm \mathcal{O}(\varepsilon \log u)) \|\psi(x) - \psi(y)\|^2$$

with high probability. Moreover, ψ takes $\mathcal{O}(\frac{\log^2 n \log u}{\varepsilon^3})$ time to evaluate.

Proof. Let us define informally $\pi_i = \varphi'_i(\varphi'_{i-1}(\dots, \dots), \varphi'_{i-1}(\dots, \dots))$ to be unfolded version of φ' , that is a linear map $\mathbb{R}^{d \cdot 2^i} \rightarrow \mathbb{R}^d$. Formally $\pi_0 = \text{id}$, and for $x = (x_1, \dots, x_{d \cdot 2^i})$, defining

$$\pi_i((x_1, \dots, x_{d \cdot 2^i})) = \varphi'_i(\pi_{i-1}(x_{\text{left}}), \pi_{i-1}(x_{\text{right}})),$$

where $x_{\text{left}} = (x_1, \dots, x_{d \cdot 2^{i-1}})$, $x_{\text{right}} = (x_{d \cdot 2^{i-1} + 1}, \dots, x_{d \cdot 2^i})$.

We now observe that $s_i = \pi_i(\underbrace{(1, \dots, 1)}_{2^i d})$ and then (by induction)

$$\text{PROJECT}(x, i) = \pi_i(\underbrace{(1, 1, \dots, 1)}_x, \underbrace{(0, \dots, 0)}_{2^i d - x}).$$

Inductively, each iteration $1, \dots, \log(u/d)$ results in extra multiplicative $(1 \pm \varepsilon)$ distortion. Computation time is dominated by applications of $\varphi'_1, \dots, \varphi'_{\log(u/d)}$, both in the preprocessing time and the evaluation time. Since each linear map φ'_i is applied in time $\mathcal{O}(\frac{\log^2 n}{\varepsilon^3})$, the time complexity bound follows. ◀

► **Theorem 3.** Text-to-pattern ℓ_1 distances over alphabet $[u]$ for some constant $u = \text{poly}(n)$ can be approximated by an algorithm using only basic arithmetic operations and not using convolution. The approximation is $1 \pm \varepsilon$ multiplicative with high probability, computed in $\mathcal{O}(\frac{n \log^2 n (\log^2 n + \log^4 u)}{\varepsilon^3})$ time.

Proof. We use Lemma 12 to reduce the problem to estimating ℓ_2^2 text-to-pattern distance between $\psi(P)$ and $\psi(T)$ at starting positions $1, d+1, 2d+1, \dots$. We use procedure SINGLESKETCH to compute sketch of $\mu(P)$, and procedure ALLSKETCH to compute sketch of every (dm) -substring of $\varphi(T)$ starting at selected positions. Denote by $\varepsilon' = \Omega(\varepsilon/k)$ the stretch constant in procedures SINGLESKETCH and ALLSKETCH, and by $\varepsilon'' = \Omega(\varepsilon/\log u)$ the stretch constant in procedures PROJECT and PREPROCESS. The total run-time of ALLSKETCH is then $\mathcal{O}(\frac{n \log^3 n}{\varepsilon^2 \varepsilon'}) = \mathcal{O}(\frac{n \log^4 n}{\varepsilon^3})$ and total run-time of computing $\psi(T)$ and $\psi(P)$ is $\mathcal{O}(\frac{n \log^2 n \log u}{(\varepsilon'')^3}) = \mathcal{O}(\frac{n \log^2 n \log^4 u}{\varepsilon^3})$. ◀

