

# 29th Annual Symposium on Combinatorial Pattern Matching

CPM 2018, July 2–4, 2018, Qingdao, China

Edited by

Gonzalo Navarro

David Sankoff

Binhai Zhu



### *Editors*

Gonzalo Navarro  
Department of Computer Science  
University of Chile, Chile  
gnavarro@dcc.uchile.cl

David Sankoff  
Department of Math and Statistics  
University of Ottawa, Canada  
sankoff@uottawa.ca

Binhai Zhu  
Gianforte School of Computing  
Montana State University, USA  
bhz@montana.edu

### *ACM Classification 2012*

Mathematics of computing → Discrete mathematics, Mathematics of computing → Information theory, Information systems → Information retrieval, Theory of computation → Design and analysis of algorithms, Applied computing → Computational biology

## **ISBN 978-3-95977-074-3**

### *Published online and open access by*

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <http://www.dagstuhl.de/dagpub/978-3-95977-074-3>.

### *Publication date*

May, 2018

### *Bibliographic information published by the Deutsche Nationalbibliothek*

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

### *License*

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): <http://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.2018.0

**ISBN 978-3-95977-074-3**

**ISSN 1868-8969**

**<http://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)
- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

**ISSN 1868-8969**

**<http://www.dagstuhl.de/lipics>**





To all algorithmic stringologists in the world



## ■ Contents

Preface	
<i>Gonzalo Navarro, David Sankoff, and Binhai Zhu</i> .....	0:vii
<b>Regular Papers</b>	
Maximal Common Subsequence Algorithms	
<i>Yoshifumi Sakai</i> .....	1:1–1:10
Order-Preserving Pattern Matching	
Indeterminate Strings	
<i>Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai</i> ....	2:1–2:15
On Undetected Redundancy in the Burrows-Wheeler Transform	
<i>Uwe Baier</i> .....	3:1–3:15
Quasi-Periodicity Under Mismatch Errors	
<i>Amihood Amir, Avivit Levy, and Ely Porat</i> .....	4:1–4:15
Fast Matching-based Approximations for Maximum Duo-Preservation String Mapping and its Weighted Variant	
<i>Brian Brubach</i> .....	5:1–5:14
Nearest constrained circular words	
<i>Guillaume Blin, Alexandre Blondin Massé, Marie Gasparoux, Sylvie Hamel, and Élise Vandomme</i> .....	6:1–6:14
Online LZ77 Parsing and Matching Statistics with RLBWTs	
<i>Hideo Bannai, Travis Gagie, and Tomohiro I</i> .....	7:1–7:12
Non-Overlapping Indexing – Cache Obliviously	
<i>Sahar Hooshmand, Paniz Abedin, M. Oğuzhan Külekci, and Sharma V. Thankachan</i>	8:1–8:9
Faster Online Elastic Degenerate String Matching	
<i>Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda</i> .....	9:1–9:10
A Simple Linear-Time Algorithm for Computing the Centroid and Canonical Form of a Plane Graph and Its Applications	
<i>Tatsuya Akutsu, Colin de la Higuera, and Takeyuki Tamura</i> .....	10:1–10:12
Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms	
<i>Amihood Amir and Itai Boneh</i> .....	11:1–11:13
Longest substring palindrome after edit	
<i>Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda</i> .....	12:1–12:14
A Succinct Four Russians Speedup for Edit Distance Computation and One-against-many Banded Alignment	
<i>Brian Brubach and Jay Ghurje</i> .....	13:1–13:12

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu



Leibniz International Proceedings in Informatics

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

Can a permutation be sorted by best short swaps? <i>Shu Zhang, Daming Zhu, Haitao Jiang, Jingjing Ma, Jiong Guo, and Haodi Feng</i>	14:1–14:12
Computing longest common square subsequences <i>Takafumi Inoue, Shunsuke Inenaga, Heikki Hyvrö, Hideo Bannai, and Masayuki Takeda</i>	15:1–15:13
Slowing Down Top Trees for Better Worst-Case Compression <i>Bartłomiej Dudek and Paweł Gawrychowski</i>	16:1–16:8
On the Maximum Colorful Arborescence Problem and Color Hierarchy Graph Structure <i>Guillaume Fertin, Julien Fradin, and Christian Komusiewicz</i>	17:1–17:15
Dualities in Tree Representations <i>Rayan Chikhi and Alexander Schönhuth</i>	18:1–18:12
Longest Lyndon Substring After Edit <i>Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda</i>	19:1–19:10
The Heaviest Induced Ancestors Problem Revisited <i>Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan</i>	20:1–20:13
Superstrings with multiplicities <i>Bastien Cazaux and Eric Rivals</i>	21:1–21:16
Linear-time algorithms for the subpath kernel <i>Kilho Shin and Taichi Ishikawa</i>	22:1–22:13
Linear-Time Algorithm for Long LCF with $k$ Mismatches <i>Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen</i>	23:1–23:16
Lyndon Factorization of Grammar Compressed Texts Revisited <i>Isamu Furuya, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda</i>	24:1–24:10

## ■ Preface

The Annual Symposium on Combinatorial Pattern Matching is the international research forum in the areas of combinatorial pattern matching, string algorithms and related applications. The objects people work on include trees, regular expressions, graphs, point sets, and sequences. The goal is to design efficient algorithms based on the properties of these structures. The problems dealt 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 29th Annual Symposium on Combinatorial Pattern Matching (CPM-2018) held on July 2-4, 2018 in Qingdao, China. CPM-2018 was run jointly with COCOON-2018. In fact, it is the first time that CPM is held in China.

The conference program included 24 contributed papers and three invited talks, held jointly with COCOON-2018. The three invited talks are by Ming Li (University of Waterloo, Canada), on “Challenges from cancer immunotherapy”; Russell Schwartz (Carnegie-Mellon University, USA), on “Reconstructing tumor evolution and progression in structurally variant cancer cells”; and Michael Segal (Ben-Gurion University of the Negev, Israel), on “Privacy aspects in data querying”.

The contributed papers were selected out of 38 submissions, giving an acceptance ratio of 63%. Each submission was reviewed by at least three Program Committee members (who might have been assisted by external reviewers, all listed below). We thank all the Program Committee members and external reviews for their hard work, on which this excellent scientific program is based.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has taken place annually since then. 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 (Canada), Pisa, Lille, New York, Palermo, Helsinki, Bad Herrenalb, Moscow, Ischia, Tel Aviv, and Warsaw. From the 3rd to the 26th meeting, all the proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, CPM proceedings have been published in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM-2016) and volume 78 (CPM-2017) respectively.

The submission and review process was carried out with EasyChair. We thank the CPM Steering Committee for all the advice and support. We thank Haitao Jiang and Daming Zhu (and their students) at Shandong University for their extensive involvement and local arrangements. Finally we would like to thank the National Natural Science Foundation of China for providing generous financial support to the conference.

Gonzalo Navarro  
David Sankoff  
Binhai Zhu





## ■ Program Committee

Gonzalo Navarro (Co-Chair)	University of Chile, Chile
David Sankoff (Co-Chair)	University of Ottawa, Canada
Binhai Zhu (Co-Chair)	Montana State University, USA
Amihood Amir	Bar Ilan University, Israel
Djamal Belazzougui	DTISI-CERIST, Algeria
Maxime Crochemore	King's College London, UK and Universite Paris-est, France
Travis Gagie	University of Helsinki, Finland
Pawel Gawrychowski	University of Wroclaw, Poland
Raffaele Giancarlo	Universita degli Studi di Palermo, Italy
Roberto Grossi	University of Pisa, Italy
Jiong Guo	Shandong University, China
Wing-Kai Hon	National Tsinghua University, Taiwan
Shunsuke Inenaga	Kyushu University, Japan
Haitao Jiang	Shandong University, China
Gregory Kucherov	Universite Paris-Est Marne-la-Vallee, France
Tak-Wah Lam	University of Hong Kong, China
Gad Landau	University of Haifa, Israel
Moshe Lewenstein	Bar Ilan University, Israel
Inge Li Gortz	Technical University of Denmark, Denmark
Veli Makinen	University of Helsinki, Finland
Sebastian Maneth	University of Edinburgh, UK
Yakov Nekrich	University of Waterloo, Canada
Nicola Prezza	Technical University of Denmark, Denmark
Rahul Shah	Louisiana State University, USA
Ayumi Shinohara	Tohoku University, Japan
Tatiana Starikovskaya	ENS, PSL Research University, France
Esko Ukkonen	University of Helsinki, Finland
Filippo Utro	IBM T.J. Watson Research Center, USA







## ■ External Reviewers

Diego Arroyuelo	Dominik Kempa
Golnaz Badkobeh	Vladimir Kolesnikov
Hideo Bannai	Dmitry Kosolobov
Guillaume Blin	Henry Leung
Bastien Cazaux	Chi Man Liu
Panagiotis Charalampopoulos	Giovanni Manzini
Edgar Chavez	Andrea Marino
Alessio Conte	Laurent Mouchard
Fabio Cunial	Yuto Nakashima
Zanoni Dias	Solon Pissis
Bartłomiej Dudek	Simon Puglisi
Maciej Duleba	Jakub Radoszewski
Simone Faro	Carl Philip Reh
Gabriele Fici	Jamie Simpson
Johannes Fischer	William F. Smyth
Arnab Ganguly	Sagi Snir
Samah Ghazawi	Jens Stoye
Alice Heliou	Dekel Tsur
Diptarama Hendrian	Daniel Valenzuela
Sahar Hooshmand	Luca Versari



## ■ List of Authors

Paniz Abedin (8,20)  
Tatsuya Akutsu (10)  
Amihood Amir (4,11)  
Kotaro Aoyama (9)  
Uwe Baier (3)  
Hideo Bannai (2,7,9,12,15,19,24)  
Guillaume Blin (6)  
Alexandre Blondin Massé (6)  
Itai Boneh (11)  
Brian Brubach (5,13)  
Bastien Cazaux (21)  
Panagiotis Charalampopoulos (23)  
Rayan Chikhi (18)  
Maxime Crochemore (23)  
Colin de La Higuera (10)  
Bartłomiej Dudek (16)  
Haodi Feng (14)  
Guillaume Fertin (17)  
Julien Fradin (17)  
Alexandre Francisco (2)  
Mitsuru Funakoshi (12)  
Isamu Furuya (24)  
Travis Gagie (7)  
Arnab Ganguly (20)  
Marie Gasparoux (6)  
Pawel Gawrychowski (16)  
Jay Ghurye (13)  
Jiong Guo (14)  
Sylvie Hamel (6)  
Rui Henriques (2)  
Sahar Hooshmand (8,20)  
Heikki Hyyrö (15)  
Tomohiro I (7,9,24)  
Costas Iliopoulos (23)  
Shunsuke Inenaga (9,12,15,19,24)  
Takafumi Inoue (15)  
Taichi Ishikawa (22)  
Haitao Jiang (14)  
Tomasz Kociumaka (23)  
Christian Komusiewicz (17)  
M. Oguzhan Külekci (8)  
Avivit Levy (4)  
Jingjing Ma (14)  
Yuto Nakashima (9,12,19,24)  
Solon Pissis (23)  
Ely Porat (4)  
Jakub Radoszewski (23)  
Eric Rivals (21)  
Luis Russo (2)  
Wojciech Rytter (23)  
Yoshifumi Sakai (1)  
Alexander Schönhuth (18)  
Kilho Shin (22)  
Masayuki Takeda (9,12,15,19,24)  
Takeyuki Tamura (10)  
Sharma V. Thankachan (8,20)  
Yuki Urabe (19)  
Élise Vandomme (6)  
Tomasz Walen (23)  
Shu Zhang (14)  
Daming Zhu (14)

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu



Leibniz International Proceedings in Informatics

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



# Maximal Common Subsequence Algorithms

Yoshifumi Sakai

Graduate School of Agricultural Science, Tohoku University  
468-1, Aza-Aoba, Aramaki, Aoba-ku, Sendai 980-0845, Japan  
yoshifumi.sakai.c7@tohoku.ac.jp

---

## Abstract

A common subsequence of two strings is maximal, if inserting any character into the subsequence can no longer yield a common subsequence of the two strings. The present article proposes a (sub)linear-time, linear-space algorithm for finding a maximal common subsequence of two strings and also proposes a linear-time algorithm for determining if a common subsequence of two strings is maximal.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** algorithms, string comparison, longest common subsequence, constrained longest common subsequence

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.1

## 1 Introduction

A subsequence of a string of characters is obtained from the string by deleting any number of not necessarily contiguous characters at any position. A common subsequence of two strings can be thought of as a pattern common to the strings. A common subsequence is maximal, if inserting any character into the subsequence can no longer yield a common subsequence. Hence, any common subsequence can be found as a subsequence of some maximal common subsequence. The present article considers the problem of finding a maximal common subsequence of two strings both of length  $O(n)$  over an alphabet set of  $O(n)$  characters for some positive integer  $n$  and also considers the problem of determining if a given common subsequence of the two strings is maximal.

A longest one of maximal common subsequences is called a longest common subsequence (an LCS). It is well known that the dynamic programming algorithm of Wagner and Fisher [10] finds an LCS of two  $O(n)$ -length strings in  $O(n^2)$  time and  $O(n^2)$  space. Moreover, the divide-and-conquer version developed by Hirschberg [6] reduces the required space to  $O(n)$  without increasing the asymptotic execution time. On the other hand, Abboud et al. [1] revealed that, for any positive constant  $\epsilon$ , there exist no  $O(n^{2-\epsilon})$ -time algorithms for computing the LCS length, unless the strong exponential time hypothesis (SETH) [7, 8] is false. This immediately implies that, under assumption of SETH, neither an LCS can be found nor whether a common subsequence is an LCS can be determined in  $O(n^{2-\epsilon})$  time. Problems of finding a conditional LCS have also been considered. The constrained LCS (CLCS) problem [9, 3] (also called the SEQ-IC-LCS problem [2]) and the restricted LCS (RLCS) problem [5] (also called the SEQ-EC-LCS problem [2]) are such problems. Given a common subsequence  $P$  as essentially “relevant” (resp. “irrelevant”) to relationship between the two strings, the CLCS (RLCS) problem consists of finding an LCS that has (resp. does not have)  $P$  as a subsequence and was shown to be solvable in  $O(n^3)$  time [3] (resp. [5, 2]). From definition, the CLCS found is maximal. In contrast, the RLCS found is not necessarily maximal and, unless maximal, the RLCS might not be very informative in



© Yoshifumi Sakai;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 1; pp. 1:1–1:10

Leibniz International Proceedings in Informatics



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

certain applications because it is just obtained from some common subsequence, which has the “irrelevant”  $P$  perfectly as a subsequence, only by deleting a single character.

The reason why it takes at least an almost quadratic time to find an LCS or a conditional LCS as a pattern common to the two strings is due to condition that the pattern to be found should have a maximum length. Possibly for an analogous reason, the best asymptotic running time known for finding a shortest maximal common subsequence of two strings remains cubic [4]. The present article shows that, ignoring such conditions with respect to the length of a maximal common subsequence to be found, we can find a maximal common subsequence much faster, by proposing an  $O(n \log \log n)$ -time,  $O(n)$ -space algorithm. This algorithm can also be used to find a constrained maximal common subsequence, which hence has  $P$  as a subsequence, in the same asymptotic time and space, where  $P$  is an arbitrary common subsequence given as a “relevant” pattern. It is also shown that we can determine whether any given common subsequence, such as an RLCS, is maximal further faster, by proposing an  $O(n)$ -time algorithm.

This article is organized as follows. Section 2 defines notations and terminology used in this article. Section 3 proposes an  $O(n \log \log n)$ -time,  $O(n)$ -space algorithm that finds a maximal common subsequence of two strings of length  $O(n)$ . Section 4 modifies the above algorithm so as to output a maximal common subsequence having a given common subsequence as a subsequence in the same asymptotic time and space. Section 5 proposes an  $O(n)$ -time algorithm that determines if a given common subsequence is maximal. Section 6 concludes this article.

## 2 Preliminaries

For any sequences  $S$  and  $S'$ , let  $S \circ S'$  denote the concatenation of  $S$  followed by  $S'$ . Let  $\varepsilon$  denote the empty sequence. For any sequence  $S$ , let  $|S|$  denote the length of  $S$ . For any index  $i$  with  $1 \leq i \leq |S|$ , let  $S[i]$  denote the  $i$ th element of  $S$ , so that  $S = S[1] \circ S[2] \circ \dots \circ S[|S|]$ . A subsequence of  $S$  is a sequence obtained from  $S$  by deleting elements at any position, i.e.,  $S[i_1] \circ S[i_2] \circ \dots \circ S[i_l]$  for some indices  $i_1, i_2, \dots, i_l$  with  $0 \leq l \leq |S|$  and  $1 \leq i_1 < i_2 < \dots < i_l \leq |S|$ . For any sequences  $S$  and  $S'$ , we say that  $S$  contains  $S'$ , if  $S'$  is a subsequence of  $S$ . For any indices  $i'$  and  $i$  with  $0 \leq i' \leq i \leq |S|$ , let  $S(i', i]$  denote the contiguous subsequence of  $S$  consisting of all elements at position between  $i' + 1$  and  $i$ , i.e.,  $S[i' + 1] \circ S[i' + 2] \circ \dots \circ S[i]$ . Note that  $S(i, i) = \varepsilon$ . We call  $S(i', i]$  a prefix (resp. suffix) of  $S$ , if  $i' = 0$  (resp.  $i = |S|$ ).

Let  $\Sigma = \{c_1, c_2, \dots, c_{|\Sigma|}\}$  be an alphabet set of  $|\Sigma|$  characters, which are totally ordered. A string is a sequence of characters over  $\Sigma$ . For any strings  $X$  and  $Y$ , a common subsequence of  $X$  and  $Y$  is a subsequence of  $X$  that is also a subsequence of  $Y$ . We say that  $X$  and  $Y$  are disjoint, if they have no non-empty common subsequences. Let a common subsequence  $W$  of  $X$  and  $Y$  be maximal, if inserting any character into  $W$  can no longer yield a common subsequence of  $X$  and  $Y$ .

## 3 Algorithm for finding a maximal common subsequence

This section proposes an  $O(n \log \log n)$ -time algorithm that outputs, for any strings  $X$  and  $Y$  of length  $O(n)$  with  $|\Sigma| = O(n)$  given as input, a maximal common subsequence of  $X$  and  $Y$ .

For technical reasons, we assume without loss of generality that  $X[1] = Y[1] = c_1$ ,  $X[|X|] = Y[|Y|] = c_{|\Sigma|}$ , which will work as sentinels, and neither  $c_1$  nor  $c_{|\Sigma|}$  appears in  $X(1, |X| - 1]$  and also in  $Y(1, |Y| - 1]$ . Note that  $W[1] = c_1$ ,  $W[|W|] = c_{|\Sigma|}$ , and  $W(1, |W| - 1]$  is a maximal common subsequence of  $X(1, |X| - 1]$  and  $Y(1, |Y| - 1]$  for any maximal common subsequence  $W$  of  $X$  and  $Y$ .

We also assume that the array  $\mathcal{I}$  (resp.  $\mathcal{J}$ ) of arrays  $I_c$  (resp.  $J_c$ ) for all characters  $c$  in  $\Sigma$  is available, where  $I_c$  (resp.  $J_c$ ) is an appropriate data structure supporting queries of the following index, indicating the nearest occurrence of a specific character  $c$  in  $X$  (resp.  $Y$ ) from a specific position  $i$  (resp.  $j$ ).

► **Definition 1.** For any character  $c$  in  $\Sigma$  and any index  $i$  with  $0 \leq i \leq |X|$ , let  $I_c^{\leftarrow}(i)$  (resp.  $I_c^{\rightarrow}(i)$ ) denote the least (greatest) index such that  $c$  does not appear in  $X(I_c^{\leftarrow}(i), i]$  (resp.  $X(i, I_c^{\rightarrow}(i))$ ). Define index  $J_c^{\leftarrow}(j)$  (resp.  $J_c^{\rightarrow}(j)$ ) analogously with respect to  $Y$ .

In what follows, we adopt as data structure  $I_c$  (resp.  $J_c$ ) the y-fast trie [11] maintaining all indices  $i$  (resp.  $j$ ) with  $X[i] = c$  (resp.  $Y[j] = c$ ), because array  $\mathcal{I}$  (resp.  $\mathcal{J}$ ) is constructible in  $O(n \log \log n)$  time and  $O(n)$  space and supports  $O(\log \log n)$ -time queries of any index introduced above. However, in implementation for practical use, if  $n$  is not very large, then due to hidden constant factors in big-O notation, adopting as  $I_c$  (resp.  $J_c$ ) the array consisting of the same indices as the y-fast trie in ascending order, supporting  $O(\log n)$ -time queries based on a binary search of the array, might be more suitable. Furthermore, if  $|\Sigma|$  is a small constant, then we can adopt as  $I_c$  (resp.  $J_c$ ) the table of indices  $I_c^{\leftarrow}(i)$  and  $I_c^{\rightarrow}(i)$  (resp.  $J_c^{\leftarrow}(i)$  and  $J_c^{\rightarrow}(j)$ ) for all indices  $i$  (resp.  $j$ ), which supports  $O(1)$ -time queries.

We design the proposed algorithm based on the following property of a common subsequence  $W$ , which is naturally derived from the fact that  $W$  is not maximal if and only if inserting some character between some prefix and the remaining suffix of  $W$  still yields a common subsequence of  $X$  and  $Y$ .

► **Lemma 2.** For any common subsequence  $W$  of  $X$  and  $Y$ ,  $W$  is maximal if and only if  $X_k$  and  $Y_k$  are disjoint for any index  $k$  with  $0 \leq k \leq |W|$ , where  $X_k$  (resp.  $Y_k$ ) is the remaining substring obtained from  $X$  (resp.  $Y$ ) by deleting both the shortest prefix containing  $W(0, k]$  and the shortest suffix containing  $W(k, |W|]$ .

**Proof.** The lemma follows from the fact that, for any index  $k$  with  $0 \leq k \leq |W|$  and any character  $c$  in  $\Sigma$ ,  $W(0, k] \circ c \circ W(k, |W|]$  is a common subsequence of  $X$  and  $Y$  if and only if  $c$  appears in both  $X_k$  and  $Y_k$ . ◀

The proposed algorithm solves the problem using string variable  $W$ , which is initially set to  $c_1 \circ c_{|\Sigma|}$  and is eventually updated to a maximal common subsequence of  $X$  and  $Y$ . For any index  $k$  with  $0 \leq k \leq |W|$ , let  $X_k$  and  $Y_k$  be the substrings in Lemma 2. The algorithm updates  $W$  by iteratively replacing it by  $W(0, k] \circ c \circ W(k, |W|]$ , where  $k$  is the least index such that  $X_k$  and  $Y_k$  are not disjoint and  $c$  is a certain character appearing both in  $X_k$  and  $Y_k$ , until  $X_k$  and  $Y_k$  become disjoint for all indices  $k$  with  $0 \leq k \leq |W|$ . Note that the resulting string  $W$  is a maximal common subsequence of  $X$  and  $Y$  due to Lemma 2. The algorithm adopts as  $c$  the character that appears both in the shortest possible suffix of  $X_k$  or  $Y_k$  and the entire string of the other. As shown later, this choice is crucial to executing the algorithm in  $O(n \log \log n)$  time.

In order to execute the above, the algorithm maintains a sequence variable,  $\hat{W} = (i_1, j_1) \circ (i_2, j_2) \circ \cdots \circ (i_{|W|}, j_{|W|})$ , consisting of  $|W|$  index pairs so that  $X(i_k, i_{k+1}]$  and  $Y(j_k, j_{k+1}]$  are respectively certain prefixes of  $X_k$  and  $Y_k$  such that they are disjoint if and only if  $X_k$  and  $Y_k$  are disjoint. The character to be inserted at position between  $W(0, k]$  and  $W(k, |W|]$  is searched for by iteratively updating  $\hat{W}$  by replacing  $(i_{k+1}, j_{k+1})$  by  $(i_{k+1} - 1, j_{k+1} - 1)$ . If  $i_{k+1}$  becomes  $i_k$  or  $j_{k+1}$  becomes  $j_k$ , then, since  $X_k$  and  $Y_k$  are disjoint, the algorithm updates  $\hat{W}$  by replacing  $\hat{W}[k+1]$  by  $(i', j')$  and then updates  $k$  to  $k+1$ , where  $i'$  (resp.  $j'$ ) is the index such that  $X(0, i']$  (resp.  $Y(0, j']$ ) is the shortest prefix of  $X$  (resp.  $Y$ ) containing  $W(0, k+1]$ , i.e.  $i' = I_{W[k+1]}^{\rightarrow}(i_k) + 1$  (resp.  $j' = J_{W[k+1]}^{\rightarrow}(j_k) + 1$ ). Otherwise,

**Algorithm 1:** Algorithm findMCS

---

```

1:  $W \leftarrow c_1 \circ c_{|\Sigma|}$ ;
2:  $\hat{W} \leftarrow (1, 1) \circ (|X| - 1, |Y| - 1)$ ;
3:  $k \leftarrow 1$ ;
4: while  $k < |W|$ ,
5:    $(i', j') \leftarrow \hat{W}[k]$ ;
6:    $(i, j) \leftarrow \hat{W}[k + 1]$ ;
7:   while  $i' < i, j' < j, J_{X[i]}^<(j) \leq j'$ , and  $I_{Y[j]}^<(i) \leq i'$ ,
8:      $\hat{W}[k + 1] \leftarrow (i - 1, j - 1)$ ;
9:      $(i, j) \leftarrow \hat{W}[k + 1]$ ;
10:  if  $i = i'$  or  $j = j'$ , then
11:     $\hat{W}[k + 1] \leftarrow (I_{W[k+1]}^>(i') + 1, J_{W[k+1]}^>(j') + 1)$ ;
12:     $k \leftarrow k + 1$ ,
13:  otherwise, if  $J_{X[i]}^<(j) > j'$ , then
14:     $W \leftarrow W(0, k] \circ X[i] \circ W(k, |W|]$ ;
15:     $\hat{W} \leftarrow \hat{W}(0, k] \circ (i - 1, J_{X[i]}^<(j) - 1) \circ \hat{W}(k, |\hat{W}|]$ ,
16:  otherwise,
17:     $W \leftarrow W(0, k] \circ Y[j] \circ W(k, |W|]$ ;
18:     $\hat{W} \leftarrow \hat{W}(0, k] \circ (I_{Y[j]}^<(i) - 1, j - 1) \circ \hat{W}(k, |\hat{W}|]$ ;
19:  output  $W$ .
```

---

if  $X[i_{k+1}]$  appears in  $Y(j_k, j_{k+1})$  (i.e., if  $J_{X[i_{k+1}]}^<(j_{k+1}) > j_k$ ), then the algorithm updates  $W$  to  $W(0, k] \circ X[i_{k+1}] \circ W(k, |W|]$  and also updates  $\hat{W}$  to  $\hat{W}(0, k] \circ (i, j) \circ \hat{W}(k, |\hat{W}|]$ , where  $i$  (resp.  $j$ ) is the index such that  $X(i, |X|)$  (resp.  $Y(j, |Y|)$ ) is the shortest suffix of  $X$  (resp.  $Y$ ) containing  $X[i_{k+1}] \circ W(k, |W|]$ , i.e.,  $i = i_{k+1} - 1$  (resp.  $j = J_{X[i_{k+1}]}^<(j_{k+1}) - 1$ ). Otherwise, since  $Y[j_{k+1}]$  appears in  $X(i_k, i_{k+1})$ , the algorithm updates  $W$  and  $\hat{W}$  in a symmetric manner with respect to  $Y[j_{k+1}]$ .

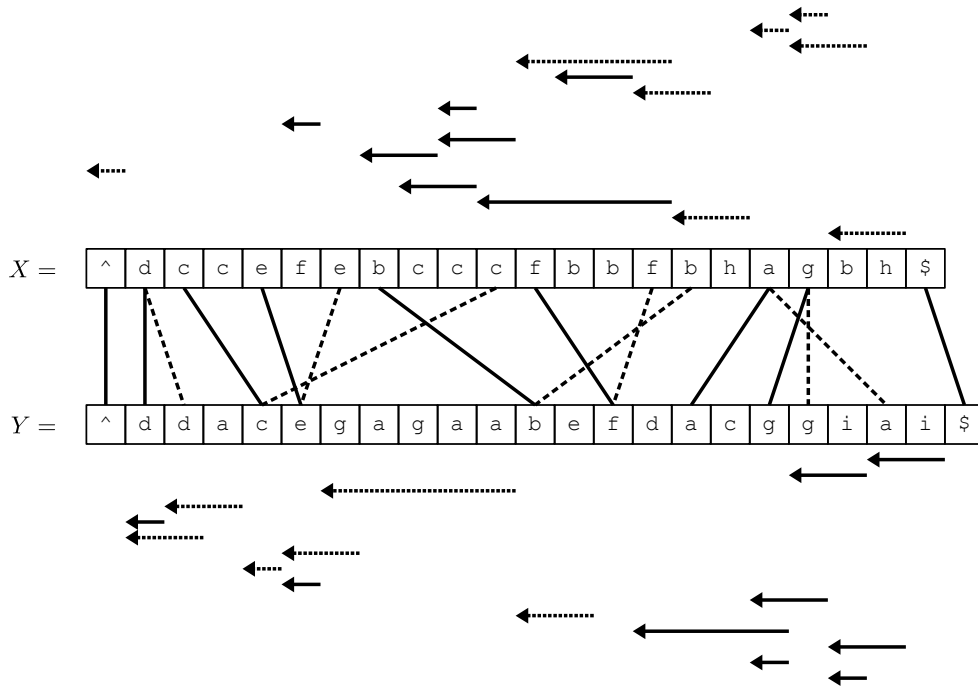
A pseudocode of the proposed algorithm is given as Algorithm findMCS in Algorithm 1, where we assume that, by an  $O(n \log \log n)$ -time preprocessing, arrays  $\mathcal{I}$  and  $\mathcal{J}$  are available as data structures supporting  $O(\log \log n)$ -time queries of any of indices  $I_c^<(i)$ ,  $I_c^>(i)$ ,  $J_c^<(j)$ , and  $J_c^>(j)$ . In this pseudocode, variables  $i'$ ,  $j'$ ,  $i$ , and  $j$  are respectively used to represent indices  $i_k$ ,  $j_k$ ,  $i_{k+1}$ , and  $j_{k+1}$ , where  $(i_k, j_k) = \hat{W}[k]$  and  $(i_{k+1}, j_{k+1}) = \hat{W}[k + 1]$ . A concrete example of how this algorithm works is presented in Figure 1.

As mentioned earlier, the following condition holds at any execution of line 7 of this algorithm and also at the last execution of line 4.

► **Definition 3.** For any string  $W$ , any index pair sequence  $\hat{W} = (i_1, j_1) \circ (i_2, j_2) \circ \dots \circ (i_{|W|}, j_{|W|})$ , and any index  $k$ , let  $C(W, \hat{W}, k)$  denote the condition that

- $W$  is a common subsequence of  $X$  and  $Y$  containing  $c_1 \circ c_{|\Sigma|}$ ,
- $1 \leq k \leq |W|$ ,
- for any index  $k'$  with  $1 \leq k' \leq k - 1$ ,  $X(i_{k'}^+, i_{k'+1}^+]$  and  $Y(j_{k'}^+, j_{k'+1}^+]$  are disjoint,
- $(i_k, j_k) = (i_k^+, j_k^+)$ , and
- for any index  $k'$  with  $k \leq k' \leq |W| - 1$ ,
  - $i_k \leq i_{k'+1} \leq i_{k'+1}^+$ ,
  - $j_k \leq j_{k'+1} \leq j_{k'+1}^+$ ,
  - $X(i_{k'+1}, i_{k'+1}^+]$  and  $Y(j_k, j_{k'+1}^+]$  are disjoint, and





■ **Figure 1** A maximal common subsequence  $W = \hat{dcebfag}\$$  of  $X = \hat{dccefcbcccfbbfbhagbh}\$$  and  $Y = \hat{ddacegagaabefdacggiaai}\$$  with  $c_1 = \hat{\ }$  and  $c_{|\Sigma|} = \$$ , which is output by Algorithm findMCS. Lines 5 through 18 of the algorithm are executed fifteen times and for each number  $t$  with  $1 \leq t \leq 15$ , the  $t$ th most inner pair of arrows (one solid and the other dotted, which are of the same length) indicates which index pairs  $(i, j)$  are considered by line 7 throughout the  $t$ th iteration of lines 5 through 18, where the dotted arrow is chosen so as to show that the sum of the length of all dotted arrows is at most  $2(|X| + |Y|)$ . Each dashed line between  $X[i]$  and  $Y[j]$  indicates that  $\hat{W}$  is replaced by  $\hat{W}(0, k) \circ (i - 1, j - 1) \circ \hat{W}(k, |\hat{W}|)$  by either line 15 or line 18. Each solid line between  $X[i']$  and  $Y[j']$ , other than the leftmost one, indicates that  $\hat{W}[k + 1]$  is set to index pair  $(i', j')$  by line 11.

–  $X(i_k, i_{k'+1}^+]$  and  $Y(j_{k'+1}, j_{k'+1}^+]$  are disjoint, where, for any index  $k'$  with  $0 \leq k' \leq |W|$ ,  $i_{k'}^-$  (resp.  $j_{k'}^-$ ) is the least index such that  $X(0, i_{k'}^-]$  (resp.  $Y(0, j_{k'}^-]$ ) contains  $W(0, k']$  and  $i_{k'+1}^+$  (resp.  $j_{k'+1}^+$ ) is the greatest index such that  $X(i_{k'+1}^+, |X|)$  (resp.  $Y(j_{k'+1}^+, |Y|)$ ) contains  $W(k', |W|)$ .

► **Lemma 4.** *Condition  $C(W, \hat{W}, k)$  holds at any execution of line 7 of Algorithm findMCS and also at the last execution of line 4.*

**Proof.** The lemma is proven by induction. Since  $C(c_1 \circ c_{|\Sigma|}, (1, 1) \circ (|X| - 1, |Y| - 1), 1)$  holds at the first execution of line 7, assume that  $C(W, \hat{W}, k)$  holds at an arbitrary execution of line 7. Let  $(i', j') = \hat{W}[k]$  and let  $(i, j) = \hat{W}[k + 1]$ . If  $i' < i, j' < j, J_{X[i]}^<(j) \leq j'$ , and  $I_{Y[j]}^<(i) \leq i'$ , then  $C(W, \hat{W}(0, k) \circ (i - 1, j - 1) \circ \hat{W}(k + 1, |W|), k)$  holds, because  $i' \leq i - 1, j' \leq j - 1, X[i]$  does not appear in  $Y(j', j]$ , and  $Y[j]$  does not appear in  $X(i', i]$ . If  $i = i'$  or  $j = j'$ , then  $C(W, \hat{W}(0, k) \circ (I_{W[k+1]}^>(i') + 1, J_{W[k+1]}^>(j') + 1) \circ \hat{W}(k + 1, |W|), k + 1)$  holds, because  $X(i_k^-, i_{k+1}^+]$  and  $Y(j_k^-, j_{k+1}^+]$  are disjoint. If  $i' < i, j' < j$ , and  $J_{X[i]}^<(j) > j'$ , then  $C(W(0, k) \circ X[i] \circ W(k, |W|), \hat{W}(0, k) \circ (i - 1, J_{X[i]}^<(j) - 1) \circ \hat{W}(k, |W|), k)$  holds, because  $X(i - 1, |X|)$  (resp.  $Y(J_{X[i]}^<(j) - 1, |Y|)$ ) is the shortest suffix of  $X$  (resp.  $Y$ )

that contains  $X[i] \circ W(k, |W|)$ . Analogously, if  $i' < i$ ,  $j' < j$ , and  $I_{Y[j]}^<(i) > i'$ , then  $C(W(0, k] \circ Y[j] \circ W(k, |W|), \hat{W}(0, k] \circ (I_{Y[j]}^<(i) - 1, j - 1) \circ \hat{W}(k, |W|), k)$  holds.  $\blacktriangleleft$

The following simple lemma plays a key role in estimating execution time of the algorithm. This lemma claims, for example, that the situation where any solid line other than the leftmost and rightmost ones in Figure 1 shares at least one of endpoints with a unique dotted line is inevitable.

► **Lemma 5.** *At least one of  $I_{W[k+1]}^>(i') = i_{k+1}^+$  or  $J_{W[k+1]}^>(j') = j_{k+1}^+$  holds at any execution of line 11 in Algorithm findMCS, where  $i_{k+1}^+$  and  $j_{k+1}^+$  are the indices in Definition 3.*

**Proof.** Since  $X(i', i_{k+1}^+]$  and  $Y(j', j_{k+1}^+]$  are disjoint due to Lemma 4,  $W[k+1]$  does not appear in at least one of  $X(i', i_{k+1}^+]$  or  $Y(j', j_{k+1}^+]$ .  $\blacktriangleleft$

► **Theorem 6.** *For any strings  $X$  and  $Y$  of length  $O(n)$  with  $|\Sigma| = O(n)$ , Algorithm findMCS outputs a maximal common subsequence of  $X$  and  $Y$  in  $O(n \log \log n)$  time and  $O(n)$  space.*

**Proof.** Since  $C(W, \hat{W}, |W|)$  holds at the last execution of line 4 of the algorithm due to Lemma 4, it follows from Lemma 2 that  $W$  output by the algorithm is a maximal common subsequence of  $X$  and  $Y$ .

Execution time of the algorithm is estimated as follows. Let  $V$  be the eventual string  $W$  output by line 19. For any index  $k$  with  $0 \leq k \leq |V|$ , let  $g_k^-$  (resp.  $h_k^-$ ) denote the least index such that  $X(0, g_k^-]$  (resp.  $Y(0, h_k^-]$ ) contains  $V(0, k]$ . Let  $k$  be an arbitrary index with  $1 \leq k \leq |V|$  and consider  $W$  and  $\hat{W}$  just before execution of line 11. Let  $i_k^-, j_k^-, i_{k+1}^+$ , and  $j_{k+1}^+$  be the indices in Definition 3. Let  $(i', j') = \hat{W}[k]$  and let  $(i, j) = \hat{W}[k+1]$ . Note that  $i' = i_k^-$  and  $j' = j_k^-$  due to Lemma 4. Since  $i = i'$  or  $j = j'$ ,  $\hat{W}[k+1]$  is obtained from  $(i_{k+1}^+, j_{k+1}^+)$  by executing either line 15 or line 18 and then executing line 8 iteratively  $\min(i_{k+1}^+ - i_k^-, j_{k+1}^+ - j_k^-)$  times. This implies that execution time of the algorithm is  $O(\sum_{k=1}^{|V|-1} \min(i_{k+1}^+ - i_k^-, j_{k+1}^+ - j_k^-) \log n)$ . Since  $W(0, k] = V(0, k]$ , both  $i_k^- = g_k^-$  and  $j_k^- = h_k^-$  hold. Similarly, since  $W(0, k+1] = V(0, k+1]$ , both  $I_{W[k+1]}^>(i') + 1 = g_{k+1}^-$  and  $J_{W[k+1]}^>(j') + 1 = h_{k+1}^-$  hold. Therefore, from Lemma 5,  $i_{k+1}^+ + 1 = g_{k+1}^-$  or  $j_{k+1}^+ + 1 = h_{k+1}^-$  holds and hence we have that  $\min(i_{k+1}^+ - i_k^-, j_{k+1}^+ - j_k^-) \leq \max(g_{k+1}^- - g_k^-, h_{k+1}^- - h_k^-)$ . Since  $g_1^- = 1$ ,  $h_1^- = 1$ ,  $g_{|V|}^- = |X|$ , and  $h_{|V|}^- = |Y|$ ,  $\sum_{k=1}^{|V|-1} \max(g_{k+1}^- - g_k^-, h_{k+1}^- - h_k^-) \leq |X| + |Y| = O(n)$ . Thus,  $\sum_{k=1}^{|V|-1} \min(i_{k+1}^+ - i_k^-, j_{k+1}^+ - j_k^-) = O(n)$ , implying that the algorithm outputs  $V$  in  $O(n \log \log n)$  time.

The algorithm uses variables  $W$ ,  $\hat{W}$ ,  $k$ ,  $i'$ ,  $j'$ ,  $i$ , and  $j$ , together with data structures  $\mathcal{I}$  and  $\mathcal{J}$ , which all require  $O(n)$  space.  $\blacktriangleleft$

#### 4 Algorithm for finding a constrained maximal common subsequence

This section modifies Algorithm findMCS so as to output, for any common subsequence  $P$  of  $X$  and  $Y$  given as an additional input string, a maximal common subsequence of  $X$  and  $Y$  that contains  $P$  in  $O(n \log \log n)$  time and  $O(n)$  space, where we assume the same condition of  $X$  and  $Y$  as in Section 3 and also assume that  $P[1] = c_1$ ,  $P[|P|] = c_{|\Sigma|}$  and neither  $c_1$  nor  $c_{|\Sigma|}$  appears in  $P(1, |P| - 1]$ . Note that  $W[1] = c_1$ ,  $W[|W|] = c_{|\Sigma|}$ , and  $W(1, |W| - 1]$  is a maximal common subsequence of  $X(1, |X| - 1]$  and  $Y(1, |Y| - 1]$  containing  $P(1, |P| - 1]$  for any maximal common subsequence  $W$  of  $X$  and  $Y$  containing  $P$ .

The only difference of the modified algorithm from the original algorithm is to initialize  $W$  to  $P$ , instead of  $c_1 \circ c_{|\Sigma|}$ , and  $\hat{W}$  to a certain index pair sequence  $\hat{P}$  satisfying  $C(P, \hat{P}, 1)$ ,

**Algorithm 2:** Algorithm findCMCS

---

```

1:  $W \leftarrow P$ ;
2:  $\hat{W} \leftarrow \varepsilon$ ;
3:  $i \leftarrow |X| - 1$ ;  $j \leftarrow |Y| - 1$ ;
4: for each index  $k$  from  $|P| - 1$  down to 1,
5:     while  $X[i + 1] \neq P[k + 1]$ ,
6:          $i \leftarrow i - 1$ ;
7:     while  $Y[j + 1] \neq P[k + 1]$ ,
8:          $j \leftarrow j - 1$ ;
9:      $\hat{W} \leftarrow (i, j) \circ \hat{W}$ ;
10:     $i \leftarrow i - 1$ ;  $j \leftarrow j - 1$ ;
11:  $\hat{W} \leftarrow (1, 1) \circ \hat{W}$ ;
12:  $k \leftarrow 1$ ;
13: do the same as lines 4 through 19 of Algorithm findMCS.

```

---

instead of  $(1, 1) \circ (|X| - 1, |Y| - 1)$ . Since lines 4 through 19 of the original algorithm delete no characters from  $W$ , the modified algorithm eventually outputs a maximal common subsequence of  $X$  and  $Y$  that contains  $P$  in  $O(n \log \log n)$  time after initialization of  $(W, \hat{W}, 1)$  to  $(P, \hat{P}, 1)$ . For any index  $k$  with  $1 \leq k \leq |P|$ , let  $i_{k+1}^+$  (resp.  $j_{k+1}^+$ ) be the greatest index such that  $X(i_{k+1}^+, |X|)$  (resp.  $Y(j_{k+1}^+, |Y|)$ ) contains  $P(k, |P|)$ . Then, Definition 3 immediately suggests that  $\hat{P}$  can be set to  $(1, 1) \circ (i_2^+, j_2^+) \circ (i_3^+, j_3^+) \circ \cdots \circ (i_{|P|}^+, j_{|P|}^+)$ . Thus, we have Algorithm findCMCS presented in Algorithm 2 as an  $O(n \log \log n)$ -time algorithm for finding a maximal common subsequence of  $X$  and  $Y$  containing  $P$ .

► **Theorem 7.** *For any strings  $X$  and  $Y$  of length  $O(n)$  with  $|\Sigma| = O(n)$  and any common subsequence  $P$  of  $X$  and  $Y$ , Algorithm findCMCS outputs a maximal common subsequence of  $X$  and  $Y$  containing  $P$  in  $O(n \log \log n)$  time and  $O(n)$  space.*

**Proof.** It is easy to verify by induction that, for any index  $k$  with  $1 \leq k \leq |P| - 1$ ,  $X(i, |X|)$  (resp.  $Y(j, |Y|)$ ) at execution of line 9 of the algorithm are the shortest suffix of  $X$  (resp.  $Y$ ) that contains  $P(k, |P|)$ . Therefore,  $W$ ,  $\hat{W}$ , and  $k$  just after execution of line 12 satisfy  $C(W, \hat{W}, k)$ . Since lines 1 through 12 are executed in  $O(n)$  time, the theorem can be proven in a way similar to the proof of Theorem 6. ◀

## 5 Algorithm for determining if a common subsequence is maximal

This section proposes an  $O(n)$ -time algorithm that determines, for any strings  $X$  and  $Y$  of length  $O(n)$  with  $|\Sigma| = O(n)$  and any common subsequence  $W$  of  $X$  and  $Y$  given as input, whether  $W$  is maximal or not.

The proposed algorithm is based on Lemma 2. Using an array of  $|\Sigma|$  bits, each being used to indicate if a distinct character in  $\Sigma$  appears in  $Y_k$ , we can determine if  $X_k$  and  $Y_k$  are disjoint in  $O(|X_k| + |Y_k|)$  time for any index  $k$  with  $0 \leq k \leq |W|$ , where  $X_k$  and  $Y_k$  are the substrings of  $X$  and  $Y$  in Lemma 2, respectively. However, this naive approach provides only an  $O(n^2)$ -time algorithm, because both  $|X_k|$  and  $|Y_k|$  can be  $\Theta(n)$  for all indices  $k$  and  $|W|$  can also be  $\Theta(n)$ . In order to reduce this execution time to  $O(n)$ , the algorithm exploits the fact that if  $X_{k-1}$  and  $Y_{k-1}$  are disjoint, then the prefix  $X_k^{\triangleleft}$  of  $X_k$  overlapping  $X_{k-1}$  and the prefix  $Y_k^{\triangleleft}$  of  $Y_k$  overlapping  $Y_{k-1}$  are also disjoint; otherwise,  $W$  is not maximal due to Lemma 2, where  $X_{-1} = X(0, 0]$  and  $Y_{-1} = Y(0, 0]$ . From this fact, if  $X_{k-1}$  and  $Y_{k-1}$  are

**Algorithm 3:** Algorithm `determinelfMCS`


---

```

1:   $i_0^- \leftarrow 0; j_0^+ \leftarrow 0; i_{|W|+1}^- \leftarrow |X|; j_{|W|+1}^+ \leftarrow |Y|;$ 
2:   $k \leftarrow 1;$ 
3:  for each index  $i$  from 1 to  $|X|$ ,
4:    if  $k \leq |W|$  and  $X[i] = W[k]$ , then
5:       $i_k^- \leftarrow i;$ 
6:       $k \leftarrow k + 1;$ 
7:   $k \leftarrow |W| - 1;$ 
8:  for each index  $i$  from  $|X|$  down to 1,
9:    if  $k \geq 1$  and  $X[i] = W[k + 1]$ , then
10:      $i_{k+1}^- \leftarrow i - 1;$ 
11:      $k \leftarrow k - 1;$ 
12:   $k \leftarrow 1;$ 
13:  for each index  $j$  from 1 to  $|Y|$ ,
14:    if  $k \leq |W|$  and  $Y[j] = W[k]$ , then
15:       $j_k^+ \leftarrow j;$ 
16:       $k \leftarrow k + 1;$ 
17:   $k \leftarrow |W| - 1;$ 
18:  for each index  $j$  from  $|Y|$  down to 1,
19:    if  $k \geq 1$  and  $Y[j] = W[k + 1]$ , then
20:      $j_{k+1}^+ \leftarrow j - 1;$ 
21:      $k \leftarrow k - 1;$ 
22:  for each character  $c$  in  $\Sigma$ ,
23:     $i_c \leftarrow 0;$ 
24:     $j_c \leftarrow 0;$ 
25:  for each index  $k$  from 0 to  $|W|$ ,
26:    for each index  $i$  from  $i_k^- + 1$  to  $i_{k+1}^-$ , where  $i_0^- = 0$ ,
27:       $i_{X[i]} \leftarrow i;$ 
28:      if  $j_{X[i]} > j_k^-$ , then
29:        output “not maximal” and halt;
30:    for each index  $j$  from  $j_k^+ + 1$  to  $j_{k+1}^+$ , where  $j_0^+ = 0$ ,
31:       $j_{Y[j]} \leftarrow j;$ 
32:      if  $i_{Y[j]} > i_k^+$ , then
33:        output “not maximal” and halt;
34:  output “maximal”.

```

---

disjoint, then whether  $X_k$  and  $Y_k$  are disjoint can be determined only by checking if  $X_k^\triangleright$  and  $Y_k^\triangleleft$  are disjoint as well as checking if  $X_k$  and  $Y_k^\triangleright$  are disjoint, where  $X_k^\triangleright$  (resp.  $Y_k^\triangleright$ ) are the remaining suffix of  $X_k$  (resp.  $Y_k$ ) after deleting prefix  $X_k^\triangleleft$  (resp.  $Y_k^\triangleleft$ ). Note however that, as long as using the array of  $|\Sigma|$  bits, it still takes  $O(|X_k| + |Y_k|)$  time to determine if  $X_k$  and  $Y_k$  are disjoint. The algorithm reduces this execution time to  $O(|X_k^\triangleright| + |Y_k^\triangleright|)$  by using, instead of the bit array, a pair of arrays of  $|\Sigma|$  indices. Each index in one (resp. the other) of the arrays in the pair is used to represent the last position at which a distinct character in  $\Sigma$  appears in the prefix of  $Y$  (resp.  $X$ ) having  $Y_k^\triangleleft$  (resp.  $X_k$ ) as a suffix. This index array allows the algorithm to determine if any character in  $X_k^\triangleright$  (resp.  $Y_k^\triangleright$ ) appears in  $Y_k^\triangleleft$  (resp.  $X_k$ ) in  $O(1)$  time. Furthermore, since the prefix of  $Y$  (resp.  $X$ ) having  $Y_k^\triangleleft$  (resp.  $X_k$ ) as a

suffix is the concatenation of the prefix of  $Y$  (resp.  $X$ ) having  $Y_{k-1}^{\leftarrow}$  (resp.  $X_{k-1}$ ) as a suffix followed by  $Y_{k-1}^{\rightarrow}$  (resp.  $X_k^{\rightarrow}$ ), for each  $k$ , this index array can be updated appropriately in  $O(|Y_{k-1}^{\rightarrow}|)$  (resp.  $O(|X_k^{\rightarrow}|)$ ) time.

We show that Algorithm `determinelfMCS` presented in Figure 3 works as the proposed algorithm.

► **Theorem 8.** *For any strings  $X$  and  $Y$  of length  $O(n)$  with  $|\Sigma| = O(n)$  and any common subsequence  $W$  of  $X$  and  $Y$ , Algorithm `determinelfMCS` outputs message “not maximal”, if  $W$  is not a maximal common subsequence of  $X$  and  $Y$ , or outputs message “maximal”, otherwise, in  $O(n)$  time.*

**Proof.** For any index  $k$  with  $0 \leq k \leq |W|$ , let  $X_k$  and  $Y_k$  be the strings in Lemma 2 and let indices  $i_k^{\leftarrow}, i_{k+1}^{\leftarrow}, j_k^{\leftarrow}$ , and  $j_{k+1}^{\leftarrow}$  be such that  $X_k = X(i_k^{\leftarrow}, i_{k+1}^{\leftarrow}]$  and  $Y_k = Y(i_k^{\leftarrow}, i_{k+1}^{\leftarrow}]$ . Furthermore, let  $X_k^{\leftarrow} = X(i_k^{\leftarrow}, \max(i_k^{\leftarrow}, i_k^{\leftarrow})]$  and let  $X_k^{\rightarrow} = X(\max(i_k^{\leftarrow}, i_k^{\leftarrow}), i_{k+1}^{\leftarrow}]$ , where  $i_0^{\leftarrow} = 0$ . Similarly, let  $Y_k^{\leftarrow} = Y(j_k^{\leftarrow}, \max(j_k^{\leftarrow}, j_k^{\leftarrow})]$  and let  $Y_k^{\rightarrow} = Y(\max(j_k^{\leftarrow}, j_k^{\leftarrow}), j_{k+1}^{\leftarrow}]$ , where  $j_0^{\leftarrow} = 0$ . The algorithm uses index variable  $i_c$  (resp.  $j_c$ ) for any character  $c$  in  $\Sigma$ . Let  $I$  (resp.  $J$ ) denote the array consisting of variables  $i_c$  (resp.  $j_c$ ) for all characters  $c$  in  $\Sigma$ . For any index  $k$  with  $-1 \leq k \leq |W|$ , let  $C_I(k)$  (resp.  $C_J(k)$ ) denote the condition that, for any character  $c$  in  $\Sigma$ ,  $X(i_c, i_{k+1}^{\leftarrow}]$  (resp.  $Y(j_c, j_{k+1}^{\leftarrow}]$ ) is the longest suffix of  $X(0, i_{k+1}^{\leftarrow}]$  (resp.  $Y(0, j_{k+1}^{\leftarrow}]$ ) in which  $c$  does not appear.

After computing indices  $i_k^{\leftarrow}, j_k^{\leftarrow}, i_k^{\leftarrow}$ , and  $j_k^{\leftarrow}$  for all indices  $k$  with  $0 \leq k \leq |W|$  by lines 1 through 21 of the algorithm, lines 22 through 24 initialize variables  $i_c$  and  $j_c$  so that  $C_I(-1)$  and  $C_J(-1)$  hold. Then, for any index  $k$  from 0 to  $|W|$ , lines 25 through 33 check if  $X_k$  and  $Y_k$  are disjoint as follows. Since either  $k = 0$  or  $X_{k-1}$  and  $Y_{k-1}$  are disjoint,  $X_k^{\leftarrow}$  and  $Y_k^{\leftarrow}$  are disjoint. Therefore, it suffices to check if  $X_k^{\rightarrow}$  and  $Y_k^{\leftarrow}$  are disjoint and check if  $X_k$  and  $Y_k^{\rightarrow}$  are disjoint. Lines 27 through 29 update  $I$  so as to satisfy  $C_I(k)$  by iteratively executing line 27 and also check if  $X_k^{\rightarrow}$  and  $Y_k^{\leftarrow}$  are disjoint by iteratively executing line 28 using array  $J$  satisfying  $C_J(k-1)$ . If  $X_k^{\rightarrow}$  and  $Y_k^{\leftarrow}$  are not disjoint, then, since  $j_{X[i]} > j_k^{\leftarrow}$  holds for some index  $i$  with  $i_k^{\leftarrow} + 1 \leq i \leq i_{k+1}^{\leftarrow}$  due to  $C_J(k-1)$ , line 29 outputs message “not maximal” and terminates the algorithm; otherwise, line 29 is never executed also due to  $C_J(k-1)$  and hence lines 30 through 33 are executed. Lines 30 through 33 update array  $J$  so as to satisfy  $C_J(k)$  and check if  $X_k$  and  $Y_k^{\rightarrow}$  are disjoint using array  $I$  satisfying  $C_I(k)$  in a similar manner. Thus, the algorithm works correctly.

It is easy to verify that the algorithm runs in  $O(n)$  time. ◀

## 6 Conclusion

The present article proposed an  $O(n \log \log n)$ -time,  $O(n)$ -space algorithm that finds a maximal common subsequence of two  $O(n)$ -length strings over an alphabet set of  $O(n)$  characters, which are totally ordered, where  $n$  is an arbitrary positive integer and a common subsequence is maximal, if inserting any character into it can no longer yields a common subsequence. It is also shown that, without increasing asymptotic time and space complexities, this algorithm can be used to find a constrained maximal common subsequence, which contains a common subsequence given arbitrarily as a “relevant” pattern, after an appropriate initialization of some variables. Furthermore, an  $O(n)$ -time algorithm that determines if a given common subsequence is maximal was also proposed.

There remain some questions to be solved, which are related to the problems considered in the present article. Our algorithms run much faster than those proposed so far (and also all possible algorithms under SETH) for the LCS-related problems corresponding to ours. One reason for this difference is that any common subsequence is certainly a subsequence

of some maximal common subsequence but is not necessarily a subsequence of any LCS. This fact naturally poses a question whether we can find a restricted maximal common subsequence, which does not contain a common subsequence given as an “irrelevant” pattern, in  $O(n \log \log n)$  time and  $O(n)$  space, because some restricted non-maximal common subsequences are not necessarily subsequences of any restricted maximal common subsequence. The gap between asymptotic execution time of the proposed algorithms for finding a maximal common subsequence and for determining if a common subsequence given is maximal immediately poses another natural question whether we can find a maximal common subsequence in  $O(n)$  time.

---

## References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 59–78. IEEE, 2015.
- 2 Yi-Ching Chen and Kun-Mao Chao. On the generalized constrained longest common subsequence problems. *Journal of Combinatorial Optimization*, 21(3):383–392, 2011.
- 3 Francis YL Chin, Alfredo De Santis, Anna Lisa Ferrara, NL Ho, and SK Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4):175–179, 2004.
- 4 Campbell B Fraser, Robert W Irving, and Martin Middendorf. Maximal common subsequences and minimal common supersequences. *Information and Computation*, 124(2):145–153, 1996.
- 5 Zvi Gotthilf, Danny Hermelin, Gad M Landau, and Moshe Lewenstein. Restricted LCS. In *International Symposium on String Processing and Information Retrieval*, pages 250–257. Springer, 2010.
- 6 Daniel S Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.
- 7 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.
- 8 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.
- 9 Yin-Te Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.
- 10 Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.
- 11 Dan E Willard. Log-logarithmic worst-case range queries are possible in space  $\Theta(N)$ . *Information Processing Letters*, 17(2):81–84, 1983.

# Order-Preserving Pattern Matching Indeterminate Strings

**Rui Henriques**

INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Portugal  
rmch@tecnico.ulisboa.pt (correspondence)

**Alexandre P. Francisco**

INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Portugal

**Luís M. S. Russo**

INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Portugal

**Hideo Bannai**

Department of Computer Science, Kyushu University, Japan

---

## Abstract

Given an indeterminate string pattern  $p$  and an indeterminate string text  $t$ , the problem of order-preserving pattern matching with character uncertainties ( $\mu$ OPPM) is to find all substrings of  $t$  that satisfy one of the possible orderings defined by  $p$ . When the text and pattern are determinate strings, we are in the presence of the well-studied exact order-preserving pattern matching (OPPM) problem with diverse applications on time series analysis. Despite its relevance, the exact OPPM problem suffers from two major drawbacks: 1) the inability to deal with indetermination in the text, thus preventing the analysis of noisy time series; and 2) the inability to deal with indetermination in the pattern, thus imposing the strict satisfaction of the orders among all pattern positions.

In this paper, we provide the first polynomial algorithms to answer the  $\mu$ OPPM problem when: 1) indetermination is observed on the pattern or text; and 2) indetermination is observed on both the pattern and the text and given by uncertainties between pairs of characters. First, given two strings with the same length  $m$  and  $O(r)$  uncertain characters per string position, we show that the  $\mu$ OPPM problem can be solved in  $O(mr \lg r)$  time when one string is indeterminate and  $r \in \mathbb{N}^+$  and in  $O(m^2)$  time when both strings are indeterminate and  $r=2$ . Second, given an indeterminate text string of length  $n$ , we show that  $\mu$ OPPM can be efficiently solved in polynomial time and linear space.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Pattern matching

**Keywords and phrases** Order-preserving pattern matching, Indeterminate string analysis, Generic pattern matching, Satisfiability

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.2

**Funding** This work was developed in the context of a secondment granted by the BIRDS MASC RISE project funded in part by EU H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no.690941. This work was further supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2013.

## 1 Introduction

Given a pattern string  $p$  and a text string  $t$ , the exact order preserving pattern matching (OPPM) problem is to find all substrings of  $t$  with the same relative orders as  $p$ . The problem is applicable to strings with characters drawn from numeric or ordinal alphabets.



© Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 2; pp. 2:1–2:15

Leibniz International Proceedings in Informatics



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



Illustrating, given  $p=(1,5,3,3)$  and  $t=(5,1,4,2,2,5,2,4)$ , substring  $t[1..4]=(1,4,2,2)$  is reported since it satisfies the character orders in  $p$ ,  $p[0] \leq p[2] = p[3] \leq p[1]$ . Despite its relevance, the OPPM problem has limited potential since it prevents the specification of errors, uncertainties or don't care characters within the text.

Indeterminate strings allow uncertainties between two or more characters per position. Given indeterminate strings  $p$  and  $t$ , the problem of order preserving pattern matching uncertain text ( $\mu$ OPPM) is to find all substrings of  $t$  with an assignment of values that satisfy the orders defined by  $p$ . For instance, let  $p=(1,2|5,3,3)$  and  $t=(5,0,1,2|1,2,5,2|3,3|4)$ . The substrings  $t[1..4]$  and  $t[4..7]$  are reported since there is an assignment of values that preserve either  $p[0] < p[1] < p[2] = p[3]$  or  $p[0] < p[2] = p[3] < p[1]$  orderings: respectively  $t[1..4]=(0,1,2,2)$  and  $t[4..7]=(2,5,3,3)$ .

Order-preserving pattern matching captures the structural isomorphism of strings, therefore having a wide-range of relevant applications in the analysis of financial times series, musical sheets, physiological signals and biological sequences [32, 39, 36]. Uncertainties often occur across these domains. In this context, although the OPPM problem is already a relaxation of the traditional pattern matching problem, the need to further handle localized errors is essential to deal with noisy strings [33]. For instance, given the stochasticity of gene regulation (or markets), the discovery of order-preserving patterns in gene expression (or financial) time series needs to account for uncertainties [35, 34]. Numerical indexes of amino-acids (representing physiochemical and biochemical properties) are subjected to errors difficulting the analysis of protein sequences [38]. Another example are ordinal strings obtained from the discretization of numerical strings, often having two uncertain characters in positions where the original values are near a discretization boundary [33].

Let  $m$  and  $n$  be the length of the pattern  $p$  and text  $t$ , respectively. The exact OPPM problem has a linear solution on the text length  $O(n+m \lg m)$  based on the Knuth-Morris-Pratt algorithm [41, 39, 22]. Alternative algorithms for the OPPM problem have also been proposed [21, 12, 20]. Contrasting with the large attention given to the resolution of the OPPM problem, to our knowledge there are no polynomial-time algorithms to solve the  $\mu$ OPPM problem. Naive algorithms for  $\mu$ OPPM assess all possible pattern and text assignments, bounded by  $O(nr^m)$  when considering up to  $r$  uncertain characters per position.

This work proposes the first polynomial algorithms able to answer the  $\mu$ OPPM problem. Accordingly, the contributions are organized as follows. First, we show that an indeterminate string of length  $m$  order-preserving matches a determinate string with the same length in  $O(mr \lg r)$  time based on their monotonic properties. Second, we show that  $\mu$ OPPM two indeterminate strings with the same length and  $r=2$  can be solved in  $O(m^2)$  time by reducing  $\mu$ OPPM to a 2-satisfiability task. Third, given a text string of length  $n$ , we show that the  $\mu$ OPPM problem is in polynomial time and linear space, and efficiently solved using effective filtration procedures.

## 2 Background

Let  $\Sigma$  be a totally ordered alphabet and an element of  $\Sigma^*$  be a string. The length of a string  $w$  is denoted by  $|w|$ . The empty string  $\varepsilon$  is a string of length 0. For a string  $w=xyz$ ,  $x$ ,  $y$  and  $z$  are called a prefix, substring, and suffix of  $w$ , respectively. The  $i$ -th character of a string  $w$  is denoted by  $w[i]$  for each  $1 \leq i \leq |w|$ . For a string  $w$  and integers  $1 \leq i \leq j \leq |w|$ ,  $w[i..j]$  denotes the substring of  $w$  from position  $i$  to position  $j$ . For convenience, let  $w[i..j]=\varepsilon$  when  $i > j$ .

Given strings  $x$  and  $y$  with equal length  $m$ ,  $y$  is said to order-preserving against  $x$  [41], denoted by  $x \approx y$ , if the orders between the characters of  $x$  and  $y$  are the same, i.e.



$x[i] \leq x[j] \Leftrightarrow y[i] \leq y[j]$  for any  $1 \leq i, j \leq m$ . A non-empty pattern string  $p$  is said to order-preserving match (*op-match* in short) a non-empty text string  $t$  iff there is a position  $i$  in  $t$  such that  $p \approx t[i - |p| + 1..i]$ . The *order-preserving pattern matching* (OPPM) problem is to find all such text positions.

## 2.1 The Problem

Given a totally ordered alphabet  $\Sigma$ , an indeterminate string is a sequence of disjunctive sets of characters  $x[1]x[2]..x[n]$  where  $x[i] \subseteq \Sigma$ . Each position is given by  $x[i] = \sigma_1.. \sigma_r$  where  $r \geq 1 \wedge \sigma_i \in \Sigma$ .

Given an indeterminate string  $x$ , a *valid assignment*  $\$x$  is a (determinate) string with a single character at position  $i$ , denoted  $\$x[i]$ , contained in the  $x[i]$  set of characters, i.e.  $\$x[1] \in x[1], \dots, \$x[m] \in x[m]$ . For instance, the indeterminate string  $(1|3, 3|4, 2|3, 1|2)$  has  $2^4$  valid assignments. Given an indeterminate position  $x[i] \subseteq \Sigma$ ,  $\$x_j[i]$  is the  $j^{\text{th}}$  ordered value of  $x[i]$  (e.g.  $\$x_0[i] = 1$  for  $x[i] = 1|2$ ). Given an indeterminate string  $x$ , let a *partially assigned* string  $\$x$  be an indeterminate string with an arbitrary number of uncertain characters removed, i.e.  $\$x[1] \subseteq x[1], \dots, \$x[m] \subseteq x[m]$ .

Given a determinate string  $x$  of length  $m$ , an indeterminate string  $y$  of equal length is said to be *order-preserving* against  $x$ , identically denoted by  $x \approx y$ , if there is a valid assignment  $\$y$  such that the relative orders of the characters in  $x$  and  $\$y$  are the same, i.e.  $x[i] \leq x[j] \Leftrightarrow \$y[i] \leq \$y[j]$  for any  $1 \leq i, j \leq m$ . Given two indeterminate strings  $x$  and  $y$  with length  $m$ ,  $y$  preserves the orders of  $x$ ,  $x \approx y$ , if exists  $\$y$  in  $y$  that respects the orders of a valid assignment  $\$x$  in  $x$ .

A non-empty indeterminate pattern string  $p$  is said to order-preserving match (*op-match* in short) a non-empty indeterminate text string  $t$  iff there is a position  $i$  in  $t$  such that  $p \approx t[i - |p| + 1..i]$ . The problem of *order-preserving pattern matching with character uncertainties* ( $\mu$ OPPM) problem is to find all such text positions.

To understand the complexity of the  $\mu$ OPPM problem, let us look to its solution from a naive stance yet considering state-of-the-art OPPM principles. The algorithmic proposal by Kubica et al. [41] is still up to this date the one providing a lowest bound,  $O(n+q)$ , where  $q=m$  for alphabets of size  $m^{O(1)}$  ( $q=m \lg m$  otherwise). Given a determinate string  $x$  of length  $m$ , an integer  $i$  ( $1 \leq i < m$ ) is said in the context of this work to be an *order-preserving border* of  $x$  if  $x[1..i] \approx x[m-i+1..m]$ . In this context, given a pattern string  $p$ , the orders between the characters of  $p$  are used to linearly infer the order borders. The order borders can then be used within the Knuth-Morris-Pratt algorithm to find op-matches against a text string  $t$  in linear time [41].

Given a determinate string  $p$  of length  $m$  and an indeterminate string  $t$  of length  $n$ , the previous approach is a direct candidate to the  $\mu$ OPPM problem by decomposing  $t$  in all its possible assignments,  $O(r^n)$ . Since determinate assignments to  $t$  are only relevant in the context of  $m$ -length windows, this approach can be improved to guarantee a maximum of  $O(r^m)$  assignments at each text position. Despite its simplicity, this solution is bounded by  $O(nr^m)$ . This complexity is further increased when indetermination is also considered in the pattern, stressing the need for more efficient alternatives.

## 2.2 Related work

The exact OPPM problem is well-studied in literature. Kubica et al. [41], Kim et al. [39] and Cho et al. [22] presented linear time solutions on the text length by respectively combining order-borders, rank-based prefixes and grammars with the Knuth-Morris-Pratt (KMP)

algorithm [40]. Cho et al. [21], Belazzougui et al. [12], and Chhabra et al. [20] presented  $O(nm)$  algorithms that show a sublinear average complexity by either combining bad character heuristics with the Boyer–Moore algorithm [13] or applying filtration strategies. Recently, Chhabra et al. [18] proposed further principles to solve OPPM using word-size packed string matching instructions to enhance efficiency.

In the context of numeric strings, multiple relaxations to the exact pattern matching problem have been pursued to guarantee that approximate matches are retrieved. In norm matching [7, 44, 1, 47], matches between numeric strings occur if a given distance threshold  $f(x, y) \leq \theta$  is satisfied. In  $(\delta, \gamma)$ -matching [14, 26, 24, 23, 42, 43, 45], strings are matched if the maximum difference of the corresponding characters is at most  $\delta$  and the sum of differences is at most  $\gamma$ .

In the context of nominal strings, variants of the pattern matching task have also been extensively studied to allow for don't care symbols in the pattern [37, 25, 9], transposition-invariant [42], parameterized matching [11, 6], less than matching [5], swapped matching [2, 46], gaps [15, 16, 31], overlap matching [4], and function matching [3, 8].

Despite the relevance of the aforementioned contributions to answer the exact order-preserving pattern matching and generic pattern matching, they cannot be straightforwardly extended to efficiently answer the  $\mu$ OPPM problem.

### 3 Polynomial time $\mu$ OPPM for equal length pattern and text

*Section 3.1* introduces the first efficient algorithm to solve the  $\mu$ OPPM problem when one string is indeterminate ( $r \in \mathbb{N}^+$ ). *Section 3.2* shows the existence of a polynomial solution when both strings are indeterminate and uncertainties are observed between pairs of characters ( $r=2$ ). Based on the reducibility of the graph coloring problem to the formulations proposed in *Section 3.2*, we hypothesize that op-matching indeterminate strings with an arbitrary number of uncertain characters per position ( $r \in \mathbb{N}^+$ ) is in class **NPC**. The proof of this intuition is, nevertheless, considered out of the scope, being regarded as future work.

#### 3.1 $O(mr \lg r)$ time $\mu$ OPPM when one string is indeterminate

Given a determinate string  $x$  of length  $m$ , there is a well-defined permutation of positions,  $\pi$ , that specifies a non-monotonic ascending order of characters in  $x$ . For instance, given  $x=(1,4,3,1)$ , then  $x[0]=x[3]<x[2]<x[1]$  and  $\pi=(0,3,2,1)$ . Given a determinate string  $y$  with the same length,  $y$  op-matches  $x$  if it  $y$  satisfies the same  $m-1$  orders. For instance, given  $x=(1,4,3,1)$  and  $y=(2,5,4,3)$ ,  $x$  orders are not preserved in  $y$  since  $y[0] \neq y[3] < y[2] < y[1]$ .

The monotonic properties can be used to answer  $\mu$ OPPM when one string is indeterminate. Given an indeterminate string  $y$ , let  $x_\pi$  and  $y_\pi$  be the permuted strings in accordance with  $\pi$  orders in  $x$ . To handle equality constraints, positions in  $y_\pi$  with identical characters in  $x_\pi$  can be intersected, producing a new string  $y'_\pi$  with  $s$  length ( $s \leq m$ ). Illustrating, given  $x=(4,1,4,2)$  and  $y=(2|7, 2, 7|8, 1|4|8)$ , then  $\pi=(1,3,0,2)$ ,  $x_\pi=(1,2,4,4)$ ,  $y_\pi=(2, 8|4|1, 7|2, 8|7)$  and  $y'_\pi=(y_\pi[0], y_\pi[1], y_\pi[2] \cap y_\pi[3])=(2, 8|4|1, 7)$ . To handle monotonic inequalities,  $y'_\pi[i]$  characters can be concatenated in descending order to compose  $z=y'_\pi[0]y'_\pi[1]..y'_\pi[s]$  and the orders between  $x$  and  $y$  verified by testing if the longest increasing subsequence (LIS) [29] of  $z$  has  $s$  length. In the given example,  $z=(2, 8, 4, 1, 7)$ , and the LIS of  $z=(2, 8, 4, 1, 7)$  is  $w=(2,4,7)$ . Since  $|w|=|y'_\pi|=3$ ,  $y$  op-matches  $x$ .

► **Theorem 1.** *Given a determinate string  $x$  and an indeterminate string  $y$ , let  $x_\pi$  and  $y_\pi$  be the sorted strings in accordance with  $\pi$  order of characters in  $x$ . Let the positions with equal*

**Algorithm 1:**  $O(mr \lg r)$   $\mu$ OPPM algorithm with one indeterminate string

---

```

Input: determinate  $x$ , indeterminate  $y$  ( $|x|=|y|=m$ )
 $\pi \leftarrow \text{sortedIndexes}(x);$  //  $O(m)$  if  $|\Sigma| = m^{\Omega(1)}$  ( $O(m \lg m)$  otherwise)
 $x_\pi \leftarrow \text{permute}(x, \pi), y_\pi \leftarrow \text{permute}(y, \pi);$  //  $O(m + mr)$ 
 $j \leftarrow 0; y'_\pi[0] \leftarrow \{y_\pi[0]\};$ 
foreach  $i \in 1..m-1$  do //  $O(mr \lg r)$ 
  | if  $x_\pi[i] = x_\pi[i-1]$  then  $y'_\pi[j] \leftarrow y'_\pi[j] \cup \{y_\pi[i]\};$  //  $O(r \lg r)$ 
  | else  $j \leftarrow j+1; y'_\pi[j] \leftarrow \{y_\pi[i]\};$ 
 $s \leftarrow |y'_\pi|, \text{nextMin} \leftarrow -\infty;$ 
foreach  $i \in 0..s-1$  do //  $O(mr)$ 
  |  $\text{nextMin} \leftarrow \min\{a \mid a \in y'_\pi[i], a > \text{nextMin}\};$  //  $O(r)$ 
  | if  $\exists \text{nextMin}$  then return false};
return true};

```

---

characters in  $x_\pi$  be intersected in  $y_\pi$  to produce a new indeterminate string  $y'_\pi$ . Consider  $z_i$  to be a string with  $y'_\pi[i]$  characters in descending order and  $z = z_1 z_2 \dots z_m$ , then:

$$|w| = |y'_\pi| \Leftrightarrow y \approx x \quad \text{where } w = \text{longest increasing subsequence in } z \quad (1)$$

**Proof.** ( $\Rightarrow$ ) If the length of the longest increasing subsequence (LIS),  $|w|$ , equals the number of monotonic relations in  $x$ ,  $|y'_\pi|$ , then  $y \approx x$ . By sorting characters in descending order per position, we guarantee that at most one character per position in  $y'_\pi$  appears in the LIS (respecting monotonic orders in  $x$  given  $y'_\pi$  properties). By intersecting characters in positions of  $y$  with identical characters in  $x$ , we guarantee the eligibility of characters satisfying equality orders in  $x$ , otherwise empty positions in  $y'_\pi$  are observed and the LIS length is less than  $|y'_\pi|$ . ( $\Leftarrow$ ) If  $|w| < |y'_\pi|$ , there is no assignment in  $y$  that op-matches  $x$  due to one of two reasons: 1) there are empty positions in  $y'_\pi$  due to the inability to satisfy equalities in  $x$ , or 2) it is not possible to find a monotonically increasing assignment to  $y'_\pi$  and, given the properties of  $y'_\pi$ ,  $y_\pi$  cannot preserve the orders of  $x_\pi$ .  $\blacktriangleleft$

Solving the LIS task on a string of size  $n$  is  $O(n \lg n)$  [29] where  $n = |z| = O(rm)$ . In addition, set intersection operations are performed  $O(m)$  times on sets with  $O(r)$  size, which can be accomplished in  $O(rm \lg r)$  time. As a result, the  $\mu$ OPPM problem with one indeterminate string can be solved in  $O(rm \lg(rm))$ .

Given the fact that the candidate string for the LIS task has properties of interest, we can improve the complexity of this calculus (*Theorem 2*) in accordance with Algorithm 1.

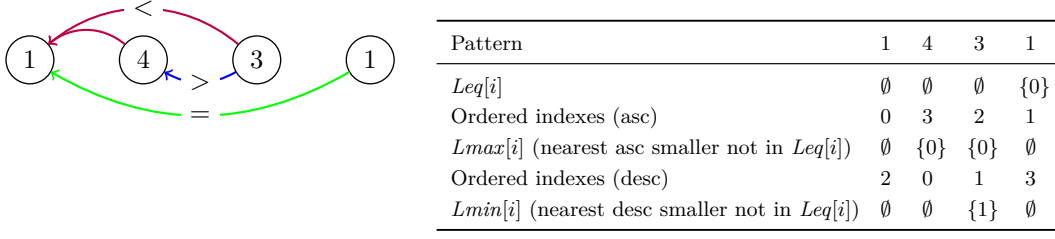
► **Theorem 2.**  $\mu$ OPPM two strings of length  $m$ , one being indeterminate, is in  $O(mr \lg r)$  time, where  $r \in \mathbb{N}^+$ .

**Proof.** In accordance with Algorithm 1,  $\mu$ OPPM is bounded by the verification of equalities,  $O(mr \lg r)$  [27]. Testing inequalities after set intersections can be linearly performed on the size of  $y$ ,  $O(mr)$  time, improving the  $O(mr \lg(mr))$  bound given by the LIS calculus.  $\blacktriangleleft$

The analysis of Algorithm 1 further reveals that the  $\mu$ OPPM problem with one indeterminate string requires linear space in the text length,  $O(mr)$ .

### 3.2 $O(m^2)$ time $\mu$ OPPM ( $r=2$ ) with indeterminate pattern and text

As indetermination in real-world strings is typically observed between pairs of characters [33], a key question is whether  $\mu$ OPPM on two indeterminate strings is in class **P** when  $r=2$ . To explore this possibility, new concepts need to be introduced. In OPPM research, character orders in a string of length  $m$  can be decomposed in 3 sequences with  $m$  unit sets:



■ **Figure 1** Orders identified for  $p=(1,4,3,1)$  in accordance with Kubica et al. [41].

► **Definition 3.** For  $i=0, \dots, m-1$ :

- $Leq_x[i] = \{\max\{k : k < i, x[i] = x[k]\}\}$  ( $\emptyset$  if there is no eligible  $k$ )
- $Lmax_x[i] = \{\max\{\text{argmax}_k\{x[k] : k < i, x[i] > x[k]\}\}\}$  ( $\emptyset$  if there is no eligible  $k$ )
- $Lmin_x[i] = \{\max\{\text{argmin}_k\{x[k] : k < i, x[i] < x[k]\}\}\}$  ( $\emptyset$  if there is no eligible  $k$ )

$Leq$ ,  $Lmax$  and  $Lmin$  capture  $=$ ,  $>$  and  $<$  relationships between each character  $x[i]$  in  $x$  and the closest preceding character  $x[k]$ . These orders can be inferred in linear time for alphabets of size  $m^{O(1)}$  and in  $O(m \lg m)$  time for other alphabets by answering the “all nearest smaller values” task on the sorted indexes [41]. Figure 1 depicts  $Leq$ ,  $Lmax$  and  $Lmin$  for  $x=(1,4,3,1)$ . Given determinate strings  $x$  and  $y$ ,  $A=Leq_x[t+1]$ ,  $B=Lmax_x[t+1]$  and  $C=Lmin_x[t+1]$ , if  $x[1..t] \approx y[1..t]$  then:

$$x[1..t+1] \approx y[1..t+1] \Leftrightarrow \forall a \in A (y[t+1] = y[a]) \wedge \forall b \in B (y[t+1] > y[b]) \wedge \forall c \in C (y[t+1] < y[c]). \quad (2)$$

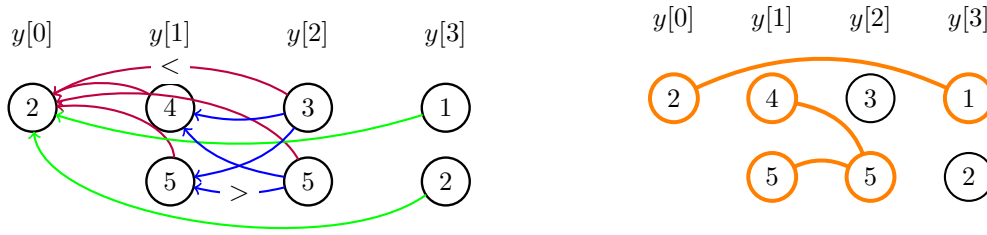
When allowing uncertainties between pairs of characters, previous research on the OPPM problem cannot be straightforwardly extended due to the need to trace  $O(2^m)$  assignments on indeterminate strings.

► **Lemma 4.** Given a determinate string  $x$ , an indeterminate string  $y$ , and the singleton sets  $A=Leq_x[t+1]$ ,  $B=Lmax_x[t+1]$  and  $C=Lmin_x[t+1]$  containing a position in  $1..t$ . If  $x[1..t] \approx y[1..t]$  is verified on a specific assignment of  $y$  characters, denoted  $\$y$ , then:

$$x[1..t+1] \approx y[1..t+1] \Leftrightarrow \exists_{\$y[t+1] \in \$y[t+1]} \forall_{a \in A} \exists_{\$y[a] \in \$y[a]} \forall_{b \in B} \exists_{\$y[b] \in \$y[b]} \forall_{c \in C} \exists_{\$y[c] \in \$y[c]} \\ \$y[t+1] = \$y[a] \wedge \$y[t+1] > \$y[b] \wedge \$y[t+1] < \$y[c]$$

**Proof.** ( $\Rightarrow$ ) In accordance with  $Leq$ ,  $Lmax$  and  $Lmin$  definition, for any  $a \in A$ ,  $b \in B$  and  $c \in C$  we have  $x[t+1]=x[a]$ ,  $x[t+1]>x[b]$  and  $x[t+1]<x[c]$ . If there is an assignment to  $y[1..t+1]$  in  $\$y$  that preserves the orders of  $x[1..t+1]$ , then for each  $a \in A$ ,  $b \in B$  and  $c \in C$   $\$y[t+1]=\$y[a]$ ,  $\$y[t+1]>\$y[b]$  and  $\$y[t+1]<\$y[c]$  (where  $\$y[t+1] \in \$y[t+1]$ ,  $\$y[a] \in \$y[a]$ ,  $\$y[b] \in \$y[b]$ ,  $\$y[c] \in \$y[c]$ ). ( $\Leftarrow$ ) We need to show that  $x[1..t+1] \approx y[1..t+1]$ . Since  $x[1..t] \approx y[1..t]$ , for  $i < t$ ,  $\exists_{\$y[i] \in \$y[i], \$y[t+1] \in \$y[t+1]}: x[t+1]>x[i] \Leftrightarrow \$y[t+1]>\$y[i]$ . Assuming  $x[t+1]>x[i]$  for some  $i \in \{1..t\}$ : by the definition of  $Lmax$ ,  $\forall_{b \in B} x[b]>x[i]$ ; by the order-isomorphism of  $x[1..t]$  and  $\$y[1..t]$  in  $\$y[1..t]$ , there is  $\$y[i] \in \$y[i]$  and  $\$y[b] \in \$y[b]$  that  $\forall_{b \in B} \$y[b]>\$y[i]$ ; and by the assumption of the lemma,  $\forall_{b \in B} \$y[t+1]>\$y[b]$ ; hence  $\$y[t+1]>\$y[i]$ . Similarly,  $x[t+1]<x[i]$  (and  $x[t+1]=x[i]$ ) implies  $\$y[t+1]<\$y[i]$  (and  $\$y[t+1]=\$y[i]$ ), yielding the stated equivalence. ◀

Given two strings of equal length, the  $\mu$ OPPM problem can be schematically represented according to the identified order restrictions. Figure 2 represents restrictions on the indeterminate string  $y=(2,4|5,3|5,1|2)$  in accordance with the observed orders in  $x=(1,4,3,1)$ .



■ **Figure 2** Schematic representation of the pairwise ordering restrictions for text  $y=(2,4|5,3|5,1|2)$  and pattern  $x=(1,4,3,1)$ . In the left side, all order verifications are represented, while in the right side only the order conflicts are signaled (e.g.  $y[1]=4$  cannot be selected together with  $y[2]=5$ ).

The left side edges are placed in accordance with *Lemma 4* and capture assessments on the orders between pairs of characters. The right side edges capture incompatibilities detected after the assessments, i.e. pairs of characters that cannot be selected simultaneously (for instance,  $y[0]=2$  and  $y[3]=1$ , or  $y[1]=4$  and  $y[2]=5$ ). For the given example, there are two valid assignments,  $\$y_1=(2,4,3,2)$  and  $\$y_2=(2,5,4,2)$ , that satisfy  $x[0]=x[3]<x[2]<x[1]$ , thus  $y$  op-matches  $x$ .

To verify whether there is an assignment that satisfies the identified ordering restrictions, we propose the reduction of  $\mu$ OPPM problem to a Boolean satisfiability problem.

Given a set of Boolean variables, a formula in conjunctive normal form is a conjunction of clauses, where each clause is a disjunction of literals, and a literal corresponds to a variable or its negation. Let a 2CNF formula be a formula in the conjunctive normal form with at most two literals per clause. Given a CNF formula, the *satisfiability* (SAT) problem is to verify if there is an assigning of values to the Boolean variables such that the CNF formula is satisfied.

► **Theorem 5.** *The  $\mu$ OPPM problem over two strings of equal length, one being indeterminate, can be reduced to a satisfiability problem with the following CNF formula:*

$$\begin{aligned}
 \phi = & \bigwedge_{i=0}^{m-1} \left( \bigvee_{\$y[i] \in y[i]} z_{i, \$y[i]} \right) \\
 & \wedge \bigwedge_{i=0}^{m-1} \left( \bigwedge_{\$y[i] \in y[i]} \bigwedge_{j \in \text{Leq}[i], \$y[j] \in y[j]} (\neg z_{i, \$y[i]} \vee \neg z_{j, \$y[j]} \vee \$y[i] = \$y[j]) \right. \\
 & \wedge \bigwedge_{\$y[i] \in y[i]} \bigwedge_{\substack{j \in \text{Lmax}[i] \\ \$y[j] \in y[j]}} (\neg z_{i, \$y[i]} \vee \neg z_{j, \$y[j]} \vee \$y[i] > \$y[j]) \\
 & \left. \wedge \bigwedge_{\$y[i] \in y[i]} \bigwedge_{\substack{j \in \text{Lmin}[i] \\ \$y[j] \in y[j]}} (\neg z_{i, \$y[i]} \vee \neg z_{j, \$y[j]} \vee \$y[i] < \$y[j]) \right) \quad (3)
 \end{aligned}$$

**Proof.** Let us show that if  $x$  op-matches  $y$  then  $\phi$  is satisfiable, and if  $x$  does not op-match  $y$  then  $\phi$  is not satisfiable. ( $\Rightarrow$ ) When  $x \approx y$ , there is an assignment of values to  $y$ ,  $\$y$ , that satisfy the orderings of  $x$ .  $\phi$  is satisfiable if there is at least one variable assigned to true per clause  $\bigvee_{\$y[i] \in y[i]} z_{i, \$y[i]}$  given conflicts  $\neg z_{i, \$y[i]} \vee \neg z_{j, \$y[j]}$ . As conflicts do not prevent the existence of a valid assignment (by assumption), then  $\exists_{\$y} \bigwedge_{i \in \{0..m-1\}} z_{i, \$y[i]}$  and  $\phi$  is satisfiable. ( $\Leftarrow$ ) When  $x$  does not op-match  $y$ , there is no assignment of values  $\$y \in y$  that

can satisfy the orders of  $x$ . Per formulation, the conflicts  $\neg z_{i,\$y[i]} \vee \neg z_{j,\$y[j]}$  prevent the satisfiability of one or more clauses  $\vee_{\$y[i] \in y[i]} z_{i,\$y[i]}$ , leading to a non-satisfiable formula. ◀

If the established  $\phi$  formula is satisfiable, there is a Boolean assignment to the variables that specify an assignment of characters in  $y$ ,  $\$y$ , preserving the orders of  $x$  (as defined by  $Leq$ ,  $Lmax$  and  $Lmin$ ). Otherwise, it is not possible to select an assignment  $\$y$  op-matching  $x$ .  $\phi$  has at most  $r \times m$  variables,  $\{z_{i,\sigma} \mid i \in \{0..m-1\}, \sigma \in \Sigma\}$ . The Boolean value assigned to a variable  $z_{i,\sigma}$  simply defines that the associated character  $\sigma$  from  $y[i]$  can be either considered (when true) or not (when false) to compose a valid assignment  $\$y$  that op-matches the given determinate string  $x$ . The reduced (3) formula is composed of two major types of clauses:  $\vee_{\$y[i] \in y[i]} z_{i,\$y[i]}$ , and  $(\neg z_{i,\$y[i]} \vee \neg z_{j,\$y[j]} \vee \mathbf{bool})$  where  $\mathbf{bool}$  is either given by  $\$y[i] = \$y[j]$ ,  $\$y[i] < \$y[j]$  or  $\$y[i] > \$y[j]$ . Clauses of the first type specify the need to select at least one character per position in  $y$  to guarantee the presence of valid assignments. The remaining clauses specify ordering constraints between characters. If an inequality, such as  $\$y[i] > \$y[j]$ , is assessed as true, the associated clause is removed. Otherwise,  $(\neg z_{i,\sigma_1} \vee \neg z_{j,\sigma_2})$  is derived, meaning that these  $\sigma_1$  and  $\sigma_2$  characters should not be selected simultaneously since they do not satisfy the orders defined by a given pattern. For instance, the pairs of characters in orange from Figure 2 should not be simultaneously selected due to order conflicts. To this end,  $(\neg z_{0,2} \vee \neg z_{3,1})$  and  $(\neg z_{1,4} \vee \neg z_{2,5})$  clauses need to be included to verify if  $y \approx x$ . Considering  $y=(2, 4|5, 4|5, 1|2)$  and  $x=(1,4,3,1)$ , schematically represented in Figure 2, the associated CNF formula is:

$$\phi = z_{0,2} \wedge (z_{1,4} \vee z_{1,5}) \wedge (z_{2,4} \vee z_{2,5}) \wedge (z_{3,1} \vee z_{3,2}) \wedge (\neg z_{0,2} \vee \neg z_{3,1}) \wedge (\neg z_{1,4} \vee \neg z_{2,5})$$

► **Theorem 6.** *Given two strings of length  $m$ , one being indeterminate with  $r=2$ , the  $\mu$ OPPM problem can be reduced to a 2SAT problem with a CNF formula with  $O(m)$  size.*

**Proof.** Given *Theorem 5* and the fact that the reduced CNF formula has at most two literals per clause –  $\phi$  is a composition of  $\vee_{\$y[i] \in y[i]} z_{i,\$y[i]}$  clauses with  $|y[i]| \in \{1, 2\}$  and  $(\neg z_{i,\$y[i]} \vee \neg z_{j,\$y[j]} \vee \mathbf{bool})$  clauses –  $\mu$ OPPM with  $r=2$  and one indeterminate string is reducible to 2SAT. The reduced formula has at most  $10m$  clauses with 2 literals each, being linear in  $m$ :

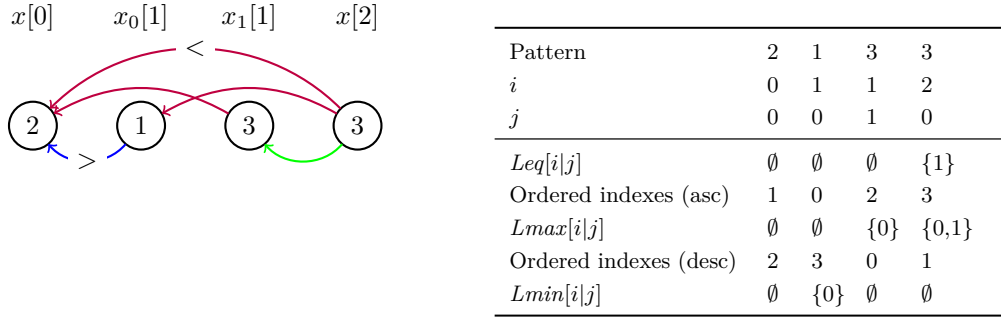
- [clauses that impose the selection of at least one character per position in  $y$ ] Since  $y$  has  $m$  positions, and each position is either determinate (unitary clause) or defines an uncertainty between a pair of characters, there are  $m$  clauses and at most  $2m$  literals;
- [clauses that define the ordering restrictions between two variables] A position in the indeterminate string  $y[i]$  needs to satisfy at most two order relations. Considering that  $i$ ,  $Leq[i]$ ,  $Lmax[i]$  and  $Lmin[i]$  specify uncertainties between pairs of characters, there are up to 12 restrictions per position: 4 ordering restrictions between characters in  $y[i]$  and  $y[Leq[i]]$ ,  $y[Lmax[i]]$  and  $y[Lmin[i]]$ . Whenever the order between two characters is not satisfied, a clause is added per position, leading to at most  $12m$  clauses. ◀

► **Theorem 7.** *The  $\mu$ OPPM between determinate and indeterminate strings of equal length can be solved in linear time when  $r=2$ .*

**Proof.** Given the fact that a 2SAT problem can be solved in linear time [10]<sup>1</sup>, this proof directly derives from *Theorem 6* as it guarantees the soundness of reducing  $\mu$ OPPM ( $r=2$ ) to a 2SAT problem with a CNF formula with  $O(m)$  size. ◀

<sup>1</sup> 2SAT problems have linear time and space solutions on the size of the input formula. Consider for instance the original proposal [10], the formula  $\phi$  is modeled by a directed graph  $G=(V, E)$ , with two nodes per variable  $z_i$  in  $\phi$  ( $z_i$  and  $\neg z_i$ ) and two directed edges for each clause  $z_i \vee z_j$  (the equivalent





■ **Figure 3** Order relationships of  $x=(2, 1|3, 3)$  and the corresponding  $Lmax$  and  $Lmin$  vectors.

As the size of the mapped CNF formula  $\phi$  is  $O(m)$  and the a valid algorithm to verify its satisfiability would require the construction of a graph with  $O(m)$  nodes and edges, the required memory for the target  $\mu$ OPPM problem is  $\Theta(m)$ .

When moving from one to two indeterminate strings, previous contributions are insufficient to answer the  $\mu$ OPPM problem. In this context, the  $Leq$ ,  $Lmax$  and  $Lmin$  vectors need to be redefined to be inferred from an indeterminate string:

► **Definition 8.**

- $Leq_x[i|j]=\{k : k < i, \exists_p \$x_j[i]=\$x_p[k]\}$  ( $\emptyset$  if there is no eligible  $k$ ), for  $i=0, \dots, m-1$
- $Lmax_x[i|j]=\{k : k < i, \exists_p \$x_j[i] > \$x_p[k]\}$  ( $\emptyset$  if there is no eligible  $k$ ), for  $i=0, \dots, m-1$
- $Lmin_x[i|j]=\{k : k < i, \exists_p \$x_j[i] < \$x_p[k]\}$  ( $\emptyset$  if there is no eligible  $k$ ), for  $i=0, \dots, m-1$

Figure 3 schematically represents the order relationships of  $x=(2, 1|3, 3)$  and the associated  $Leq$ ,  $Lmax$  and  $Lmin$  vectors. In this scenario,  $x[2]$  needs to be verified not only against  $x_0[1]$  but also against  $x_1[1]$  in case  $x_0[1]$  is disregarded. Understandably, due to character uncertainties,  $O(m^2)$  ordering verifications are required (Def.8).