References

- 1 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.
- 2 Dimitris Achlioptas. Database-friendly random projections: Johnson–Lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003. doi:10.1016/S0022-0000(03)00025-4.
- 3 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.
- 4 Amihood Amir, Ohad Lipsky, Ely Porat, and Julia Umanski. Approximate matching in the L_1 metric. In *Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings*, pages 91–103, 2005. doi:10.1007/11496656_9.
- 5 Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern hamming distances. In *STOC*, 2020 (to appear).
- 6 Peter Clifford, Raphaël Clifford, and Costas S. Iliopoulos. Faster algorithms for δ, γ -matching and related problems. In *CPM*, pages 68–78, 2005. doi:10.1007/11496656_7.
- 7 Raphaël Clifford. Matrix multiplication and pattern matching under Hamming norm. <http://www.cs.bris.ac.uk/Research/Algorithms/events/BAD09/BAD09/Talks/BAD09-Hammingnotes.pdf>. Retrieved March 2017.
- 8 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k -mismatch problem revisited. In *SODA*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.
- 9 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k -mismatch problem. In *SODA*, pages 1106–1125, 2019. doi:10.1137/1.9781611975482.68.
- 10 Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming distance in a stream. In *ICALP*, pages 20:1–20:14, 2016. doi:10.4230/LIPIcs.ICALP.2016.20.
- 11 Michael B. Cohen, T. S. Jayram, and Jelani Nelson. Simple analyses of the sparse johnson-lindenstrauss transform. In *1st Symposium on Simplicity in Algorithms, SOSA 2018, January 7-10, 2018, New Orleans, LA, USA*, pages 15:1–15:9, 2018. doi:10.4230/OASIcs.SOSA.2018.15.
- 12 M. J. Fischer and M. S. Paterson. String-matching and other products. Technical report, Massachusetts Institute of Technology, 1974.
- 13 Paweł Gawrychowski and Przemysław Uznański. Towards unified approximate pattern matching for Hamming and L_1 distance. In *ICALP*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.
- 14 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *ICALP*, pages 65:1–65:16, 2018. doi:10.4230/LIPIcs.ICALP.2018.65.
- 15 William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space, 1984.
- 16 Daniel M. Kane and Jelani Nelson. Sparser johnson-lindenstrauss transforms. *J. ACM*, 61(1):4:1–4:23, 2014. doi:10.1145/2559902.
- 17 Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.
- 18 Howard J. Karloff. Fast algorithms for approximately counting mismatches. *Inf. Process. Lett.*, 48(2):53–60, 1993. doi:10.1016/0020-0190(93)90177-B.

- 19 Tsvi Kopelowitz and Ely Porat. Breaking the variance: Approximating the hamming distance in $1/\epsilon$ time per alignment. In *FOCS*, pages 601–613, 2015. doi:10.1109/FOCS.2015.43.
- 20 Tsvi Kopelowitz and Ely Porat. A simple algorithm for approximating the text-to-pattern hamming distance. In *SOSA@SODA*, pages 10:1–10:5, 2018. doi:10.4230/OASIcs.SOSA.2018.10.
- 21 S. R. Kosaraju. Efficient string matching. Manuscript, 1987.
- 22 Karim Labib, Przemysław Uznański, and Daniel Wolleb-Graf. Hamming distance completeness. In *CPM*, pages 14:1–14:17, 2019. doi:10.4230/LIPIcs.CPM.2019.14.
- 23 Gad M. Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theor. Comput. Sci.*, 43:239–249, 1986. doi:10.1016/0304-3975(86)90178-7.
- 24 Ohad Lipsky and Ely Porat. L_1 pattern matching lower bound. *Inf. Process. Lett.*, 105(4):141–143, 2008. doi:10.1016/j.ipl.2007.08.011.
- 25 Ohad Lipsky and Ely Porat. Approximate pattern matching with the L_1 , L_2 and L_∞ metrics. *Algorithmica*, 60(2):335–348, 2011. doi:10.1007/s00453-009-9345-9.
- 26 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *FOCS*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.
- 27 Tatiana Starikovskaya, Michal Svagerka, and Przemysław Uznański. L_p pattern matching in a stream. *CoRR*, abs/1907.04405, 2019. arXiv:1907.04405.
- 28 Jan Studený and Przemysław Uznański. Approximating approximate pattern matching. In *CPM*, volume 128, pages 15:1–15:13, 2019. doi:10.4230/LIPIcs.CPM.2019.15.
- 29 AL Toom. The complexity of a scheme of functional elements simulating the multiplication of integers. In *Doklady Akademii Nauk*, volume 150(3), pages 496–498. Russian Academy of Sciences, 1963.
- 30 Przemysław Uznański. Approximating text-to-pattern distance via dimensionality reduction. *CoRR*, abs/2002.03459, 2020. arXiv:2002.03459.