► **Lemma 9.** Given indeterminate strings  $x$  and  $y$ , let  $A_j=Leq_x[t+1|j]$ ,  $B_j=Lmax_x[t+1|j]$  and  $C_j=Lmin_x[t+1|j]$  (Def.8) be the orders associated with  $\$x_j[t+1]$ . If  $x[1..t] \approx y[1..t]$  is verified on a partial assignment of  $y$  characters, denoted by  $\S y$ , then:

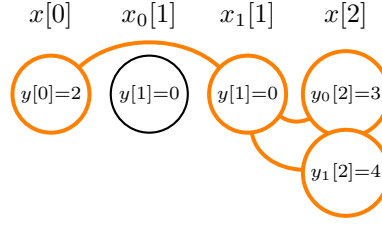
$$x[1..t+1] \approx y[1..t+1] \Leftrightarrow \exists_{j \in \{0,1\}} \exists_{\$y[t+1] \in \S y[t+1]} \forall_{a \in A_j, b \in B_j, c \in C_j} \exists_{\$y[a] \in \S y[a], \$y[b] \in \S y[b], \$y[c] \in \S y[c]} (\$y[t+1] = \$y[a] \wedge \$y[t+1] > \$y[b] \wedge \$y[t+1] < \$y[c])$$

**Proof.** ( $\Rightarrow$ ) Similar to the proof of Lemma 4, yet  $A$ ,  $B$  and  $C$  conditional to  $x[t+1]$  (Def.3) are now given by  $A_j$ ,  $B_j$  and  $C_j$  conditional to  $x_j[t+1]$  (Def.8). If there is an assignment to  $y[1..t+1]$  in  $\S y$  that preserves one of the possible orders in  $x[1..t+1]$ , then for any  $a \in A_j$ ,  $b \in B_j$  and  $c \in C_j$ :  $\$y[t+1]=\$y[a]$ ,  $\$y[t+1]>\$y[b]$  and  $\$y[t+1]<\$y[c]$  (where  $\$y[t+1] \in \S y[t+1]$ ,  $\$y[a] \in \S y[a]$ ,  $\$y[b] \in \S y[b]$ ,  $\$y[c] \in \S y[c]$ ).

( $\Leftarrow$ ) We need to show that  $x[1..t+1] \approx y[1..t+1]$ . Since  $x[1..t] \approx y[1..t]$ , it is sufficient to prove that for  $i \leq t$ : exists  $\$x[i] \in \S x[i]$ ,  $\$x[t+1] \in \S x[t+1]$ ,  $\$y[i] \in \S y[i]$ ,  $\$y[t+1] \in$

---

implicative forms  $\neg z_i \Rightarrow z_j$  and  $\neg z_j \Rightarrow z_i$ ). Given  $G$ , the strongly connected components (SCCs) of  $G$  can be discovered in  $O(|V| + |E|)$ . During the traversal if a variable and its complement belong to the same SCC, then the procedure stops as  $\phi$  is determined to be unsatisfiable. Given the fact that both  $V=O(m)$  and  $E=O(m)$  by Lemma 6, this procedure is  $O(m)$  time and space.



■ **Figure 4** Conflicts when op-matching  $y=(2,0,3|4)$  against  $x=(2,1|3,3)$ .

$\$y[t+1]$  such that  $\$x[t+1]=\$x[i] \Leftrightarrow \$y[t+1]=\$y[i]$ ,  $\$x[t+1]>\$x[i] \Leftrightarrow \$y[t+1]>\$y[i]$  and  $\$x[t+1]<\$x[i] \Leftrightarrow \$y[t+1]<\$y[i]$ . This results from Def.8, the order-isomorphism property and Lemma 4. ◀

Figure 4 represents encountered restrictions when op-matching  $x=(2,1|3,3)$  against  $y=(2,0,3|4)$ . The right side edges capture the detected incompatibilities, i.e. pairs of characters that cannot be selected simultaneously. For the given example, there are 2 valid assignments –  $\$y_1=(2,0,3)$  and  $\$y_2=(2,0,4)$  – satisfying  $\$x_0[1]<\$x_0[0]<\$x_0[2]$ , thus  $x \approx y$ .

To verify whether there is an assignment that satisfies the identified ordering restrictions, Theorem 10 extends the previously introduced SAT mapping given by (3).

► **Theorem 10.** *Given  $Lmax$  and  $Lmin$  (Def.8),  $\mu OPPM$  problem over two indeterminate strings of equal length can be reduced to a satisfiability problem with the following CNF formula:*

$$\phi = \bigwedge_{x[i] \in x \wedge |x[i]|=1} w_{i,x_0[i]} \wedge \bigwedge_{x[i] \in x \wedge |x[i]|>1} \left( \left( \bigvee_{\$x[i] \in x[i]} w_{i,\$x[i]} \right) \wedge \left( \bigvee_{\$x[i] \in x[i]} \neg w_{i,\$x[i]} \right) \right) \quad (4.1)$$

$$\wedge \bigwedge_{i=0}^{m-1} \left( \bigwedge_{\$x[i] \in x[i]} \left( \bigvee_{\$y[i] \in y[i]} z_{i,\$x[i],\$y[i]} \right) \wedge \left( \bigvee_{\$y[i] \in y[i]} \neg z_{i,\$x[i],\$y[i]} \right) \right) \quad (4.2)$$

$$\wedge \bigwedge_{i=0}^{m-1} \left( \left( \bigwedge_{\$x[i] \in x[i]} \bigwedge_{\$y[i] \in y[i]} \left( \neg z_{i,\$x[i],\$y[i]} \vee w_{i,\$x[i]} \right) \right) \right) \quad (4.3)$$

$$\wedge \bigwedge_{i=0}^{m-1} \bigwedge_{\$y[i] \in y[i], \$x[i] \in x[i]} \left( \bigwedge_{j \in Leq[i]} \bigwedge_{\$y[j] \in y[j], \$x[j] \in x[j]} \left( \neg z_{i,\$x[i],\$y[i]} \vee \neg z_{j,\$x[j],\$y[j]} \vee \$y[i] = \$y[j] \right) \right. \\ \wedge \bigwedge_{j \in Lmax[i]} \bigwedge_{\substack{\$y[j] \in y[j] \\ \$x[j] \in x[j]}} \left( \neg z_{i,\$x[i],\$y[i]} \vee \neg z_{j,\$x[j],\$y[j]} \vee \$y[i] > \$y[j] \right) \\ \left. \wedge \bigwedge_{j \in Lmin[i]} \bigwedge_{\substack{\$y[j] \in y[j] \\ \$x[j] \in x[j]}} \left( \neg z_{i,\$x[i],\$y[i]} \vee \neg z_{j,\$x[j],\$y[j]} \vee \$y[i] < \$y[j] \right) \right) \quad (4.4)$$

**Proof.** If  $x \approx y$  then  $\phi$  is satisfiable, and if  $x$  does not op-match  $y$  then  $\phi$  is not satisfiable.

( $\Rightarrow$ ) When  $x$  op-matches  $y$ , there is an assignment of values in  $x$  and  $y$  such that  $\$x \approx \$y$ .  $\phi$  is satisfiable if there is one and only one variable  $w_{i,\$x[i]}$  per  $i^{th}$  position (4.1). This occurs iff one of the variables  $z_{i,\$x[i],\$y[i]}$  is true for a given  $i^{th}$  position in accordance with (4.2) and



(4.3). As conflicts (4.4) do not prevent the existence of a valid assignment (by assumption), one or more variables  $z_{i,\$x[i],\$y[i]}$  can be selected per position.  $\phi$  can then be satisfied by fixing a single variable  $z_{i,\$x[i],\$y[i]}$  per  $i^{th}$  position as **true** and the remaining variables as **false**. Given (4.3) equivalences between  $w_{i,\$x[i]}$  and  $z_{i,\$x[i],\$y[i]}$  variables,  $\phi$  is consequently satisfiable.

( $\Leftarrow$ ) When  $x$  does not op-match  $y$ , there is no assignment of values  $\$x \in x$  and  $\$y \in y$  such that  $\$x \approx \$y$ . In this context, the conflicts (4.4) can prevent the satisfiability of clauses (4.2) or (4.1), thus leading to an unsat formula. Per formulation, in the absence of an order-preserving match, conflicts will prevent the assignment of at least one variable  $z_{i,\$x[i],\$y[i]}$  on compatible  $(i, \$x[i])$  pairs, which are necessarily shown as conflicts on (4.1) clauses as a consequence of the assignment constraints placed by (4.2) and (4.3) clauses.  $\blacktriangleleft$

If the formula is satisfiable, there is a Boolean assignment to the variables such that there is an assignment of characters in  $y$ ,  $\$y$ , and in  $x$ ,  $\$x$ , such that both strings op-match. Otherwise, it is not possible to select assignments such that  $x \approx y$ . Given  $r=2$ , the established  $\phi$  formula has at most  $6m$  Boolean variables: a) at most  $2m$  variables of the type  $\{w_{i,\sigma} \mid i \in \{0..m-1\}, \sigma \in \Sigma\}$  corresponding to characters in  $x$ ; and b) at most  $4m$  variables of the type  $\{z_{i,\sigma_1,\sigma_2} \mid i \in \{0..m-1\}, \sigma_1, \sigma_2 \in \Sigma\}$  defining combinations of characters in the  $i^{th}$  position of  $x$  and  $y$ . Boolean values assigned to variables  $w_{i,\sigma}$  are used to find a valid assignment of characters in  $x$ . Boolean values assigned to variables  $z_{i,\sigma_1,\sigma_2}$  define whether characters  $\sigma_1 \in x[i]$  and  $\sigma_2 \in y[i]$  belong to an op-match. The reduced formula is composed of four major types of clauses:

- (4.1) a single character per  $x$  position should be selected if exists  $\$x$  such that  $\$x \approx y$ ;
- (4.2) a single character per indeterminate  $y$  position should be selected if there is a valid assignment  $\$y$  such that  $\$x \approx \$y$ , where  $\$x$  is given by assignments to (4.1) clauses;
- (4.3) clauses that guarantee an association between  $x$  and  $y$ :  $z_{i,\$x[i],\$y[i]} \Rightarrow w_{i,\$x[i]}$ ;
- (4.4) clauses specify ordering constraints between pairs of characters  $\sigma_1 \in y[i]$  and  $y[Leq[i]]$ ,  $y[Lmax[i]]$  and  $y[Lmin[i]]$ . If the inequalities  $\$y[i]=\$y[j]$ ,  $\$y[i]>\$y[j]$  and  $\$y[i]<\$y[j]$  are assessed as **false**, these leads to clauses of the form  $(\neg z_{i,\sigma_1} \vee \neg z_{j,\sigma_2})$ , meaning that these characters should not be selected simultaneously in the given positions (see Figure 4).

To instantiate the proposed mapping, consider  $x=(2, 1|3, 3)$  and  $y=(2, 0, 3|4)$ , schematically represented in Figure 3. The associated CNF formula is:

$$\begin{aligned} \phi = & w_{0,2} \wedge (w_{1,1} \vee w_{1,3}) \wedge (\neg w_{1,1} \vee \neg w_{1,3}) \wedge w_{2,3} \quad // (4.1) \text{ one valid assignment to } x \\ & \wedge (z_{2,3,3} \vee z_{2,3,4}) \wedge (\neg z_{2,3,3} \vee \neg z_{2,3,4}) \quad // (4.2) \text{ assignment to indeterminate } y \text{ positions} \\ & \wedge (\neg z_{0,2,2} \vee w_{0,2}) \wedge (\neg z_{1,1,0} \vee w_{1,1}) \wedge (\neg z_{1,3,0} \vee w_{1,3}) \wedge (\neg z_{2,3,3} \vee w_{2,3}) \\ & \quad \wedge (\neg z_{2,3,4} \vee w_{2,3}) \quad // (4.3) \text{ implications between } x \text{ and } y: z_{i,\$x[i],\$y[i]} \Rightarrow w_{i,\$x[i]} \\ & \wedge (\neg z_{0,0,2} \vee \neg z_{1,3,0}) \wedge (\neg z_{1,3,0} \vee \neg z_{2,3,3}) \wedge (\neg z_{1,3,0} \vee \neg z_{2,3,4}) \quad // 4.4 \text{ character conflicts} \end{aligned}$$

► **Theorem 11.** *When  $r=2$ , the  $\mu$ OPPM problem for two indeterminate strings of equal length is reducible to a 2-satisfiability problem over a CNF formula with  $O(m^2)$  size.*

**Proof.** The reduced formula (4) is in the two conjunctive normal form (2CNF). (4.1), (4.2) and (4.3) clauses have at most two literals given  $r=2$ . (4.4) clauses contain inequalities dynamically assigned to **true** or **false** during the reduction phase, producing clauses with at most two literals. There are at most  $2m$  clauses given by (4.1) as  $x$  has at most 2 characters per position; and at most  $4m$  clauses given by (4.2) (as well as  $4m$  clauses given by (4.3)) resulting from the combination of possible characters from a position in  $x$  and  $y$ . Since there is a maximum of  $O(m)$  orders per position, there can be at most  $O(m^2)$  order conflicts between characters and thus  $O(m^2)$  clauses given by (4.4) of the form  $(\neg z_{i,\$x[i],\$y[i]} \vee \neg x_{j,\$x[j],\$y[j]})$ .  $\blacktriangleleft$

► **Theorem 12.**  $\mu$ OPPM indeterminate strings of equal length is in  $O(m^2)$  time when  $r=2$ .

**Proof.** Given *Theorem 11* and the ability to solve 2SAT tasks linearly in the size of the CNF formula [10], the proof of this theorem follows naturally. ◀

As linear time algorithms to solve 2SAT problems require linear space (see appendix) and the size of the mapped satisfiability formula  $\phi$  is  $O(m^2)$  (*Theorem 11*), the memory complexity of the  $\mu$ OPPM problems between indeterminate strings with  $r=2$  and equal length is  $O(m^2)$ .

#### 4 Polynomial time $\mu$ OPPM

► **Lemma 13.** Given a pattern string of length  $m$  and a text string of length  $n$ , one being indeterminate, the  $\mu$ OPPM problem can be solved in  $O(nmr \lg r)$  time. When both the pattern and text are indeterminate with  $r=2$ , the  $\mu$ OPPM problem can be solved in  $O(nm^2)$  time.

**Proof.** From *Lemmas 7* and *12*: verifying if two strings of length  $m$  op-match can be either done in  $O(mr \lg r)$  time (indetermination in one string) or  $O(m^2)$  time (indetermination on both strings and  $r=2$ ). At most  $n-m+1$  verifications need to be performed. ◀

*Lemma 13* confirms that the  $\mu$ OPPM problem with one indeterminate strings or uncertainties between characters ( $r=2$ ) is in class **P**. This lemma further triggers the research question “Are  $O(nmr)$  and  $O(nm^2)$  tight bounds to solve the  $\mu$ OPPM?”, here left as an open research question.

Irrespectively of the answer, the analysis of the average complexity is of complementary relevance. State-of-the-art research on the exact OPPM problem shows that the average performance of algorithms in  $O(nm)$  time can outperform linear algorithms [20, 17, 19].

Motivated by the evidence gathered by these works, we suggest the use of filtration procedures to improve the average complexity of the proposed  $\mu$ OPPM algorithm while still preserving its complexity bounds. A filtration procedure encodes the input pattern and text, and relies on this encoding to efficiently find positions in the text with a high likelihood to op-match a given pattern. Despite the diversity of string encodings, simplistic binary encodings are considered to be the state-of-the-art in OPPM research [20, 17]. In accordance with Chhabra et al. [20], a pattern  $p$  can be mapped into a binary string  $p'$  expressing increases (1), equalities (0) and decreases (0) between subsequent positions. By searching for exact pattern matches of  $p'$  in an analogously transformed text string  $t'$ , we guarantee that the verification of whether  $p[0..m-1]$  and  $t[i..i+m-1]$  orders are preserved is only performed when exact binary matches occur. Illustrating, given  $p=(3,1,2,4)$  and  $t=(2,4,3,5,7,1,4,8)$ , then  $p'=(1,0,1,1)$  and  $t'=(1,1,0,1,1,0,1,1,0)$ , revealing two matches  $t'[1..4]$  and  $t'[4..7]$ : one spurious match  $t[1..5]$  and one true match  $t[4..8]$ .

When handling indeterminate strings the concept of increase, equality and decrease needs to be redefined. Given an indeterminate string  $x$ , consider  $x'[i]=1$  if  $\max(x[i]) < \min(x[i+1])$ ,  $x'[i]=0$  if  $\min(x[i]) \geq \max(x[i+1])$ , and  $x'[i]=*$  otherwise. Under this encoding, the pattern matching problem is identical under the additional guard that a character in  $p'$  always matches a don't care position,  $t'[i]=*$ , and vice-versa. Illustrating, given  $p=(6,2|3,5)$  and  $t=(3|4,5,6|8,6|7,3,5,4|6,7|8,4)$ , then  $p'=(0,1)$  and  $t'=(11*01*10)$ , leading to one true match  $t[3..5]$  – e.g.  $\$t[3..5]=(6,3,5)$  – and one spurious match  $t[5..7]$ . Exact pattern matching algorithms, such as Knuth-Morris-Pratt and Boyer-Moore, can be adapted to consider don't care positions while preserving complexity bounds [40, 13].

The properties of the proposed encoding guarantee that the exact matches of  $p'$  in  $t'$  cannot skip any op-match of  $p$  in  $t$ . Thus, when combining the premises of *Lemma 13* with the previous observation, we guarantee that the computed  $\mu$ OPPM solution is sound.

The application of this simple filtration procedure prevents the recurring  $O(mr \lg r)$  or  $O(m^2)$  verifications  $n-m+1$  times. Instead, the complexity of the proposed method to solve the  $\mu$ OPPM problem becomes  $O(dmr \lg r + n)$  (when one string is indeterminate) or  $O(dm^2 + n)$  (when both strings are indeterminate and  $r=2$ ) where  $d$  is the number of exact matches ( $d \ll n$ ). According to previous work on exact OPPM with filtration procedures [20], SBNDM2 and SBNDM4 algorithms [28] (Boyer-Moore variants) were suggested to match binary encodings. In the presence of small patterns, Fast Shift-Or (FSO) [30] can be alternatively applied [20].

A given string text can be read and encoded incrementally from the standard input as needed to perform  $\mu$ OPPM, thus requiring  $O(mr)$  space. When filtration procedures are considered, the aforementioned algorithms for exact pattern matching require  $O(m)$  space [20], thus  $\mu$ OPPM space requirements are bound by substring verifications (*Section 3*):  $O(mr)$  space when one string is indeterminate and  $O(m^2)$  when indetermination is considered on both strings and  $r=2$ .

## 5 Concluding remark

This work addressed the relevant yet scarcely studied problem of finding order-preserving pattern matches on indeterminate strings ( $\mu$ OPPM). We showed that the problem has a polynomial solution when uncertainties are verified between two characters by reducing the  $\mu$ OPPM problem to a 2-satisfiability problem. To this end, we first demonstrated that the problem of matching two strings with equal length can be solved in linear time and space when considering indetermination in one string and in quadratic time when considering indetermination on both the pattern and text strings. Finally, we showed that the  $\mu$ OPPM problem can be efficiently solved in polynomial time by combining the proposed verifications with filtration procedures.

---

## References

- 1 Amihood Amir, Yonatan Aumann, Piotr Indyk, Avivit Levy, and Ely Porat. Efficient computations of  $l_1$  and  $l_{\infty}$  rearrangement distances. *Theor. Comput. Sci.*, 410(43):4382–4390, 2009. doi:10.1016/j.tcs.2009.07.019.
- 2 Amihood Amir, Yonatan Aumann, Gad M. Landau, Moshe Lewenstein, and Noa Lewenstein. Pattern matching with swaps. *J. Algorithms*, 37(2):247–266, 2000. doi:10.1006/jagm.2000.1120.
- 3 Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.
- 4 Amihood Amir, Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. Overlap matching. *Inf. Comput.*, 181(1):57–74, 2003. doi:10.1016/S0890-5401(02)00035-4.
- 5 Amihood Amir and Martin Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Inf. Comput.*, 118(1):1–11, 1995. doi:10.1006/inco.1995.1047.
- 6 Amihood Amir, Martin Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994. doi:10.1016/0020-0190(94)90086-8.
- 7 Amihood Amir, Ohad Lipsky, Ely Porat, and Julia Umanski. Approximate matching in the  $l_1$  metric. In *CPM*, volume 5, pages 91–103. Springer, 2005.

- 8 Amihoud Amir and Igor Nor. Generalized function matching. *J. Discrete Algorithms*, 5(3):514–523, 2007. doi:10.1016/j.jda.2006.10.001.
- 9 Alberto Apostolico. General pattern matching. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook*, pages 15–15. Chapman & Hall/CRC, 2010.
- 10 Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- 11 Brenda S Baker. A theory of parameterized pattern matching: algorithms and applications. In *ACM symposium on Theory of computing*, pages 71–80. ACM, 1993.
- 12 Djamel Belazzougui, A. Pierrot, M. Raffinot, and Stéphane Vialette. Single and multiple consecutive permutation motif search. In *Int. Symposium on Algorithms and Computation*, pages 66–77. Springer, 2013.
- 13 Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
- 14 Emilios Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Yoan Pinzon. Algorithms for computing approximate repetitions in musical sequences. *Int. Journal of Computer Mathematics*, 79(11):1135–1148, 2002.
- 15 Domenico Cantone, Salvatore Cristofaro, and Simone Faro. An efficient algorithm for  $\delta$ -approximate matching with  $\alpha$ -bounded gaps in musical sequences. In *IW on Experimental and Efficient Algorithms*, pages 428–439. Springer, 2005.
- 16 Domenico Cantone, Salvatore Cristofaro, and Simone Faro. On tuning the  $(\delta, \alpha)$ -sequential-sampling algorithm for  $\delta$ -approximate matching with alpha-bounded gaps in musical sequences. In *ISMIR*, pages 454–459, 2005.
- 17 Domenico Cantone, Simone Faro, and M Oguzhan Külekci. An efficient skip-search approach to the order-preserving pattern matching problem. In *Stringology*, pages 22–35, 2015.
- 18 Tamanna Chhabra, Simone Faro, M. Oguzhan Külekci, and Jorma Tarhio. Engineering order-preserving pattern matching with SIMD parallelism. *Softw., Pract. Exper.*, 47(5):731–739, 2017. doi:10.1002/spe.2433.
- 19 Tamanna Chhabra, M Oguzhan Külekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In *Stringology*, pages 36–46, 2015.
- 20 Tamanna Chhabra and Jorma Tarhio. A filtration method for order-preserving matching. *Inf. Process. Lett.*, 116(2):71–74, 2016. doi:10.1016/j.ipl.2015.10.005.
- 21 Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. Fast order-preserving pattern matching. In *Combinatorial Optimization and Applications*, pages 295–305. Springer, 2013.
- 22 Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.
- 23 Peter Clifford, Raphaël Clifford, and Costas Iliopoulos. Faster algorithms for  $\delta, \gamma$ -matching and related problems. In *Annual Symposium on Combinatorial Pattern Matching*, pages 68–78. Springer, 2005.
- 24 Raphaël Clifford and C Iliopoulos. Approximate string matching for music analysis. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):597–603, 2004.
- 25 Richard Cole, C. Iliopoulos, T. Lecroq, W. Plandowski, and Wojciech Rytter. On special families of morphisms related to  $\delta$ -matching and don't care symbols. *Information Processing Letters*, 85(5):227–233, 2003.
- 26 Maxime Crochemore, Costas S Iliopoulos, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Three heuristics for delta-matching: delta-bm algorithms. In *CPM*, pages 178–189. Springer, 2002.

- 27 Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Citeseer, 2000.
- 28 Branislav Đurian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Improving practical exact string matching. *Information Processing Letters*, 110(4):148–152, 2010.
- 29 Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975. doi:10.1016/0012-365X(75)90103-X.
- 30 Kimmo Fredriksson and Szymon Grabowski. Practical and optimal string matching. In *SPIRE*, volume 3772, pages 376–387. Springer, 2005.
- 31 Kimmo Fredriksson and Szymon Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Information Retrieval*, 11(4):335–357, 2008.
- 32 Xianping Ge. Pattern matching in financial time series data. *final project report for ICS*, 278, 1998.
- 33 Rui Henriques. *Learning from High-Dimensional Data using Local Descriptive Models*. PhD thesis, Instituto Superior Tecnico, Universidade de Lisboa, Lisboa, 2016.
- 34 Rui Henriques, Cláudia Antunes, and Sara C. Madeira. Methods for the efficient discovery of large item-indexable sequential patterns. In *New Frontiers in Mining Complex Patterns*, volume 8399 of *LNCS*, pages 100–116. Springer International Publishing, 2014.
- 35 Rui Henriques and Sara C Madeira. Bicspam: flexible biclustering using sequential patterns. *BMC bioinformatics*, 15(1):130, 2014.
- 36 Rui Henriques and Ana Paiva. Seven principles to mine flexible behavior from physiological signals for effective emotion recognition and description in affective interactions. In *PhyCS*, pages 75–82, 2014.
- 37 Jan Holub, William F. Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008. doi:10.1016/j.jda.2006.10.003.
- 38 Shuichi Kawashima and Minoru Kanehisa. Aaindex: amino acid index database. *Nucleic acids research*, 28(1):374–374, 2000.
- 39 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S Iliopoulos, Kunsoo Park, Simon J Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.
- 40 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.
- 41 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
- 42 Inbok Lee, Raphaël Clifford, and Sung-Ryul Kim. Algorithms on extended  $(\delta, \gamma)$ -matching. *Computational Science and Its Applications-ICCSA 2006*, pages 1137–1142, 2006.
- 43 Inbok Lee, Juan Mendivelso, and Yoan J Pinzón.  $\delta\gamma$ -parameterized matching. In *International Symposium on String Processing and Information Retrieval*, pages 236–248. Springer, 2008.
- 44 Ohad Lipsky and Ely Porat. Approximate matching in the  $l_{\infty}$  metric. *Inf. Process. Lett.*, 105(4):138–140, 2008. doi:10.1016/j.ipl.2007.08.012.
- 45 Juan Mendivelso, Inbok Lee, and Yoan J Pinzón. Approximate function matching under  $\delta$ -and  $\gamma$ -distances. In *SPIRE*, pages 348–359. Springer, 2012.
- 46 S Muthukrishnan. New results and open problems related to non-standard stringology. In *Combinatorial Pattern Matching*, pages 298–317. Springer, 1995.
- 47 Ely Porat and Klim Efremenko. Approximating general metric distances between a pattern and a text. In *ACM-SIAM symposium on Discrete algorithms*, pages 419–427. SIAM, 2008.




# On Undetected Redundancy in the Burrows-Wheeler Transform

Uwe Baier

Institute of Theoretical Computer Science, Ulm University

D-89069 Ulm, Germany

uwe.baier@uni-ulm.de

 <https://orcid.org/0000-0002-0145-0332>

---

## Abstract

The Burrows-Wheeler-Transform (BWT) is an invertible permutation of a text known to be highly compressible but also useful for sequence analysis, what makes the BWT highly attractive for lossless data compression. In this paper, we present a new technique to reduce the size of a BWT using its combinatorial properties, while keeping it invertible. The technique can be applied to any BWT-based compressor, and, as experiments show, is able to reduce the encoding size by 8 – 16% on average and up to 33 – 57% in the best cases (depending on the BWT-compressor used), making BWT-based compressors competitive or even superior to today’s best lossless compressors.

**2012 ACM Subject Classification** Theory of computation → Data compression, Applied computing → Document analysis, Mathematics of computing → Coding theory

**Keywords and phrases** Lossless data compression, BWT, Tunneling

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.3

**Related Version** Full version available at <https://arxiv.org/abs/1804.01937>.

**Supplement Material** Implementation available at <https://github.com/waYne1337/tbwt>.

## 1 Introduction

Lossless data compression plays an important role in modern digitization, as it enables us to shift and save computation resources during information exchange. For example, consider a setting where a computationally strong computer has to distribute data over a limited channel—by use of data compression, the storage requires less resources, and the file can be transmitted faster due to reduced data size—with the drawback of extra computation time for en- and decoding the information. Data compression is widespread today, current challenges are not only to compress data, but also to serve special features like resource-efficient decompression or even working on the compressed data directly, because the only way to fit it in memory (and thus process it fast) consists of using a compressed representation.

Compressors for the first mentioned feature typical make use of LZ77 [33]—a compression technique that, briefly speaking, replaces repeats in a text by references—resulting in very good compression rates and very fast decompression. Popular examples of compressors using LZ77 are `gzip` [12] or `7-zip` [27], which can be categorized as file transmission compressors.

A different technique for the second mentioned feature makes use of the Burrows-Wheeler-Transform (BWT) [3], which is an invertible permutation of the characters in the original text. The BWT itself does not compress data, but the transformed string tends to have some properties which make it highly compressible. The most popular compressor of such kind is `bzip2` [30], but compression rates are not the only aspect that make the BWT interesting:



© Uwe Baier;

licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics



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



also, the BWT in combination with wavelet trees [14] is well known to be an extremely useful and efficient tool for sequence analysis, commonly known as FM-index [8].

As the BWT has very interesting combinatorial properties, it has been heavily studied during the last two decades. For pure compression, a summary of the most relevant ideas can be found in [1]. Surprisingly, as Fenwick [6] stated, less attention was given to the encoding of runs (a run is a continuous sequence of the same character). This was reasonable, because, also stated by Fenwick [6], “the original scheme proposed by Wheeler is extremely efficient and unlikely to be much improved.”

Finally, this is the point where our paper draws on. In this paper, we describe a new technique called “tunneling”, which relies on observations about combinatorics of a BWT and can be proved to maintain the invertibility (Theorem 13). The technique is independent from the way a run gets encoded, but reduces the size of the BWT to be encoded eminently by shortening runs. In numbers, we are able to reduce the encoding size of a compressed BWT by 8 – 16% on average and up to 33 – 57% in the best cases, depending on the backend used to compress a BWT. This not only makes BWT compressors competitive with today’s best compressors, but also leaves the combinatorial properties of the BWT intact, what indicates that the technique is applicable to index structures such as the above mentioned FM-index<sup>1</sup>.

This paper is organized as follows: Section 2 contains basics about the BWT and BWT compression. Section 3 presents our new technique, followed by a proof of invertibility of our representation in Section 4. Section 5 shows a way to implement the technique, which is followed by experimental results in Section 6 and conclusions in Section 7.

## 2 Preliminaries

First we want to describe the major parts of the BWT and its use in data compression. Throughout this Paper, any interval  $[i, j]$  or  $[i, j)$  is meant to be an interval over the natural numbers, every logarithm is of base 2, and indices start with 1, except when stated differently.

Let  $\Sigma$  be a totally ordered set (alphabet) of elements (characters). A string  $S$  of length  $n$  over alphabet  $\Sigma$  is a finite sequence of  $n$  characters originating from  $\Sigma$ . We call  $S$  nullterminated if it ends with the lowest ordered character  $\$ \in \Sigma$  occurring only at the end of  $S$ . The empty string with length 0 is denoted by  $\varepsilon$ . Unless stated differently, we assume that  $S$  is nullterminated. Let  $S$  be a string of length  $n$ , and let  $i, j \in [1, n]$ . We denote by

- $S[i]$  the  $i$ -th character of  $S$ .
- $S[i..j]$  the substring of  $S$  starting at the  $i$ -th and ending at the  $j$ -th position.  
We state  $S[i..j] = \varepsilon$  if  $i > j$ , and define  $S[i..j] := S[i..j - 1]$ .
- $S_i$  the suffix of  $S$  starting at the  $i$ -th position, i.e.  $S_i = S[i..n]$ .
- $S_i <_{\text{lex}} S_j$  if the suffix  $S_i$  is lexicographically smaller than  $S_j$ ,  
i.e. there exists a  $k \geq 0$  with  $S[i..i + k] = S[j..j + k]$  and  $S[i + k] < S[j + k]$ .

► **Definition 1.** Let  $S$  be a string of length  $n$ . The *suffix array* [22] SA of  $S$  is a permutation of integers in range  $[1, n]$  satisfying  $S_{SA[1]} <_{\text{lex}} S_{SA[2]} <_{\text{lex}} \dots <_{\text{lex}} S_{SA[n]}$ .

► **Definition 2.** Let  $S$  be a string of length  $n$ , and SA be its corresponding suffix array. The Burrows-Wheeler-Transform (BWT) of  $S$  is a string  $L$  of length  $n$  defined as  $L[i] := S[SA[i] - 1]$  if  $SA[i] > 1$  and  $L[i] := \$$  if  $SA[i] = 1$ . Also, we define the F-Column  $F$  as a string of length  $n$  by  $F[i] := S[SA[i]]$ , which can also be obtained by sorting the characters in  $L$ .

<sup>1</sup> We give some hints towards this goal; there exist however more technical problems to be solved, outreaching the scope of this paper.



$i$	$SA[i]$	$S[1..SA[i]]$	$S_{SA[i]}$	$L[i]$	$F[i]$	$rlencode(L)[i]$
1	10	easypeasy	\$	y	\$	y
2	7	easype	asy\$	e	a	e
3	2	e	asypeasy\$	e	a	0
4	6	easyp	easy\$	p	e	p
5	1	ε	easypeasy\$	\$	e	\$
6	5	easy	peasy\$	y	p	y
7	8	easypea	sy\$	a	s	a
8	3	ea	sypeasy\$	a	s	0
9	9	easypeas	y\$	s	y	s
10	4	eas	ypeasy\$	s	y	0

■ **Figure 1** Suffix array, Burrows-Wheeler-Transformation L, F, run-length encoded BWT  $rlencode(L)$  and the prefixes preceding a suffix (third column) for  $S = \text{easypeasy}\$$ . The BWT is almost the same string as the concatenation of the last characters from prefixes preceding suffixes, except for the sentinel character. Also one can see that those prefixes often share more than one character with the prefixes standing next to them.

In other words, the Burrows-Wheeler Transform is the concatenation of characters which cyclically precede suffixes in the suffix array. An example of a suffix array and a BWT can be found in Figure 1. An important property of the BWT is its invertibility, i.e. it's possible to reconstruct the original string  $S$  solely from its BWT. Therefore, we use the notation  $rank_S(c, i)$  to denote the number of occurrences of character  $c$  in the string  $S[1..i]$ ,  $select_S(c, i)$  to denote the position of the  $i$ -th occurrence of  $c$  in  $S$  and  $C_S[c]$  to denote the number of characters smaller than  $c$  in  $S$ , that is,  $C_S[c] := |\{ i \in [1, n] \mid S[i] < c \}|$ .

► **Definition 3.** Let  $S$  be a string of length  $n$ , SA and L be its corresponding suffix array and BWT. The LF-mapping is a permutation of integers in range  $[1, n]$  defined as follows:

$$LF[i] := C_L[L[i]] + rank_L(L[i], i)$$

We write  $LF^x[i]$  for the  $x$ -fold application of LF, i.e.  $LF^x[i] := \underbrace{LF[LF[\dots LF[i]\dots]]}_{x \text{ times}}$ , and define  $LF^0[i] := i$ .

The LF-mapping carries its name because it maps each character in L to its corresponding position in F. To put it differently, the LF-mapping induces a walk through the suffix array in reverse text order, which commonly is called a “backward step”.

► **Lemma 4.** Let  $S$  be a string of length  $n$ , SA and LF be its suffix array and LF-mapping. Then,  $SA[LF[i]] = SA[i] - 1$  holds for all  $i \in [1, n]$  with  $SA[i] \neq 1$ .

**Proof.** See [3]. ◀

Thus, any BWT can be inverted by computing LF (which can solely be done using L), taking a walk through the suffix array in reverse text order using LF and meanwhile collecting characters from L, what yields the reverse string of  $S$  (see [3] for more details). Our next concern of the LF-mapping which will be important later is its parallelism property inside runs, that is, a consecutive sequence of the same character in the BWT.

► **Lemma 5.** Let  $S$  be a string of length  $n$ , L and LF be its corresponding BWT and LF-mapping. For any interval  $[i, j] \subseteq [1, n]$  with  $L[i] = L[i+1] = \dots = L[j]$ ,  $LF[i] + k = LF[i+k]$  holds for all  $0 \leq k \leq j - i$ .

**Proof.** Follows directly from Definition 3. ◀

To get an understanding why the BWT is useful for data compression, we need a better understanding of it. The suffix array places lexicographically similar suffixes next to each other. Therefore, suffixes in subsequences of the suffix array often share a common prefix (context). As the BWT consists of the cyclic preceding characters of those suffixes, a subsequence of the BWT can be seen as a collection of characters preceding the same context in  $S$ . As a result, the character distribution inside a subsequence of the BWT gets skew, i.e. it is dominated by just a few characters which frequently appear before the context.

Typical BWT compressors make use of this fact by transforming the BWT such that the locally skew character distributions turn into a global skew character distribution—an example for such a transformation is given by the Move-To-Front Transformation [29, 3]. Finally, the global skew character distribution of the transform is useful for the last stage of typical BWT compressors: entropy encoding. The target of entropy encoding is to minimize the middle cost for the encoding of a character in a string using its character distribution. A well-known lower bound for this cost is given by the entropy definition of Shannon [31]:

► **Definition 6.** Let  $S$  be a string of length  $n$ , and let  $c_c$  be the count of character  $c$  in  $S$ . The entropy  $H(S)$  of string  $S$  is defined as  $H(S) := \sum_{c \in \Sigma} \frac{c_c}{n} \log \frac{n}{c_c} = \log n - \frac{1}{n} \sum_{c \in \Sigma} c_c \log c_c$ .

Over the years, a couple of methods were developed to achieve cost-optimal entropy coding; most famous of such methods are Huffman coding [15] and Arithmetic coding [28]. However, it can be shown that the more skew a character distribution of an underlying source is, the more the entropy decreases. Consequently, the BWT transformation normally is highly compressible using an entropy encoder.

Another trick for improving compression used by most state-of-the-art BWT compressors is run-length-encoding. First of all, a run is a (length-maximal) subsequence in which all characters are equal. Run-length-encoding transforms a run with height (length)  $h$  of character  $c$  into the string  $ch_k h_{k-1} \cdots h_1$ , where  $(1 h_k h_{k-1} \cdots h_1)_2$  is the binary representation of  $h$  (in the encoding, the leading one is cut, and the symbols 0 and 1 are assumed to be distinct from symbols in  $S$ ). Figure 1 shows an example of run-length-encoding which often reduces the length of a BWT drastically. We refer to [7] for a survey about further BWT compression methods, and introduce our last preliminary definition: the indicator function.

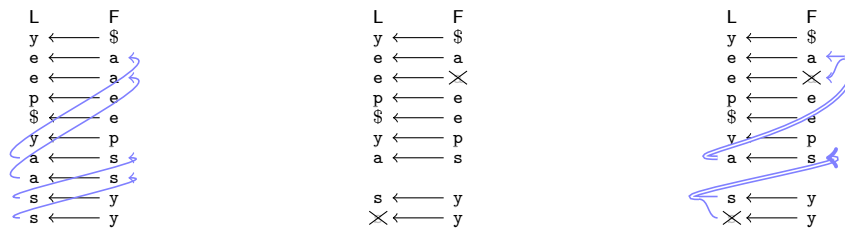
► **Definition 7.** Let  $P$  be any boolean predicate. The indicator function  $\mathbf{1}_P$  then is defined as  $\mathbf{1}_P := 1$  if predicate  $P$  is true, and  $\mathbf{1}_P := 0$  otherwise.

### 3 Tunneling

The last section explained why the BWT produces long runs of the same character. Briefly speaking, the consecutive characters in the BWT are followed by similar contexts (suffixes) in the original text, and similar contexts tend to be preceded by the same character. The BWT limits the strings preceding contexts to just 1 character, but there is no reason why longer preceding strings shouldn't be similar too<sup>2</sup>. In fact, this often is the case in repetitive texts: In Figure 1, the suffixes  $S_{SA[9]}$  and  $S_{SA[10]}$  are both preceded by the same string **eas**.

The intention of this section is to show a way how to use the similarity of the preceding strings to reduce the size of the BWT while keeping the invertibility and combinatorial properties of the BWT intact. Unlike existing approaches [24], we will not use word substitutions to achieve this goal; the problem of word substitutions is to find a good substitution scheme,

<sup>2</sup> A similar observation was made in the context of self-repetitions in suffix arrays; see [21, 26].



(a) determine the block in the BWT (b) cross out positions and remove doubly crossed-out ones (c) Reconstruction idea: use one block row for both rows

Figure 2 Process of tunneling as described in Definition 9. Above, block 2 – [9, 10] from the running example is tunneled. Any lines colored blue are related to the LF-mapping, cross-outs in L and F are displayed by crosses.

as well as storing the word dictionary efficiently. Instead, we present a method based on the combinatorics of the BWT, which contains the “word dictionary” implicitly in the remaining BWT, and offers an easier way to find a good substitution scheme—completely without substitution. Our first step will be the definition of blocks, which are short speaking repetitions of the same preceding string in lexicographically consecutive suffixes.

► **Definition 8.** Let  $S$  be a string of length  $n$  and  $L$  be its BWT. A block  $B$  is a pair of an integer  $d$  and an interval  $[i, j] \subseteq [1, n]$  ( $d - [i, j]$ -block) such that

$$L[LF^x[i]] = L[LF^x[i + 1]] = \dots = L[LF^x[j]] \text{ for all } 0 \leq x \leq d$$

We call  $[i, j]$  the start interval of  $B$ ,  $[LF^d[i], LF^d[j]]$  the end interval of  $B$ ,  $h_B := j - i + 1$  the height of  $B$  and  $w_B := d + 1$  the width of  $B$ .

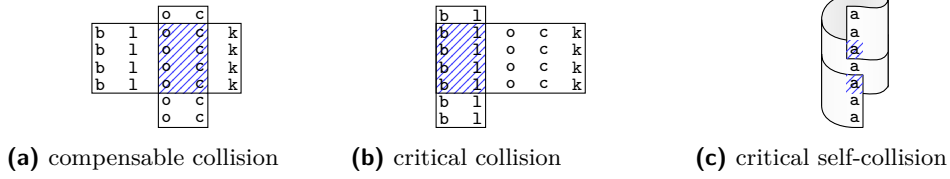
Blocks can also be seen as character matrices where each column consists of the same character and each row is build by picking characters in BWT during  $d$  consecutive applications of the LF-mapping, see also Figure 3. An example of blocks can be found in Figure 1: exemplary blocks are 2 – [9, 10], 1 – [7, 8], 0 – [2, 3] or 0 – [5, 5]. We also want to note that each column of a block can be mapped to a substring in the BWT which consists of the same character.

► **Definition 9.** Let  $S$  be a string of length  $n$ ,  $L$  and  $F$  be its corresponding BWT and F-Column, and let  $B = d - [i, j]$  be a block of  $S$ . The process of tunneling block  $B$  is defined as follows:

1. cross out position  $LF^x[k]$  in  $L$  (mark it in a bitvector  $\text{cntL}$  of size  $n$ ) for all  $0 \leq x < d$  and  $i < k \leq j$ .
2. cross out position  $LF^x[k]$  in  $F$  (mark it in a bitvector  $\text{cntF}$  of size  $n$ ) for all  $0 < x \leq d$  and  $i < k \leq j$ .
3. remove positions  $k$  that were crossed out both in  $L$  and  $F$  from  $L$ ,  $F$  and the bitvectors  $\text{cntL}$  and  $\text{cntF}$ .

To tunnel a whole set  $\mathcal{B}$  of blocks, we apply each step to all blocks  $B \in \mathcal{B}$  before continuing with the next step, where the result  $(L, \text{cntL}$  and  $\text{cntF}$ ) is called a tunneled BWT.

A simpler description of tunneling a block can be given as follows: Remove all positions of the block from the BWT, except for those in the rightmost or leftmost column or uppermost row of the block. Afterwards, cross out positions in the interval start from  $L$  and the interval end from  $F$ , both except for the uppermost position. An example can be seen in Figure 2.



■ **Figure 3** Visualization of block collisions. Blocks are displayed as continuous stripes, shared positions are marked using diagonal lines.

The interesting question will be if we are able to reconstruct the text, even if we remove positions from the BWT. Beforehand however, we need to care about block intersections, since they can produce side effects when tunneling more than one block.

► **Definition 10.** Let  $B = d - [i, j]$  and  $\tilde{B} = \tilde{d} - [\tilde{i}, \tilde{j}]$  be two different blocks of a string  $S$  with BWT  $L$  and LF-mapping  $LF$ . We say that  $B$  and  $\tilde{B}$  collide if they share positions in  $L$ , i.e. there exists  $x \in [0, d]$  and  $\tilde{x} \in [0, \tilde{d}]$  such that

$$[LF^x[i], LF^x[j]] \cap [LF^{\tilde{x}}[\tilde{i}], LF^{\tilde{x}}[\tilde{j}]] \neq \emptyset$$

We call each position in the intersection a shared position. Analogously, we say that a block  $B = d - [i, j]$  is self-colliding if some  $x, \tilde{x} \in [0, d]$  with  $x < \tilde{x}$  exist such that

$$[LF^x[i], LF^x[j]] \cap [LF^{\tilde{x}}[i], LF^{\tilde{x}}[j]] \neq \emptyset$$

Furthermore, let  $B_{in}$  and  $B_{out}$  be two colliding blocks and w.l.o.g.  $h_{B_{in}} \geq h_{B_{out}}$ . We call the collision between  $B_{in}$  and  $B_{out}$  compensable if following conditions are fulfilled:

1. The leftmost and rightmost columns of  $B_{out}$  do not intersect:  
The positions  $[i_{out}, j_{out}]$  and  $[LF^{d_{out}}[i_{out}], LF^{d_{out}}[j_{out}]]$  are not shared
2. At least one row of  $B_{in}$  does not intersect:  
There exists a  $y \in [i_{in}, j_{in}]$  such that the positions  $y, LF[y], \dots, LF^{d_{in}}[y]$  are not shared
3. The intersection area forms a block of height  $B_{in} - B_{out}$ :  
For all  $x_{in} \in [0, d_{in}]$  and  $x_{out} \in [0, d_{out}]$  following holds:

$$\left| \left[ LF^{x_{in}}[i_{in}], LF^{x_{in}}[j_{in}] \right] \setminus \left[ LF^{x_{out}}[i_{out}], LF^{x_{out}}[j_{out}] \right] \right| \in \{0, h_{B_{in}} - h_{B_{out}}\}$$

If the conditions are not fulfilled, or if a collision is a self-collision, we call it critical.

Figure 3 visualizes the different kinds of block collisions. Examples of block collisions can be found in Figure 1: the blocks  $2 - [9, 9]$  and  $0 - [7, 8]$  form a compensable collision, while the blocks  $2 - [9, 9]$  and  $1 - [7, 8]$  form a critical collision. Furthermore, for  $L = \mathbf{a a \cdots a}$  any block of width and height greater than one is self-colliding.

Now let us discuss the effect of the classifying criteria of collisions from Definition 10 a bit further. By the criteria, a compensable collision always consists of a “wider” (outer) and a “shorter but higher” (inner) block: The start and end interval of the outer block contain no shared position (condition 1), which in conjunction with condition 3 implies that the outer block must be wider than the inner one. For the inner block, at least one row must be unshared (condition 2), what in conjunction with condition 3 analogously shows that the inner block must be higher than the outer one. In the sense of visualization, these conditions build kind of a cross overlay of blocks, as depicted in Figure 3a. Extending this idea to more than two blocks, the criteria forms a natural hierarchy on colliding blocks, which will be useful for invertibility issues.

## 4 Invertibility

The purpose of this section is to show that a tunneled BWT can be inverted, i.e. the original string from which a BWT was constructed from can be rebuilt. As a first step, we will introduce a new generalized LF-mapping.

► **Definition 11.** Let  $L$  be the Burrows-Wheeler-Transformation of a string of length  $n$  and let  $\text{cntL}$  and  $\text{cntF}$  be two bitvectors of size  $n$ . The generalized LF-mapping is defined by

$$\text{LF}_{\text{cntF}}^{\text{cntL}}[i] := \underbrace{\text{select}_{\text{cntF}}(1, \dots)}_{\text{skip removed positions}} \left( \underbrace{\sum_{k=1}^n \text{cntL}[k] \cdot \mathbf{1}_{L[k] < L[i]}}_{\hat{=} C_L[L[i]]} + \underbrace{\sum_{k=1}^i \text{cntL}[k] \cdot \mathbf{1}_{L[k] = L[i]}}_{\hat{=} \text{rank}_L(L[i], i)} \right)$$

Definition 11 is almost equal to the normal LF-mapping—except that characters crossed out in  $L$  are ignored, while characters crossed out in  $F$  are skipped. Next, we will see that this definition is reasonable for tunneling, as it maintains its structure when removing positions from  $L$  in the “right” manner.

► **Lemma 12.** Let  $L$  be the Burrows-Wheeler-Transformation of a string of length  $n$  with LF-mapping  $\text{LF}$  and let  $\text{cntL}$  and  $\text{cntF}$  be two bitvectors of size  $n$ . Then following properties hold for the generalized LF-mapping  $\text{LF}_{\text{cntF}}^{\text{cntL}}$ :

1. Let  $\text{cntF}[i] = \text{cntL}[i] = 1$  for all  $i \in [1, n]$ . Then,  $\text{LF}_{\text{cntF}}^{\text{cntL}}$  is identical to the normal LF-mapping  $\text{LF}$ .
2. Let  $\text{cntF}$  and  $\text{cntL}$  be two bitvectors such that  $\text{LF}_{\text{cntF}}^{\text{cntL}}[j] = \text{LF}[j]$  for all  $j$  with  $\text{cntL}[j] = 1$ . Let  $i$  be an integer with  $\text{cntL}[i] = \text{cntF}[L[i]] = 1$ . Crossing out position  $i$  in  $L$  and position  $\text{LF}[i]$  in  $F$  (setting  $\text{cntL}[i] = \text{cntF}[L[i]] = 0$ ) does not change the mapping: Define

$$\widetilde{\text{cntL}}[j] := \text{cntL}[j] \cdot \mathbf{1}_{j \neq i} \quad \text{and} \quad \widetilde{\text{cntF}}[j] := \text{cntF}[j] \cdot \mathbf{1}_{j \neq \text{LF}[i]}$$

Then,  $\text{LF}_{\widetilde{\text{cntF}}}^{\widetilde{\text{cntL}}}[j] = \text{LF}[j]$  for all  $j$  with  $\widetilde{\text{cntL}}[j] = 1$ .

3. Let  $i$  be an integer with  $\text{cntL}[i] = \text{cntF}[i] = 0$ . Removing position  $i$  (crossed out both in  $L$  and  $F$ ) from  $L$ ,  $\text{cntL}$  and  $\text{cntF}$  does not change the mapping: Define

$$\widetilde{\text{cntL}}[j] := \text{cntL}[j + \mathbf{1}_{j \geq i}], \quad \widetilde{\text{cntF}}[j] := \text{cntF}[j + \mathbf{1}_{j \geq i}] \quad \text{and} \quad \widetilde{L}[j] := L[j + \mathbf{1}_{j \geq i}]$$

Then, for the corresponding mapping  $\widetilde{\text{LF}}_{\widetilde{\text{cntF}}}^{\widetilde{\text{cntL}}}$  the following holds:

$$\widetilde{\text{LF}}_{\widetilde{\text{cntF}}}^{\widetilde{\text{cntL}}}[j] = \text{LF}_{\text{cntF}}^{\text{cntL}}[j + \mathbf{1}_{j \geq i}] - \mathbf{1}_{\text{LF}_{\text{cntF}}^{\text{cntL}}[j + \mathbf{1}_{j \geq i}] \geq i}$$

Lemma 12 looks really bulky at the first moment, but reflects the operations required for tunneling, as we will see by discussing its different properties. Property 1 says that the new LF-mapping is identical to the old if we cross out nothing, and is straightforward. The more interesting property 2 tells us that the LF-mapping stays identical if we cross out consecutive positions in  $L$  and  $F$  in terms of text order. To explain why this works, think of a character  $c$  at position  $i$  in a BWT. If we cross out  $c$  in  $L$ , it is ignored, and thus the LF-mapping of all characters in  $L$  which are greater than  $c$  (or equal to  $c$  but placed below of  $i$ ) gets shifted one position upwards, and thus is modified. Now, as we also cross out  $\text{LF}[i]$  in  $F$ , and crossed-out positions in  $F$  are skipped, all of the modified positions are shifted one position downward, because their original LF-mapping was greater than  $\text{LF}[i]$ , and thus, the mapping stays identical. Property 3 also is easy to understand, as it says that a position which is

---

**Algorithm 1:** Inverting a tunneled Burrows-Wheeler-Transform.
 

---

**Data:** Tunneled Burrows-Wheeler-Transform  $L$  of size  $n$  from string  $S$  of size  $\tilde{n}$ , bitvectors  $\text{cntL}$  and  $\text{cntF}$  of size  $n + 1$  with  $\text{cntL}[n + 1] = \text{cntF}[n + 1] = 1$ .

**Result:** String  $S$  from which  $L$ ,  $\text{cntL}$  and  $\text{cntF}$  were build from.

```

1 let  $\text{LF}^*$  be the generalized LF-mapping (Definition 11).
2 initialize an empty stack  $s$ 
3  $S[\tilde{n}] \leftarrow \$$ 
4  $j \leftarrow 1$ 
5 for  $i \leftarrow \tilde{n} - 1$  to 1 do
6   if  $\text{cntF}[j + 1] = 0$  then // end of a tunnel
7      $j \leftarrow j + s.\text{top}()$ 
8      $s.\text{pop}()$ 
9    $S[i] \leftarrow L[j]$ 
10  if  $\text{cntL}[j] = 0$  or  $\text{cntL}[j + 1] = 0$  then // start of a tunnel
11     $k \leftarrow \max\{l \in [1, j] \mid \text{cntL}[l] = 1\}$  // uppermost row of block
12     $s.\text{push}(j - k)$ 
13   $j \leftarrow \text{LF}^*[j]$ 

```

---

ignored in  $L$  and skipped in  $F$  can be completely removed. A formal proof of the properties is omitted due to reasons of space, for now let us concentrate on the reconstruction of the original text.

The idea for reconstruction will be as follows: According to Lemma 12, tunneling will leave the LF-mapping identical, so we need only to clarify how to deal with tunneled blocks. As a remainder, tunneling means that we remove all of the characters except for the rightmost and leftmost column and uppermost row of a block. In blocks, we know that each row of the block is identical, and all of the rows run in parallel (in terms of the LF-mapping). Thus, if we reach the start of a tunnel, we will save the offset to the uppermost row, proceed at the uppermost row, and, once we leave the tunnel, use the saved offset to get back to the correct “lane”, as shown Figure 2c and described in Algorithm 1.

The reconstruction process also inspired us to name the method tunneling: once the start of a block is reached, the offset to the uppermost row gets saved, and we enter the “tunnel”, namely the uppermost row. After the tunnel ended, the temporarily information is used to get back to the correct “lane”, that is the row on which we entered the tunnel.

► **Theorem 13.** *Let  $S$  be a string of length  $\tilde{n}$ , let  $\mathcal{B}$  be a set of blocks in its BWT containing no critical block collisions, and let  $L$ ,  $\text{cntL}$  and  $\text{cntF}$  (each of size  $n$ ) be the components emerging by tunneling the blocks of set  $\mathcal{B}$ . Then, Algorithm 1 reconstructs the string  $S$  from the tunneled BWT in  $O(\tilde{n})$  time.*

**Proof.** First, note that the generalized LF-mapping in a tunneled BWT conforms to the normal LF-mapping in a traditional BWT: Definition 9 tells us that for each marked position  $i$  in  $\text{cntL}$ , the associated position  $\text{LF}[i]$  is marked in  $\text{cntF}$  during tunneling process. Furthermore, tunneling only removes positions  $i$  with  $\text{cntL}[i] = \text{cntF}[i] = 0$ —note that this argumentation still is true for any colliding blocks, with only the only difference that some of those “position pairs” are marked more than once. Thus Lemma 12 ensures that a walk through the generalized LF-mapping (Line 13 of Algorithm 1) will reproduce the same string during character pickup (Line 9)—except for positions  $i$  with either  $\text{cntL}[i]$  or  $\text{cntF}[i]$  equal zero, which are positions in a start or end interval of a block.

As we know that each row of a block is identical (in terms of characters), and that the LF-mapping in a block runs in parallel (Lemma 5), for correct reconstruction, it’s sufficient to store the relative offset to the top of a block when entering it, reconstruct any of the rows of the block, and at the end of the block use the stored offset to step back to the relative

position on which the block was entered—as performed by Algorithm 1. In case of block collisions, by the hierarchy build from the condition of compensable collisions, a tunnel will not be left until all its inner colliding block tunnels are left, thus the stack in Algorithm 1 correctly matches each tunnel end with the offset the tunnel was entered.

Finally, Algorithm 1 can be implemented to require  $O(\tilde{n})$  time by precomputing the generalized LF-mapping in  $O(n)$  time, and by implementing line 11 with an array mapping each position to the nearest previous position  $i$  with  $\text{cntL}[i] = 1$ , which obviously also can be computed in  $O(n)$  time, requiring  $O(1)$  time per query. ◀

In contrast, when dealing with critical collisions, Algorithm 1 will not be able to correctly match a tunnel start or end to the corresponding tunnel due to the intersections of start intervals or end intervals of blocks. In the case of self-collisions, the tunneling process from Definition 9 will remove entries of the topmost row of the self-colliding block from the BWT, thus falsifying the correctness proof of Algorithm 1.

## 5 Practical Implementation

This section’s purpose is to give a brief summary on how to use BWT tunneling practically. Our first restriction therefore is to focus only on such-called run-blocks, what will make tunneling easier to handle. A run-block is a block whose start and end intervals have the same height as the runs in the BWT where the intervals occur. Furthermore, we will focus only on width-maximal run-blocks having height and width both greater than one.

► **Definition 14.** Let  $S$  be a string of length  $n$  and  $L$  be its BWT. Furthermore, assume that the border cases  $L[0]$  and  $L[n+1]$  contain characters such that  $L[0] \neq L[1]$  and  $L[n] \neq L[n+1]$ .

A  $d - [i, j]$ -block is called a run-block if  $L[i] \neq L[i-1]$ ,  $L[j] \neq L[j+1]$ ,  $L[\text{LF}^d[i]] \neq L[\text{LF}^d[i]-1]$  and  $L[\text{LF}^d[j]] \neq L[\text{LF}^d[j]+1]$  holds.

A run-block  $RB$  is called width-maximal if it is wider than any colliding run-block  $\widetilde{RB}$  with same height, i.e.  $RB$  and  $\widetilde{RB}$  collide and  $h_B = h_{\widetilde{RB}} \Rightarrow w_{RB} \geq w_{\widetilde{RB}}$ .

An example of run-blocks is given in Figure 1:  $0 - [2, 3]$ ,  $1 - [7, 8]$  are run-blocks,  $2 - [9, 10]$  is the only width-maximal run-block. As a counter example,  $2 - [9, 9]$  is no run-block, as  $L[9] = L[10]$ , thus the height is not identical to that of the run where the block starts.

Run-blocks with height greater than one will never be self-colliding<sup>3</sup>—also, any collision between width-maximal run-blocks always is compensable: a run-block is height-maximal in sense of its start- and end-interval, thus any collision enforces one block to be higher and on the “inside” of the other, as required by Definition 10.

### 5.1 Block Computation

Our first concern is how blocks can be computed. For arbitrary blocks, a simple solution would be to compute the pairwise longest common suffixes of  $S[1..SA[i])$  and  $S[1..SA[i+1])$ , and afterwards enumerate the blocks using a stack-based approach—which is possible in  $O(n)$ , see [16] and [18]. However, as we want to compute the restricted set of width-maximal run-blocks, we will choose a different approach.

<sup>3</sup> Self-collisions are related to overlapping repeats in the text. Thus, a self-colliding block has to share positions in its start interval with itself, what cannot happen in run-blocks: as the overlapping intervals of a block cannot be equal (LF is a permutation), the start interval wouldn’t be height-maximal.



<table style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: left;">L</th> <th style="text-align: left;">cntL</th> <th style="text-align: left;">cntF</th> </tr> </thead> <tbody> <tr><td>y</td><td>1</td><td>1</td></tr> <tr><td>e</td><td>1</td><td>1</td></tr> <tr><td>e</td><td>1</td><td>0</td></tr> <tr><td>p</td><td>1</td><td>1</td></tr> <tr><td>\$</td><td>1</td><td>1</td></tr> <tr><td>y</td><td>1</td><td>1</td></tr> <tr><td>a</td><td>1</td><td>1</td></tr> <tr><td>s</td><td>1</td><td>1</td></tr> <tr><td>s</td><td>0</td><td>1</td></tr> </tbody> </table> <p><b>(a)</b> tunneled BWT from Figure 2</p>	L	cntL	cntF	y	1	1	e	1	1	e	1	0	p	1	1	\$	1	1	y	1	1	a	1	1	s	1	1	s	0	1	<table style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: left;">L</th> <th style="text-align: left;"><math>\text{aux} = 2 \cdot \text{cntL} + \text{cntF}</math></th> </tr> </thead> <tbody> <tr><td>y</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>e</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>e</td><td><math>2 = 2 \cdot 1 + 0</math></td></tr> <tr><td>p</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>\$</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>y</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>a</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>s</td><td><math>3 = 2 \cdot 1 + 1</math></td></tr> <tr><td>s</td><td><math>1 = 2 \cdot 0 + 1</math></td></tr> </tbody> </table> <p><b>(b)</b> merge bitvectors cntL and cntL to vector aux</p>	L	$\text{aux} = 2 \cdot \text{cntL} + \text{cntF}$	y	$3 = 2 \cdot 1 + 1$	e	$3 = 2 \cdot 1 + 1$	e	$2 = 2 \cdot 1 + 0$	p	$3 = 2 \cdot 1 + 1$	\$	$3 = 2 \cdot 1 + 1$	y	$3 = 2 \cdot 1 + 1$	a	$3 = 2 \cdot 1 + 1$	s	$3 = 2 \cdot 1 + 1$	s	$1 = 2 \cdot 0 + 1$	<table style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: left;">L</th> <th style="text-align: left;">aux</th> </tr> </thead> <tbody> <tr><td><math>\rightarrow</math>y</td><td rowspan="4" style="vertical-align: middle;">2 ]</td></tr> <tr><td><math>\rightarrow</math>e</td></tr> <tr><td>e</td></tr> <tr><td><math>\rightarrow</math>p</td></tr> <tr><td><math>\rightarrow</math>\$</td><td rowspan="4" style="vertical-align: middle;">1 ]</td></tr> <tr><td><math>\rightarrow</math>y</td></tr> <tr><td><math>\rightarrow</math>a</td></tr> <tr><td><math>\rightarrow</math>s</td></tr> <tr><td>s</td><td></td></tr> </tbody> </table> <p><b>(c)</b> remove “run-heads” from aux and trim symbols beside of runs</p>	L	aux	$\rightarrow$ y	2 ]	$\rightarrow$ e	e	$\rightarrow$ p	$\rightarrow$ \$	1 ]	$\rightarrow$ y	$\rightarrow$ a	$\rightarrow$ s	s	
L	cntL	cntF																																																																
y	1	1																																																																
e	1	1																																																																
e	1	0																																																																
p	1	1																																																																
\$	1	1																																																																
y	1	1																																																																
a	1	1																																																																
s	1	1																																																																
s	0	1																																																																
L	$\text{aux} = 2 \cdot \text{cntL} + \text{cntF}$																																																																	
y	$3 = 2 \cdot 1 + 1$																																																																	
e	$3 = 2 \cdot 1 + 1$																																																																	
e	$2 = 2 \cdot 1 + 0$																																																																	
p	$3 = 2 \cdot 1 + 1$																																																																	
\$	$3 = 2 \cdot 1 + 1$																																																																	
y	$3 = 2 \cdot 1 + 1$																																																																	
a	$3 = 2 \cdot 1 + 1$																																																																	
s	$3 = 2 \cdot 1 + 1$																																																																	
s	$1 = 2 \cdot 0 + 1$																																																																	
L	aux																																																																	
$\rightarrow$ y	2 ]																																																																	
$\rightarrow$ e																																																																		
e																																																																		
$\rightarrow$ p																																																																		
$\rightarrow$ \$	1 ]																																																																	
$\rightarrow$ y																																																																		
$\rightarrow$ a																																																																		
$\rightarrow$ s																																																																		
s																																																																		

■ **Figure 4** Exemplary aux-vector generation from a tunneled BWT. “Run-heads” (first symbols of runs) are marked using arrows, runs with height greater 1 by right square brackets.

We will describe the idea of the approach only; see the full version of this paper for an algorithm. The idea is to use runs as a start point, and use the LF-mapping to proceed over the BWT. Every time a run is reached which allows to width-extend the current block, the current block is pushed on a stack and the run is used as new block. Then, as soon as the current block cannot be extended (because the current run is not high enough), blocks are popped from the stack until an extendable block is reached. During the removal of blocks, the necessary conditions for width-maximal run-blocks can be checked. Also, to increase efficiency, a side-effect similar to pointer jumping is used, which allows to skip already processed blocks.

## 5.2 Tunneled BWT Encoding

The encoding of a tunneled BWT requires to encode three components: the remaining BWT L, as well as the bitvectors cntL and cntF. To reduce a component, we merge cntL and cntF to a new vector named aux by setting  $\text{aux}[i] := 2 \cdot \text{cntL}[i] + \text{cntF}[i]$ . The vector aux now contains 3 distinct symbols<sup>4</sup>, and is further shortened by removing all positions where runs in L start (all of this positions must be unmarked as the topmost row of a run-block stays unchanged) and by trimming the (identical) remaining symbols beside of each run of height greater than one to just one symbol (reconstruction is possible as only run-blocks are tunneled). An example is listed in Figure 4.

As the components L and aux originate from the same source and are quite similar, we’ll encode both components with the same BWT backend encoder, like, for example, Move-To-Front-Transformation + run-length-encoding of zeros + entropy encoding. This not only simplifies the implementation, but also allows to uncouple the block choice from the used backend encoder: As tunneling leaves the uppermost row of a block intact, the effect of tunneling can be summarized as shortening runs in L at cost of increasing the number of runs in aux due to the tunnel-marking symbols. For a good block-choice, it is useful to think of a tax system: a good choice maximizes the net-benefit, which is given by gross-benefit (amount of information removed from L) minus the tax (amount of information required to encode aux). Now, as long as different backend encoders encode runs in a similar fashion, an optimal block choice for a specific backend encoder will be near-optimal for all the other backend encoders, as the efficiency of such encoders can be seen as a constant  $c$  which does not affect the maximization of the net-benefit.

<sup>4</sup> Positions  $i$  containing markings in both  $\text{cntL}[i]$  and  $\text{cntF}[i]$  were removed by tunneling.



---

**Algorithm 2:** Greedy block choice
 

---

**Data:** a set  $\mathcal{RB}$  of width-maximal run-blocks and a function `score` which for each block returns the amount of run-characters it removes.  
**Result:** the array `BS` and a number  $t_{\text{best}}$ , whereby `BS[1.. $t_{\text{best}}$ ]` contains the blocks of a greedy block choice.

```

1 let BS be an array of size  $|\mathcal{RB}|$ 
2  $t_{\text{best}} \leftarrow 0$  // number of tunnels allowing best benefit
3  $b_{\text{best}} \leftarrow 0$  // best tunneling net-benefit in bits
4  $tc \leftarrow 0$  // number of run-characters removed by tunneling
5 for  $t \leftarrow 1$  to  $|\text{BS}|$  do
6   let  $B \in \mathcal{RB}$  be the block with score(B) maximal
7   reduce score( $\tilde{B}$ ) for all colliding blocks  $\tilde{B}$  of  $B$  depending on the kind of collision
8   remove collisions between all inner and outer colliding blocks of  $B$  // intersecting area gets
   tunneled
9    $\mathcal{RB} \leftarrow \mathcal{RB} \setminus \{B\}$ 
10   $\text{BS}[t] \leftarrow B$ 
11   $tc \leftarrow tc + \text{score}(B)$  // update number of removed run-characters from tunneling
12  if gross-benefit - tax  $>$   $b_{\text{best}}$  then // update block choice; see equations (1) and (2)
13     $t_{\text{best}} \leftarrow t$ 
14     $b_{\text{best}} \leftarrow \text{gross-benefit} - \text{tax}$ 

```

---

As most state-of-the-art compressors use run-length-encoding, we estimate net-benefit and tax in terms of run-length-encoding. Consequently, the net-benefit between a normal BWT  $L$  and a tunneled BWT  $\tilde{L}$  is given by

$$\begin{aligned} \text{gross-benefit} &:= |\text{rlencode}(L)| \cdot H(\text{rlencode}(L)) - |\text{rlencode}(\tilde{L})| \cdot H(\text{rlencode}(\tilde{L})) \\ &\approx n \log \left( \frac{n}{n - tc} \right) - rc \log \left( \frac{rc}{rc - tc} \right) + tc \left( 1 + \log \left( \frac{n - tc}{rc - tc} \right) \right) \end{aligned} \quad (1)$$

where  $n := |\text{rlencode}(L)|$  is the number of characters of the run-length-encoding of  $L$ ,  $H$  is the entropy function from Definition 6,  $rc$  is the number of run-characters in  $\text{rlencode}(L)$  (all characters except for the run-heads) and  $tc$  is the number of removed run-characters in  $\text{rlencode}(\tilde{L})$ . The tax is given by

$$\begin{aligned} \text{tax} &:= |\text{rlencode}(\text{aux})| \cdot H(\text{rlencode}(\text{aux})) \\ &\approx 2 \cdot t \cdot (1 + \log(h^2 - 1)) + 2 \cdot t \cdot h \cdot \log \left( 1 + \frac{2}{h - 1} \right) \end{aligned} \quad (2)$$

where  $t$  is the number of tunneled Blocks and  $h := \log \left( \frac{r_{h>1} - 2 \cdot t}{2 \cdot t} \right)$  with  $r_{h>1}$  being the number of runs with height greater than 1.

### 5.3 Block Choice

The estimators from the last section give a clear indication that tunneling will not always improve compression—picking too small blocks may result in a tax whose size overcomes the gross-benefit. It is clear however that bigger blocks are preferable to smaller ones—thus a greedy approach will produce best results. Algorithm 2 sketches our strategy for choosing blocks, also considering that the gross-benefit and the tax do not grow in a likewise manner.

The collision handling of Algorithm 2 can be done using a collision graph, that is, a graph connecting colliding blocks whose intersecting area is not overlaid by a third block. Line 7 then can be implemented using a graph traversal together with block information, while Line 8 can be performed by removing the node corresponding to block  $B$  from the graph.

Implementing block scores is a bit complicated, as run-length-encoding is used. Initially, the score of a Block  $B$  gets to the sum of the run-lengths of all runs  $B$  points in minus the sum of all reduced run-lengths, i.e. the sum of  $\log(h) - \log(h - h_B)$  for each run of height

$h$  where  $B$  points in. Score updates of outer collisions are approximated by multiplying the score with the ratio by which the block width was shortened. Score updates of inner collisions can be done by subtracting the width-difference-ratio-multiplied score of  $B$ .<sup>5</sup>

By using a heap, Algorithm 2 can be shown to run in  $O(n \log |\mathcal{RB}|)$  time. Unfortunately, we have to mention that the greedy strategy is not always optimal: think of three blocks whose colliding picture forms a shape like a big “H”. If the middle block has a score bigger than that of the outer blocks (but close enough), for  $t = 1$ , the middle block is the optimal choice, while for  $t = 2$  the outer blocks would be preferred, what is not covered by the algorithm. These situations however should not occur often in practice, so we neglected them.

## 6 Experimental Results

This section contains experimental results showing the effectiveness of our new technique. We applied our technique as described in the last section to three different BWT compressors:

- **bwz** : **bzip2** [30] without memory limitations.
- **bcm** : one of the strongest open-source-available BWT compressors [25, 19].
- **wt** : BWT encoded in a huffman-shaped wavelet tree<sup>6</sup> using hybrid bitvectors [11, 17] (good compression for miscellaneous data [17]) provided by the **sdsl-lite** library [13].

Our test data comes from three different text corpora, namely 12 medium-sized files from the Silesia Corpus [4] (6 – 49 MB), 6 bigger files from the Pizza & Chili Corpus [9] (54 – 1130 MB), as well as 9 repetitive files from the Repetitive Corpus [10] (45 – 446 MB). The full benchmark and all results are available at [2].

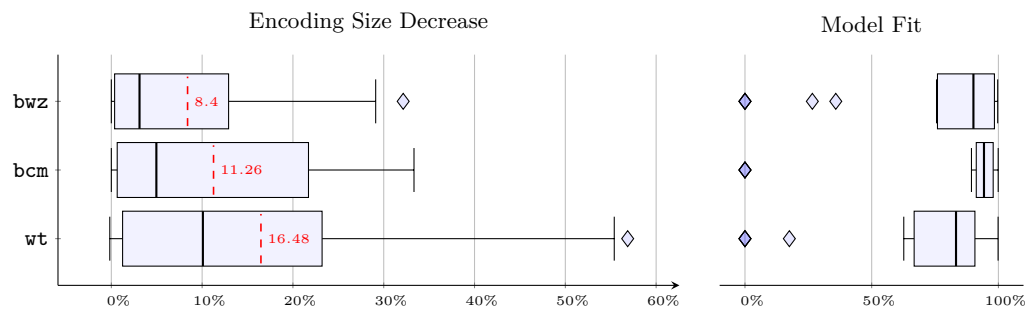
Beside of compression improvements, it is important to see if tunneling exploits its full potential not only in theory (as shown in the heuristics and approximations from last section) but also in practice. Therefore, we measured how well the theoretical model fits to the respective compressor by comparing the gross-net benefit ratios of model and compressor: as the efficiency of a compressor can be seen as some constant which is canceled when dividing two benefits from the same source, the gross-net benefit ratio should mostly be independent from the efficiency of a compressor, making it nicely comparable. Figure 5 shows compression improvements and the model fit as min-max distance  $\frac{\min\{x,y\}}{\max\{x,y\}}$  of both ratios.

As the box plots show, the compression improvements of tunneling are significant and the theoretical model fits quite well. However, for half of the test files the encoding size decrease lies below 4 – 11%—not very surprisingly, this half typically consists of small to medium-sized files where the normal BWT-based compressors work very good already. Compression improvements for the upper half of the data however are significant and range from 4 – 11% encoding size decrease up to 33 – 57% decrease. Also, it seems that the better the compressor works (in terms of compression rate), the better the model fits.

The outliers in the model fit can be ignored for two reasons: first, as the figure shows, tunneling never worsens compression by a serious amount; second, the potential of tunneling in all those files (fraction of net-benefit of theoretical model and the size of the **bwz** encoding)

<sup>5</sup> Let  $B_{in}$  be the inner colliding block of  $B$ . We set  $\mathbf{score}(B_{in}) = \mathbf{score}(B_{in}) - \frac{w_{B_{in}}}{w_B} \cdot \mathbf{score}(B)$ , which is motivated by the fact that  $\log(h - h_B) - \log(h - h_B - (h_B - h_{B_{in}})) = \log(h - h_B) - \log(h - h_{B_{in}}) = (\log(h) - \log(h - h_{B_{in}})) - (\log(h) - \log(h - h_B))$  for a single run in which  $B_{in}$  and  $B$  collide.

<sup>6</sup> Note that this data structure is **not capable** of text indexing. By permuting the bits in **cntL** according to the permutation induced by stably sorting column **L**, the generalized LF-mapping can be computed using  $\mathbf{select}_{\mathbf{cntF}}(1, \mathbf{rank}_{\mathbf{cntL}}(1, \mathbf{C}_L[\mathbf{L}[i]] + \mathbf{rank}_L(\mathbf{L}[i], i)))$ , allowing backward steps using wavelet trees. Unfortunately, more technical problems must be solved, outreaching the scope of this paper.



■ **Figure 5** Compression improvements and model fit of tunneling displayed as Tukey boxplots. The boxplots contain the whole tested data set. Boxes consist of lower quartile, median, upper quartile and average (red dashed line), whisker range is given by 1.5 times the interquartile range, outliers are shown as diamond markers. Compression improvements use the untunneled versions as baseline, model fits are given by the min-max distance of the gross-net benefit ratios of theoretical model and compressor.

■ **Table 1** Compression comparison of different compressors on a selection of the used test data. All values are shown in bits per symbol, that is, original file size times bits per symbol gives the compression size. Best compression results are marked bold.

Compressor	Silesia Corpus			Pizza & Chili Corpus			Repetitive Corpus		
	nci (32 MB)	samba (21 MB)	webster (40 MB)	proteins (1130 MB)	dna (386 MB)	english (1024 MB)	coreutils (196 MB)	para (410 MB)	world-forecasts (45 MB)
<b>bwz</b>	0.34	1.81	1.48	2.29	1.83	1.84	0.23	0.31	0.12
<b>bwz -tunneled</b>	0.33	1.75	1.48	2.00	1.81	1.66	0.17	0.21	0.11
<b>bcm</b>	0.29	1.49	1.24	2.33	1.72	1.56	0.23	0.32	0.13
<b>bcm -tunneled</b>	<b>0.28</b>	1.42	1.24	<b>1.95</b>	<b>1.70</b>	<b>1.34</b>	0.16	0.21	0.11
<b>wt</b>	0.61	2.70	2.08	3.97	2.05	2.45	0.69	0.49	0.40
<b>wt -tunneled</b>	0.54	2.45	2.07	2.72	2.03	1.99	0.38	0.42	0.29
<b>xz -9e -M 100% [32]</b>	0.35	1.38	1.61	2.22	1.78	1.93	<b>0.14</b>	<b>0.11</b>	<b>0.09</b>
<b>zpaq isc [20]</b>	0.36	<b>1.20</b>	<b>1.21</b>	2.61	1.86	1.64	0.62	1.85	0.09

is below 0.3%. The model fit itself however shouldn't be overestimated, as we tested tunneling with different models, always getting a quite similar compression result.

Table 1 shows a comparison of our technique with compressors following different paradigms: **xz** [32] is a very effective Lempel-Ziv compressor similar to **7zip** [27], while **zpaq** [20] uses context-mixing. As the table shows, the tunneled version of **bcm** performs best among all shown BWT compressors, **xz** remains the best choice for repetitive data. Beside of pure compression, we want to note that tunneling has its price: the time and space requirements for encoding roughly double, while decoding time and space requirements are reduced by a small amount.

## 7 Conclusion

As we have seen in the last Section, compression gains due to our technique are a significant improvement to BWT-based compression. The technique however is in early stage of development, and therefore has some outstanding problems like making a good and economic block choice. A solution might be to use heuristics for similar problems in the LCP array

[5, 23], our presented approach is a nice baseline but too complicated and resource-expensive. It also would be nice to get rid of the restriction of run-based blocks; Section 5.1 indicates that this is possible, but collisions complicate the situation. In our opinion, the “big deal” will be to build a compressed FM-index [8] with full functionality; the footnote on page 12 clearly indicates that this should be possible, although correct pattern counting might be a bit tricky. Thinking of a text index with half of the size of the currently best implementations seems utopian, but this paper shows that this goal should be achievable, giving a lot of motivation for further research on the topic.

---

## References

- 1 Jürgen Abel. Post BWT Stages of the Burrows-Wheeler Compression Algorithm. *Software – Practice and Experiences*, 40(9):751–777, 2010.
- 2 Uwe Baier. Tunneled BWT Implementation and Benchmark. <https://github.com/waYne1337/tbwt>. last visited January 2018.
- 3 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- 4 Sebastian Deorowicz. Silesia Corpus. <http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia>. last visited January 2018.
- 5 Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane. Compression with the tudocomp Framework. In *16th International Symposium on Experimental Algorithms*, SEA ’17, pages 13:1–13:22, 2017.
- 6 Peter M. Fenwick. Burrows-Wheeler compression: Principles and reflections. *Theoretical Computer Science*, 387(3):200–219, 2007.
- 7 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The Engineering of a Compression Boosting Library: Theory vs Practice in BWT Compression. In *Algorithms – ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings*, ESA ’06, pages 756–767, 2006.
- 8 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.
- 9 Paolo Ferragina and Gonzalo Navarro. Pizza & Chili Corpus. <http://pizzachili.dcc.uchile.cl/texts.html>. last visited January 2018.
- 10 Paolo Ferragina and Gonzalo Navarro. Repetitive Corpus. <http://pizzachili.dcc.uchile.cl/repcorpus.html>. last visited January 2018.
- 11 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*, 2(4):611–639, 2006.
- 12 Jean-Loup Gailly and Mark Adler. gzip File Compressor. <http://www.gzip.org/>. last visited January 2018.
- 13 Simon Gog. sds1-lite Library. <https://github.com/simongog/sds1-lite>. last visited January 2018.
- 14 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order Entropy-compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’03, pages 841–850, 2003.
- 15 David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- 16 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Slashing the Time for BWT Inversion. In *Proceedings of the 2012 Data Compression Conference*, DCC ’12, pages 99–108, 2012.

- 17 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid Compression of Bitvectors for the FM-Index. In *Proceedings of the 2014 Data Compression Conference, DCC '14*, pages 302–311, 2014.
- 18 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 12th Annual Conference on Combinatorial Pattern Matching, CPM '01*, pages 181–192, 2001.
- 19 Matt Mahoney. Large Text Compression Benchmark. <http://mattmahoney.net/dc/text.html>. last visited January 2018.
- 20 Matt Mahoney. zpaq File Compressor. <http://mattmahoney.net/dc/zpaq.html>. last visited January 2018.
- 21 Veli Mäkinen. Compact Suffix Array. In *Proceedings of the 11th Annual Conference on Combinatorial Pattern Matching, CPM '00*, pages 305–319, 2000.
- 22 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 5:935–948, 1993.
- 23 Markus Mauer, Timo Beller, and Enno Ohlebusch. A Lempel-Ziv-style Compression Method for Repetitive Texts. In *Proceedings of the Prague Stringology Conference 2017, PSC '17*, pages 96–107, 2017.
- 24 Alistair Moffat and R. Yugo Kartono Isal. Word-based Text Compression Using the Burrows-Wheeler Transform. *Information Processing and Management*, 41:1175–1192, 2005.
- 25 Ilya Muravyov. bcm File Compressor. <https://github.com/encode84/bcm>. last visited January 2018.
- 26 Gonzalo Navarro and Veli Mäkinen. Compressed Full-text Indexes. *ACM Computing Surveys*, 39(1), 2007.
- 27 Igor Pavlov. 7zip File Compressor. <http://www.7-zip.org/>. last visited January 2018.
- 28 Jorma J. Rissanen and Glen G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979.
- 29 B. Ya Ryabko. Data compression by means of a “book stack”. *Problems of Information Transmission*, 16:265–269, 1980.
- 30 Julian Seward. bzip2 File Compressor. <http://bzip.org/>. last visited January 2018.
- 31 Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.
- 32 Tukaani. xz File Compressor. <https://tukaani.org/xz/>. last visited January 2018.
- 33 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.



# Quasi-Periodicity Under Mismatch Errors

**Amihood Amir**

Bar-Ilan University and Johns Hopkins University  
Ramat-Gan, Israel  
amir@cs.biu.ac.il

**Avivit Levy**

Shenkar College  
Ramat-Gan, Israel  
avivitlevy@shenkar.ac.il

**Ely Porat**

Bar-Ilan University  
Ramat-Gan, Israel  
porately@cs.biu.ac.il

---

## Abstract

Tracing regularities plays a key role in data analysis for various areas of science, including coding and automata theory, formal language theory, combinatorics, molecular biology and many others. Part of the scientific process is understanding and explaining these regularities. A common notion to describe regularity in a string  $T$  is a *cover* or *quasi-period*, which is a string  $C$  for which every letter of  $T$  lies within some occurrence of  $C$ . In many applications finding exact repetitions is not sufficient, due to the presence of errors. In this paper we initiate the study of *quasi-periodicity persistence* under mismatch errors, and our goal is to characterize situations where a given quasi-periodic string remains quasi-periodic even after substitution errors have been introduced to the string. Our study results in proving necessary conditions as well as a theorem stating sufficient conditions for quasi-periodicity persistence. As an application, we are able to close the gap in understanding the complexity of *Approximate Cover Problem* (ACP) relaxations studied by [5, 4] and solve an open question.

**2012 ACM Subject Classification** Mathematics of computing → Combinatorics on words, Theory of computation → Pattern matching

**Keywords and phrases** Periodicity, Quasi-Periodicity, Cover, Approximate Cover

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.4

## 1 Introduction

Tracing regularities plays a key role in data analysis for various areas of science, including coding and automata theory, formal language theory, combinatorics, molecular biology and many others. Part of the scientific process is understanding and explaining these regularities. A typical form of regularity is *periodicity*, meaning that a “long” string  $T$  can be represented as a concatenation of copies of a “short” string  $P$ , possibly ending in a prefix of  $P$ . Periodicity has been extensively studied in Computer Science over the years (see [26]).

For many phenomena the definition of periodicity is too restrictive and it is necessary to study wider classes of repetitive patterns. One common such notion is that of a *cover* or a *quasi-period*, defined as follows.



© Amihood Amir, Avivit Levy, and Ely Porat;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 4; pp. 4:1–4:15

Leibniz International Proceedings in Informatics



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

► **Definition 1 (Cover).** A length  $m$  substring  $C$  of a string  $T$  of length  $n$ , is said to be a *cover* of  $T$ , if  $n > m$  and every letter of  $T$  lies within some occurrence of  $C$ .

Note that by the definition of cover, the string  $C$  is both a prefix and a suffix of the string  $T$ . For example, consider the string  $T = abaababaaba$ . Clearly,  $T$  is “almost” periodic with period  $aba$ , however, as it is not completely periodic, the algorithms that exploit repetitions cannot be applied to it. On the other hand, the string  $C = aba$  is a cover of  $T$ , which allows applying to  $T$  cover-based algorithms. In this paper we study quasi-periodicity under mismatch errors.

Quasi-periodicity was introduced by Ehrenfeucht in 1990 (according to [8]). The earliest paper in which it was studied is by Apostolico, Farach and Iliopoulos [9], which defined the *quasi-period* of a string to be the length of its shortest cover and presented an algorithm for computing the quasi-period of a given string in  $O(n)$  time and space. The new notion attracted immediately several groups of researchers (e.g. [10], [27, 28], [25], [11]). An overview on the first decade of the research on covers can be found in the surveys [8, 18, 31].

While covers are a significant generalization of the notion of periods as formalizing regularities in strings, they are still restrictive, in the sense that it remains unlikely that an arbitrary string has a cover shorter than the word itself. One direction to deal with this is to study different variants of quasi-periodicity. Variants that were introduced include *seeds* [20], *maximal quasi-periodic substring* [7], the notion of *k-covers* [19],  *$\lambda$ -cover* [32], *enhanced covers* [16], *partial cover* [21]. Since the notion of a seed is necessary to our study, we give its formal definition here.

► **Definition 2 (Seed).** A length  $m$  substring  $C$  of a string  $T$  of length  $n$ , is said to be a *seed* of  $T$ , if  $n > m$  and there exists a superstring  $T'$  of  $T$  such that  $C$  is a cover of  $T'$ .

Note that the definition of a seed allows the first and last occurrence of the seed  $C$  in  $T$  to be incomplete. Other recently explored directions include the inverse problem for cover arrays [14], extensions to strings in which not all letters are uniquely defined, such as *indeterminate strings* [6] or *weighted sequences* [33]. Some of the related problems are  $\mathcal{NP}$ -hard (see e.g., [6, 12, 21]).

Another direction to deal with the restrictiveness of quasi-periodicity definition is to consider the presence of errors. This direction was motivated by some applications, such as molecular biology and computer-assisted music analysis, where finding exact repetitions is not sufficient. In these applications, a more appropriate notion is that of *approximate repetitions*, where errors are allowed (see, e.g., [13, 15]). This notion was first studied in 1993 by Landau and Schmidt [23, 24] who concentrated on approximate tandem repeats. Note that, the natural definition of an approximate repetition is not clear. One possible definition is that the distance between any two adjacent repeats is small. Another possibility is that all repeats lie at a small distance from a single “original”. Such a definition of *approximate seeds* is studied in [12, 29, 17]. Indeed, all these definitions along with other ones were proposed and studied (see [1, 22, 30]). Yet another possibility is that all repeats must be equal, but we allow a fixed total number of mismatches. The possibility presented in [1] is a global one, assuming that an original unknown string is a sequence of repeats without errors, but the process of sequence creation or transmission incurs errors to the sequence of repeats, and, thus, the examined input string is not a sequence of repeats. [5] extend this approach to quasi-periodicity and study the *approximate cover problem (ACP)*, in which the input text is a sequence of some cover repetitions with possible mismatch errors, and a string that covers the text with the minimum number of errors is sought. We continue this line of research by studying quasi-periodic strings that have been introduced to substitution errors.



**Our Results.** In this paper we initiate the study of *quasi-periodicity persistence* under mismatch errors, and our goal is to characterize situations where a given quasi-periodic string remains quasi-periodic even after substitution errors have been introduced to the string. An implicit study of this question was introduced by [3], while proving that two strings with Hamming distance 1 cannot be both quasi-periodic. In our terminology this means that quasi-periodicity does not persist under a single substitution error. Broadening the study to situations where more than one substitution error may happen necessitates a deep understanding of the structure of quasi-periodic strings as well as their behaviour under mismatch errors. Our study results in proving necessary conditions as well as a theorem stating sufficient conditions for quasi-periodicity persistence. As an application, we are able to close the gap in understanding the complexity of ACP relaxations studied by [5, 4] and solve an open question [5] regarding the complexity of the full-tiling relaxation of the ACP.

**Paper Contributions.** The main contributions of this paper are:

- Giving a first explicit study of quasi-periodicity persistence, while broadening the knowledge on quasi-periodic strings under mismatch errors.
- Proving that the full-tiling relaxation of the ACP is polynomial-time computable, which was beyond the reach of current research prior to this paper. This result closes the gap in understanding the complexity hierarchy of ACP relaxations, where the ACP itself was proven to be  $\mathcal{NP}$ -hard [5].
- Proving properties of covers, seeds and quasi-periodic strings under mismatch errors that can serve future research of approximate regularities.

The paper is organized as follows. In Section 2, we give formal definitions and basic lemmas. Section 3 is devoted to the study of quasi-periodicity persistence under mismatch errors. In Section 4, we give a description of the full-tiling relaxation of the ACP and prove it is polynomial-time computable by applying the results from Section 3.

## 2 Preliminaries

In this section we give the needed formal definitions and prove some basic combinatorial properties of covers and seeds.

► **Definition 3 (Tiling).** Let  $T$  be a string over alphabet  $\Sigma$  such that the string  $C$  over alphabet  $\Sigma$  is a cover of  $T$ . Then, the sorted list of indices representing the start positions of occurrences of the cover  $C$  in the text  $T$  is called the *tiling* of  $C$  in  $T$ .

In this paper we have a text  $T$  which may contain substitution errors and, therefore, is not coverable. However, we would like to refer to a retained tiling of an unknown string  $C$  in  $T$  although  $C$  does not cover  $T$  because of mismatch positions. The following definition makes a distinction between a list of indices that may be assumed to be a tiling of the text before mismatch errors occurred and a list of indices that cannot be such a tiling.

► **Definition 4 (A Valid Tiling).** Let  $T$  be an  $n$ -length string over alphabet  $\Sigma$  and let  $L$  be a sorted list of indices  $L \subset \{1, \dots, n\}$ . Let  $m = n + 1 - L_{last}$ , where  $L_{last}$  is the last index in  $L$ . Then,  $L$  is called a *valid tiling* of  $T$ , if  $i_1 = 1$  and for every  $i_k, i_{k+1} \in L$ , it holds that  $i_{k+1} - i_k \leq m$ .

► **Notation 5.** Let  $C$  be an  $m$  length string over alphabet  $\Sigma$ . Denote by  $S(C)$  a string of length  $n$ ,  $n > m$ , such that  $C$  is a cover of  $S(C)$ .

## 4:4 Quasi-Periodicity Under Mismatch Errors

Note that  $S(C)$  is not uniquely defined even for a fixed  $n > m$ , since different valid tilings of the  $m$ -length string  $C$  may generate a different  $n$ -length string  $S(C)$ . A unique version is obtained if a unique appropriate valid tiling  $L$  is also given.

► **Notation 6.** Let  $T$  be an  $n$ -length string over alphabet  $\Sigma$  and let  $L$  be a valid tiling of  $T$ . Let  $m = n + 1 - L_{last}$ , where  $L_{last}$  is the last index in the tiling  $L$ . For any  $m$ -length string  $C'$ , let  $S_L(C')$  be the  $n$ -length string obtained using  $C'$  as a cover and  $L$  as the tiling as follows:  $S_L(C')$  begins with a copy of  $C'$  and for each index  $i$  in  $L$  a new copy of  $C'$  is concatenated starting from index  $i$  of  $S_L(C')$  (maybe running over a suffix of the last copy of  $C'$ ).

► **Definition 7.** Let  $T$  be a string of length  $n$  over alphabet  $\Sigma$ . Let  $H$  be the Hamming distance. The *distance of  $T$  from being covered* is:

$$dist = \min_{C \in \Sigma^*, |C| < n, S(C) \in \Sigma^n} H(S(C), T).$$

We will also refer to *dist* as *the number of errors in  $T$* .

► **Definition 8.** Let  $T$  be a string of length  $n$  over alphabet  $\Sigma$ . An  $m$ -long string  $C$  over  $\Sigma$ ,  $m \in \mathbb{N}$ ,  $m < n$ , is called an  *$m$ -length approximate cover of  $T$* , if for every string  $C'$  of length  $m$  over  $\Sigma$ ,  $\min_{S(C') \in \Sigma^n} H(S(C'), T) \geq \min_{S(C) \in \Sigma^n} H(S(C), T)$ , where  $H$  is the Hamming distance of the given strings.

We refer to  $\min_{S(C) \in \Sigma^n} H(S(C), T)$  as the *number of errors of an  $m$ -length approximate cover of  $T$* .

► **Definition 9 (Approximate Cover).** Let  $T$  be a string of length  $n$  over alphabet  $\Sigma$ . A string  $C$  over alphabet  $\Sigma$  is called an *approximate cover of  $T$*  if:

1.  $C$  is an  $m$ -length approximate cover of  $T$  for some  $m \in \mathbb{N}$ ,  $m < n$ , for which

$$\min_{S(C) \in \Sigma^n} H(S(C), T) = dist.$$

2. for every  $m'$ -length approximate cover of  $T$ ,  $C'$ , s.t.  $\min_{S(C') \in \Sigma^n} H(S(C'), T) = dist$ , it holds that:  $m' \geq m$ .

**Primitivity.** By definition, an approximate cover  $C$  should be *primitive*, i.e., it cannot be covered by a string other than itself (otherwise,  $T$  has a cover with a smaller length). Note that a periodic string can be covered by a smaller string (not necessarily the period), and therefore, is not primitive.

► **Definition 10.** The *Approximate Cover Problem (ACP)* is the following:

*INPUT:* String  $T$  of length  $n$  over alphabet  $\Sigma$ .

*OUTPUT:* An approximate cover of  $T$ ,  $C$ , and the number of errors in  $T$ .

### 2.1 Properties of Covers and Seeds

Our analysis of quasi-periodicity persistence under mismatch errors in Section 3 requires a preliminary study of properties of covers and seeds. We use the following easy observations:

► **Observation 11.** *If a string  $W$  is coverable but non-periodic then the shortest cover  $c$  of  $W$  satisfies  $|c| < |W|/2$ .*

► **Observation 12.** *Coverability is transitive, i.e., a cover of a cover of  $W$  is a cover of  $W$ .*

► **Observation 13.** *If a string  $c$  is a cover of a string  $W$  then  $c$  is a seed of every factor of  $W$  of length at least  $|c|$ .*

► **Observation 14.** *If a string  $s$  is a seed of a string  $W$  then  $s$  is a seed of every factor of  $W$  of length at least  $|s|$ .*

► **Lemma 15.** *A periodic string  $W$  is coverable, and one of the following must hold:*

1. *Its shortest cover is of length more than  $|W|/2$ . In this case,  $W = p^2p'$ , where  $p'$  is a non-empty prefix of  $p$ .*
2. *Its shortest cover  $c$  is of length at most  $|W|/2$  and  $|c| \neq |p|$ , where  $p$  is the period of  $W$ .*
3. *Its shortest cover  $c$  is of length at most  $|W|/2$  and  $|c| = |p|$ , where  $p$  is the period of  $W$ . In this case,  $c = p$  and  $W = p^i$ .*

**Example:** Consider the following periodic strings:

- The string  $W = abaabaa$  is periodic with period  $p = aba$ . It is coverable by  $c = abaa = pp'$ , where  $|c| > |W|/2$ . Note that  $W = p^2p'$ , where  $p' = a$  is a prefix of  $p$ .
- The string  $W = abaababaababa$  is periodic with period  $p = abaab$ . It is coverable by the string  $pp' = abaababa$ , however, its shortest cover is  $c = aba$ , where  $|c| < |W|/2$  and  $|c| < |p|$ . Note that  $c$  does not cover  $p$ .
- The string  $W = abababa$  is periodic with period  $p = ab$ . It is coverable by  $c = pp' = aba$ , where  $|c| < |W|/2$ , and  $|c| > |p|$ .
- The string  $W = ababaababa$  is periodic with period  $p = ababa$ . It is coverable by its period  $p$  since  $W = p^2$ , however, its shortest cover is  $c = aba$ , where  $|c| < |W|/2$  and  $|c| < |p|$ . Note that  $c$  covers  $p$ . We can make a longer string with this period  $W_1 = ababaababaaba$  which is still periodic with  $p$  and covered by  $c$ . However, if we take  $W_2 = ababaababaa$  which is again still periodic with  $p$  but no longer coverable by  $aba$  (which remains its seed). The shortest cover of  $W_2$  is  $c_2 = ababaa$ .

The next properties require an additional notation:

► **Notation 16.** *Let  $w$  be a string. Denote by  $w^j$  the string  $w$  with the  $j$ -th symbol substituted by some other symbol, i.e.,  $|w^{(j)}| = |w|$ , for all  $i = 1, \dots, |w|$ ,  $i \neq j$ ,  $w_i = w_i^{(j)}$ , and  $w_j \neq w_j^{(j)}$ . In this case we also write  $w =_j w^{(j)}$ .*

► **Theorem 17.** *Let  $S$  be a length  $n$  periodic string with period  $P$ . Then for any  $j \in \{1, \dots, n\}$ ,  $S^{(j)}$  is not periodic.*

Theorem 17 can be easily proven using a lemma proved in [2]. A similar result for covers is also known [3]:

► **Theorem 18.** [Amir et al. [3]] For every string  $W$  of length  $n$  and index  $j \in \{1, \dots, n\}$ , at most one of the strings  $W$ ,  $W^{(j)}$  is coverable.

We will also need the following auxiliary lemma from [3].

► **Lemma 19.** [Amir et al. [3]] Let  $w$  be a string and  $j$  be an index. Then  $w$  is not a seed of  $w^{(j)}$ .

► **Corollary 20.** [Amir et al. [3]] Let  $U$  and  $V$  be two coverable strings of the same length such that  $U \neq V$ . Then,  $H(U, V) > 1$ .

We also make use of the following lemma.

► **Lemma 21.** *Let  $W$  be a coverable string and let  $j$  be an index. Then no prefix (suffix) of  $W^{(j)}$  of length at most  $|W|/2$  is a seed of  $W^{(j)}$ .*

### 3 Quasi-Periodicity Persistence Under Mismatch Errors

In this section we analyze the extent to which quasi-periodicity may persist under mismatch errors, and characterize the structure of strings, the number and positions of mismatch errors, where such a persistence of quasi-periodicity is assured. First note that Corollary 20 means that quasi-periodicity is *not* persistent under a single mismatch error, no matter its position. Our analysis effort is, therefore, devoted to study the case where more mismatch errors occur. In such situations the number of errors as well as their exact positions determine whether the quasi-periodicity persists or not, as the following example shows.

**Example:** Consider the quasi-periodic (specifically, also periodic) string:  $S = abbbbabbbb$ . Two mismatch errors in positions 2 and 3 give the primitive string  $S' = aaabbabbbb$ . However, two mismatch errors in positions 2 and 6 give the (quasi-)periodic string  $S'' = aabbaabbbb$ .

We, therefore, need to refer to the structure of a quasi-periodic string, in order to analyze persistence of quasi-periodicity. Since the structure of the special case of periodic strings is known, giving sufficient conditions for periodicity persistence is easy, as the following observation states.

► **Observation 22.** *Let  $S$  be a periodic string with period length  $p$ . Let  $q = a \cdot p$ ,  $a \in \mathbb{N}$ , such that  $\lfloor \frac{n}{q} \rfloor \geq 2$ , and let  $S'$  be the string obtained from  $S$  by  $k = \lfloor \frac{n}{q} \rfloor$  errors of substitution to a character  $\sigma \in \Sigma$  where the difference between each subsequent error positions is  $q$ , then  $S'$  is periodic.*

We also want to give sufficient conditions for quasi-periodicity persistence, which is much more complicated to understand. Our first goal is to give a formal characterization of a structure of quasi-periodic strings. We begin by characterizing the structure of any string that may serve as a cover of another string. We call this string of variables *the free variable scheme* of the cover. We first give a definition of this term.

► **Definition 23.** Let  $\Gamma$  be an alphabet, called the free variables alphabet. A string in  $\Gamma^*$  is a *free variable scheme*. A free variable scheme  $\alpha$  over alphabet  $\Sigma$  defines a subset  $S_\alpha$  of  $\Sigma^*$ , where  $s$  is in  $S_\alpha$  if there is a function  $\Phi : \Sigma \rightarrow \Gamma$  such that  $\Phi(s) = \alpha$ .

► **Lemma 24.** *The free variable scheme representing any primitive cover is of the form  $\alpha\beta\alpha$  where:  $\beta$  is a non empty string of distinct variables that do not occur in  $\alpha$ ,  $\alpha$  is defined recursively, as follows:*

1.  $\alpha$  is empty, or
2.  $\alpha$  is of the form  $\alpha'\beta'\alpha'$ , where  $\beta'$  is a, possibly empty, string of variables that do not occur in  $\alpha'$ , and  $\alpha'$  is recursively defined similarly as  $\alpha$ .

**Proof.** There are two cases to consider:

1. If there are no overlaps of the cover in the string, then the cover is actually a period (having only complete appearances in the string it covers). In this case, it has the form  $\alpha\beta\alpha$ , where  $\alpha$  is empty and  $\beta$  is a non empty string of distinct variables that do not occur in  $\alpha$ .
2. If there is at least one overlap of the cover in the string it covers, then the cover cannot be a period. Such an overlap forces the structure of the cover to begin and end with the same string, represented by the sequence of variables  $\alpha$ . It is, therefore, of the form  $\alpha\beta\alpha$  where  $\beta$  is a string of distinct variables that do not occur in  $\alpha$ . The fact that the cover is primitive, means that  $\beta$  is non empty. Otherwise, it is periodic, therefore, by Lemma 15

it is coverable, contradiction to its primitivity. Now, if all the overlaps of the cover are of the same length as  $\alpha$ , then the recursion ends and we get  $\alpha = \beta'$  and  $\alpha'$  is empty. Otherwise, any overlap of different length forces a recursive structure as follows. Assume without loss of generality that the longer overlap is of length as  $\alpha$ , then the existence of a smaller overlap means that  $\alpha$  begins and ends with the same string, represented by the sequence of variables  $\alpha'$ . These occurrences of  $\alpha'$  in  $\alpha$  may be separated by another string, represented by the sequence of variables  $\beta'$ . ◀

**Example:** The recursive structure of the following free variable scheme

WWYWWZWYWWABCWWYWWZWYWW

is:

**level 1:**  $\alpha = WWYWWZWYWW$  and  $\beta = ABC$

**level 2:**  $\alpha = WWYWW$  and  $\beta = Z$

**level 3:**  $\alpha = WW$  and  $\beta = Y$

**level 4:**  $\alpha = W$  and  $\beta$  is the empty string

**level 5:**  $\alpha$  is the empty string and  $\beta = W$ .

Lemma 24 serves us in two directions. On the one hand, it is used to characterize the structure of any primitive cover, and on the other hand, it serves to characterize the structure of any quasi-periodic string with more than two occurrences of the cover. Note that, by Lemma 15, any quasi-periodic string with only two occurrences of the cover is periodic, for which Observation 22 applies. We, therefore, refer only to quasi-periodic strings having more than two occurrences of their cover. We call such strings *non-degenerate* quasi-periodic. The second direction is applied as formalized by Observation 25.

► **Observation 25.** *The free variable scheme applies to any non-degenerate quasi-periodic string  $S$  by defining  $C$  as follows. Take the longest prefix of  $S$  with length  $q_0$  smaller than  $|S|/2$ , which equals the suffix of  $S$  with the same length. Define  $\alpha$  to have length  $q_0$ . The length of  $\beta$  will be  $|S| - 2 \cdot q_0$ . Define  $\alpha_0 = \alpha$ ,  $\beta_0 = \beta$ . The definition of  $C$  continues recursively as long as  $S_{\alpha_i}$  has a prefix of length at most  $|S_{\alpha_i}|/2$  which equals its suffix of the same length. Let  $S_{\alpha_{i+1}}$  be the prefix, and define:  $\alpha_i = \alpha_{i+1}\beta_{i+1}\alpha_{i+1}$ . Otherwise, the recursion stops with  $\alpha_i = \beta_{i+1}$ . For all  $i$ ,  $\beta_i$  is defined to be a sequence of distinct variables that do not occur elsewhere.*

**Example:** Consider the string  $S = abaababaabaababaababa$ , which is quasi-periodic with cover  $aba$ . Applying Observation 25 to  $S$  gives:

**level 1:**  $S_{\alpha_0} = abaababa$ ,  $|\alpha_0| = 8$  and  $|\beta_0| = 5$

**level 2:**  $S_{\alpha_1} = aba$ ,  $|\alpha_1| = 3$  and  $|\beta_1| = 2$

**level 3:**  $S_{\alpha_2} = a$ ,  $|\alpha_2| = 1$  and  $|\beta_2| = 1$ .

The free variable scheme we get is:  $C = Y_1Y_2Y_1Y_3Y_4Y_1Y_2Y_1Y_5Y_6Y_7Y_8Y_9Y_1Y_2Y_1Y_3Y_4Y_1Y_2Y_1$ . The assignment:  $Y_1 = a$ ,  $Y_2 = b$ ,  $Y_3 = a$ ,  $Y_4 = b$ ,  $Y_5 = a$ ,  $Y_6 = b$ ,  $Y_7 = a$ ,  $Y_8 = a$ ,  $Y_9 = b$ , to the variables of  $C$  gives the string  $S$ . Note that  $C$  has a structure of a primitive cover, however, the above assignment results in a non-primitive string.

Observation 25 enables us to refer to any non-degenerate quasi-periodic string as an assignment to its free variable scheme. Our analysis takes into account the positions of the mismatch errors inserted to this string by referring to them as changing the assignment of a set of variables in its free variable scheme. In order to continue with our analysis, we need the following notation.

► **Notation 26.** Let  $C$  be a given free variable scheme. Denote by  $\mathcal{A}_C$  the string created by an assignment  $\mathcal{A}$  of alphabet symbols to the variables of  $C$ . Denote by  $\mathcal{A}_C(X_{i_1}, \dots, X_{i_j})$  the string created by assigning the variables  $X_{i_1}, \dots, X_{i_j}$  of  $C$  an alphabet symbol that is different from the one assigned by  $\mathcal{A}$ , and all other variables get assigned the same alphabet symbol as in the assignment  $\mathcal{A}$ .

We begin by showing that, assuming we already have an assignment  $\mathcal{A}$  giving a quasi-periodic string  $\mathcal{A}_C$ , then changing the assignment of **any** variable that appears once in the  $\alpha$  or  $\beta$  parts of the free variable scheme  $C$  results in a primitive string.

► **Lemma 27.** Let  $C$  be a given free variable scheme, and let  $Y$  be a variable that appears only once in the  $\alpha$  or  $\beta$  parts of  $C$ . Assume that  $\mathcal{A}_C$  is non-primitive, then  $\mathcal{A}_C(Y)$  is primitive.

We can now proceed with checking the case of variables that appear more than once in  $\alpha$ . Changing the assignment of such a variable may in some cases result in a primitive string, but in other cases result in another non-primitive string, as the next example shows.

**Example:** Consider  $C = XYVXYZXYVXY$ , where  $\alpha = XYVXY$  and  $\beta = Z$ . Note that both  $X$  and  $Y$  appear twice in  $\alpha$ . Now the string  $\mathcal{A}_C = aaaaaaaaaa$  is non-primitive, however, both  $\mathcal{A}_C(X) = baabaabaaba$  and  $\mathcal{A}_C(Y) = abaabaabaab$  are also non-primitive with covers  $baaba$  and  $abaab$ , respectively. Nonetheless, changing the assignment of both  $X$  and  $Y$  also yields a non-primitive string  $\mathcal{A}_C(X, Y) = bbabbabbabb$  with a cover  $bbabb$ .

Lemma 30 characterizes the case of a variable  $Y$  for which  $\mathcal{A}_C(Y)$  is also non-primitive. We need Definition 28 and Lemma 29 for the statement and proof of Lemma 30.

► **Definition 28 (Superimposed Tiling).** Let  $L_1$  and  $L_2$  be valid tilings over length  $n$ . Then,  $L_1$  is said to be *superimposed on*  $L_2$ , if for every index  $r$  in  $L_2$ , let  $s$  be the greatest index in  $L_1$  satisfying  $s \leq r$ , then  $r + m_2 \leq s + m_1$ , where  $m_1, m_2$  are the lengths of the covers in  $L_1$  and  $L_2$ , respectively ( $m = n - L_{last}$ , where  $L_{last}$  is the last index in a given tiling  $L$ ).

► **Lemma 29.** Let  $\mathcal{A}_C = \mathcal{A}_\alpha S \mathcal{A}_\alpha$  and  $\mathcal{A}_C(Y) = \mathcal{A}_\alpha(Y) S \mathcal{A}_\alpha(Y)$ , where  $\mathcal{A}_\alpha$  is a sequence of at least 2 appearances of  $W$  and  $\mathcal{A}_\alpha(Y)$  is sequence of the same number of appearances of  $W^{(j)}$ , then only one of the strings  $\mathcal{A}_C, \mathcal{A}_C(Y)$  can be non-primitive and the other must be primitive.

► **Lemma 30 (Quasi-Periodicity Persistence Necessary Condition).** If there exists a variable  $Y$  in  $C$  such that both  $\mathcal{A}_C$  and  $\mathcal{A}_C(Y)$  are non-primitive, then:

1.  $Y$  appears  $2^i$  times in  $C$ , where  $i \geq 2$ .
2. If  $Y$  appears  $2^i$  times in  $C$  for some  $i \geq 2$ , then  $C = \alpha_i \beta_i \alpha_i$ , and for each  $1 < \ell \leq i$ ,  $\alpha_\ell = \alpha_{\ell-1} \beta_{\ell-1} \alpha_{\ell-1}$ , where  $Y$  appears once in  $\alpha_1$ .
3. There exists an  $\ell$ ,  $1 \leq \ell < i$  such that  $\beta_\ell$  is non-empty.
4. Let  $L_c$  and  $L_{c'}$  be the tiling of the cover  $c$  in  $\mathcal{A}_C$  and  $c'$  in  $\mathcal{A}_C(Y)$ , respectively. Then, if  $|c| \leq |c'|$  then  $L_{c'}$  is superimposed on  $L_c$ , else,  $L_c$  is superimposed on  $L_{c'}$ .

**Proof.** First, by Lemma 27,  $Y$  must appear at least twice in  $\alpha$  (and therefore, exactly twice as that in  $C$ ). Thus, by the recursive structure of  $C$  (Lemma 24),  $Y$  appears a power of 2 times in  $\alpha$  and, therefore, a power of two times in  $C$ . Also, by Lemma 24, this recursive structure is of the form  $C = \alpha_i \beta_i \alpha_i$ , and for each  $1 < \ell \leq i$ ,  $\alpha_\ell = \alpha_{\ell-1} \beta_{\ell-1} \alpha_{\ell-1}$ , where  $Y$  appears once in  $\alpha_1$ . Denote by  $W$  the string  $\mathcal{A}_{\alpha_1}$  and by  $S_\ell$  the string  $\mathcal{A}_{\beta_\ell}$ , for all  $\ell$ . Note that since  $Y$  only appears once in  $\alpha_1$  and doesn't appear elsewhere, we have that

$\mathcal{A}_{\beta_\ell}(Y) = \mathcal{A}_{\beta_\ell} = S_\ell$ , for all  $\ell$ , and  $\mathcal{A}_{\alpha_1}(Y) = W^{(j)}$  for the index  $j$  that indicates the position of  $Y$  in  $\alpha_1$ .

We now prove that there exists an  $\ell$ ,  $1 \leq \ell < i$  such that  $\beta_\ell$  is non-empty. Assume to the contrary that only  $S_i$  is nonempty (because  $\beta_i$  must be nonempty by the definition of  $C$ ), then we have that  $\mathcal{A}_C = \mathcal{A}_{\alpha_i} S_i \mathcal{A}_{\alpha_i}$  and  $\mathcal{A}_C(Y) = \mathcal{A}_{\alpha_i}(Y) S_i \mathcal{A}_{\alpha_i}(Y)$ , where  $\mathcal{A}_{\alpha_i}$  is a sequence of  $2^{i-1}$  appearances of  $W$  and  $\mathcal{A}_{\alpha_i}(Y)$  is a sequence of  $2^{i-1}$  appearances of  $W^{(j)}$ . Note that since  $2^{i-1} \geq 2$ , then by Lemma 29, only one of the strings  $\mathcal{A}_C$ ,  $\mathcal{A}_C(Y)$  can be non-primitive and the other must be primitive, contradiction. Therefore, there exists an  $\ell$ ,  $1 \leq \ell < i$  such that  $\beta_\ell$  is non-empty.

Since both  $\mathcal{A}_C$  and  $\mathcal{A}_C(Y)$  are non-primitive, they have shortest covers  $c$  and  $c'$ , respectively. Assume without loss of generality that  $|c| \leq |c'|$ . It remains to show that  $L_{c'}$  is superimposed on  $L_c$ . Assume to the contrary that this is not the case, and let  $r$  be the first index in  $L_c$ , such that for the greatest index  $s$  in  $L_{c'}$  satisfying  $s \leq r$ , we have  $r + |c| > s + |c'|$ . First, note that  $s \neq r$ , otherwise, we necessarily have:  $r + |c| \leq s + |c'|$ , since  $|c| \leq |c'|$ , which contradicts the assumption. Therefore,  $s < r$ .

Let  $\hat{c}$  be the  $|c'|$ -length prefix of  $\mathcal{A}_C$ . Since  $c$  covers  $\mathcal{A}_C$  then  $c$  is both a prefix and a suffix of  $G_C$  and a prefix of  $\hat{c}$ . Also, since  $c'$  covers  $\mathcal{A}_C(Y)$ , then  $c'$  is both a prefix and a suffix of  $\mathcal{A}_C(Y)$  and, therefore,  $\hat{c}$  is both a prefix and a suffix of  $\mathcal{A}_C$ . Thus,  $c$  is also a suffix of  $\hat{c}$ . We get that the last complete occurrence of  $c$ , before  $r$  is at index  $s + |c'| - |c|$ . Now, any occurrence of  $c'$  in  $\mathcal{A}_C(Y)$  before index  $r$  (including  $r$ ) contradicts the maximality of  $s$ . Also, any occurrence of  $c'$  in index greater than  $r$  and at most  $s + |c'|$  contradicts the choice of  $r$  as the first index to contradict the assumption due to the occurrence of  $c$  in index  $s + |c'| - |c| < r$ . Therefore, the next occurrence of  $c'$  in  $\mathcal{A}_C(Y)$  is at index  $s + |c'| + 1$ . Thus, we have in  $\mathcal{A}_C$  two consecutive occurrences of  $\hat{c}$ , where  $c$  is a suffix of the first occurrence (due to the occurrence of  $c$  in index  $s + |c'| - |c|$ ) and a prefix of the second occurrence (due to the fact that  $c$  is a prefix of  $\hat{c}$ ), and there is an occurrence of  $c$  overlapping to suffix of the first and the prefix of the second (due to the occurrence in index  $r$ ). However, this means that  $c$  is periodic, which contradicts the minimality of  $c$  by Lemma 15. This concludes the proof of the lemma.  $\blacktriangleleft$

The following observations and lemma describe basic properties of the superimposition relation between tilings and the equation systems that tilings impose. Lemma 34 follows.

► **Observation 31.** *Any tiling  $L$  of the string created by an assignment  $\mathcal{A}$  on the  $m$ -length free variable scheme  $C$  imposes an equation system  $E$  on the variables of  $C$ . Moreover, if  $L_1$  is superimposed on  $L_2$ , then  $E_1 \subseteq E_2$ , where  $E_1$  and  $E_2$  are the equation systems imposed by  $L_1$  and  $L_2$ , respectively.*

► **Observation 32.** *Assume that the strings  $\mathcal{A}_C(Y_1)$  and  $\mathcal{A}_C(Y_2)$  are both non-primitive, and let  $L_1$  and  $L_2$  be the tilings of the strings  $\mathcal{A}_C(Y_1)$  and  $\mathcal{A}_C(Y_2)$ , respectively. Let  $E_1$  and  $E_2$  be the equation systems imposed by  $L_1$  and  $L_2$ , respectively. Then,  $\mathcal{A}_C(Y_1, Y_2)$  imposes the equation system  $E = E_1 \cap E_2$  (which may be empty).*

We need the following lemma for the proof of Lemma 34 below.

► **Lemma 33.** *Assume that the strings  $\mathcal{A}_C$ ,  $\mathcal{A}_C(Y_1)$  and  $\mathcal{A}_C(Y_2)$  are non-primitive, and let  $L$ ,  $L_1$  and  $L_2$  be the tilings of the strings  $\mathcal{A}_C$ ,  $\mathcal{A}_C(Y_1)$  and  $\mathcal{A}_C(Y_2)$ , respectively. Assume that both  $L_1$  and  $L_2$  are superimposed on  $L$ . Then,  $L_1$  is superimposed on  $L_2$  or  $L_2$  is superimposed on  $L_1$ .*



► **Lemma 34.** *If there exist variables  $Y_1, \dots, Y_k$ ,  $k \geq 1$ , in  $C$  such that  $\mathcal{A}_C, \mathcal{A}_C(Y_i)$ , for every  $i$ ,  $1 \leq i \leq k$ , are non-primitive, then  $\mathcal{A}_C(Y_1, \dots, Y_k)$  is also non-primitive. Moreover, let  $L$  and  $L'$  be the tilings of the covers in the strings  $\mathcal{A}_C$  and  $\mathcal{A}_C(Y_1, \dots, Y_k)$ , respectively, then  $L$  is superimposed on  $L'$  or viceversa.*

**Proof.** The proof is by induction on  $k$ . The case  $k = 1$  follows trivially. Let  $Y_1, \dots, Y_{k+1}$ ,  $k \geq 1$ , be variables in  $C$  such that  $\mathcal{A}_C, \mathcal{A}_C(Y_i)$ , for every  $i$ ,  $1 \leq i \leq k+1$ , are non-primitive. By induction hypothesis,  $\mathcal{A}_C(Y_1, \dots, Y_k)$  is also non-primitive. Moreover, let  $L$  and  $L'$  be the tilings of the covers  $c, c'$  in the strings  $\mathcal{A}_C$  and  $\mathcal{A}_C(Y_1, \dots, Y_k)$ , respectively, then  $L$  is superimposed on  $L'$  or viceversa. Also, let  $L''$  be the tiling of the cover  $c''$  of the string  $\mathcal{A}_C(Y_{k+1})$ . By Lemma 30, we have that if  $|c| \leq |c''|$  then  $L''$  is superimposed on  $L$ , else,  $L$  is superimposed on  $L''$ .

Let  $E_c, E_{c'}$  and  $E_{c''}$  be the equation systems imposed, according to Observation 31, by the tilings  $L, L'$  and  $L''$ , respectively. There are four cases to consider:

1. If  $L'$  is superimposed on  $L$  and  $L''$  is superimposed on  $L$ , then by Lemma 33, either  $L'$  is superimposed on  $L''$  or viceversa. Therefore, either  $E_{c'} \subseteq E_{c''} \subseteq E_c$  or  $E_{c''} \subseteq E_{c'} \subseteq E_c$ . Thus, by Observation 32,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  imposes the equation system  $E = E_{c'} \cap E_{c''} = E_{c'}$  or  $E = E_{c'} \cap E_{c''} = E_{c''}$ . Consequently,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  is non-primitive. Moreover, we have that  $\hat{L}$  is superimposed on  $L$ , where  $\hat{L}$  is the tiling defined by  $E$ .
2. If  $L'$  is superimposed on  $L$  and  $L$  is superimposed on  $L''$ , then  $E_{c'} \subseteq E_c$  and  $E_c \subseteq E_{c''}$ . Therefore,  $E_{c'} \subseteq E_{c''}$ . Thus, by Observation 32,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  imposes the equation system  $E = E_{c'} \cap E_{c''} = E_{c'}$ . Consequently,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  is non-primitive. Moreover, we have that  $\hat{L}$  is superimposed on  $L$ , where  $\hat{L}$  is the tiling defined by  $E$ .
3. If  $L$  is superimposed on  $L'$  and  $L''$  is superimposed on  $L$ , then  $E_c \subseteq E_{c'}$  and  $E_{c''} \subseteq E_c$ . Therefore,  $E_{c''} \subseteq E_{c'}$ . Thus, by Observation 32,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  imposes the equation system  $E = E_{c'} \cap E_{c''} = E_{c''}$ . Consequently,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  is non-primitive. Moreover, we have that  $\hat{L}$  is superimposed on  $L$ , where  $\hat{L}$  is the tiling defined by  $E$ .
4. If  $L$  is superimposed on  $L'$  and  $L$  is superimposed on  $L''$ , then  $E_c \subseteq E_{c'}$  and  $E_c \subseteq E_{c''}$ . Thus, by Observation 32,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  imposes the equation system  $E = E_{c'} \cap E_{c''} \supseteq E_c$ . Consequently,  $\mathcal{A}_C(Y_1, \dots, Y_{k+1})$  is non-primitive. Moreover, we have that  $L$  is superimposed on  $\hat{L}$ , where  $\hat{L}$  is the tiling defined by  $E$ .

This concludes the proof of the lemma. ◀

Theorem 35 follows.

► **Theorem 35. [Quasi-Periodicity Persistence Theorem]**

*Let  $S$  be a non-degenerate quasi-periodic string. Let  $C$  be the free variable scheme of  $S$ ,  $\mathcal{A}$  be the assignment on the variables of  $C$  such that  $\mathcal{A}_C = S$ , and*

$$V = \{Y \in C \mid Y \text{ appears } k(Y) = 2^i \text{ times, } i \geq 2, \text{ and } \mathcal{A}_C(Y) \text{ is quasi-periodic}\}.$$

*Let  $V' \subseteq V$  and let  $S'$  be the string obtained from  $S$  by  $k = \sum_{Y \in V'} k(Y)$  substitution errors in positions where the variables  $Y \in V'$  appear in  $C$  by an assignment  $\mathcal{A}_C(V')$ , then  $S'$  is quasi-periodic.*

## 4 Application: Closing the Complexity Gap in ACP Relaxations Study

In this section we apply the analysis of quasi-periodicity persistence described in Section 3, to study the full-tiling relaxation of the approximate cover problem, in which we are given a retained tiling of the cover before the errors has occurred together with the input string



itself. This relaxation was first introduced and studied in [5] and its complexity remained open. Proving the correctness of this algorithm was beyond the reach of current research of quasi-periodicity in the presence of mismatch errors prior to this paper. The results of Section 3 enable proving its correctness, thus showing that the full-tiling relaxation of the ACP is polynomial time computable, which closes the gap in understanding the complexity hierarchy of ACP relaxations, presented in [5]. In order to formally define the relaxation we need Definition 36.

► **Definition 36.** Let  $T$  be an  $n$ -length string over alphabet  $\Sigma$  and let  $L$  be a valid tiling of  $T$ . Let  $m = n + 1 - L_{last}$ , where  $L_{last}$  is the last index in the tiling  $L$ . Then, an  $L$ -approximate cover of  $T$  is a primitive string  $C$  such that for every string  $C'$  of length  $m$  over  $\Sigma$ ,  $H(S_L(C'), T) \geq H(S_L(C), T)$ , where  $H$  is the Hamming distance of the given strings.

$\min_{C \in \Sigma^m} H(S_L(C), T)$  is the number of errors of an  $L$  approximate cover of  $T$ .

The formal definition of the full-tiling relaxation of the ACP is given below.

► **Definition 37** (The Full-Tiling Relaxation of the ACP). *INPUT:* String  $T$  of length  $n$  over alphabet  $\Sigma$ , and a valid tiling  $L$  of  $T$ .

*OUTPUT:* An  $L$ -approximate cover  $C$  of  $T$ .

[5] suggest a polynomial-time algorithm for the full-tiling relaxation of the approximate cover problem in two parts. The algorithm has a mandatory part, called the Histogram Greedy Algorithm. This algorithm does the main work in finding an approximate cover subject to the tiling  $L$ . It returns a candidate for the final  $L$  approximate cover to be output. This candidate is legal if it is primitive and illegal, otherwise. In the latter case, a second part of the algorithm is needed: the Full-Tiling Primitivity Coercion. In this part, the legality of the candidate is checked, and if needed, the candidate is corrected in order to coerce the primitivity requirement. In order to give a self-contained presentation of our results, we give a description of the Histogram Greedy Algorithm in Subsection 4.1 and the Full-Tiling Primitivity Coercion Algorithm in Subsection 4.2. We then complete its analysis in Subsection 4.3.

## 4.1 The Histogram Greedy Algorithm

This part of the algorithm performs the following steps given the text  $T$  and the valid tiling  $L$ :

1. Find  $m$ , the length of an approximate cover subject to the tiling  $L$ , by computing the difference between  $n + 1$ , and the last index in the tiling  $L$ ,  $L_{last}$ , which indicates the last occurrence of the cover in  $T$ .
2. Compute the  $m$ -length mask  $M$  of an approximate cover, by initializing  $M$  to zeroes, setting  $M[1] = 1$ , then reading the tiling  $L$  from beginning to end and for each  $i_k, i_{k+1} \in L$  setting  $M[i_{k+1} - i_k] = 1$ .
3. Compute the  $m$ -long string  $V_C$  of variables from an auxiliary alphabet

$$\Sigma_V = \{v_1, v_2, \dots, v_m\}.$$

First, we initialize the  $m$ -long string  $V_C$  to  $v_1 v_2 \dots v_m$ . Then, we read the mask  $M$  from end to beginning, and for every  $j$  such that  $M[j] = 1$ , we update the string  $V_C$  by equalizing the substrings  $V_C[1..m - j + 1]$  and  $V_C[j..m]$ . In the equalization process, when we obtain an equation  $v_k = v_\ell$  for  $k < \ell$ , we replace both letters by  $v_k$ . The resulting

string  $V_C$  represents  $C$  in the following sense: for any pair of indices  $1 \leq i < j \leq m$ , if  $V_C[i] = V_C[j]$  then  $C[i] = C[j]$ . However, it can be that  $V_C[i] \neq V_C[j]$ , while  $C[i] = C[j]$ . In other words,  $V_C$  carries the information on equalities imposed by the mask  $M$  between indices of  $C$ .

4. Compute the string of length  $n$ ,  $V_T$ , with variables from the auxiliary alphabet  $\Sigma_V$ , which is a string covered by  $V_C$  according to the tiling  $L$  of  $T$ .  $V_C$  is computed using the tiling  $L$  and  $V_C$  as follows: it begins with a copy of  $V_C$  and for each index  $i$  in  $L$  a new copy of  $V_C$  is concatenated starting from index  $i$  of  $V_T$  (maybe running over a suffix of the last copy of  $V_C$ ).
5. Compute the histogram  $Hist_{V_C, \Sigma}$  using the alignment of  $T$  with  $V_T$  and counting for each variable  $V \in V_C$  and each  $\sigma \in \Sigma$ , the number of indices  $i$  in  $T, V_T$  for which  $V_T[i] = V$  and  $T[i] = \sigma$ .
6. Compute an  $L$ -approximate cover candidate  $C$  greedily according to the histogram  $Hist_{V_C, \Sigma}$ , as follows: for every index  $1 \leq i \leq m$ , set  $C[i] = \sigma_0$ , where  $Hist_{V_C, \Sigma}[V_C[i], \sigma_0] = \max_{\sigma \in \Sigma} Hist_{V_C, \Sigma}[V_C[i], \sigma]$ , i.e., for each index in  $C$  we choose the alphabet symbol that minimizes the number of mismatch errors between  $S_L(C)$  and  $T$  in the relevant indices according to the tiling  $L$ .

The algorithm outputs the  $m$ -length string  $C$  from its last step and the histogram table  $Hist_{V_C, \Sigma}$ .

As discussed in [5], the output  $C$  of the Histogram Greedy algorithm might not be an  $L$ -approximate cover of  $T$ , because it might not be primitive, as the following example shows.

**Example:** Assume that  $V_C = XYZWXY$  and  $\Sigma = \{a, b\}$  and that the histogram  $Hist_{V_C, \Sigma}$  computed by the algorithm is the following:

$V_C \setminus \Sigma$	a	b
X	4	1
Y	2	3
Z	2	1
W	0	3

Then, the Histogram Greedy algorithm chooses:  $X = a$ ,  $Y = b$ ,  $Z = a$ ,  $W = b$ , and outputs  $C = ababab$ , which cannot be considered a legal cover since it is not primitive, i.e.,  $C$  itself can be covered by the shorter string  $ab$ . Note, that the input tiling  $L$  requires an  $m$ -length string as an output. Therefore, the (primitive) 2-length approximate cover  $ab$  is precluded as an  $L$ -approximate cover. Assuming that the input tiling  $L$  is the retained tiling of the cover of the original text before the errors occurred, such a case means that, though  $ab$  is a string covering  $T$  subject to a partial tiling  $L$  with the least number of errors, it does not cover  $T$  with  $L$  as a full tiling. In this sense,  $L$  is an evidence that the original cover is of larger length than  $ab$  and that more errors actually happened.

In order to impose the requirement of the definition of an  $L$ -approximate cover of  $T$  to be a primitive string such that all its repetitions to cover  $T$  (with minimum number of errors) are marked in the tiling  $L$ , we need a primitivity coercion algorithm. This algorithm was suggested by [5], and is described in Subsection 4.2.

## 4.2 The Full-Tiling Primitivity Coercion Algorithm

This part of the algorithm gets as input the string  $C$  returned by the Histogram Greedy algorithm (Subsection 4.1) and performs the following steps:

1. Check the primitivity of  $C$  (using the linear-time algorithm of [9]). If  $C$  is primitive, return  $C$ .
2. Else, find  $V_k \in V_C$  such that if the assignment of  $V_k$  is changed from the symbol with the largest value in the row of  $V_k$  in  $Hist_{V_C, \Sigma}$  to the symbol with the second largest value in this row, thus obtaining a new  $m$ -length candidate string  $C'$ , such that the difference  $H(S_L(C'), T) - H(S_L(C), T)$  is minimized and where  $C'$  is primitive.

Lemma 38 below describes the time complexity of the Full-Tiling Primitivity Coercion algorithm and immediately follows from the linear-time complexity of the algorithm [9] used in the first step and the description of the second step.

► **Lemma 38** ([5]). *The time complexity of the Full-Tiling Primitivity Coercion algorithm is  $O(|\Sigma| \cdot m)$ .*

The following subsection is devoted to proving the correctness of the Full-Tiling Primitivity Coercion algorithm, thus proving that the full-tiling relaxation of the ACP is polynomial-time computable.

### 4.3 Correctness of the Full-Tiling Primitivity Coercion Algorithm

We begin by noting that the structure of the string of variables created by the Histogram Greedy algorithm has the free variable scheme, as defined by Lemma 24.

If the cover generated by the assignment of the Histogram Greedy algorithm to this scheme is not primitive, then Corollary 20 guarantees that changing the value of any free variable in  $\beta$  results in a primitive cover. The problem is that this may not necessarily be the cover with minimum cost. Checking, for every single free variable, if changing it for the alphabet symbol with the second largest value in the histogram results in a primitive cover, and choosing the cover that generates the smallest Hamming distance, will indeed guarantee that we have a primitive cover with the smallest Hamming distance that results from changing a **single** variable. We need to show that it is impossible to get a primitive cover that generates an even smaller Hamming distance, by choosing the alphabet symbol with the second highest histogram in a **set** of free variables. We prove that this situation can not happen.

Note that if  $C$  is the free variable scheme generated by the Histogram Greedy algorithm. Then  $\mathcal{A}_C$  can be the string created by the assignment of this algorithm to the variables of  $C$ , and  $\mathcal{A}_C(X_{i_1}, \dots, X_{i_j})$  can be the string created by assigning the variables  $X_{i_1}, \dots, X_{i_j}$  of  $C$  an alphabet symbol whose histogram value is *second highest*, and all other variables get assigned an alphabet symbol whose histogram value is the highest.

Theorem 39 follows.

► **Theorem 39.** *Given a text  $T$  of length  $n$  over alphabet  $\Sigma$  and a valid tiling  $L$ . Let  $L_{last}$  be the last index in  $L$ . Then, the full-tiling relaxation of the approximate cover problem of  $T$  can be solved in  $O(\Sigma \cdot m + n)$  time, where  $m = n + 1 - L_{last}$ .*

**Proof.** First, note that an  $L$ -approximate cover of  $T$  must have length  $m = n + 1 - L_{last}$ , where  $L_{last}$  is the last index in  $L$ . If the string  $C'$  returned by the Histogram greedy algorithm is primitive, then  $C'$  is an  $L$ -approximate cover of  $T$  since by its construction, it is the  $m$ -length primitive string such that its  $n$ -length tiled string according to the given tiling  $L$ ,  $S_L(C')$  has the minimum Hamming distance from  $T$ . In this case  $C'$  is also the string  $C$  returned by the Full-Tiling Primitivity Coercion algorithm.

Assume then that  $C'$  is not a primitive string and, therefore, the second step of the Full-Tiling Primitivity Coercion algorithm is performed. By Lemma 27, a change in a variable that appear once in the  $\beta$  or  $\alpha$  parts of the free variable form of  $C'$  results in a primitive  $m$ -length string. Lemma 30 characterizes  $C'$  in case there exists a variable such that changing its assignment does not yield a primitive string. Note that by Lemma 34 taking a set of such variables and changing their assignment also does not yield a primitive string. Therefore, any set of variables such that changing their assignment yields a primitive string, necessarily contains a variable that changing its assignment only is enough to yield a primitive string. However, it is obvious that changing this variable only gives a primitive string that covers the input string with less mismatch errors.

The second step of the Full-tiling Primitivity Coercion algorithm chooses a character that minimizes the difference  $H(S_L(C'), T) - H(S_L(C), T)$ . Therefore, the resulting  $m$ -length string  $C$  is the  $m$ -length primitive string such that its  $n$ -length tiled string according to the given tiling  $L$ ,  $S_L(C)$  has the minimum Hamming distance from  $T$ . ◀

---

### References

- 1 A. Amir, E. Eisenberg, and A. Levy. Approximate periodicity. In *Proc. ISAAC 2010*, LNCS 6506, pages 25–36. Springer, 2010.
- 2 A. Amir, E. Eisenberg, A. Levy, E. Porat, and N. Shapira. Cycle detection and correction. *ACM Transactions on Algorithms*, 9(1)(13), 2012.
- 3 A. Amir, C. S. Iliopoulos, and J. Radoszewski. Two strings at hamming distance 1 cannot be both quasiperiodic. *Information Processing Letters*, 128:54–57, 2017.
- 4 A. Amir, A. Levy, M. Lewenstein, R. Lubin, and B. Porat. Can we recover the cover? In *Proc. CPM*, 2017.
- 5 A. Amir, A. Levy, R. Lubin, and E. Porat. Approximate cover of strings. In *Proc. CPM*, 2017.
- 6 P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau. Conservative string covering of indeterminate strings. In *Proc. Stringology*, pages 108–115, 2008.
- 7 A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoret. Comput. Sci.*, 119:247–265, 1993.
- 8 A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *Proc. Structures in Logic and Computer Science*, LNCS 1261, pages 236–248, 1997.
- 9 A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39:17–20, 1991.
- 10 D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44:345–347, 1992.
- 11 D. Breslauer. Testing string superprimitivity in parallel. *Information Processing Letters*, 49(5):235–241, 1994.
- 12 M. Christodoulakis, C. S. Iliopoulos, K. Park, and J. S. Sim. Approximate seeds of strings. *Journal of Automata, Languages and Combinatorics*, 10:609–626, 2005.
- 13 T. Crawford, C. S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Comput. Musicol.*, 11:73–100, 1998.
- 14 M. Crochemore, C. S. Iliopoulos, S. P. Pissis, and G. Tischler. Cover array string reconstruction. In *Proc. CPM*, pages 251–259, 2010.
- 15 M. Crochemore, C. S. Iliopoulos, and H. Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In *Proc. 9th Austral. Workshop on Combinatorial Algorithms*, pages 172–185, 1998.
- 16 T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczynski. Enhanced string covering. *Theor. Comput. Sci.*, 506:102–114, 2013.


- 17 O. Guth and B. Melichar. *Using Finite Automata Approach for Searching Approximate Seeds of Strings*, pages 347–360. Springer, 2010.
- 18 C. S. Iliopoulos and L. Mouchard. Quasiperiodicity and string covering. *Theor. Comput. Sci.*, 218(1):205–216, 1999.
- 19 C. S. Iliopoulos and W. F. Smyth. An on-line algorithm of computing a minimum set of  $k$ -covers of a string. In *Proc. 9th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 97–106, 1998.
- 20 C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996.
- 21 T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Fast algorithm for partial covers in words. In *Proc. CPM*, pages 177–188, 2013.
- 22 R. M. Kolpakov and G. Kucherov. Finding approximate repetitions under hamming distance. *Theor. Comput. Sci.*, 303:135–156, 2003.
- 23 G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proc. 4th Symp. Combinatorial Pattern Matching*, LNCS 648, pages 120–133, 1993.
- 24 G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *J. of Computational Biology*, 8(1):1–18, 2001.
- 25 Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
- 26 M. Lothaire. *Combinatorics on words*. Addison-Wesley, 1983.
- 27 D. Moore and W. F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994.
- 28 D. Moore and W. F. Smyth. A correction to: An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 54:101–103, 1995.
- 29 B. Melichar O. Guth and M. Balik. *All Approximate Covers and Their Distance using Finite Automata*, pages 21–26. CEUR-WS, 2009.
- 30 J. S. Sim, C. S. Iliopoulos, K. Park, and W. F. Smyth. Approximate periods of strings. *Theor. Comput. Sci.*, 262:557–568, 2001.
- 31 W. F. Smyth. Repetitive perhaps, but certainly not boring. *Theor. Comput. Sci.*, 249(2):343–355, 2000.
- 32 H. Zhang, Q. Guo, and C. S. Iliopoulos. Algorithms for computing the lambda-regularities in strings. *Fundam. Inform.*, 84(1):33–49, 2008.
- 33 H. Zhang, Q. Guo, and C. S. Iliopoulos. Varieties of regularities in weighted sequences. In *Proc. AAIM*, LNCS 6142, pages 271–280, 2010.



# Fast Matching-based Approximations for Maximum Duo-Preservation String Mapping and its Weighted Variant

Brian Brubach<sup>1</sup>

Department of Computer Science, University of Maryland, College Park, MD 20742, USA  
bbrubach@cs.umd.edu

 <https://orcid.org/0000-0003-1520-2812>

---

## Abstract

We present a new approach to approximating the Maximum Duo-Preservation String Mapping Problem (MPSM) based on massaging the constraints into a tractable matching problem. MPSM was introduced in Chen, Chen, Samatova, Peng, Wang, and Tang [10] as the complement to the well-studied Minimum Common String Partition problem (MCSP). Prior work also considers the  $k$ -MPSM and  $k$ -MCSP variants in which each letter occurs at most  $k$  times in each string. The authors of [10] showed a  $k^2$ -approximation for  $k \geq 3$  and 2-approximation for  $k = 2$ . Boria, Kurpisz, Leppänen, and Mastrolilli [6] gave a 4-approximation independent of  $k$  and showed that even 2-MPSM is APX-Hard. A series of improvements led to the current best bounds of a  $(2 + \epsilon)$ -approximation for any  $\epsilon > 0$  in  $n^{O(1/\epsilon)}$  time for strings of length  $n$  and a 2.67-approximation running in  $O(n^2)$  time, both by Dudek, Gawrychowski, and Ostropolski-Nalewaja [16]. Here, we show that a 2.67-approximation can surprisingly be achieved in  $O(n)$  time for alphabets of constant size and  $O(n + \alpha^7)$  for alphabets of size  $\alpha$ .

Recently, Mehrabi [28] introduced the more general weighted variant, Maximum Weight Duo-Preservation String Mapping (MWPSM) and provided a 6-approximation. Our approach gives a 2.67-approximation to this problem running in  $O(n^3)$  time. This approach can also find an  $8/(3(1 - \epsilon))$ -approximation to MWPSM for any  $\epsilon > 0$  in  $O(n^2 \epsilon^{-1} \lg \epsilon^{-1})$  time using the approximate weighted matching algorithm of Duan and Pettie [15].

Finally, we introduce the first streaming algorithm for MPSM. We show that a single pass suffices to find a 4-approximation on the size of an optimal solution using only  $O(\alpha^2 \lg n)$  space.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** approximation algorithm, maximum duo-preservation string mapping, minimum common string partition, string comparison, streaming algorithm, comparative genomics

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.5

**Acknowledgements** The author wishes to thank advisors Mihai Pop and Aravind Srinivasan for their support, guidance, and encouragement.

## 1 Introduction

String comparison is a fundamental problem in many fields such as bioinformatics and data compression. The difference between two strings is often measured by some notion of edit distance, the number of edit operations required to transform one string into another.

---

<sup>1</sup> Supported in part by NSF awards CCF-1422569 and CCF-1749864 as well as the NIH, grant R01-AI-100947 to Mihai Pop.



© Brian Brubach;

licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 5; pp. 5:1–5:14

Leibniz International Proceedings in Informatics



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

The classic Levenshtein distance definition includes insertion, deletion, and/or substitution operations on single characters. However, the more general edit distance with moves problem studied in [13] allows an additional operation wherein an entire block of text is shifted within a string.

Variations of these shift operations, also known as rearrangements, are commonly studied in genomics [31, 11] with several biologically motivated twists on the above definition. String comparison of DNA or protein sequences can provide an estimate of how closely related different species are. In data compression, we may want to store many similar strings as a single string along with the edits required to recover all strings. These two applications even overlap naturally in the field of bioinformatics where extremely large datasets of biological sequences are common. For example, the challenge of pan-genome storage is to store many highly similar sequences from the same clade such as a bacterial species.

One way to capture just the “moves” operation on two strings which are permutations of each other is the Minimum Common String Partition problem (MCSP). In that problem, we cut (partition) each string into a multi-set of substrings such that the two multi-sets are identical and the number of cuts is minimized. This paper studies the complementary problem to MCSP, the Maximum Duo-Preservation String Mapping Problem (MPSM) and its weighted variant (MWPSM). Our goal is to find a one-to-one mapping from the letters of one string to the other. The objective is to maximize the pairs of consecutive letters (duos) which map to pairs of consecutive letters in the other string (i.e. pairs that are not cut in MCSP).

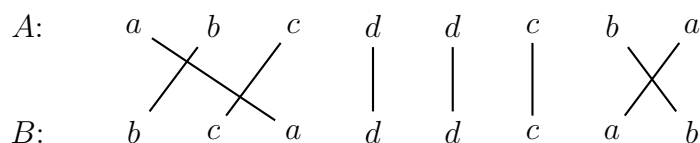
While MCSP has been well-studied for some time, a recent flurry of work on MPSM has given us a deeper understanding of that problem. Mehrabi [28] introduced the Maximum Weight Duo-Preservation String Mapping Problem (MWPSM) to better capture applications in comparative genomics. Beyond identifying the number of block moves, the weighted variant allows us to address questions like, “How far did these blocks move?” This better captures the concept of “synteny” in genetics [22, 29]. Also addressing practical considerations, Dudek, et al [16] included a quadratic time version of their approximation algorithm whereas much of the prior work has focused on improving the approximation in polynomial time.

## 1.1 Problem Description

The Maximum Duo-Preservation String Mapping Problem (MPSM) is defined as follows. We are given two strings  $A = a_1a_2 \dots a_n$  and  $B = b_1b_2 \dots b_n$  of length  $n$  such that  $B$  is a permutation of  $A$ . Let  $a_i$  and  $b_j$  be the  $i^{\text{th}}$  and  $j^{\text{th}}$  characters of their respective strings. A *proper* mapping  $\pi$  from  $A$  to  $B$  is a one-to-one mapping with  $a_i = b_{\pi(i)}$  for all  $i = 1, \dots, n$ . A *duo* is simply two consecutive characters from the same string. We say that a duo  $(a_i, a_{i+1})$  is *preserved* if  $a_i$  is mapped some  $b_j$  and  $a_{i+1}$  is mapped to  $b_{j+1}$ . The objective is to return a proper mapping from the letters of  $A$  to the letters of  $B$  which preserves the maximum number of duos. Note that the number of duos preserved in each string is identical and by convention we count the number of duos preserved in a single string rather than the sum over both strings. Let  $OPT_{MPSM}$  denote the number of duos preserved from a single string in an optimal solution to the MPSM problem. Figure 1 shows an example of an optimal mapping which preserves the maximum possible number of duos.

The complementary Minimum Common String Partition problem (MCSP) seeks to find partitions of the strings  $A$  and  $B$  where a partition  $P_A$  of  $A$  is defined as a set of substrings whose concatenation is  $A$ . The objective is to find minimum cardinality partitions  $P_A$  of  $A$  and  $P_B$  of  $B$  such that  $P_B$  is a permutation of  $P_A$ . Let  $OPT_{MCSP}$  denote the cardinality of a partition in an optimal solution. We can see that  $OPT_{MCSP} = |P_A| = |P_B| = n - OPT_{MPSM}$ .





■ **Figure 1** Illustration of a mapping  $\pi$  from  $A$  to  $B$  that preserves 3 duos:  $bc$ ,  $dd$ , and  $dc$ . A solution to the complementary MCSP problem on the same strings would be partitions  $P_A = a, bc, ddc, b, a$  and  $P_B = bc, a, ddc, a, b$  with  $|P_A| = |P_B| = 5$ .

The variants,  $k$ -MPSM and  $k$ -MCSP, add the restriction that each letter occurs at most  $k$  times in each string. For a given algorithm, let  $ALG_{MPSM}$  be number of duos preserved by the algorithm. The approximation ratio for that algorithm is defined as  $OPT_{MPSM}/ALG_{MPSM}$ .

In MWPSM, a weight is assigned to every pair of preservable duos and we seek to maximize the weight of the solution. While [28], discusses using weights to capture the positions of preserved duos within their respective strings, the weights in MWPSM can be arbitrary and are not required to be a function of position.

## 1.2 Related Work

The Maximum Duo-Preservation String Mapping Problem (MPSM) was introduced in [10] along with the related Constrained Maximum Induced Subgraph (CMIS) and Constrained Minimum Induced Subgraph (CNIS) problems. They used a linear programming and randomized rounding approach to approximate the  $k$ -CMIS problem which they show is a generalization of  $k$ -MPSM. This led to a  $k^2$ -approximation for  $k \geq 3$  and a 2-approximation for  $k = 2$ . This was improved by [6] to a 4-approximation independent of  $k$  and running in  $O(n^{3/2})$  time as well as approximation ratios of 3 for  $k = 3$  and  $8/5$  for  $k = 2$ . [6] also showed that  $k$ -MPSM is APX-hard even for  $k = 2$ , meaning no polynomial-time approximation scheme (PTAS) exists assuming  $P \neq NP$ . The approximation was subsequently improved to 3.5 using local search [5], 3.25 using a combinatorial triplet matching approach [7], and finally  $2 + \epsilon$  for any  $\epsilon > 0$  in  $n^{O(1/\epsilon)}$  time, again using local search [16]. The work of [16] also presented a 2.67-approximation running in  $O(n^2)$  time.

The recent interest in MPSM led to the study of several variants including Maximum Weight Duo-preservation String Mapping (MWPSM),  $k$ -MPSM, and fixed-parameter tractability (FPT). The weighted variant of MPSM was introduced in [28] along with an algorithm achieving a 6-approximation. That work was the first to apply the local ratio technique developed by Bar-Yehuda and Even [2] to an MPSM problem. Recent work on  $k$ -MPSM led to a  $(1.4 + \epsilon)$ -approximation for 2-MPSM [32]. On the FPT side, [3] showed that MPSM is fixed-parameter tractable when parameterized by the number of preserved duos and [27] achieved a faster running time with a randomized algorithm.

The Minimum Common String Partition problem (MCSP) has been extensively studied from many angles including polynomial-time approximation [10, 12, 13, 20, 26, 25], fixed-parameter tractability [8, 14, 23, 9], and heuristics [17, 4, 18]. FPT algorithms have been parameterized by maximum number of times any character occurs, minimum block size, and the size of the optimal minimum partition. Heuristic approaches range from an ant colony optimization algorithm [17] to integer linear programming (ILP) based strategies [4, 18] which in some cases solve the problem optimally for strings up to 2,000 characters in length.

The problem was shown to be NP-hard (thus implying MPSM is also NP-hard) and APX-hard even for 2-MCSP [20]. The current best approximations are an  $O(\log n \log^* n)$ -

approximation due to [13] for general MCSP bases on the related edit distance with moves problem and an  $O(k)$ -approximation for  $k$ -MCSP due to [26]. Applications to evolutionary distance and genome rearrangement can be found in [31, 11].

**Unclaimed results in prior work:** An analysis of prior work shows that 4-approximations to both problems studied here can be achieved using slight modifications to existing work. For MWPSM, the algorithm in [6] can be extended by choosing a maximum weight matching and partition rather than maximum cardinality. For the unweighted problem, Goldstein and Lewenstein [21] showed an  $O(n)$  time greedy algorithm for MCSP. Although not discussed in their paper which pre-dated MPSM, we note that the greedy algorithm for MCSP achieves a 4-approximation for MPSM by a fairly straightforward charging argument. Formal proofs of these claims are outside the scope of this paper and we leave them to the interested reader. Additionally, we will not refer to these approximations when comparing our work to previous best known results. We simply mention them here for completeness and to give a nod to two nice papers in the area.

### 1.3 Our Contributions

We show a transformation of the Maximum Duo-Preservation String Mapping (MPSM) problem into a related tractable problem. This transformation leads to new algorithms for both weighted and unweighted MPSM. For the weighted case, we present an  $8/3$ -approximation running in  $O(n^3)$  time. This improves upon the previous best 6-approximation in polynomial time [28] (a tighter bound on the running time is not given in the paper). It also matches the best quadratic time approximation for the unweighted problem of 2.67 and approaches the best unweighted approximation of  $2 + \epsilon$  for any  $\epsilon > 0$  in  $n^{O(1/\epsilon)}$  time, both due to [16]. We further show in Corollary 2 that we can improve the running time at the cost of a weaker approximation. For the unweighted case, we present the first linear time algorithm with an  $8/3$ -approximation again matching the previous best quadratic time algorithm and coming fairly close to the best known  $(2 + \epsilon)$ -approximation achieved by a significantly larger running time. In particular, the move from quadratic to linear time in length of the strings is significant for practical settings wherein the string length may be long enough that quadratic time is prohibitive. Finally, we introduce the first streaming algorithm for MPSM in the streaming model where each string is read one character at a time. We show that a single pass suffices to find a 4-approximation on the size of an optimal solution using only  $O(\alpha^2 \lg n)$  space.

In addition, the techniques here are novel to this problem and may inspire future improvements. While [7] also used a form of triplet matching, the structure of the triplet matching is different as is the approach to achieving a feasible solution to MPSM. Our main results are summarized in the theorems below.

► **Theorem 1.** *There exists an algorithm which finds an  $8/3$ -approximation to MWPSM on strings of length  $n$  in  $O(n^3)$  time.*

► **Corollary 2.** *Using the approximate weighted matching algorithm of [15], we can find an  $8/(3(1-\epsilon))$ -approximation to MWPSM on strings of length  $n$  for any  $\epsilon > 0$  in  $O(n^2 \epsilon^{-1} \lg \epsilon^{-1})$  time.*

► **Theorem 3.** *There exists an algorithm which finds an  $8/3$ -approximation to MPSM on strings of length  $n$  over alphabets of size  $\alpha$  in  $O(n + \alpha^7)$  time.*

► **Corollary 4.** *There exists an algorithm which finds an 8/3-approximation to MPSPM on strings of length  $n$  over constant-sized alphabets in  $O(n)$  time.*

► **Theorem 5.** *There exists a single-pass streaming algorithm which finds a 4-approximation to the size of an MPSPM on strings of length  $n$  over alphabets of size  $\alpha$  using only  $O(\alpha^2 \lg n)$  space.*

## 1.4 Preliminaries

Let  $A = a_1 a_2 \dots a_n$  and  $B = b_1 b_2 \dots b_n$  be two strings of length  $n$  with  $a_i$  and  $b_j$  being the  $i^{\text{th}}$  and  $j^{\text{th}}$  characters of their respective strings. A *duo*  $D_i^A = (a_i, a_{i+1})$  contains a pair of consecutive characters  $a_i$  and  $a_{i+1}$ . We use  $D^A = (D_1^A, \dots, D_{n-1}^A)$  and  $D^B = (D_1^B, \dots, D_{n-1}^B)$  to denote the sets of *duos* for  $A$  and  $B$ , respectively. We similarly define a *triplet*  $T_i^A = (a_i, a_{i+1}, a_{i+2})$  as a set of three consecutive characters  $a_i$ ,  $a_{i+1}$ , and  $a_{i+2}$  in the string and sets of *triplets*  $T^A = (T_1^A, \dots, T_{n-2}^A)$  and  $T^B = (T_1^B, \dots, T_{n-2}^B)$  for strings  $A$  and  $B$ , respectively. Observe that the duos  $D_i^A$  and  $D_{i+1}^A$  correspond to the first two and last two characters, respectively, of the triplet  $T_i^A$ . We refer to duos  $D_i^A$  and  $D_{i+1}^A$  as *subsets* of the triplet  $T_i^A$ .

A proper mapping  $\pi$  from  $A$  to  $B$  is a one-to-one mapping from the letters of  $A$  to the letters of  $B$  with  $a_i = b_{\pi(i)}$  for all  $i = 1, \dots, n$ . Recall that a duo  $(a_i, a_{i+1})$  is preserved if and only if  $a_i$  is mapped to some  $b_j$  and  $a_{i+1}$  is mapped to  $b_{j+1}$ . We call a pair of duos  $(D_i^A, D_j^B)$  *preservable* if and only if  $a_i = b_j$  and  $a_{i+1} = b_{j+1}$ . For MWPSM, let  $w(D_i^A, D_j^B)$  be the weight gained by mapping  $D_i^A$  to  $D_j^B$ .

For consistency, we define the concept of conflicting pairs of duos using the terminology of [6]. Two preservable pairs of duos  $(D_i^A, D_j^B)$  and  $(D_h^A, D_\ell^B)$  are said to be *conflicting* if no proper mapping can preserve both of them. These conflicts can be of two types type 1 and type 2. In *type 1 conflicts*, either  $i = h \wedge j \neq \ell$  or  $i \neq h \wedge j = \ell$ . In *type 2 conflicts*, either  $i = h + 1 \wedge j \neq \ell + 1$  or  $i \neq h + 1 \wedge j = \ell + 1$ .

The algorithms here only show how to map the characters of the preserved duos. In all cases, note that any unmapped characters can be mapped arbitrarily to identical characters in the other string in linear time.

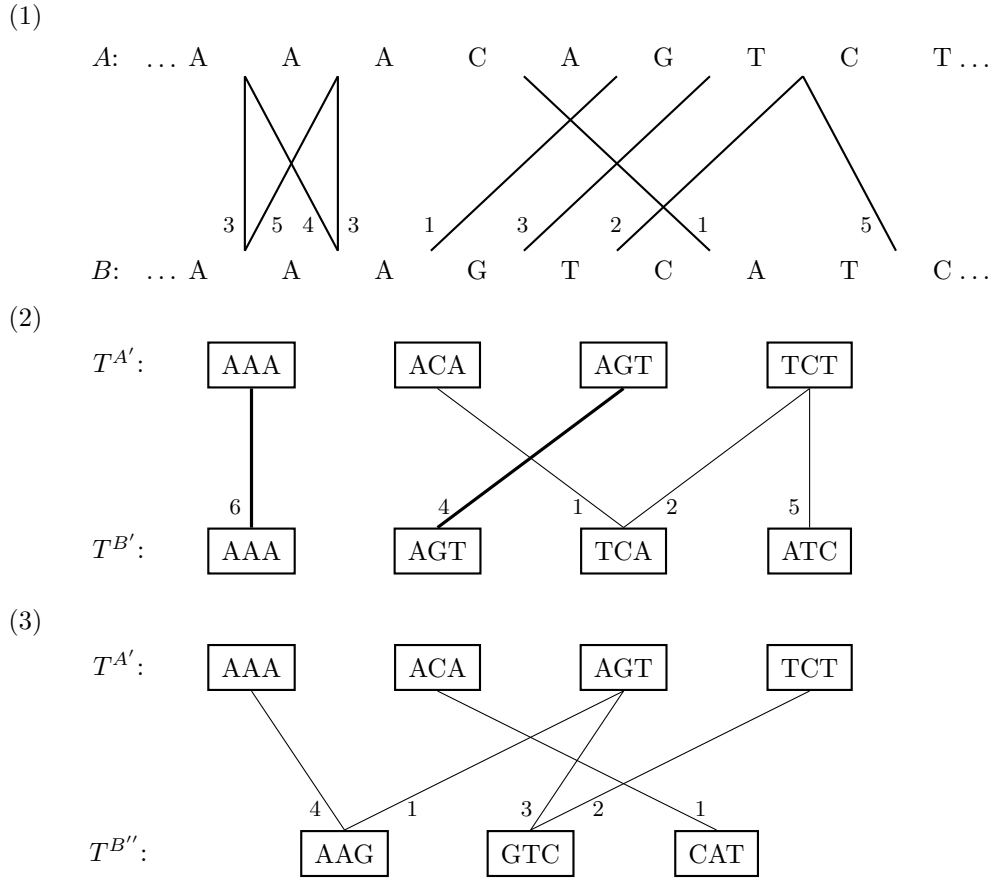
## 2 Main techniques and algorithm for MWPSM

For both algorithms, we first solve a weighted bipartite matching problem we call Alternating Triplet Matching (ATM). In this section, we define ATM, show that a solution to ATM has weight at least 3/4 of an optimal solution to MWPSM, and finally show that we can convert a solution to ATM to a feasible duo mapping while preserving 1/2 of its weight. Combining these facts leads to an 8/3-approximation to MWPSM.

### 2.1 The Alternating Triplet Matching (ATM) problem

Here, we define this problem in terms of MWPSM. Modifications for the unweighted variant (to admit a faster solution) will be defined in Section 3. Let  $T^{A'} = \{T_i^A \mid i \text{ is odd}\}$ ,  $T^{B'} = \{T_i^B \mid i \text{ is odd}\}$  and  $T^{B''} = \{T_i^B \mid i \text{ is even}\}$ . Throughout the paper, we refer to triplets starting at odd indices in their respective strings as *odd triplets* and similarly use the term *even triplets*. Note, we do not use the even triplets from  $A$ .

Using these subsets, we formulate bipartite matching problems on two separate graphs  $G' = \{T^{A'}, T^{B'}, E'\}$  and  $G'' = \{T^{A'}, T^{B''}, E''\}$ . The edges of  $G'$  depend on the letters in the triplets. Consider triplets  $T_i^{A'} = (D_i^A, D_{i+1}^A)$  and  $T_j^{B'} = (D_j^B, D_{j+1}^B)$ . For each pair of duos



■ **Figure 2** Illustration of how to generate an ATM instance from an MWPSM instance. (1) Substrings of the original two strings,  $A$  and  $B$ , starting at some odd index and featuring weighted edges representing the weight of preserving a pair of duos. (2) The graph  $G'$  with thicker edges representing an exact match between two triplets. In the case of multiple edges between a pair of triplets (e.g. the five edges between the “AAA” triplets), we only show the heaviest weight edge. (3) The graph  $G''$ . Note that that the weight of a mapping which maps the two “AGTC” strings to each other is 6, which can be achieved by a matching in  $G'$ , but not in  $G''$ .

$D_h^A$  and  $D_\ell^B$  with  $h \in \{i, i + 1\}$ ,  $\ell \in \{j, j + 1\}$ , and  $D_h^A = D_\ell^B$ , we add an edge  $e = (T_i^{A'}, T_j^{B'})$  with weight  $w(e) = w(D_h^A, D_\ell^B)$ . Additionally, if  $T_i^{A'} = T_j^{B'}$ , we add an edge  $e = (T_i^{A'}, T_j^{B'})$  between them with weight  $w(e) = w(D_i^A, D_j^B) + w(D_{i+1}^A, D_{j+1}^B)$ . In other words, the edge gets the combined weight of the duo pairs preserved by mapping the substring  $T_i^{A'}$  to the substring  $T_j^{B'}$ . The graph  $G''$  is defined similarly. There could be up to five edges total if the triplets contain one letter repeated (e.g. “AAA”). In the case of multiple edges between a pair of triplets, we only need to consider the heaviest edge among them since each triplet can be matched at most once. However, we keep all edges for the sake of simplifying some of the proofs. Figure 2 illustrates the procedure of generating an ATM instance.

## 2.2 MWPSM algorithm and analysis

Let  $OPT_{G'}$  and  $OPT_{G''}$  be the weights of maximum weight matchings in  $G'$  and  $G''$ , respectively. Note that we can find these matchings in the time it takes to compute maximum weight bipartite matching. Since our graphs have  $O(n)$  vertices and could have  $O(n^2)$  edges,

this takes  $O(n^2 \lg n + n \cdot n^2) = O(n^3)$  time [19]. Lemma 6 states that either  $OPT_{G'}$  or  $OPT_{G''}$  will be a  $(3/4)$ -approximation to the weight of an optimal solution to MWPSM,  $OPT_{MWPSM}$ . Let  $OPT_{ATM} = \max(OPT_{G'}, OPT_{G''})$ .

► **Lemma 6.**  $OPT_{ATM} \geq (3/4)OPT_{MWPSM}$ .

**Proof.** We divide the edges of  $OPT_{MWPSM}$  into two partitions. The first partition,  $P^{same}$ , includes mappings, in which both letters occur at odd indices or both letters occur at even indices. The second partition,  $P^{diff}$ , includes the remaining mappings wherein one letter is at an odd index and the other is at an even index (this could be odd from  $A$ , even from  $B$  or even from  $A$ , odd from  $B$ ).

Note that the mapping of each preserved pair of duos  $(D_i^A, D_j^B)$  will be contained in one of these two partitions. Without loss of generality, let the weight of  $P^{same}$  be at least the weight of  $P^{diff}$ . We show how to transform  $OPT_{MWPSM}$  into a feasible bipartite matching in  $G'$  while retaining the full weight of  $P^{same}$  and at least half of the weight of  $P^{diff}$ . Thus, we retain at least  $3/4$  of the weight of  $OPT_{MWPSM}$ .

For each triplet in the vertex set of  $G'$  that contains one or two preserved duos from  $P^{same}$ , we can add an edge to our matching with weight equal to the weight of the preserved duos. This works because consecutive pairs of preserved duos  $(D_i^A, D_j^B)$  and  $(D_{i+1}^A, D_{j+1}^B)$  with  $i$  and  $j$  both being odd will correspond to a “double” edge in the ATM instance with weight equal to  $w(D_i^A, D_j^B) + w(D_{i+1}^A, D_{j+1}^B)$ . On the other hand, if  $i$  and  $j$  are both even, then the duos of  $(D_i^A, D_j^B)$  and  $(D_{i+1}^A, D_{j+1}^B)$  are contained in four different triplets and will be added separately. Thus, we can maintain all of the weight of  $P^{same}$  in a matching in  $G'$ .

A slightly trickier case arises with  $P^{diff}$ . Any consecutive pairs of preserved duos  $(D_i^A, D_j^B)$  and  $(D_{i+1}^A, D_{j+1}^B)$  in  $P^{diff}$  will have  $i$  and  $j$  of different parity. This results in the duos being contained in three triplets, two from one partition and one from the other. That means the edges in the ATM instance capturing the weights of the two pairs will be conflicting. Thus we can only preserve the weight of one of the two pairs in our ATM solution. To guarantee that we add at least half of the weight of  $P^{diff}$  to our solution, we further partition it into pairs  $(D_i^A, D_j^B)$  with  $i$  being odd and those with  $i$  being even. Then we simply choose the heavier of those two partitions to add to our ATM solution.

For the case where  $P^{diff}$  is heavier than  $P^{same}$ , we can do a similar construction for  $G''$ . Thus, our ATM solution in either  $G'$  or  $G''$  could have at least  $3/4$  the weight of an optimal solution to MWPSM. ◀

We can now show how to transform an optimal solution to ATM (the heavier of the two matchings) into a feasible string mapping which preserves at least half of the weight of the ATM solution. Let  $G = (D^A, D^B, E)$  be a bipartite graph on the duos of  $A$  and  $B$  with edge weights equal to the weight of preserving each pair of duos. We first show how to convert an ATM solution into a matching  $M$  in  $G$ . Then, we show how to resolve conflicts of type 2 (conflicts of type 1 will not arise since  $M$  is a matching).

The transformation is simply a reversal of how we constructed the ATM graphs. For each edge between triplets in our ATM solution (the heavier of the two matchings in  $G'$  and  $G''$ ), we add an edge or edges to  $M$  corresponding to the duos that “created” that triplet edge.

To resolve conflicts, we consider the conflict graph  $C$  wherein we have a node for each edge in  $M$  and an arc between nodes if their corresponding edges are in conflict. We can prove that  $C$  has maximum degree 2, meaning it will be a collection of paths and cycles. Further, we note that each cycle will have even length due to Lemma 7 and the fact that the underlying graph is bipartite. Thus, for each path or cycle, we choose the heavier of

the two maximal independent sets in that path or cycle to add to our final MPSM solution. Lemma 7 establishes that  $C$  has maximum degree 2.

► **Lemma 7.** *Each edge in  $M$  conflicts with at most one other edge at each endpoint.*

**Proof.** First, we note that each duo is contained in at most one triplet edge from the ATM solution and therefore can only be matched once in  $M$ . In other words,  $M$  is a classical matching in the bipartite graph of duos. This follows from the fact that consecutive triplets in a string starting at only odd (or only even) indices will overlap at exactly one letter.

This ensures that no conflicts of type 1 can arise since that would require a duo to be matched twice. We can also show that at most one conflict of type 2 arises at each endpoint. Without loss of generality, consider the endpoint  $D_i^A$ . Consider the duos  $D_{i-1}^A$  and  $D_{i+1}^A$  where such a conflict might arise. Notice that one of these duos must have come from the same triplet as  $D_i^A$ , while the other comes from a different triplet. The duo from the same triplet will either be unmatched or matched as a non-conflicting parallel edge. Thus no conflict arises from that duo. The duo from a different triplet could contribute at most one conflicting edge by the above claim that each duo is matched at most once. Applying this argument to both endpoints of a given edge completes the proof. ◀

► **Lemma 8.**  *$M$  can be converted into  $M'$ , a feasible solution to MWPSM, such that the weight of  $M'$  is at least  $(1/2)OPT_{ATM} \geq (3/8)OPT_{MWPSM}$ .*

**Proof.** The conflict graph on the edges of  $M$  must be a collection of paths and even length cycles since it has maximum degree 2 and  $G$  is bipartite. We can simply decompose each path or cycle into two independent sets and choose the heavier of the two. This operation discards at most half of the weight of  $M$  while removing all conflicts and leaving us with a feasible solution to MWPSM. ◀

The proofs of Theorem 1 and Corollary 2 follow from the preceding lemmas.

### 3 Linear time algorithm for unweighted MPSM

The basic approach follows roughly the same steps as the weighted algorithm from Section 2: construct an ATM instance, solve the matching problem, transform the solution into a duo matching on the strings, and resolve conflicts. We show that with a small modification, each step can be done in linear time for the unweighted problem. The key insight that allows for this speedup is that identical triplets can be collapsed into single vertices and we can solve a  $b$ -matching problem we call  $b$ -ATM. In the  $b$ -matching variant of classical matching, each vertex in the graph has a capacity and can be matched that many times. We will abuse notation a bit and refer to each vertex as having capacity  $b$ , although we actually allow the capacity of each node to be different. The following subsections illustrate how to perform the aforementioned steps and bound the running time of each step.

#### 3.1 Constructing the $b$ -ATM instance in $O(n + \alpha^4)$ time

We construct a triplet matching problem as in Section 2.1 with one crucial adjustment: identical triplets are collapsed into single vertices with capacity equal to the number of occurrences of that triplet in its given set ( $T^{A'}$ ,  $T^{B'}$ , or  $T^{B''}$ ). The number of times each vertex is allowed to be matched is equal to its capacity. Similarly, each edge can be matched multiple times up to the smaller capacity among its two endpoints. Algorithm 1 shows how to construct a  $b$ -ATM instance from the two input strings in linear time.

**Algorithm 1:** CONSTRUCT B-ATM

- 
- 1 Traverse each string to build a set of triplets with counts for  $A'$ ,  $B'$ , and  $B''$ .
  - 2 For  $G'$  and  $G''$ , create a vertex for each triplet with capacity equal to its count. Add edges between the triplets as in Section 2.1 with the following modification. If two triplets match exactly, give the edge weight 2 and if they only share a duo in common, give the edge weight 1.
- 

**Algorithm 2:** SOLVE B-ATM

- 
- 1 Add each edge with weight 2, corresponding to two identical triplets, to the matching.
  - 2 Find a maximum b-matching in the remaining “unweighted” graph using maximum flow techniques.
- 

As in Section 2.1, let  $OPT_{G'}$  and  $OPT_{G''}$  be the weights of maximum weight b-matchings in  $G'$  and  $G''$ , respectively. Lemma 9 states that either  $OPT_{G'}$  or  $OPT_{G''}$  will be a  $(3/4)$ -approximation to the size of an optimal solution to MPSM,  $OPT_{MPSM}$ . Let  $OPT_{b-ATM} = \max(OPT_{G'}, OPT_{G''})$  as constructed by Algorithm 1.

► **Lemma 9.**  $OPT_{b-ATM} \geq (3/4)OPT_{MPSM}$ .

**Proof.** This proof follows from Lemma 6. Suppose we constructed an ATM instance as in Section 2.1, but for the unweighted problem. By Lemma 6, we would have  $OPT_{ATM} \geq (3/4)OPT_{MPSM}$ . Now note that we can collapse all identical triplet vertices in each partition of  $OPT_{ATM}$  to get a feasible solution to the  $b$ -ATM problem without reducing the weight. ◀

► **Lemma 10.** *Algorithm 1 constructs a graph with  $O(\alpha^3)$  vertices and  $O(\alpha^4)$  edges in  $O(n + \alpha^4)$  time.*

**Proof.** Step 1 of the algorithm clearly runs in less than  $O(n + \alpha^4)$  time. It simply traverses each string once, storing the triplets in some appropriate data structure with constant insert and query time.

To bound the running time of step 2, we first bound the number of edges created. Note that the bipartite graph of  $b$ -ATM has  $O(\alpha^3)$  vertices in each partition since that is the maximum number of 3-mers in an alphabet of size  $\alpha$ . To bound the edge set, notice that for any 3-mer, there exist at most  $4\alpha$  other 3-mers with a substring of length 2 in common. Thus, the max degree of each node is  $O(\alpha)$  and the size of the edge set  $E$  is at most  $O(\alpha^4)$ . When adding edges, we can check for the existence of each edge in constant time, again assuming the triplet are stored in some appropriate data structure. ◀

### 3.2 Solving $b$ -ATM quickly

Algorithm 2 shows how to solve  $b$ -ATM within our time constraints. Lemma 11 proves the correctness of this algorithm while Lemma 12 bounds its running time.

► **Lemma 11.** *Algorithm 2 finds a maximum weight b-matching in the  $b$ -ATM instance.*

**Proof.** Here, we need to justify Step 1 of Algorithm 2 by showing that there always exists some maximum  $b$ -matching which contains all of the edges corresponding to identical pairs of triplets. First note that it is feasible to include all such edges since they can never conflict with each other. For each triplet in one partition, there is at most one identical triplet in the other partition.

**Algorithm 3:** TRANSFORM B-ATM TO MPSM

- 
- 1 Assign each copy of a 3-mer and its edge from the b-ATM solution to a triplet from the original strings to get an ATM solution.
  - 2 Transform the ATM solution into a duo matching as detailed in Section 2.2
  - 3 Resolve conflicts by traversing the paths/cycles of the conflict graph and discarding every other edge.
- 

We apply the following claim iteratively to complete the proof. Given a maximum weight  $b$ -matching  $M$  which does not include all identical pair edges, we can always add one such edge without decreasing the weight of the solution. Consider an arbitrary identical pair edge  $e$  that is not in  $M$ . To add  $e$  to  $M$  we need to remove at most two edges from  $M$ , one for each endpoint of  $e$ . Since  $e$  has a weight of 2 while the removed edges have weights of 1 each, swapping those edges for  $e$  will not reduce the weight of the solution. ◀

► **Lemma 12.** *Algorithm 2 runs in  $O(n)$  time plus the time to compute an unweighted maximum  $b$ -matching on a graph with  $O(\alpha^3)$  vertices and  $O(\alpha^4)$  edges and total capacity  $O(n)$ . Using current maximum flow algorithms, Algorithm 2 can run in  $O(n + \alpha^7)$  time.*

**Proof.** If the graph were unweighted, we could find a maximum  $b$ -matching in  $O(|V||E|) = O(\alpha^7)$  time using the maximum flow approach in [30]. Fortunately, by Lemma 11, we can first add all edges with weight 2 to our solution. Thus, we are left with an “unweighted” residual problem that can be solved using a maximum flow algorithm. ◀

### 3.3 Transforming the $b$ -ATM solution to a duo matching and resolving conflicts

Now that we have solved our  $b$ -ATM problem we need transform it back to a duo matching. The obvious challenge here is that each  $b$ -ATM vertex represents roughly  $b$  copies of a given 3-mer that must each be assigned to a triplet in the original string in linear time while preserving the weight of the  $b$ -ATM solution. There are  $b!$  such assignments and  $b$  could be on the order of  $n$ . However, the important observation here is that we can do this *arbitrarily* and still preserve the size of the  $b$ -ATM solution.

► **Lemma 13.** *Algorithm 3 constructs a feasible solution to MPSM with size equal to half the weight of  $OPT_{b-ATM}$ .*

**Proof.** The proof follows from Lemma 8. Notice that we assign exactly one copy of a 3-mer to each triplet and the result is a feasible solution to the ATM problem. ◀

► **Lemma 14.** *Algorithm 3 runs in  $O(n)$  time.*

**Proof.** Assigning each copy of a 3-mer and its edge to a triplet can be done in constant time if we maintain lists of the indices at which each 3-mer occurs in each string, resulting in  $O(n)$  time overall. Similarly, generating the duo-matching can easily be done in  $O(n)$  time. Resolving conflicts in the unweighted problem involves traversing  $O(n)$  edges and removing every other one which can be done in  $O(n)$  time as well. ◀

The proofs of Theorem 3 and Corollary 4 follow from the preceding lemmas.



#### 4 A streaming algorithm for MPSM

We observe that the algorithm of [6] can be adapted into a single-pass streaming algorithm in the streaming model where each string is read one character at a time. We present an algorithm using  $O(\alpha^2 \lg n)$  space and giving a 4-approximation of the size of an MPSM solution without providing an explicit mapping. In [6], they upper bound MPSM by a maximum matching in the duo graph. Then they show that a feasible MPSM solution can be found while preserving at least  $1/4$  of the edges in the matching.

The algorithm is simple. Maintain a counter for each 2-mer in the alphabet and a counter for the size of the matching. While processing the first string, count the number of occurrences of each 2-mer. For the second string, each time you encounter a duo with a nonzero count, decrease its count by 1 and increase the size of the matching by 1. At the end, divide the size of the matching by 4 to get a 4-approximation to the size of the optimal MPSM. The following Lemmas establish the space-efficiency and correctness of the the algorithm.

► **Lemma 15.** *The streaming algorithm uses only  $O(\alpha^2 \lg n)$  space where  $\alpha$  is the alphabet size and  $n$  is the length of the strings.*

**Proof.** The number of 2-mers from an alphabet of size  $\alpha$  is  $\alpha^2$ . We require only  $O(\lg n)$  bits of space for each 2-mer counter since no 2-mer could appear more than  $O(n)$  times where  $n$  is the length of the strings. Similarly, we keep just one counter for the size of the matching which requires only  $O(\lg n)$  bits of space since the size of the matching is at most  $n$ . In addition to the counters, we must store the previously seen letter since our streaming model involves reading one character at a time, but we are counting duos. However, this only requires  $O(\lg \alpha)$  space. ◀

► **Lemma 16.** *The streaming algorithm achieves a 4-approximation to MPSM.*

**Proof.** We first show that the size of a maximum matching in a bipartite duo graph  $G$  as defined in [6] is equal to the sum of the minimum number of occurrences of each duo among the two strings. Notice that  $G$  can be decomposed into a set of connected components for each 2-mer since each vertex only has edges to other vertices corresponding to the same 2-mer. Further, each of these connected components is a complete bipartite graph with maximum matching size equal to the minimum size of the two partitions.

Thus, computing the above sum gives us the size of the maximum matching. We note that the number of times the matching size counter increase due to vertices of a given 2-mer is exactly equal to the minimum number of times that 2-mer appears in either of the two strings.

Finally, as shown in [6], a maximum matching in the duo graph is an upper bound on the optimal solution to MPSM and can always be converted into a feasible MPSM solution while preserving at least  $1/4$  of its size. ◀

The proof of Theorem 5 follows from Lemmas 15 and 16.

#### 5 Conclusion and future directions

We showed a transformation of the Maximum Duo-Preservation String Mapping (MPSM) problem into a related tractable problem. This led to new algorithms for both MWPSM and MPSM. For the weighted case, we presented a tighter approximation closing in on the best unweighted result using a reasonably fast algorithm. We also showed that the

running time could be improved at the expense of a slightly weaker approximation. For the unweighted case, we presented the first linear time algorithm with an approximation matching the previous best quadratic time algorithm and fairly close to the best known approximation achieved by a significantly larger running time. Finally, we presented the first streaming algorithm for MPSM showing that a constant approximation is achievable in the single-pass streaming model.

We believe the most pressing future direction is to explore the applications and utility of this problem further. The complementary relationship with Minimum Common String Partition (MCSP) has driven much of the current interest in MPSM. However, given their relationship, new approximations for MPSM do not directly lead to any improvements for MCSP. It is reasonable to ask if the study of MPSM can teach us anything about MCSP or at least inspire new heuristics. We note that some current linear-time algorithms for MCSP are greedy algorithms [21] with a proven lower bound of  $\Omega(n^{0.46})$  [24] (Although this bound arises from carefully constructed strings over a  $(\log n)$ -sized alphabet). This is in contrast to the best known approximation for MCSP,  $O(\log n \log^* n)$  [13]. Perhaps the linear time MPSM algorithm presented here could be combined with greedy approaches leading to better, more robust heuristics. Further, since MPSM currently appears to be “easier” than MCSP, it would be fruitful to explore more applications for MPSM itself in bioinformatics, data compression, and beyond.

On the theoretical side, the biggest questions revolve around the factor of 2 approximation. Is this tight for MPSM conditioned on some hardness conjecture or can we do better? It surely seems like a natural bound. Regardless, can we achieve a 2-approximation in linear time? Likewise, for MWPSM, a 2-approximation could be seen as the next major goal. All of this seems within reach, using existing ideas or different tools such as LP rounding techniques. Another direction would be to add edit operations. It seems that MWPSM could be adapted to handle the cost of substitutions. However, this is nontrivial since existing algorithms assume that letters which do not belong to preserved duos can be mapped at no penalty.

Finally, we propose variants of MWPSM that may admit a faster approximation than we see in this paper. Suppose the weights are not arbitrary, but follow some “rules”. [28] suggested the weight of a duo-preservation could be a function of the “closeness” of the mapping in terms of the positions of the characters in their respective strings. However, [28] and this paper consider only arbitrary weights. One could imagine a weight function like  $w(D_i^A, D_j^B) = n - |i - j|$  that does not require us to examine every edge in the duo graph. Of course, the function need not be so naive as any metric or geometric weight functions admit faster matching algorithms [1].

---

## References

- 1 Pankaj K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 555–564, New York, NY, USA, 2014. ACM.
- 2 R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. In G. Ausiello and M. Lucertini, editors, *Analysis and Design of Algorithms for Combinatorial Problems*, volume 109 of *North-Holland Mathematics Studies*, pages 27–45. North-Holland, 1985.
- 3 Stefano Beretta, Mauro Castelli, and Riccardo Dondi. Parameterized tractability of the maximum-duo preservation string mapping problem. *CoRR*, abs/1512.03220, 2015. [arXiv: 1512.03220](https://arxiv.org/abs/1512.03220).

- 4 Christian Blum, José A. Lozano, and Pinacho Davidson. Mathematical programming strategies for solving the minimum common string partition problem. *European Journal of Operational Research*, 242(3):769–777, 2015.
- 5 Nicolas Boria, Gianpiero Cabodi, Paolo Camurati, Marco Palena, Paolo Pasini, and Stefano Quer. A  $7/2$ -approximation algorithm for the maximum duo-preservation string mapping problem. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:8, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 6 Nicolas Boria, Adam Kurpisz, Samuli Leppänen, and Monaldo Mastrolilli. Improved approximation for the maximum duo-preservation string mapping problem. In *Algorithms in Bioinformatics: 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 14–25, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 7 Brian Brubach. Further improvement in approximating the maximum duo-preservation string mapping problem. In Martin Frith and Christian Nørgaard Storm Pedersen, editors, *Algorithms in Bioinformatics*, pages 52–64, Cham, 2016. Springer International Publishing.
- 8 Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, and Irena Rusu. A fixed-parameter algorithm for minimum common string partition with few duplications. In Aaron Darling and Jens Stoye, editors, *Algorithms in Bioinformatics: 13th International Workshop, WABI 2013, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 244–258, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 9 Laurent Bulteau and Christian Komusiewicz. Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '14*, pages 102–121, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.
- 10 Wenbin Chen, Zhengzhang Chen, Nagiza F. Samatova, Lingxi Peng, Jianxiong Wang, and Maobin Tang. Solving the maximum duo-preservation string mapping problem with linear programming. *Theoretical Computer Science*, 530:1–11, 2014.
- 11 Xin Chen, Jie Zheng, Zheng Fu, Peng Nan, Yang Zhong, S. Lonardi, and Tao Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(4):302–315, Oct 2005.
- 12 Marek Chrobak, Petr Kolman, and Jiří Sgall. The greedy algorithm for the minimum common string partition problem. In Klaus Jansen, Sanjeev Khanna, José D. P. Rolim, and Dana Ron, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques: 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2004, and 8th International Workshop on Randomization and Computation, RANDOM 2004, Cambridge, MA, USA, August 22-24, 2004. Proceedings*, pages 84–95, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 13 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.
- 14 Peter Damaschke. Minimum common string partition parameterized. In Keith A. Crandall and Jens Lagergren, editors, *Algorithms in Bioinformatics: 8th International Workshop, WABI 2008, Karlsruhe, Germany, September 15-19, 2008. Proceedings*, pages 87–98, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 15 Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1:1–1:23, 2014.
- 16 Bartłomiej Dudek, Paweł Gawrychowski, and Piotr Ostropolski-Nalewaja. A family of approximation algorithms for the maximum duo-preservation string mapping problem. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz Interna-*

- tional Proceedings in Informatics (LIPIcs)*, pages 10:1–10:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 17 S. M. Ferdous and M. Sohel Rahman. Solving the minimum common string partition problem with the help of ants. In Ying Tan, Yuhui Shi, and Hongwei Mo, editors, *Advances in Swarm Intelligence: 4th International Conference, ICSI 2013, Harbin, China, June 12–15, 2013, Proceedings, Part I*, pages 306–313, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
  - 18 S. M. Ferdous and M. Sohel Rahman. An integer programming formulation of the minimum common string partition problem. *PLoS ONE*, 10(7):1–16, 07 2015.
  - 19 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987.
  - 20 Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem: Hardness and approximations. In *Proceedings of the 15th International Conference on Algorithms and Computation, ISAAC’04*, pages 484–495, Berlin, Heidelberg, 2004. Springer-Verlag.
  - 21 Isaac Goldstein and Moshe Lewenstein. Quick greedy computation for minimum common string partition. *Theor. Comput. Sci.*, 542:98–107, 2014.
  - 22 RC Hardison. Comparative genomics. *PLoS Biol*, 1(2):e58, 2003.
  - 23 Haitao Jiang, Binhai Zhu, Daming Zhu, and Hong Zhu. Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23(4):519–527, 2012.
  - 24 Haim Kaplan and Nira Shafir. The greedy algorithm for edit distance with moves. *Information Processing Letters*, 97(1):23–27, 2006.
  - 25 Petr Kolman and Tomasz Waleń. Approximating reversal distance for strings with bounded number of duplicates. *Discrete Applied Mathematics*, 155(3):327–336, 2007.
  - 26 Petr Kolman and Tomasz Waleń. Reversal distance for strings with duplicates: Linear time approximation using hitting set. In Thomas Erlebach and Christos Kaklamanis, editors, *Approximation and Online Algorithms: 4th International Workshop, WAOA 2006, Zurich, Switzerland, September 14–15, 2006. Revised Papers*, pages 279–289, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
  - 27 Christian Komusiewicz, Mateus de Oliveira Oliveira, and Meirav Zehavi. Revisiting the parameterized complexity of maximum-duo preservation string mapping. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
  - 28 Saeed Mehrabi. Approximating weighted duo-preservation in comparative genomics. In Yixin Cao and Jianer Chen, editors, *Computing and Combinatorics*, pages 396–406, Cham, 2017. Springer International Publishing.
  - 29 A. R. Mushegian. *Foundations of Comparative Genomics*. Elsevier, 1 2007.
  - 30 James B. Orlin. Max flows in  $O(nm)$  time, or better. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing, STOC ’13*, pages 765–774, New York, NY, USA, 2013. ACM.
  - 31 Krister M. Swenson, Mark Marron, Joel V. Earnest-Deyoung, and Bernard M. E. Moret. Approximating the true evolutionary distance between two genomes. *J. Exp. Algorithmics*, 12:3.5:1–3.5:17, 2008.
  - 32 Yao Xu, Yong Chen, Guohui Lin, Tian Liu, Taibo Luo, and Peng Zhang. A  $(1.4 + \epsilon)$ -approximation algorithm for the 2-max-duo problem. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th International Symposium on Algorithms and Computation (ISAAC 2017)*, volume 92 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 66:1–66:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

# Nearest constrained circular words

## Guillaume Blin

LaBRI, Université de Bordeaux, CNRS UMR 5800, Talence, France  
guillaume.blin@labri.fr

## Alexandre Blondin Massé<sup>1</sup>

Dép. d'informatique, Université du Québec à Montréal, QC, Canada  
blondin\_masse.alexandre@uqam.ca

## Marie Gasparoux<sup>2</sup>

LaBRI, Université de Bordeaux, CNRS UMR 5800, Talence, France  
Dép. d'informatique et de recherche opérationnelle, Université de Montréal, QC, Canada  
marie.gasparoux@umontreal.ca

## Sylvie Hamel<sup>3</sup>

Dép. d'informatique et de recherche opérationnelle, Université de Montréal, QC, Canada  
sylvie.hamel@umontreal.ca

## Élise Vandomme

LaCIM, Université du Québec à Montréal, QC, Canada  
elise.vandomme@lacim.ca

---

### Abstract

In this paper, we study circular words arising in the development of equipment using shields in brachytherapy. This equipment has physical constraints that have to be taken into consideration. From an algorithmic point of view, the problem can be formulated as follows: Given a circular word, find a constrained circular word of the same length such that the Manhattan distance between these two words is minimal. We show that we can solve this problem in pseudo polynomial time (polynomial time in practice) using dynamic programming.

**2012 ACM Subject Classification** Theory of computation → Dynamic programming, Mathematics of computing → Combinatorics on words, Applied computing → Bioinformatics

**Keywords and phrases** Circular constrained alignments, Manhattan distance, Application to brachytherapy

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.6

**Supplement Material** A Haskell implementation of the algorithms discussed in this paper is available at <https://gitlab.com/ablondin/nearest-constrained-circular-words>

---

<sup>1</sup> Supported by the Individual Discovery Grant RGPIN-417269-2013 from National Sciences and Engineering Research Council of Canada (NSERC).

<sup>2</sup> Partially supported by PoPRA project funded by Conseil Régional d'Aquitaine and European FEDER and IdEx Bordeaux.

<sup>3</sup> Supported by the Individual Discovery Grant RGPIN-2016-04576 from National Sciences and Engineering Research Council of Canada (NSERC).



## 1 Introduction

The profusion of circular molecular DNA sequences (plasmids, mitochondrial DNA, bacterial chromosomes, etc.) and the need to compare them have opened the way to the elaboration of time efficient algorithms for the alignment problem of circular words. The first efficient algorithm for this problem was presented by Maes in 1990 [8] and uses a divide-and-conquer approach to find the edit or Levenshtein distance [6] between two circular words of lengths  $m$  and  $n$  in  $\mathcal{O}(nm \log(n))$  time. It is based on the classical Wagner and Fischer algorithm that finds an alignment and edit distance between two words [12]. Since then, more time efficient algorithms have been developed, using dynamic programming [3], branch-and-bound techniques [9], exploring either approximate or suboptimal solutions [1, 10] or alignment-free methods [2, 4]. In this article, we focus on a related problem which is to find a constrained circular word as close as possible of a given circular word according to the Manhattan distance. This problem comes from currently explored brachytherapy techniques.

*Brachytherapy* is a form of internal radiotherapy involving short distance (*brachys* in Greek) irradiation. A treatment is performed by placing small sealed radiation sources, also called *seeds*, near or into the tumor site [11]. The seed is moved step by step through a catheter, to irradiate the whole tumor. The radiation dose received at each point depends on the time spent by the seed at this position. The resulting irradiation field is uniform around the seed, regardless of the shape and position of the tumor. Therefore, the use of shields is conceptually considered to modulate more precisely this field. A cylindrical shield encapsulating the seed would block radiation, except through some prescribed openings. Let us associate each stopping position of the seed with the prescription doses for its surrounding circular area. This area of interest is divided, as precisely as possible, in  $n$  equal sections. The dose for each section is either a positive integer  $x$  (tumor tissue; irradiation prescribed for  $x$  units of time) or a 0 (healthy tissue; no irradiation needed) so that each stopping position of the seed is represented by a circular integer word  $w$  of length  $n$ .

Let us imagine a process, such as 3D printing, allowing us to easily produce customized cylindrical shields for brachytherapy. For a given tumor, we want to irradiate during a selected time through an adequate cylindrical shield, such that each tissue section receives a radiation dose as close as possible to its prescription. Moreover, let us assume that the physical precision of our process is limited: the smallest possible closed (resp. open) sector of a produced shield is still covering several sections of the surrounding area. Then, our algorithmic problem can be formalized as follows: given a circular integer word  $w$  of length  $n$ , the cylindrical shield to be designed can be seen as a constrained circular binary word of length  $n$  where, when we replace each 1 by the selected irradiation time  $t$ , the Manhattan distance to  $w$  is minimal.

This article is organized as follows. In Section 2, we introduce the notation and mathematically describe our problem. Section 3 is dedicated to computing the minimal distance for a fixed irradiation time, using a dynamic programming algorithm. In Section 4 we provide an algorithm to compute the optimal solutions of the problem. Section 5 considers a particular case of the problem where overdoses are forbidden. Finally, we give perspectives for future work in Section 6.

## 2 Preliminaries

We now introduce the usual notation on words, as well as a formal definition of the considered problem.

## 2.1 Words

We briefly introduce the basic terminology on words. More details can be found in [7]. An *alphabet* is a non-empty set  $\Sigma$ ; its elements are called *letters*. It is called *binary* if  $|\Sigma| = 2$ . A *word*  $w = w_1 \cdots w_n$  over  $\Sigma$  is a finite sequence of letters of  $\Sigma$ , where  $w_i$  is the  $i$ -th letter of  $w$ . The *empty word* is denoted by  $\varepsilon$ . We denote by  $\Sigma^*$  the set of all words over  $\Sigma$ . A *language*  $L$  over  $\Sigma$  is a subset of  $\Sigma^*$ .

The *length* of a word  $w$  is the number of its letters and is denoted by  $|w|$ . For a positive integer  $n$ , we denote by  $\Sigma^n$  the set of all words of length  $n$  over  $\Sigma$ .

Given a word  $y \in \Sigma^*$ , we say that  $y$  is a *factor* of  $w \in \Sigma^*$  if there exist two words  $x, z \in \Sigma^*$  such that  $w = xyz$ . In particular, if  $x = \varepsilon$  (resp.  $z = \varepsilon$ ), then  $y$  is called a *prefix* (resp. *suffix*) of  $w$ . For two words  $w$  and  $u$ , let  $\text{LCP}(w, u)$  (resp.  $\text{LCS}(w, u)$ ) denote the *longest common prefix* (resp. *suffix*) of  $w$  and  $u$ . We denote by  $\text{Pref}(w)$  the *set of all prefixes* of a word  $w$ . For an integer  $i$ , we write  $\text{pref}_i(w)$  (resp.  $\text{suff}_i(w)$ ) the *length- $i$  prefix* (resp. *suffix*) of  $w$ .

Given an integer  $n \geq 0$ , we denote by  $w^n$  the word  $ww \cdots w$  ( $n$  times). If  $w = xyz$  and  $y = a^n$ , for some letter  $a \in \Sigma$  and some integer  $n \geq 1$ , we say that  $y$  is an  *$a$ -run* in  $w$ . In the sequel, each  $a$ -run  $y$  considered is *maximal*, that is, the last letter of  $x$  and the first one of  $z$  are both different from  $a$ . Let  $P \subset \Sigma^*$  be a finite set of words over  $\Sigma$ . A language  $L \subseteq \Sigma^*$  is said to *avoid*  $P$  if, for any  $w \in L$  and any  $y \in P$ ,  $y$  is not a factor of  $w$ .

Two words  $w$  and  $w'$  are *conjugate*, denoted by  $w \sim w'$ , if there exist two words  $x$  and  $y$  such that  $w = xy$  and  $w' = yx$ . For  $w = w_1 \cdots w_n \in \Sigma^n$ , the  $j$ -th conjugate of  $w$ , for  $1 \leq j \leq n$ , is the word  $w_j \cdots w_n w_1 \cdots w_{j-1}$ . It is easy to verify that the relation  $\sim$  is an equivalence relation. Any equivalence class of  $\sim$  is called a *circular word*. For instance,  $\{01001, 10010, 00101, 01010, 10100\}$  is a circular word.

## 2.2 Problem definition

For a given tumor, we want to design a cylindrical shield with openings, both enabling us to apply the most accurate treatment possible to the patient and taking into account the production process limitations. Assuming that the irradiation time is  $t$ , we represent this shield by a circular word over the alphabet  $\{0, t\}$ , where each  $t$  stands for an opened section and each 0 for a closed section. Let  $\ell_1$  be the minimal length for an opening, and  $\ell_0$  the one for a closed sector between two openings. Then, in any word representing a producible shield, each 0-run (resp.  $t$ -run) must be of length at least  $\ell_0$  (resp.  $\ell_1$ ). Thus, any such word belongs to the following language of admissible words.

► **Definition 1.** For three positive integers  $\ell_0, \ell_1, t$ , let  $A_t$  be the language over the alphabet  $\{0, t\}$  that avoids  $t0^{m_0}t$  and  $0t^{m_1}0$  for all  $0 < m_0 < \ell_0$ ,  $0 < m_1 < \ell_1$ . We say that a word  $u$  is  *$t$ -admissible* if  $u \in A_t$ .

To reduce the amount of notation, we do not write  $\ell_0, \ell_1$  as indices in Definition 1 since  $\ell_0, \ell_1$  are fixed. So we write  $A_t$  instead of  $A_{\ell_0, \ell_1, t}$ . Moreover, from now on, we fix  $\ell = \max\{\ell_0, \ell_1\}$ .

► **Example 2.** Let  $\ell_0 = 3$ ,  $\ell_1 = 5$  and  $t = 2$ . We have  $02222200022 \in A_t$ , but the word  $02222200222$  is not  $t$ -admissible since it contains the factor  $20^22$  and  $\ell_0 = 3$ .

The doses applied through a shield depend on the associated irradiation time  $t$ . Therefore, we select the most accurate shield for each relevant value of  $t$ .

► **Definition 3.** Let  $\ell_0, \ell_1, t$  be three positive integers. A word  $u \in A_t$  is said to be  *$t$ -circularly admissible* if all of its conjugates are in  $A_t$ . The set of all  $t$ -circularly admissible words is denoted by  $C_t$ .



■ **Table 1** Comparison of the results of a naive algorithm ( $u_{\text{greedy}}$  and  $u_{\text{greedy\&circ}}$ ) with an optimal solution ( $u_{\text{optimal}}$ ) for  $\ell = \ell_0 = \ell_1 = 3$  and  $w = 111100111000$ . Each difference with  $w$  is underlined. The last two runs of  $u_{\text{greedy}}$  are merged into a 0-run in  $u_{\text{greedy\&circ}}$ . Only  $u_{\text{optimal}}$  is a proper solution.

$w$	1	1	1	1	0	0	1	1	1	0	0	0	$d(u, w)$	$u \in C_1 ?$	$u \in \text{Sol}(w) ?$
$u_{\text{greedy}}$	1	1	1	1	0	0	<u>0</u>	1	1	<u>1</u>	0	0	2		
$u_{\text{greedy\&circ}}$	1	1	1	1	0	0	<u>0</u>	<u>0</u>	<u>0</u>	0	0	0	3	✓	
$u_{\text{optimal}}$	1	1	1	<u>0</u>	0	0	1	1	1	0	0	0	1	✓	✓

► **Example 4.** Let  $\ell_0 = 3$ ,  $\ell_1 = 5$  and  $t = 2$ . We have  $02222200022 \in A_t$  but it is not  $t$ -circularly admissible since its conjugate  $22222000220$  is not  $t$ -admissible. We have  $02222200000 \in C_t$ .

At last, we use a distance metric to assess the accuracy of the considered treatments.

► **Definition 5.** The *Manhattan distance*  $d$  between two words  $u$  and  $v \in \mathbb{N}^n$  is  $d(u, v) = \sum_{i=1}^n |u_i - v_i|$ , where  $|u_i - v_i|$  denotes the absolute value of  $u_i - v_i$ .

Our problem can formally be stated as follows.

► **Problem 1.** Given a word  $w$  over the alphabet  $\mathbb{N}$  and two integers  $\ell_0, \ell_1$ , find  $t \in \mathbb{N}^+$  and  $u \in C_t$  that minimizes  $d(u, w)$ .

Note that a solution to Problem 1 always exists (since  $u = 0^{|w|}$  is  $t$ -circularly admissible for any  $t$ ), but is not necessarily unique. Therefore, we set the following notation. Let us denote the set of all solutions to Problem 1 for  $w$  by

$$\text{Sol}(w) = \{u \in \cup_t \text{Sol}_t(w) \mid d(u, w) \text{ is minimal}\},$$

where  $\text{Sol}_t(w) = \{u \in C_t \mid d(u, w) \text{ is minimal}\}$ . Observe that if  $\ell = \max\{\ell_0, \ell_1\} \leq 1$ , Problem 1 is trivial. Hence, in practice, we always consider  $\ell \geq 2$ .

Also, we always assume in the following that  $|w| \geq \ell_0 + \ell_1$ . Indeed, the case where  $|w| = n < \ell_0 + \ell_1$  is also trivial since any solution for such a word  $w$  is either  $0^n$  or  $t^n$ .

It is worth mentioning that the following naive, greedy strategy fails to solve Problem 1. Let us consider the simple case where  $w$  is a binary word, and hence  $t = 1$ . Let  $u_{\text{greedy}}$  be the word obtained by reading through  $w$  and beginning a new 0-run (resp. 1-run) as soon as both a 0 (resp. 1) occurs in  $w$  and the length of the previous 1-run (resp. 0-run) is at least  $\ell_1$  (resp.  $\ell_0$ ). Else, the last created run is extended. Then, there exist instances of Problem 1 such that  $u_{\text{greedy}} \notin C_1$ , hence is not a proper solution. For such an instance, let  $u_{\text{greedy\&circ}} \in C_1$  be the word obtained by merging the last two runs of  $u_{\text{greedy}}$  into one run of either 0 or 1 (the one giving the lowest distance to  $w$ ). An instance such that neither  $u_{\text{greedy}}$  nor  $u_{\text{greedy\&circ}}$  minimizes the distance with  $w$  is presented in Table 1.

In what follows, we show that, in practice, Problem 1 can be solved by a polynomial dynamic programming algorithm.

### 3 Computing the optimal distance for a fixed irradiation time

To solve Problem 1 for a given  $w$ , we first fix the irradiation time  $t$  and compute the optimal distance for this fixed  $t$ . In other words, we compute  $d(u, w)$  for some  $u \in \text{Sol}_t(w)$ .



### 3.1 Distance matrix

Let  $\ell, \ell_0, \ell_1, n, t$  be positive integers such that  $\ell = \max\{\ell_0, \ell_1\}$  and  $n \geq \ell_0 + \ell_1$ . Let  $w$  be a word of length  $n$  over the alphabet  $\mathbb{N}$ . In this section, we describe an algorithm that computes the set  $\text{Sol}_t(w)$  for a fixed irradiation time  $t$ .

To compute the set  $\text{Sol}_t(w)$ , we build a dynamic matrix using the possible prefixes of  $u \in \text{Sol}_t(w)$ , that is, the set

$$\mathcal{P}_t = \{0^i t \mid 1 \leq i \leq \ell_0\} \cup \{t^i 0 \mid 1 \leq i \leq \ell_1\} \cup \{0^{\ell_0+1}, t^{\ell_1+1}\}.$$

In particular, for any word  $u \in A_t$ ,  $|\mathcal{P}_t \cap \text{Pref}(u)| = 1$ , i.e., there exists a unique word  $p \in \mathcal{P}_t$  such that  $p$  is a prefix of  $u$ . Note that  $|\mathcal{P}_t| = \ell_0 + \ell_1 + 2 \leq 2\ell + 2$ .

► **Example 6.** For  $\ell_0 = 3$  and  $\ell_1 = 5$ ,  $\mathcal{P}_t = \{0t, 00t, 000t, 0000, t0, tt0, ttt0, tttt0, ttttt0, tttttt\}$ .

The following definition generalizes the notion of  $t$ -circularly admissible words, to describe the (shorter) words yet to be extended into ones.

► **Definition 7.** Let  $n \in \mathbb{N}$ ,  $i \in \{1, \dots, n\}$ ,  $p \in \mathcal{P}_t$  and  $v \in A_t \cap \{0, t\}^\ell$ . Then a word  $u$  is called  $t$ -circularly preadmissible (with respect to  $p$  and  $v$ ) if it satisfies the following properties:

- (i) (admissibility)  $u \in A_t$  and  $|u| = i$ ;
- (ii) (circular extendability) There exists a word  $x$  such that  $ux \in C_t$ ;
- (iii) (prefix compatibility)  $\text{LCP}(u, p) \in \{u, p\}$ ;
- (iv) (suffix compatibility)  $\text{LCS}(u, v) \in \{u, v\}$ .

The set of  $t$ -circularly preadmissible words of length  $i$  with respect to  $p$  and  $v$  is denoted by  $C_{t,p}(v, i)$ .

Roughly speaking,  $u \in C_{t,p}(v, i)$  if  $u$  is  $t$ -admissible, starts with  $p$ , ends with  $v$  (whenever  $u$  is long enough) and can be extended into at least one  $t$ -circularly admissible word. In particular, we need to consider the longest common prefix (resp. suffix) of  $u$  and  $p$  (resp.  $u$  and  $v$ ) to take care of the cases where  $u$  is shorter than  $p$  (resp. shorter than  $v$ ). Note that each word of  $\mathcal{P}_t$  is circularly preadmissible since  $n \geq \ell_0 + \ell_1$ .

We now define a matrix whose entries indicate the minimum distance between increasingly longer prefixes  $u'$  of both  $w$  and  $u$ , so that  $u'$  is preadmissible with respect to its unique prefix  $p \in \mathcal{P}_t$ . The columns of the matrix correspond to the length of these prefixes while the rows correspond to the possible suffixes of these prefixes.

► **Definition 8 (Distance matrix).** For  $p \in \mathcal{P}_t$ , let  $D_{w,t,p}$  be the matrix of size  $|A_t \cap \{0, t\}^\ell| \times n$  such that

$$D_{w,t,p}[v, i] = \min\{d(u, w_1 \cdots w_i) \mid u \in C_{t,p}(v, i)\}.$$

for each  $v \in A_t \cap \{0, t\}^\ell$  and each  $i \in \{1, \dots, n\}$ , with the convention that  $\min \emptyset = \infty$ .

► **Example 9.** Consider for instance, the word  $w = 013331102230313210$  with  $\ell_0 = 3$  and  $\ell_1 = 5$ . Let  $t = 2$  and  $p = 02$ . Figure 3 depicts the matrix  $D_{w,t,p}$ .

Note that the size of the matrix  $D_{w,t,p}$ , which is equal to  $|A_t \cap \{0, t\}^\ell| \times n$ , is polynomial with respect to  $\ell$  and  $n$ . This comes from the following lemma.

► **Lemma 10.** The language  $A_t \cap \{0, t\}^\ell$  contains  $2\ell + \binom{\ell - \min\{\ell_0, \ell_1\}}{2}$  words.

**Proof.** Let  $v \in A_t \cap \{0, t\}^\ell$ . By Definition 1,  $v$  does not contain  $t0^{m_0}t$  and  $0t^{m_1}0$  for all  $0 < m_0 < \ell_0, 0 < m_1 < \ell_1$ . Recall that  $\ell = \max\{\ell_0, \ell_1\}$ . If  $\ell_0 = \ell_1 = \ell$ , the fact that  $|v| = \ell$  implies that  $v$  either is a 1-run followed by a 0-run or vice-versa. Since there are  $\ell$  positions where this first run can end, this gives us  $\ell$  words beginning by 1 and  $\ell$  words beginning by 0. Now, if  $\ell = \ell_0 > \ell_1$ ,  $v$  can also begin and end with a 0-run, and have a 1-run of length at least  $\ell_1$  in the middle. In this case, we need to decide where the first 0-run ends and where the last 0-run begins. Hence we have  $\ell - \ell_1$  possible positions to choose from. The case  $\ell = \ell_1$  is symmetric and so we have that the language  $A_t \cap \{0, t\}^\ell$  contains  $2\ell + \binom{\ell - \min\{\ell_0, \ell_1\}}{2}$  words. ◀

Moreover, it follows from Definition 8 that the matrices  $D_{w,t,p}$  keep track of the optimal values. More precisely, let

$$D_{w,t} = \min\{D_{w,t,p}[v, n] \mid p \in \mathcal{P}_t, v \in A_t \cap \{0, t\}^\ell\}.$$

Then, the next theorem states that any word  $u$  such that  $d(u, w) = \min_t D_{w,t}$  is a solution to Problem 1. Indeed, Definition 7 implies that all prefixes of a circularly admissible word  $u$  are preadmissible.

► **Theorem 11.** *Let  $u \in \{0, t\}^n$  and  $i \in \{1, \dots, n\}$ . If  $u \in C_t$ , then there exist  $p \in \mathcal{P}_t$  and  $v \in A_t \cap \{0, t\}^\ell$  such that  $\text{pref}_i(u) \in C_{t,p}(v, i)$ . Consequently, for any  $u \in \text{Sol}_t(w)$ ,*

$$d(u, w) = \min\{D_{w,t,p}[v, n] \mid v \in A_t \cap \{0, t\}^\ell, p \in \mathcal{P}_t\}.$$

**Proof.** Assume first that  $u \in C_t$ . Let  $p$  be the only word in  $\mathcal{P}_t \cap \text{Pref}(u)$  and  $v \in A_t \cap \{0, t\}^\ell$  such that  $\text{LCS}(u, v) \in \{\text{pref}_i(u), v\}$ . For  $i \geq \ell$ , it is obvious that such a  $v$  always exists since  $u \in C_t$  implies  $\text{pref}_i(u) \in A_t$ . For  $i < \ell$ , let  $a$  be the first letter of  $u$  and  $v = a^{\ell-i} \text{pref}_i(u)$ . Then, all the conditions of Definition 7 are satisfied and  $\text{pref}_i(u) \in C_{t,p}(v, i)$ .

For the last part of the statement, we only need to show that  $u \in C_{t,p}(v, n)$  for some  $v \in A_t \cap \{0, t\}^\ell$  and  $p \in \mathcal{P}_t$  implies  $u \in C_t$ . Assume that  $u \in C_{t,p}(v, n)$  for adequate  $p$  and  $v$ . Thus,  $u$  is admissible and  $|u| = n$ , so that  $u \cdot \varepsilon = u \in C_t$ , by the circular extendability property. The result then follows from the definition of  $\text{Sol}_t(w)$ . ◀

In the next subsection, we show that, for a fixed  $t$ , for all  $p \in \mathcal{P}_t$ ,  $v \in A_t$  and  $i \in \{|p| + 1, \dots, n\}$ , the value  $D_{w,t,p}[v, i]$  can be computed in constant time from at most two values of the form  $D_{w,t,p}[v', i - 1]$ , where  $v' \in A_t$ .

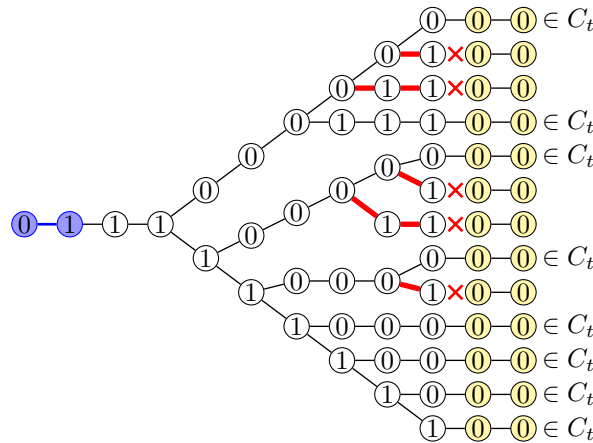
### 3.2 Dynamic programming

We describe now the dynamic computation of such matrices  $D_{w,t,p}$ . We begin by taking care of the initialization.

► **Lemma 12.** *Let  $p \in \mathcal{P}_t$  and  $D = D_{w,t,p}$ . For  $v \in A_t \cap \{0, t\}^\ell$  and  $1 \leq i \leq |p|$ , we have*

$$D[v, i] = \begin{cases} d(\text{pref}_i(w), \text{pref}_i(p)), & \text{if } \text{LCS}(v, \text{pref}_i(p)) \in \{v, \text{pref}_i(p)\}; \\ \infty, & \text{otherwise.} \end{cases}$$

**Proof.** We first prove that  $u \in C_{t,p}(v, i)$  implies  $u = \text{pref}_i(p)$ . Indeed, assume that there exists some  $u \in C_{t,p}(v, i)$ . Then, by Definition 7, we have in particular that  $u$  is admissible,  $|u| = i$  and  $\text{LCP}(u, p) \in \{u, p\}$ . Since  $i \leq |p|$ , we deduce that  $\text{LCP}(u, p) = u$ , i.e.  $u = \text{pref}_i(p)$ . Therefore, either  $C_{t,p}(v, i) = \{\text{pref}_i(p)\}$  or  $C_{t,p}(v, i) = \emptyset$ .



**Figure 1** Circularly admissible words of length 12 with  $\ell_0 = \ell_1 = 3$ ,  $t = 1$  that have  $p = 01$  as prefix. They are constructed by successive extensions of admissible words complying with the prefix. In blue is highlighted the prefix  $p$ ; in yellow is the suffix of any such circularly admissible words; and in red are the discarded extensions.

It remains to check under which conditions  $\text{pref}_i(p) \in C_{t,p}(v, i)$ . Clearly,  $\text{pref}_i(p)$  is admissible, since  $p$  is, and  $|\text{pref}_i(p)| = i$ . Also, note that  $\text{pref}_i(p)$  is circularly extendable, since  $p \in \mathcal{P}_t$  is circularly extendable whenever  $n \geq \ell_0 + \ell_1$ . The prefix compatibility is trivially verified since  $\text{LCP}(\text{pref}_i(p), p) = \text{pref}_i(p)$ . Therefore, if  $\text{LCS}(v, \text{pref}_i(p)) \in \{v, \text{pref}_i(p)\}$ , then  $C_{t,p}(v, i) = \{\text{pref}_i(p)\}$  and the result follows. Otherwise,  $C_{t,p}(v, i) = \emptyset$ . ◀

To compute the other part of the table, we need to focus on words that are admissible for  $w$  and that comply with the given prefix  $p \in \mathcal{P}_t$ . Near the end of the table, we need to add extra conditions to take into account the fact that suffixes of the circularly admissible words are prescribed by  $p$ .

**Example 13.** Let us consider the word  $w = 001100001111$ ,  $t = 1$ ,  $\ell_0 = \ell_1 = 3$  and the prefix  $p = 01 \in \mathcal{P}_t$ . Figure 1 depicts all words considered in the computation of  $D_{w,t,p}$ , that is, all admissible words that can be extended to a circularly admissible word complying with the prefix  $p$ . This prefix condition is highlighted in blue. Starting from the left, we write the possible extensions of an admissible word of length  $i$  to one of length  $i + 1$ . Observe that this rule of extension sometimes leads to dead ends. Indeed, to be circularly admissible, any admissible word of length  $|w|$  has to end with two 0's since  $\ell_0 = 3$  and the prescribed prefix  $p = 01$  begins with only one 0 (this condition is highlighted in yellow). Therefore, the extensions highlighted in red are not prefixes of any circularly admissible words, so that such words need to be discarded.

In Theorem 14, the yellow condition corresponds to the case  $n - c_p < i \leq n$  and the red dead ends are taken care of in the case  $n - c_p - \ell_a < i \leq n - c_p$ . In the remaining case, Equation (1) translates the fact that any admissible word is the extension of a shorter admissible word.

**Theorem 14.** Let  $p = p_1 \dots p_k \in \mathcal{P}_t$  and  $D = D_{w,t,p}$ . If  $p_1 = 0$ , we set  $c_p = \ell_0 - |p| + 1$ ,  $\bar{p}_1 = t$  and  $a = 1$ . Otherwise, we set  $c_p = \ell_1 - |p| + 1$ ,  $\bar{p}_1 = 0$  and  $a = 0$ . For  $v = v_1 \dots v_\ell \in A_t$  and  $|p| + 1 \leq i \leq n$ , we have

$$D[v, i] = d(w_i, v_\ell) + \min\{D[v', i - 1] \mid v' \in V'\}$$

where

$$V' = \begin{cases} \emptyset, & \text{if } n - c_p < i \leq n \text{ and } \text{suff}_{i-n+c_p}(v) \notin p_1^* \\ \text{or if } n - c_p - \ell_a < i \leq n - c_p, \text{ suff}_{i-(n-c_p-\ell_a)}(v) \notin (\overline{p_1})^* \text{ and } \text{suff}_1(v) \neq p_1; \end{cases}$$

and otherwise

$$V' = \begin{cases} \{v' \in \{0v_1 \dots v_{\ell-1}, tv_1 \dots v_{\ell-1}\} \mid v'v_\ell \in A_t \cap \{0, t\}^{\ell+1}\}, & \text{if } i > \ell; \\ \{(v_{\ell-i+1})^{\ell-i+2} v_{\ell-i+2} \dots v_{\ell-1}\}, & \text{if } i \leq \ell. \end{cases} \quad (1)$$

**Proof.** By Definition 8,  $D[v, i]$  is the minimum distance  $d(u, w_1 \dots w_i)$  over words  $u \in C_{t,p}(v, i)$ . We begin by focusing on the circular extendability of such words (Condition (ii) in Definition 7).

For any circularly admissible word  $x \in C_t$ , if  $x$  has prefix  $p$  with first letter  $p_1$ , then  $x$  must end with a suffix  $p_1^{c_p}$  since  $p$  begins with  $|p| - 1$  times the letter  $p_1$  and the constant  $c_p = \ell_p - (|p| - 1)$ . Therefore, for  $n - c_p < i \leq n$ , any  $u \in C_{t,p}(v, i)$  is the prefix of a circularly admissible word and it must end with  $p_1^{i-n+c_p}$ . Thus, if  $\text{suff}_{i-n+c_p}(v) \notin p_1^*$ , then  $C_{t,p}(v, i) = \emptyset$  and  $D[v, i] = \infty$ . In this case, we set  $V' = \emptyset$ .

Consider now the case where  $n - c_p - \ell_a < i \leq n - c_p$ . Any circularly admissible word  $x \in C_t$  which has prefix  $p$ , has suffix  $p_1^{c_p}$ . So the suffix  $s$  of length  $c_p + \ell_a$  of  $x$  ends with  $c_p$  times the letter  $p_1$ . Since  $s$  has to be admissible, the factor  $p_1 \overline{p_1}$  can not occur in  $s$ . In particular, it can not occur in position in  $\{n - c_p - \ell_a + 1, \dots, n - c_p\}$  in  $x$ . Thus,  $u \in C_{t,p}(v, i)$  implies that  $u$  has a suffix of the form  $(\overline{p_1})^{i-(n-c_p-\ell_a)}$  or  $u$  ends with  $p_1$ . Hence, if  $\text{suff}_{i-(n-c_p-\ell_a)}(v) \notin (\overline{p_1})^*$  and  $\text{suff}_1(v) \neq p_1$ , then  $C_{t,p}(v, i) = \emptyset$  and  $D[v, i] = \infty$ . So we set  $V' = \emptyset$  in this case too.

Note that when  $|p| < i \leq n - c_p - \ell_a$ , then any word  $u$  satisfying Conditions (i,iii,iv) of Definition 7 always satisfies Condition (ii). Indeed, if  $u$  ends with  $p_1$ , then we can concatenate  $u$  and  $p_1^{n-i}$  to obtain a circularly admissible word. If  $u$  ends with  $\overline{p_1}$ , we can concatenate  $u$  with  $(\overline{p_1})^{\ell_a} p_1^{n-i-\ell_a}$  to get a circularly admissible word since  $c_p \leq n - i - \ell_a$ .

We now turn our attention to the other cases where Condition (ii) is always satisfied. We assume now that  $i$  and  $v$  are such that one of the following holds:

- $|p| < i \leq n - c_p - \ell_a$ ;
- $n - c_p - \ell_a < i \leq n - c_p$  and  $(\text{suff}_{i-(n-c_p-\ell_a)}(v) \in (\overline{p_1})^* \text{ or } \text{suff}_1(v) = p_1)$ ;
- $n - c_p < i \leq n$  and  $\text{suff}_{i-n+c_p}(v) \in p_1^*$ .

We discuss two cases according to whether  $i \leq \ell$  or  $i > \ell$ .

Firstly, if  $i \leq \ell$ , then  $u \in C_{t,p}(v, i)$  implies that  $u$  is admissible such that  $\text{LCS}(u, v) = u$  and  $\text{LCP}(u, p) = p$  (as  $i \geq |p| + 1$ ). In particular,  $u$  has a prefix  $\text{pref}_{i-1}(u)$  that belongs to  $C_{t,p}(v', i - 1)$  for some  $v' \in A_t \cap \{0, t\}^\ell$ . We claim that we can always choose  $v'$  such that  $v'v_\ell \in A_t$ . Indeed,  $v'$  has suffix  $\text{pref}_{i-1}(u)$  and  $\text{pref}_{i-1}(u)$  begins with at least  $|p| - 1$  times the letter  $p_1$ . Therefore, we can choose  $v'$  to be the length- $\ell$  word of  $p_1^* \text{pref}_{i-1}(u)$ . Since  $u \in A_t$ , we have  $\text{pref}_{i-1}(u)v_\ell \in A_t$  and  $v'v_\ell \in A_t$ . As  $p_1 = v_{\ell-i+1}$  and  $\text{pref}_{i-1}(u) = v_{\ell-i+1} \dots v_{\ell-1}$ , the result follows. Similarly if  $C_{t,p}(v, i) = \emptyset$ , then  $C_{t,p}(v', i) = \emptyset$  for the length- $\ell$  word  $v' \in v_{\ell-i+1}^* v_{\ell-i+1} \dots v_{\ell-1}$ . Hence, the equation  $\infty = D[v, i] = d(v_\ell, w_i) + D[v', i - 1] = \infty$  holds also in this case.

Thus, we set  $V' = \{v' \in \{0, t\}^{\ell-i+1} \text{pref}_{i-1}(u) \mid v'v_\ell \in A_t \cap \{0, t\}^{\ell+1}\}$  and we have

$$\begin{aligned} D[v, i] &= \min\{d(u, w_1 \dots w_i) \mid u \in C_{t,p}(v, i)\} \\ &= d(v_\ell, w_i) + \min\{d(\text{pref}_{i-1}(u), w_1 \dots w_{i-1}) \mid u \in C_{t,p}(v, i)\} \\ &= d(v_\ell, w_i) + \min\{d(u', w_1 \dots w_{i-1}) \mid u' \in C_{t,p}(v', i - 1) \text{ s.t. } v' \in V'\} \end{aligned}$$

and the result follows.

**Algorithm 1** Finding an optimal solution

---

```

1: function FINDSOLUTION( $w$  : integer word,  $\ell_0, \ell_1$ : integers): integer word
2:   Let  $t_{\max}$  be the largest letter occurring in  $w$ 
3:    $\ell \leftarrow \max\{\ell_0, \ell_1\}$ ;  $m \leftarrow \min\{\ell_0, \ell_1\}$ 
4:   Let  $D_{\min}$  be a table of size  $2\ell + \binom{\ell-m}{2} \times |w|$ 
5:    $d_{\min} \leftarrow +\infty$ 
6:   for  $t \leftarrow 1, \dots, t_{\max}$  do  $\triangleright t_{\max}$  iterations
7:     for  $p \in \mathcal{P}_t$  do  $\triangleright \mathcal{O}(\ell)$  iterations
8:        $D \leftarrow \text{COMPUTEMATRIX}(w, t, p, \ell_0, \ell_1)$   $\triangleright \mathcal{O}(|w| \cdot \ell^2)$ 
9:       for  $v \in A_t \cap \{0, t\}^\ell$  do  $\triangleright \mathcal{O}(\ell^2)$  iterations
10:         $dist \leftarrow D[v, |w|]$ 
11:        if  $dist < d_{\min}$  then
12:           $(optimal, time, prefix, d_{\min}, D_{\min}) \leftarrow (v, t, p, dist, D)$ 
13:        end if
14:      end for
15:    end for
16:  end for
17:   $u \leftarrow \text{BACKTRACKING}(D_{\min}, v, time, prefix)$   $\triangleright \mathcal{O}(|w|)$ 
18:  return  $u$ 
19: end function

```

---

Secondly, if  $i > \ell$ , then  $u \in C_{t,p}(v, i)$  implies that  $u \in A_t \cap \{0, t\}^i$ ,  $\text{LCP}(u, p) = p$  and  $\text{LCS}(u, v) = v$ . In particular  $u$  ends either with  $0v_1 \dots v_\ell$  or with  $tv_1 \dots v_\ell$  where  $v = v_1 \dots v_\ell$ . As  $u \in A_t$ , its suffix of length  $\ell + 1$  belongs to  $A_t$ . Therefore we have  $D[v, i] = d(w_i, v_\ell) + \min\{D[v', i-1]\}$  where  $v' \in \{0v_1 \dots v_{\ell-1}, tv_1 \dots v_{\ell-1}\}$  such that  $v'v_\ell \in A_t \cap \{0, t\}^{\ell+1}$  as required.  $\blacktriangleleft$

#### 4

 Computing an optimal solution

Now, given any circular word with representative  $w$ , we are ready to solve Problem 1, *i.e.* to give an algorithm to find a word  $u \in \text{Sol}(w)$ . For each irradiation time  $t$ , we compute the optimal distance  $d(u, w)$  for  $u \in \text{Sol}_t(w)$ . We consider the minimum value obtained while  $t$  is varying and we store the distance matrix associated to this minimum value (Algorithm 1).

As for the complexity of the algorithm, note that line 8 is computed using Lemma 12 and Theorem 14 in  $\mathcal{O}(|w| \cdot \ell^2)$  since  $|A_t \cap \{0, t\}^\ell| = 2\ell + \binom{\ell - \min\{\ell_0, \ell_1\}}{2}$ . This gives us a pseudo polynomial time algorithm of complexity  $\mathcal{O}(|w| \cdot t_{\max} \cdot \ell^3)$  to compute an optimal distance. An important note is that since  $t_{\max}$  represents, in our context, the maximal time of an irradiation, it should be kept as small as possible and in fact be a lot smaller than  $|w|$ . So, in practice, we have a polynomial time algorithm. Finally, we obtain an optimal solution by backtracking in  $\mathcal{O}(|w|)$  with Algorithm 2.

A simple experimentation shows (see Figure 2) that, as expected, our algorithm becomes quickly faster than a naive approach, which enumerates all the words in  $A_t$  of length  $|w|$ , checks whether they are circularly admissible and computes their distance from  $w$ , and which is exponential.

**► Example 15** (Example 9 continued). Let  $w = 013331102230313210$ ,  $\ell_0 = 3$  and  $\ell_1 = 5$ . For  $t \in \{1, 2, 3\}$ , let  $\mathcal{P}_t = \{0t, 00t, 000t, 0000, t0, tt0, ttt0, tttt0, ttttt\}$ . Among the matrices  $D_{w,t,p}$ , the one with  $t = 2$  and  $p = 02$  contains the minimum value in its last column and is

**Algorithm 2** From distance to solution

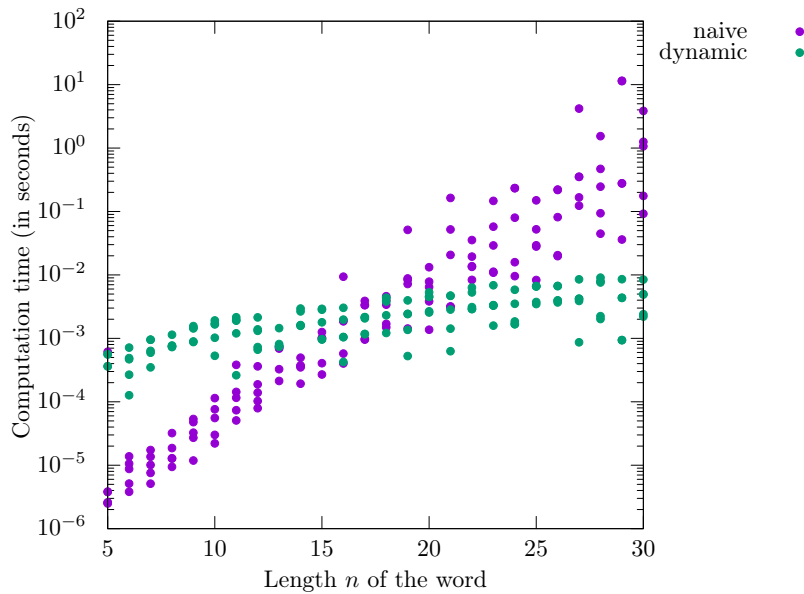
---

```

1: function BACKTRACKING( $D : \text{table}$ ,  $v = v_1 \dots v_\ell : \text{integer word}$ ,  $t : \text{integer}$ ,  $p : \text{integer}$ 
   word): integer word
2:    $n \leftarrow$  number of columns of  $D$ 
3:    $u \leftarrow \lceil v_\ell \rceil$ 
4:   for  $i \leftarrow n - 1, \dots, \ell + 1$  do
5:      $x \leftarrow v_1 \dots v_\ell$ 
6:     if  $D[0x_1 \dots x_{\ell-1}, i] \leq D[tx_1 \dots x_{\ell-1}, i]$  then
7:        $x \leftarrow 0x_1 \dots x_{\ell-1}$ ;  $u \leftarrow \lceil x_{\ell-1} \rceil + u$ 
8:     else
9:        $x \leftarrow tx_1 \dots x_{\ell-1}$ ;  $u \leftarrow \lceil x_{\ell-1} \rceil + u$ 
10:    end if
11:  end for
12:  for  $i \leftarrow \ell, \dots, |p| + 1$  do
13:     $x \leftarrow \underbrace{x_{\ell-i+1} \dots x_{\ell-i+1}}_{\ell-i+2 \text{ times}} x_{\ell-i+2} \dots x_{\ell-1}$ 
14:     $u \leftarrow \lceil x_{\ell-1} \rceil + u$ 
15:  end for
16:   $u \leftarrow p + u$ .
17:  return  $u$ 
18: end function

```

---



■ **Figure 2** Comparison of the efficiency between Algorithm 1 and a naive algorithm, based on the exhaustive enumeration of all circularly admissible factors. The computation times were obtained by generating 5 random conformations for each  $n \in \{5, 6, \dots, 30\}$ , choosing the parameters  $(w, t, \ell_0, \ell_1)$  in each case with uniform probability such that  $0 \leq w_i \leq 20$  for  $i = 1, 2, \dots, n$ ,  $1 \leq t \leq 20$ ,  $2 \leq \ell_0 \leq 5$  and  $2 \leq \ell_1 \leq 5$ . Although the dynamic programming strategy is initially more costly, it rapidly becomes faster than the naive strategy, as expected.

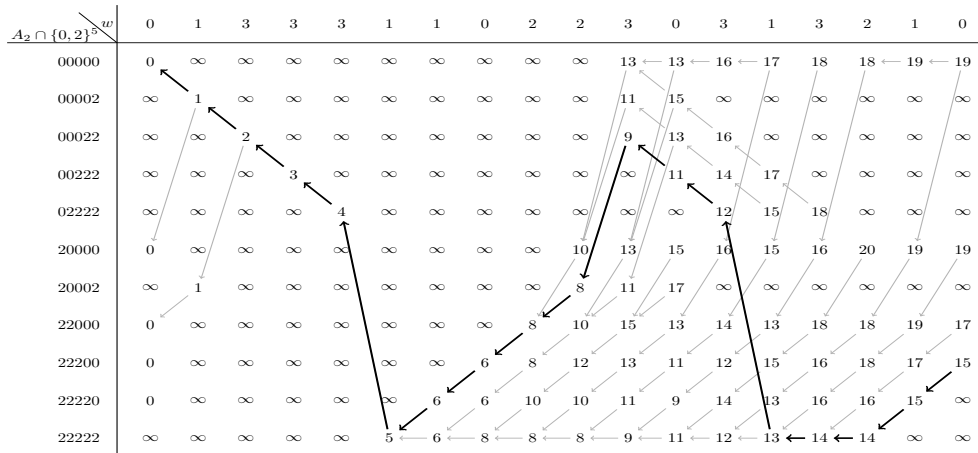


Figure 3 Matrix for the word  $w = 013331102230313210$  with  $\ell_0 = 3$ ,  $\ell_1 = 5$ ,  $t = 2$  and  $p = 02$ . An arrow between two cells indicates that the value in the arrival cell is calculated from the one in the origin cell. For ease of reading, the corresponding arrows of cells containing  $+\infty$  are not drawn. In bold is a path corresponding to an optimal solution (not necessarily unique) for the irradiation time 2.

represented in Figure 3. The path corresponding to the optimal solution, 022222000222222200 is drawn in bold. Note that it is not the only optimal solution. For instance, we also have  $0222222222222222200 \in \text{Sol}(w)$ .

### 5 Forbidding overdoses

Let us now consider a subproblem by imposing an additional condition: no tissue section can receive a dose greater than the one prescribed. That is, *overdoses* are forbidden. We then need the following stronger condition on our previous admissible words. By abuse of notation, and for the sake of simplicity, we use, in what follows, the same notation as in Sections 2 and 3.

Let  $\ell, \ell_0, \ell_1, n$  be positive integers such that  $\ell = \max\{\ell_0, \ell_1\}$  and  $n \geq \ell_0 + \ell_1$ . Let  $w$  be a word of length  $n$  over the alphabet  $\mathbb{N}$ . In this section, we show that the algorithm presented in Section 4 can also be used here, but with more efficiency, when we replace Definitions 1 and 3 by the following one.

► **Definition 16.** Let us write  $w = w_1 \cdots w_n$  with  $w_i \in \mathbb{N}$ . Let  $t$  be a letter in  $\mathbb{N}$ . A word  $u \in \mathbb{N}^n$  is *t-admissible for w* if  $u \in A_t$  and  $u_i \leq w_i$  for  $i = 1, \dots, n$ . The set of all *t-admissible words for w* is denoted by  $A_t(w)$ . In addition, if  $u \in C_t$ , then  $u$  is said to be *t-circularly admissible for w*. The set of all *t-circularly admissible words for w* is denoted by  $C_t(w)$ .

This stronger requirement affects the values of the distance matrices. We have  $D_{w,t,p}[v, i] = \infty$  if the last letter of  $v$  is greater than the  $i$ -th letter  $w_i$  of  $w$ . Otherwise,  $D_{w,t,p}[v, i]$  is computed according to Lemma 12 and Theorem 14.

By abuse of notation, we denote the set of all solutions to Problem 1 for  $w$ , under these new conditions, by  $\text{Sol}(w) = \{u \in \cup_t \text{Sol}_t(w) \mid d(u, w) \text{ is minimal}\}$ , where  $\text{Sol}_t(w) = \{u \in C_t(w) \mid d(u, w) \text{ is minimal}\}$  as before.

The “overdose-free” condition gives us the opportunity to do an efficient preprocessing of  $w$  and have a smaller set of valid prefixes  $\mathcal{P}_t$  which produces a significant gain in time

complexity. Firstly, the “overdose-free” and  $\ell_1$  constraints imply that we cannot irradiate specific sections of  $w$ . So we apply the following preprocessing on  $w$  that does not affect the solutions of Problem 1.

► **Preprocessing.** The first step is to set to 0 in  $w$  all letters that are smaller than  $t$ . Then all non-zero factors<sup>4</sup> that are too short, *i.e.* of length less than  $\ell_1$ , are replaced by 0-runs.

Secondly, since it is best to work with words beginning with a 0-run and the preprocessed  $w$  clearly does not always have that property, we need the following trivial lemma showing that optimal solutions to Problem 1 are preserved by conjugacy.

► **Lemma 17.** *Let  $t$  be a positive integer and  $u \in \text{Sol}_t(w)$ . Let  $v$  be the  $j$ -th conjugate of  $u$  and  $w'$  be the  $j$ -th conjugate of  $w$ , for an integer  $j$  such that  $1 \leq j \leq n$ . Then  $v \in \text{Sol}_t(w')$ .*

**Proof of Lemma 17.** We have  $u \in \text{Sol}_t(w)$ . Then,  $d(u, w) = d(u_1, w_1) + \dots + d(u_n, w_n)$  is minimal. Since  $v$  is the  $j$ -th conjugate of  $u$  and  $w'$  be the  $j$ -th conjugate of  $w$  we have that  $v = u_j \dots u_n u_1 \dots u_{j-1}$  and  $w' = w_j \dots w_n w_1 \dots w_{j-1}$ , which gives us  $d(v, w') = d(u_j, w_j) + \dots + d(u_n, w_n) + d(u_1, w_1) + \dots + d(u_{j-1}, w_{j-1}) = d(u, w)$ , which is minimal. So  $v \in \text{Sol}_t(w')$ . ◀

As a consequence, we can choose to work with a particular conjugate of the preprocessed input. If it contains at least one 0, let  $j \in \mathbb{N}$  be such that the  $j$ -th conjugate  $w'$  of  $w$  begins with a 0-run of length  $x$ , where  $x$  is maximal. The “overdose-free” condition implies that the possible prefixes of a solution  $u' \in \text{Sol}_t(w')$  begin with a 0-run of length at least  $x$ . So we can now compute an optimal solution  $u'$  for this  $w'$  using Algorithm 1 with the smaller set of prefixes  $\mathcal{P}_t = \{0^i t \mid x \leq i \leq \ell_0\} \cup \{0^{\ell_0+1}\}$  of length  $|\mathcal{P}_t| = \ell_0 - x + 2$ . Then, our optimal solution  $u \in \text{Sol}_t(w)$  is obtained by taking the  $(n - j + 2)$ -th conjugate of  $u'$ . Finally, if the preprocessed input  $w$  does not contain any 0, then  $\text{Sol}_t(w) = \{t^n\}$ .

Though the complexity in the worst case is not improved, in many cases the number of distance matrices to be computed is significantly lowered. For instance, the following example shows that instead of 30 distance matrices needed to find the optimal solution in the original settings, only one distance matrix is needed for the “overdose-free” problem.

► **Example 18** (Example 9 continued). For the word  $w = 013331102230313210$  with  $\ell_0 = 3$  and  $\ell_1 = 5$ , the preprocessing for the irradiation time  $t = 1$  gives the word  $013331100000313210$ . We apply our dynamic programming algorithm to the 8-th conjugate of  $w$  that is  $w' = 000003132100133311$ . Then  $\mathcal{P}_t$  contains only one word and the associated matrix  $D_{w',1,0000}$  is given in Figure 4. The word  $u' = 000001111100011111$  belongs to  $\text{Sol}_1(w')$ . Since the minimum value in the last column of  $D_{w',1,0000}$  is 12 and  $d(w', w) = 7$ , any  $u \in \text{Sol}_1(w)$  is at distance  $12 + 7$  from  $w$ .

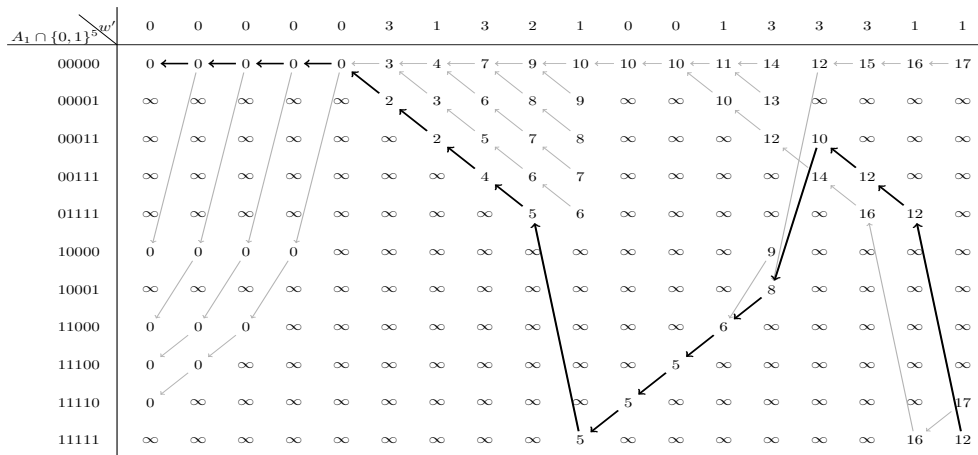
For  $t = 2$ , the first step of the preprocessing of  $w$  leads to the word  $00333000000303200$  and the second step gives  $0^{18}$ . Hence, we have  $\text{Sol}_2(w) = \{0^{18}\}$ . It follows that  $\text{Sol}_3(w) = \{0^{18}\}$  too. Hence,  $\text{Sol}(w) = \text{Sol}_1(w)$  and it contains the 12-th conjugate of  $u'$ .

## 6 Perspectives

Interesting extensions of the original problem could be considered. First, say that we have several stopping positions of our irradiation seed and that each of these stopping positions

<sup>4</sup> Here, factors are considered circularly. For instance, for  $\ell_1 = 3$  and  $t = 1$ , the word  $w = 100340022$  has only one too short non-zero factor, the one occurring in position 4.





■ **Figure 4** Matrix for the word  $w' = 000003132100133311$  with  $\ell_0 = 3$ ,  $\ell_1 = 5$ ,  $t = 1$  and  $p = 0000$ , in the case where overdoses are forbidden. An arrow between two cells indicates that the value in the arrival cell is calculated from the one in the origin cell. For ease of reading, the corresponding arrows of cells containing  $+\infty$  are not drawn. In bold is a path corresponding to an optimal solution (not necessarily unique) for the irradiation time 1.

is represented by a different word, depicting the conformation of the tumor in that exact position. If our cylindrical shield allows only one conformation of open and close sectors, the problem considered here is to find the shield conformation that is as close as possible to all the given tumor conformation words. A good heuristic here could be to find a consensus of these given words and then to use our dynamic programming algorithm to find the best shield conformation for this word. Some works has already been done on this idea of consensus of circular words. Indeed, in [5], Lee *et al.* give an  $\mathcal{O}(n^2 \log n)$  algorithm to compute, given a set of three strings, a consensus for this set under the Hamming distance, i.e. a string minimizing the sum of distances to all strings in the given set.

Another extension would be to be able to use several different irradiation times on one tumor conformation (integer word  $w$ ). Note that to be able to apply various irradiation doses, we would either need a modified device able to close each open sector after a certain irradiation time, or need to use several shield configurations (optimally, a minimal number of them) for each stopping position among the catheter.


References

- 1 Horst Bunke and Urs Bühler. Applications of approximate string matching to 2d shape recognition. *Pattern Recognition*, 26(12):1797–1812, 1993. doi:10.1016/0031-3203(93)90177-X.
- 2 Maxime Crochemore, Gabriele Fici, Robert Mercas, and Solon P. Pissis. Linear-time sequence comparison using minimal absent words & applications. In Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez, editors, *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*, volume 9644 of *Lecture Notes in Computer Science*, pages 334–346. Springer, 2016. doi:10.1007/978-3-662-49529-2\_25.
- 3 Jens Gregor and Michael G. Thomason. Dynamic programming alignment of sequences representing cyclic patterns. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(2):129–135, 1993. doi:10.1109/34.192484.

- 4 Roberto Grossi, Costas S. Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. Circular sequence comparison: algorithms and applications. *Algorithms for Molecular Biology*, 11:12, 2016. doi:10.1186/s13015-016-0076-6.
- 5 Taehyung Lee, Joong Chae Na, Heejin Park, Kunsoo Park, and Jeong Seop Sim. Finding consensus and optimal alignment of circular strings. *Theor. Comput. Sci.*, 468:92–101, 2013. doi:10.1016/j.tcs.2012.11.018.
- 6 V. I. Levenshtein. Binary codes capable of correcting insertions and reversals. *Sov. Phys. Dokl.*, 10:707–710, 1966.
- 7 M. Lothaire. *Combinatorics on words*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1997. doi:10.1017/CB09780511566097.
- 8 Maurice Maes. On a cyclic string-to-string correction problem. *Inf. Process. Lett.*, 35(2):73–78, 1990. doi:10.1016/0020-0190(90)90109-B.
- 9 Andrés Marzal and Sergio Barrachina. Speeding up the computation of the edit distance for cyclic strings. In *15th International Conference on Pattern Recognition, ICPR'00, Barcelona, Spain, September 3-8, 2000.*, pages 2891–2894. IEEE Computer Society, 2000. doi:10.1109/ICPR.2000.906217.
- 10 Ramón Alberto Mollineda, Enrique Vidal, and Francisco Casacuberta. Cyclic sequence alignments: Approximate versus optimal techniques. *IJPRAI*, 16(3):291–299, 2002. doi:10.1142/S0218001402001678.
- 11 R. Pötter, C. Haie-Meder, E. Van Limbergen, I. Barillot, M. De Brabandere, J. Dimopoulos, I. Dumas, B. Erickson, S. Lang, A. Nulens, P. Petrow, J. Rownd, and C. Kirisits. Recommendations from gynaecological (GYN) GEC-ESTRO working group (ii): Concepts and terms in 3D image-based treatment planning in cervix cancer brachytherapy—3D dose volume parameters and aspects of 3D image-based anatomy, radiation physics, radiobiology. *Radiotherapy and Oncology*, 78(1):67–77, 2006. doi:10.1016/j.radonc.2005.11.014.
- 12 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. doi:10.1145/321796.321811.

# Online LZ77 Parsing and Matching Statistics with RLBWTs


**Hideo Bannai**

Department of Informatics, Kyushu University, Japan  
RIKEN Center for Advanced Intelligence Project, Japan  
bannai@inf.kyushu-u.ac.jp  
 <https://orcid.org/0000-0002-6856-5185>

**Travis Gagie<sup>1</sup>**

Diego Portales University and CeBiB, Chile  
travis.gagie@gmail.com

**Tomohiro I<sup>2</sup>**

Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan  
tomohiro@ai.kyutech.ac.jp  
 <https://orcid.org/0000-0001-9106-6192>

---

## Abstract

Lempel-Ziv 1977 (LZ77) parsing, matching statistics and the Burrows-Wheeler Transform (BWT) are all fundamental elements of stringology. In a series of recent papers, Policriti and Prezza (DCC 2016 and *Algorithmica*, CPM 2017) showed how we can use an augmented run-length compressed BWT (RLBWT) of the reverse  $T^R$  of a text  $T$ , to compute offline the LZ77 parse of  $T$  in  $O(n \log r)$  time and  $O(r)$  space, where  $n$  is the length of  $T$  and  $r$  is the number of runs in the BWT of  $T^R$ . In this paper we first extend a well-known technique for updating an unaugmented RLBWT when a character is prepended to a text, to work with Policriti and Prezza's augmented RLBWT. This immediately implies that we can build *online* the LZ77 parse of  $T$  while still using  $O(n \log r)$  time and  $O(r)$  space; it also seems likely to be of independent interest. Our experiments, using an extension of Ohno, Takabatake, I and Sakamoto's (IWOC 2017) implementation of updating, show our approach is both time- and space-efficient for repetitive strings. We then show how to augment the RLBWT further – albeit making it static again and increasing its space by a factor proportional to the size of the alphabet – such that later, given another string  $S$  and  $O(\log \log n)$ -time random access to  $T$ , we can compute the matching statistics of  $S$  with respect to  $T$  in  $O(|S| \log \log n)$  time.

**2012 ACM Subject Classification** Theory of computation → Data compression

**Keywords and phrases** Lempel-Ziv 1977, Matching Statistics, Run-Length Compressed Burrows-Wheeler Transform

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.7

**Acknowledgements** This collaboration started at Shonan seminar 126, “Computation over Compressed Structured Data” and the results from the first part of the paper were proven there.

---

<sup>1</sup> Supported by FONDECYT Grant Number 1171058.

<sup>2</sup> Supported by JSPS KAKENHI Grant Number JP16K16009.



© Hideo Bannai, Travis Gagie, and Tomohiro I;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 7; pp. 7:1–7:12

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Indexes based on the Burrows-Wheeler Transform [4, 8] (BWT) have been in popular use for over a decade, particularly in bioinformatics, for dealing with datasets that are moderately large and compressible. Indexes based on the run-length compressed BWT [17] (RLBWT) can achieve drastically better compression on massive and highly repetitive datasets, such as databases of human genomes, but until recently their apparently inability to support fast locating queries in small space stood in the way of their widespread use. There have still been promising results published about RLBWTs, however, such as Policriti and Prezza’s [23, 24, 25] recent demonstrations that we can use an augmented RLBWT of the reverse of a text to compute offline the LZ77 parse of the text quickly in space proportional to the number of runs in the BWT of the reversed text; and Ohno, Takabatake, I and Sakamoto’s [20] recent practical implementation of an RLBWT that supports efficient prepending of characters. Notably, Policriti and Prezza’s work led directly to Gagie, Navarro and Prezza’s [10] very recent development of an RLBWT that supports fast locating while still taking space proportional to the number of runs in the BWT.

In this paper we strengthen Policriti and Prezza’s result by showing how we can compute *online* the LZ77 parse of  $T$ , while still using  $O(n \log r)$  time and  $O(r)$  space. To do this, we first extend a well-known technique for updating an unaugmented RLBWT when a character is prepended to a text, to work with Policriti and Prezza’s augmented RLBWT. This result seems likely to be of independent interest since, as we will show in the full version of this paper, it can be applied to Gagie, Navarro and Prezza’s RLBWT as well. Assuming that index will be used to store massive genomic databases to which new genomes will sometimes be added, the cost of even occasional rebuilding could be prohibitive. We have implemented our method of updating Policriti and Prezza’s augmented RLBWT on top of Ohno et al.’s implementation of updating an RLBWT, and used it to compute the LZ77 parse online for various datasets. Our experiments show our approach is both time- and space-efficient for repetitive strings.

In the second part of this paper we show how to augment the RLBWT further – albeit making it static again and increasing its space by a factor proportional to the size of the alphabet – such that later, given another string  $S$  and  $O(\log \log n)$ -time random access to  $T$ , we can compute the matching statistics of  $S$  with respect to  $T$  in  $O(|S| \log \log n)$  time. We note that there are practical compressed data structures supporting  $O(\log \log n)$ -time random access to  $T$  in theory, that also usually perform well in practice. Matching statistics are a popular tool in bioinformatics and so calculating them is of independent interest [16, 19], but in this case we are motivated by a particular application to rare-disease detection, which involves finding the minimal substrings in a genome that do not occur in a genomic database. Our result is similar to Belazzougui and Cunial’s [2, 1] with two notable differences: first, they use succinct space (i.e.,  $O(n \log \sigma)$  bits, where  $\sigma$  is the alphabet size), whereas we use compressed space, bounded in terms of  $r$ ; second, they do not consider pointers to longest matches in  $T$  as part of the matching statistics of  $S$  with respect to  $T$ , which we do.

Indexes based on the BWT, not run-length compressed, have been augmented to support functionalities far beyond standard pattern matching, but their “killer app” was DNA assembly. That may not be the case for RLBWTs: high-throughput sequencing is resulting in massive genomic databases and – although assembling genomes using entire databases as references may be interesting for, e.g., pan-genomics [26] – there are many other things we might want to do with those databases and some of them require rich query functionalities. We note that another primary task in pan-genomics, parsing a given genome into the

minimum number of phrases that can each be found in a database or, similarly, computing the genomes RLZ [13] of the genome with respect to the database, is also straightforward with an RLBWT. If we want the phrases to be aligned, we can use a combination of the RLBWT and PBWT [6].

There are already compact data structures with rich functionalities, such as straight-line programs [14], CDAWGS [3] and compressed suffix trees [18, 9], but none of them has caught on among practitioners the way BWTs did. At the same time as we try to improve the theoretical and practical time- and space-bounds of those data structures, therefore, we should try to extend the functionalities of the RLBWT (while keeping it practical). Even apart from the specific results we present in this paper, we hope it provides momentum for that effort.

## 2 Preliminaries

For the sake of brevity, we assume the reader is familiar with LZ77 (we consider the original version, with which phrases end with mismatch characters), matching statistics, the BWT and RLBWT and how to search with them, etc. In this section we first briefly describe our simplification of Policriti and Prezza's augmented RLBWT (without going into their algorithm for LZ77 parsing) and then we review how to update a standard BWT or RLBWT when a character is prepended to the text.

### 2.1 Policriti and Prezza's augmented RLBWT

In addition to all the data structures associated with the unaugmented RLBWT for a text, Policriti and Prezza's augmented RLBWT stores the suffix-array entries  $SA[i]$  and  $SA[j]$  that are the positions in the text of the first and last characters in each run  $BWT[i..j]$ . They showed how, with this extra information, a backward search for a pattern can be made to return the location of one of its occurrence (assuming it occurs at all).

We can simplify and strengthen Policriti and Prezza's result slightly, storing only the position of the first character of each run and finding the starting position of the lexicographically first suffix starting with a given pattern. When we start a backward search for a pattern  $P[1..m]$ , the initial interval is all of  $BWT[1..n]$  and we know  $SA[1]$  since  $BWT[1]$  must be the first character in a run. Now suppose we have processed  $P[i..m]$ , the current interval is  $BWT[j..k]$  and we know  $SA[j]$ . If  $BWT[j] = P[i - 1]$  then the interval for  $P[i - 1..m]$  starts with  $BWT[LF(j)]$ , where  $LF(j) = SA^{-1}[SA[j] - 1]$  can be found as usual, and so we know  $SA[LF(j)] = SA[j] - 1$ . Otherwise, the interval for  $P[i - 1..m]$  starts with  $BWT[LF(j')]$ , where  $j'$  is the position of the first occurrence of  $P[i - 1]$  in  $BWT[j..k]$ ; since  $BWT[j']$  is the first character in a run,  $j'$  is easy to compute and we have  $SA[j']$  stored and can thus compute  $SA[LF(j')]$ .

► **Lemma 1.** *We can augment an RLBWT with  $O(r)$  words, where  $r$  is the number of runs in the BWT, such that after each step in a backward search for a pattern, we can return the starting position of the lexicographically first suffix prefixed by the suffix of the pattern we have processed so far.*

### 2.2 Updating an RLBWT

Suppose we have an RLBWT for  $\$T[i + 1..n]$ , where  $\$$  does not occur in  $T$ , and know the position  $d$  of  $\$$  in the current BWT. To obtain an RLBWT for  $\$T[i..n]$ , we compute  $\text{rank}_{T[i]}(d)$  and use it to compute  $LF(p)$ , where  $p$  is the position (which we need not compute)

of the occurrence before \$ of  $T[i]$  in the current BWT. We replace \$ by  $T[i]$  in the RLBWT, which may require merging that copy of  $T[i]$  with the preceding run, the succeeding run, or both. We then insert \$ after  $\text{BWT}[\text{LF}(p)]$ , which may require splitting a run. Updating the RLBWT for  $T^R$  is symmetric when we append a character to  $T$ . Ohno et al. gave a practical implementation that supports updates in  $O(r)$  time and backward searches in  $O(\log r)$  time per character in the pattern.

► **Lemma 2** (see [20]). *We can build an RLBWT for  $T^R$  incrementally, starting with the empty string and iteratively prepending  $T[1], \dots, T[n]$  – so that after  $i$  steps we have an RLBWT for  $(T[1..i])^R$  – using a total of  $O(n \log r)$  time. Backward searches always take  $O(\log r)$  time per character in the pattern.*

### 3 Online LZ77 Parsing

Our first idea is to extend the technique from Subsection 2.2 for updating an unaugmented RLBWT, to apply to an augmented RLBWT. Then we can build an augmented RLBWT for  $T^R$  incrementally, starting with the empty string and iteratively prepending  $T[1], \dots, T[n]$ . Our second idea is to mix prepending characters to a suffix of  $T^R$  with backward searching for a prefix of that suffix, which is equivalent to appending characters to a prefix of  $T$  while searching for a suffix of that prefix.

#### 3.1 Updating an augmented RLBWT

Suppose we have an augmented RLBWT for  $\$T[i+1..n]$ , where \$ does not occur in  $T$ , and know the position  $d$  of \$ in the current BWT. To obtain an augmented RLBWT for  $\$T[i..n]$ , we compute  $\text{rank}_{T[i]}(d)$  and use it to compute  $\text{LF}(p)$ , where  $p$  is the position (which we need not compute) of the occurrence before \$ of  $T[i]$  in the current BWT. At this point we perform some calculations that were not necessary in Subsection 2.2 since, if we will split a run when we reinsert \$, we should know the position in  $T$  of the character after where we will reinsert it.

If there is a copy of  $T[i]$  after \$ in the current BWT, we find the first such copy  $\text{BWT}[q]$ , which must be the first character of a run so we have  $\text{SA}[q]$  stored. If there is no such copy, then we find the first copy  $\text{BWT}[q]$  of the smallest character lexicographically larger than  $T[i]$ , which again must be the first character of a run so we have  $\text{SA}[q]$  stored. If there is no such character, then  $\text{BWT}[\text{LF}(p)]$  is the last character in the current BWT, in which case we can proceed as in Subsection 2.2.

We replace \$ by  $T[i]$  in the RLBWT, which may require merging that copy of  $T[i]$  with the preceding run, the succeeding run, or both. We then insert \$ after  $\text{BWT}[\text{LF}(p)]$ , which may require splitting a run. If so, the position in  $T$  of the character now after \$ is  $\text{SA}[q] - 1$ . Updating the augmented RLBWT for  $T^R$  is symmetric when we append a character to  $T$ . We can extend Ohno et al.'s implementation to support updates to the augmented RLBWT for  $T^R$  in  $O(r)$  time and backward searches still in  $O(\log r)$  time per character in the pattern.

► **Lemma 3.** *We can build an augmented RLBWT for  $T^R$  incrementally, starting with the empty string and iteratively prepending  $T[1], \dots, T[n]$  – so that after  $i$  steps we have an RLBWT for  $(T[1..i])^R$  – using a total of  $O(n \log r)$  time. Backward searches always take  $O(\log r)$  time per character in the pattern.*

### 3.2 Computing the parse

Suppose we currently have an augmented RLBWT for  $(T[1..j])^R$  and the following information:

- the phrase containing  $T[j + 1]$  in the LZ77 parse of  $T$  starts at  $T[i]$ ;
- the non-empty interval  $I$  for  $(T[i..j])^R$  in the BWT for  $(T[1..j - 1])^R$ ;
- the position in  $(T[1..j - 1])^R$  of the first character in  $I$ ;
- the interval  $I'$  for  $(T[i..j + 1])^R$  in the BWT for  $(T[1..j])^R$ ;
- the position in  $(T[1..j])^R$  of the first character in  $I'$ , if  $I'$  is non-empty.

If  $I'$  is empty, then the phrase containing  $T[j + 1]$  is  $T[i..j + 1]$  with  $T[j + 1]$  being the mismatch character, and we can compute the position of an occurrence of  $T[i..j]$  in  $T[1..j - 1]$  from the position of the first character in  $I$ . We then prepend  $T[j + 1]$  to  $(T[1..j])^R$ , update the augmented RLBWT, and start a new backward search for  $T[j + 1]$ .

If  $I'$  is non-empty, then we know the phrase containing  $T[j + 2]$  starts at  $T[i]$ , so we prepend  $T[j + 1]$  to  $(T[1..j])^R$ , update the augmented RLBWT (while keeping track of the endpoints of  $I'$ ), and perform a backward step for  $T[j + 2]$  to obtain the interval  $I''$  for  $(T[i..j + 2])^R$  in the BWT for  $(T[1..j + 1])^R$ . If  $I''$  is non-empty, the augmented RLBWT returns the position in  $(T[1..j + 1])^R$  of the first character in  $I''$ .

Continuing like this, we can simultaneously incrementally build the augmented RLBWT for  $T^R$  while parsing  $T$ . Each step takes  $O(\log r)$  time and we use constant workspace on top of the augmented RLBWT, which always contains at most  $r$  runs, so we use  $O(r)$  space. This gives us our first main result:

► **Theorem 4.** *We can compute the LZ77 parse for  $T[1..n]$  online using  $O(n \log r)$  time and  $O(r)$  space, where  $r$  is the number of runs in the BWT for  $T^R$ .*

### 3.3 Experimental results

We implemented in C++ the online LZ77 parsing algorithm of Theorem 4 (the source code is available at [21]). We evaluate the performance of our method comparing with the state-of-the-art implementations for LZ77 parsing that potentially can work in the peak RAM usage smaller than  $n \lg \sigma + n \lg n$  bits. A brief explanation and setting of each method we tested is the following:

- **LZscan** [12, 15]. It runs in  $O(nd \log(n/d))$  time and  $(n/d) \lg n$  bits in addition to the input string, where  $d$  is a parameter that can be used to control time-space tradeoffs. We set  $d$  so that  $(n/d) \lg n$  is roughly half of the input size.
- **h0-lz77** [22, 7]. Online LZ77 parsing based on BWT running in  $O(n \log n)$  time and  $nH_0 + o(n \log \sigma) + O(\sigma \log n)$  bits of space. The current implementation runs in  $O(n \log n \log \sigma)$  time.
- **r1e-lz77-1** [25, 7]. Offline LZ77 parsing algorithm based on RLBWT with two sampled suffix array entries for each run. In theory it runs in  $O(n \log r)$  time and  $2r \lg n + r \lg \sigma + o(r \lg \sigma) + O(r \lg(n/r) + \sigma \lg n)$  bits of working space. The current implementation runs in  $O(n \log r \log \sigma)$  time.
- **r1e-lz77-2** [25, 7]. Offline LZ77 parsing algorithm based on RLBWT that theoretically runs in  $O(n \log r)$  time and  $z(\lg n + \lg z) + r \lg \sigma + o(r \lg \sigma) + O(r \lg(n/r) + \sigma \lg n)$  bits of working space. The current implementation runs in  $O(n \log r \log \sigma)$  time.
- **r1e-lz77-o** [Theorem 4]. To make the parsing done in a reasonable time, our online RLBWT implementation is based on [20], which runs faster (actually in  $O(n \log r)$  time) than [7] but needs  $2r \lg r$  extra bits. Online LZ77 parsing can be done in  $O(n \log r)$  time and  $2r \lg r + r \lg n + O(r \lg(n/r) + \sigma \lg n)$  bits of working space.



For the above methods other than `r1e-lz77-2`, the output space is not counted in the working space since they compute LZ77 phrases sequentially. On the other hand, `r1e-lz77-2` counts  $\lg n$  bits of working space to store the starting positions of the phrases as they are not computed sequentially.

We tested on highly repetitive datasets in `repcorpus`<sup>3</sup>, well-known corpus in this field, and some larger datasets created from git repositories. For the latter, we use the script [11] to create 1024MiB texts (obtained by concatenating source files from the latest revisions of a given repository, and truncated to be 1024MiB) from the repositories for `boost`<sup>4</sup>, `samtools`<sup>5</sup> and `sdsl-lite`<sup>6</sup> (all accessed at 2017-03-27). The programs were compiled using `g++6.3.0` with `-Ofast -march=native` option. The experiments were conducted on a 6core Xeon E5-1650V3 (3.5GHz) machine using a single core with 32GiB memory running Linux CentOS7.

In Table 1, we compare our method `r1e-lz77-o` with `r1e-lz77-2`, which is the most relevant to our method as well as the most space efficient one. The result shows that our method significantly improves the running time while keeping the increase of the space within 4 times. It can be observed that the working space of `r1e-lz77-o` gets worse as the input is less compressible in terms of RLBWT (especially for `Escherichia_Coli`).

Figure 1 compares all the tested methods for some selected datasets. It shows that `r1e-lz77-o` exhibits an interesting time-space tradeoff: running in just a few times slower than `LZscan` while working in compressed space.

## 4 Matching Statistics

The matching statistics of  $S[1..m]$  with respect to  $T$  tell us, for each suffix  $S[i..m]$  of  $S$ , what is the length  $\ell_i$  of the longest substring  $S[i..i + \ell_i - 1]$  that occurs in  $T$  and the position  $p_i$  of one of its occurrences there. We can compute  $\ell_i$  and  $p_i$  using Policriti and Prezza's augmented RLBWT for  $T^R$  by performing a backward search for each  $(S[i..m])^R$  – i.e., performing a backward step for  $S[i]$ , then another for  $S[i + 1]$ , etc. – until the interval in the BWT becomes empty, and then undoing the last backward step. However, to compute all the matching statistics this way takes time proportional to the sum of all the  $\ell$  values – which can be quadratic in  $m$  – times the time for a backward step.

Suppose we use Policriti and Prezza's augmented RLBWT for  $T$  (which stores the positions in  $T$  of both the first and last character of each run) to perform a backward search for  $S$  – i.e., performing a backward step for  $S[m]$ , then another for  $S[m - 1]$ , etc. – until the interval in the BWT becomes empty, and then undo the last backward step. This gives us the last few  $\ell$  and  $p$  values in the matching statistics for  $S$ , and the interval  $\text{BWT}[i..j]$  for some suffix  $S[k..m]$  of  $S$  such that  $S[k - 1..m]$  does not occur in  $T$  (meaning  $S[k - 1]$  does not occur in  $\text{BWT}[i..j]$ ). Consider the suffixes of  $T$  starting with the occurrences of  $S[k - 1]$  preceding  $\text{BWT}[i]$  and following  $\text{BWT}[j]$  in the BWT, which are the last and first characters in runs, respectively. By the definition of the BWT, one of these two suffixes has the longest common prefix with  $S[k - 1..m]$  – and, equivalently, with  $S[k - 1]T[p_k..n]$  – of all the suffixes of  $T$ . Therefore, if we know which of those two suffixes has the longer common prefix with  $S[k - 1]T[p_k..n]$ , we can deduce  $p_{k-1}$ .

<sup>3</sup> See <http://pizzachili.dcc.uchile.cl/repcorpus/statistics.pdf> for statistics of the datasets.

<sup>4</sup> <https://github.com/boostorg/boost>

<sup>5</sup> <https://github.com/samtools/samtools>

<sup>6</sup> <https://github.com/simongog/sdsl-lite>

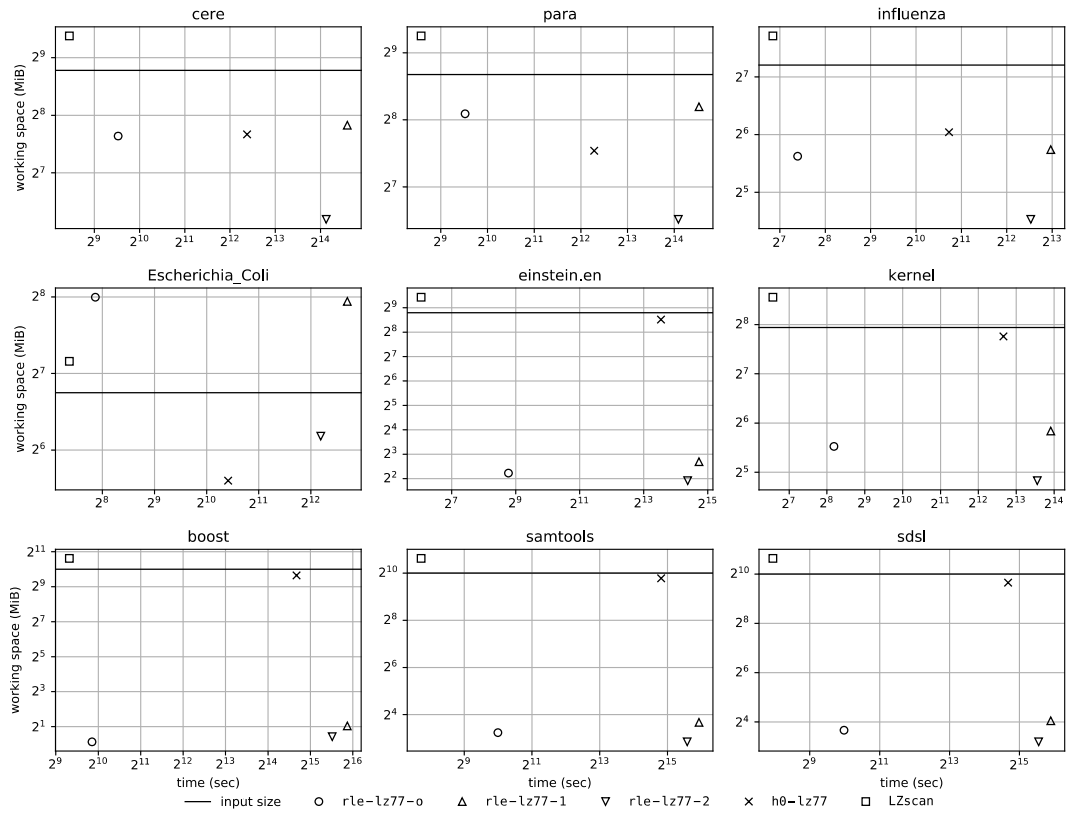


■ **Table 1** Comparison of LZ77 parsing time and working space (WS) between `rle-lz77-o` (shortened as `-o`) and `rle-lz77-2` (shortened as `-2`), where  $|T|$  is the input size (considering each character takes one byte),  $z$  is the number of LZ77 phrases for  $T$  and  $r$  is the number of runs in RLBWT for  $T^R$ .

dataset	$ T $ (MiB)	$z$	$r$	time (sec)		WS (MiB)	
				<code>-o</code>	<code>-2</code>	<code>-o</code>	<code>-2</code>
fib41	255.503	40	42	131	1334	0.065	0.071
rs.13	206.706	39	76	111	1402	0.065	0.072
tm29	256.000	54	82	104	1889	0.065	0.072
dblp.xml.00001.1	100.000	48,882	172,195	94	4754	2.694	2.258
dblp.xml.00001.2	100.000	48,865	175,278	94	4786	2.744	2.273
dblp.xml.0001.1	100.000	58,180	240,376	97	4823	3.791	2.714
dblp.xml.0001.2	100.000	58,171	269,690	97	4804	4.253	2.860
dna.001.1	100.000	198,362	1,717,162	114	3951	27.537	9.672
english.001.2	100.000	216,828	1,436,696	112	4884	23.115	10.177
proteins.001.1	100.000	221,819	1,278,264	111	4288	20.481	9.246
sources.001.2	100.000	178,138	1,211,104	105	4886	19.524	9.007
cere	439.917	1,394,808	11,575,582	737	17883	199.436	73.154
coreutils	195.772	1,286,069	4,732,794	252	9996	78.414	51.822
einstein.de.txt	88.461	28,226	99,833	82	4098	1.606	1.618
einstein.en.txt	445.963	75,778	286,697	437	21198	4.675	3.773
Escherichia_Coli	107.469	1,752,701	15,045,277	233	4674	255.363	72.527
influenza	147.637	557,348	3,018,824	168	5909	49.319	23.078
kernel	246.011	705,790	2,780,095	291	12053	46.036	28.426
para	409.380	1,879,634	15,635,177	734	17411	272.722	91.515
world_leaders	44.792	155,936	583,396	43	2002	9.092	5.932
boost	1024.000	20,630	63,710	925	46760	1.094	1.344
samtools	1024.000	158,886	562,326	1020	48967	9.445	7.190
sdsl	1024.000	210,501	758,657	1010	47964	12.677	9.138

Our first idea is to further augment Policriti and Prezza’s RLBWT such that, for any position  $i$  in the BWT and any character  $c$ , we can tell whether  $cT[SA[i]]$  has a longer common prefix with the suffix of  $T$  starting with the occurrence of  $c$  preceding  $BWT[i]$ , or with the one starting with the occurrence of  $c$  following  $BWT[i]$ . Although it sounds at first like this should use  $\Omega(n)$  space, in fact it takes  $O(\sigma)$  space per run in the BWT, where  $\sigma$  is the alphabet size. With this information, we can compute the  $p$  values for the matching statistics, using a right-to-left pass over  $S$ .

Once we have the  $p$  values, we use a left-to-right pass over  $S$  to compute the  $\ell$  values. Notice that it would again take time at least proportional to the sum of the  $\ell$  values, to start at each  $T[p_i]$  and extract characters until finding a mismatch. Since  $\ell_{i+1}$  cannot be less than  $\ell_i - 1$ , however, if we have a compact data structure that supports  $O(\log \log n)$ -time random access to  $T$  – such as the RLZ parse implemented with a y-fast trie or a bitvector [13, 5] – then we can compute all the  $\ell$  values in  $O(m \log \log n)$  total time using small space. Since the size of the RLZ parse is generally comparable to that of the RLBWT when there is a natural reference sequence, which is the case when dealing with databases of genomes from the same species, using random access to  $T$  seems unlikely to be an obstacle in practice.



■ **Figure 1** Comparison of LZ77 parsing time and working space.

#### 4.1 Further augmentation

For each run  $BWT[i..k]$  and each character  $c$  except the one in that run, we add to Policriti and Prezza's augmented RLBWT the threshold position  $j$  between  $i$  and  $k$  such that, for  $i \leq i' < j$ , each string  $cT[SA[i']..n]$  has a longer common prefix with the suffix of  $T$  starting at the copy of  $c$  preceding  $BWT[i]$  but, for  $j \leq k' \leq k$ , each string  $cT[SA[k']..n]$  has a longer common prefix with the suffix of  $T$  starting at the copy of  $c$  following  $BWT[k]$ . By the definition of the BWT, the length of the longest common prefix with the suffix of  $T$  starting at the copy of  $c$  preceding  $BWT[i]$ , is non-increasing as we go from  $BWT[i]$  to  $BWT[k]$ , and the length of the longest common prefix with the suffix of  $T$  starting at the copy of  $c$  following  $BWT[k]$  is non-decreasing; therefore there is at most one such threshold  $j$ . Dealing with special cases, such as when even  $cT[SA[i]..n]$  has a longer common prefix with the suffix of  $T$  starting at the copy of  $c$  following  $BWT[k]$ , takes a constant number of extra bits, so in total we use  $O(r\sigma)$  space for this additional augmentation, where  $r$  is now the number of runs in the BWT for  $T$  (not  $T^R$ ).

► **Lemma 5.** *We can augment an RLBWT for  $T$  with  $O(r\sigma)$  words, where  $r$  is the number of runs in the BWT for  $T$  and  $\sigma$  is the size of the alphabet, such that for any position  $i$  in the BWT and any character  $c$ , in  $O(1)$  time we can tell whether  $cT[SA[i]]$  has a longer common prefix with the suffix of  $T$  starting with the occurrence of  $c$  preceding  $BWT[i]$ , or with the one starting with the occurrence of  $c$  following  $BWT[i]$ .*

---

**Algorithm 1** Computing  $p$  values for the matching statistics of  $S$  with respect to  $T$ , using an augmented RLBWT for  $T$ . For simplicity we ignore special cases, such as when some character in  $S$  does not occur in  $T$ .

---

```

procedure COMPUTEPS( $S$ )
   $q \leftarrow$  position of the first or last character in any run
   $t \leftarrow$  position of BWT[ $q$ ] in  $T$ 
  for  $i \leftarrow m \dots 1$  do
    if BWT[ $q$ ]  $\neq$   $S[i]$  then
      if BWT[ $q$ ] is before the threshold for its run then
         $q \leftarrow$  position of the preceding occurrence of  $S[i]$  in the BWT
      else
         $q \leftarrow$  position of the following occurrence of  $S[i]$  in the BWT
       $t \leftarrow$  position of BWT[ $q$ ] in  $T$ 
     $p_i \leftarrow t$ 
     $q \leftarrow$  LF( $q$ )
     $t \leftarrow t - 1$ 

```

---

## 4.2 Algorithm

As we have said, our algorithm consists of first computing all the  $p$  values in the matching statistics using a right-to-left pass over  $S$ , then computing all the  $\ell$  values using a left-to-right pass. We first choose  $q$  to be the position of the first or last character in any run and set  $t$  to be its position in  $T$ . We then walk backward in  $S$  and  $T$  until we find a mismatch  $S[i] \neq \text{BWT}[q]$ , at which point we reset  $q$  to be the position of either the copy of  $S[i]$  preceding  $\text{BWT}[q]$  or of the one following it, depending on whether  $\text{BWT}[q]$  is before or after the threshold position for  $S[i]$  in its run. The time is dominated by backward-stepping, which can be made  $O(\log \log n)$  with Policriti and Prezza's RLBWT, so we use a total of  $O(m \log \log n)$  time. Algorithm 1 shows pseudocode.

Once we have the  $p$  values, we make a left-to-right pass over  $S$  to compute the  $\ell$  values. We start with  $S[1]$  and  $T[p_1]$  and walk forward, comparing  $S$  to  $T$  character by character, until we find a mismatch  $S[1 + \ell_1 - 1] \neq T[p_1 + \ell_1 - 1]$ , and set  $\ell_1$  appropriately. We know  $\ell_2 \geq \ell_1 - 1$ , so  $S[2..2 + \ell_1 - 2] = T[p_2..p_2 + \ell_1 - 2]$  and we can jump directly to comparing  $S[2 + \ell_1 - 1..m]$  to  $T[p_2 + \ell_1 - 1..m]$  character by character until we find a mismatch,  $S[2 + \ell_2 - 1] \neq T[p_2 + \ell_2 - 1]$ , and set  $\ell_2$  appropriately. Continuing like this with  $O(\log \log n)$ -time random access to  $T$ , we compute all the  $\ell$  values in  $O(m \log \log n)$  time. Algorithm 2 shows pseudocode. This gives us our second main result:

► **Theorem 6.** *We can augment an RLBWT for  $T$  with  $O(r\sigma)$  words, where  $r$  is the number of runs in the BWT for  $T$  and  $\sigma$  is the size of the alphabet, such that later, given  $S[1..m]$  and  $O(\log \log n)$ -time random access to  $T$ , we can compute the matching statistics for  $S$  with respect to  $T$  in  $O(m \log \log n)$  time.*

## 4.3 Application: Rare-disease detection

Each substring  $S[i..i + \ell_i - 1]$  is necessarily a right-maximal substring of  $S$  that has a match in  $T$ , but not necessarily a left-maximal one. We can easily post-process the matching statistics of  $S$  in  $O(m)$  time to find the maximal substrings with matches in  $T$ : if  $\ell_i = \ell_{i+1} + 1$ , then we discard  $\ell_{i+1}$  and  $p_{i+1}$ . Similarly, in  $O(m)$  time we can find all the minimal substrings

---

**Algorithm 2** Computing  $\ell$  values for the matching statistics of  $S$  with respect to  $T$ , using the  $p$  values and random access to  $T$ . Again, for simplicity we ignore special cases, such as when some character in  $S$  does not occur in  $T$ .

---

```

procedure COMPUTELS( $S, p_1, \dots, p_m$ )
   $\ell_0 \leftarrow 1$ 
  for  $i \leftarrow 1 \dots m$  do
     $\ell_i \leftarrow \ell_{i-1} - 1$ 
    while  $S[i + \ell_i] = T[p_i + \ell_i]$  do
       $\ell_i \leftarrow \ell_i + 1$ 

```

---

of  $S$  that have no matches in  $T$ : for each maximal matching substring of  $S$ , extending it either one character to the right or one character to the left yields a minimal non-matching substring; assuming each character in  $S$  occurs in  $T$ , this yields all the minimal non-matching substrings of  $S$ .

Finding all the non-matching substrings of a string relative to a large database of strings has applications to bioinformatics, specifically, in rare-disease discovery. For example, we might want to preprocess a large database of human genomes such that when a patient arrives with an unknown disease we suspect to be genetic, we can quickly find all the minimal substrings of his or her genome that do not occur in the database.

## 5 Recent and Future Work

We noticed recently (after the submission deadline) that we can easily remove the  $\sigma$  from the space bound in Theorem 6: between two consecutive runs of a character, we need store only a single position where suffixes switch from having a longer common prefix with the suffix following the last character in the earlier run, to having a longer common prefix with the suffix following the first character in the later run. Given a position in the BWT and a character  $c$ , we find the preceding and following runs of  $c$ 's and simply check on which side of the threshold between them the given position lies.

We think we have also found an efficient way to build the augmented RLBWT from Theorem 6, by first storing the length of the longest common prefix (LCP) of the suffixes following the characters on either side of each run boundary in the BWT. To find these LCP values, we start at the position in the BWT of the first character of the text, find the positions in the text of its neighbours in the BWT, and use random access to walk backwards in the text, performing character-by-character comparisons until we know the LCP values. We then move to the position in the BWT of the second character in the text. This time, however, we know the LCP values are not less than the previous LCP values minus 1 each, and we can use random access to skip pairs of characters we already know match, avoiding unnecessary comparisons. Repeating this for each character of the text, from left to right, we calculate all the LCP values in  $O(n \log \log n)$  total time, but we store only those at the ends of runs in the BWT. We can then support access to the array of all LCP values at the same time we support access to the suffix array. For each pair of consecutive runs of a character, we scan the LCP array entries between then to find the position of the threshold between them. This takes a total of  $O(\sigma n \log \log n)$  time.

We will update Section 4 appropriately in the full version of this paper. Now that we have a reasonable way of constructing the data structure from Theorem 6, we also plan to implement and test it, working toward a prototype for our target application of rare-disease detection.

## References

- 1 Djamal Belazzougui and Fabio Cunial. Fast matching statistics in small space. In *Proceedings of the 17th Symposium on Experimental Algorithms (SEA)*, pages 179–190, 2014.
- 2 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *Proceedings of the 21st Symposium on String Processing and Information Retrieval (SPIRE)*, pages 179–190, 2014.
- 3 Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:13, 2017.
- 4 Michael Burrows and David J. Wheeler. A block sorting lossless compression algorithm. Technical Report 124, DEC, 1994.
- 5 Anthony J. Cox, Andrea Farruggia, Travis Gagie, Simon J. Puglisi, and Jouni Sirén. RLZAP: relative Lempel-Ziv with adaptive pointers. In *Proceedings of the 23rd Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–14, 2016.
- 6 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.
- 7 Dynamic: dynamic succinct/compressed data structures library. URL: <https://github.com/xxsds/DYNAMIC>.
- 8 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.
- 9 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. Technical Report 1705.10382, arXiv.org, 2017.
- 10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the 19th Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.
- 11 get-git-revisions: Get all revisions of a git repository. URL: <https://github.com/nicolaprezza/get-git-revisions>.
- 12 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proceedings of the 13th Symposium on Experimental Algorithms (SEA)*, pages 139–150, 2013.
- 13 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.
- 14 Markus Lohrey. Algorithmics on slp-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
- 15 Lzscan. URL: <https://www.cs.helsinki.fi/group/pads/>.
- 16 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-scale algorithm design*. Cambridge University Press, 2015.
- 17 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
- 18 Gonzalo Navarro and Alberto Ordóñez Pereira. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 21(1):1.8:1–1.8:38, 2016.
- 19 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
- 20 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler transform. In *Proceedings of the 28th International Workshop on Combinatorial Algorithms (IWOCA)*, 2017. To appear.
- 21 Online rlbwt. URL: <https://github.com/itomomoti/OnlineRlbwt>.

- 22 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In *Proceedings of the 22nd Symposium on String Processing and Information Retrieval (SPIRE)*, pages 13–20, 2015.
- 23 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In *Proceedings of the Data Compression Conference (DCC)*, pages 23–32, 2016.
- 24 Alberto Policriti and Nicola Prezza. From LZ77 to the run-length encoded Burrows-Wheeler transform, and back. In *Proceedings of the 28th Symposium on Combinatorial Pattern Matching (CPM)*, pages 17:1–17:10, 2017.
- 25 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, Jul 2017.
- 26 Daniel Valenzuela and Veli Mäkinen. CHIC: a short read aligner for pan-genomic references. *bioRxiv*, 2017.

# Non-Overlapping Indexing – Cache Obliviously

**Sahar Hooshmand**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
sahar@cs.ucf.edu

**Paniz Abedin**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
paniz@cs.ucf.edu

**M. Oğuzhan Külekci**

Informatics Institute, Istanbul Technical University - Turkey  
kulekci@itu.edu.tr

**Sharma V. Thankachan**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
sharma.thankachan@ucf.edu

---

## Abstract

The *non-overlapping indexing* problem is defined as follows: pre-process a given text  $T[1, n]$  of length  $n$  into a data structure such that whenever a pattern  $P[1, p]$  comes as an input, we can efficiently report the largest set of non-overlapping occurrences of  $P$  in  $T$ . The best known solution is by Cohen and Porat [ISAAC, 2009]. Their index size is  $O(n)$  words and query time is optimal  $O(p + \text{nocc})$ , where  $\text{nocc}$  is the output size. We study this problem in the cache-oblivious model and present a new data structure of size  $O(n \log n)$  words. It can answer queries in optimal  $O(\frac{p}{B} + \log_B n + \frac{\text{nocc}}{B})$  I/Os, where  $B$  is the block size.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** Suffix Trees, Cache Oblivious, Data Structure, String Algorithms

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.8

**Funding** Part of this work was done while the last author was visiting the third author with the TÜBİTAK-BİDEB 2221 program grant number 1059B211700766. This work has also received funding from the European Union’s Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 69094 in the form of student travel grant to present a preliminary version of this work in the 12th Workshop on Compression, Text and Algorithms (WCTA), 2017.

## 1 Introduction and Related Work

Text indexing is fundamental to many areas in Computer Science such as Information Retrieval, Bioinformatics, etc. The primary goal here is to pre-process a long text  $T[1, n]$  (given in advance), such that whenever a shorter pattern  $P[1, p]$  comes as query, all occ occurrences (or simply, starting positions) of  $P$  in  $T$  can be reported efficiently. Such queries can be answered in optimal  $O(p + \text{occ})$  time using the classic *Suffix tree* data structure [14, 15]. It takes  $O(n)$  words of space. In this paper, we focus on a variation of the text indexing problem, known as the *non-overlapping indexing*. Here we are interested in finding the largest set of occurrences of  $P$  in  $T$  (denote its size by  $\text{nocc}$ ), such that any two (distinct) text positions in the output are separated by at least  $p$  characters. This primitive is central to data compression [2, 5]. The above task can be easily reduced to a set of geometric range



© Sahar Hooshmand, Paniz Abedin, M. Oğuzhan Külekci, and Sharma V. Thankachan;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 8; pp. 8:1–8:9

Leibniz International Proceedings in Informatics



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



queries, specifically  $(1 + \text{nocc})$  number of *orthogonal range next value* queries on the suffix array [12] of  $\mathbb{T}$ . Although efficient, the solutions based on this approach are not optimal in terms of query time [11, 13]. The first space-efficient (linear) and optimal  $O(p + \text{nocc})$  time solution is due to Cohen and Porat [5]. They took an alternative strategy in which the periodicity of both text and the pattern are exploited. Subsequently, Ganguly *et al.* [9] showed that the problem can also be solved in succinct space.

Unfortunately, all the aforementioned indexes heavily rely on random access over the data structure, therefore efficient only when resides in the internal memory (usually RAM, the random access memory). To this end, we revisit this problem in the secondary memory model in the context of very large input data. Here we assume that the data (and the data structure) is too big to fit within the main memory, therefore deployed in a (much larger, but slower) secondary memory. Popular models of computation are (i) the cache-aware model and (ii) the cache-oblivious model. We now present a brief description of both models.

In the **cache-aware model** (a.k.a. external memory model, I/O model and disk access model), introduced by Aggarwal and Vitter [1] the CPU is connected directly to an internal memory (of size  $M$  words), which is then connected to a very large external memory (disk). The disk is partitioned into blocks/pages and the size of each block is  $B$  words. The CPU can only work on data inside the internal memory. Therefore, to work on some data in the external memory, the corresponding blocks have to be transferred to internal memory. The transfer of a block from external memory to internal memory (or vice versa) is referred as an I/O operation. The operations inside the internal memory are orders of magnitude faster than the time for an I/O operation. Therefore, they are considered free, and the efficiency of an algorithm is measured in terms of the number of I/O operations. The **cache-oblivious model** is essentially the same as above, except the following key twist:  $M$  and  $B$  are unknown at the time of the design of algorithms and data structures [8, 7]. This means, if a cache-oblivious algorithm performs optimally between two levels of the memory hierarchy, then it is optimal at any level of the memory hierarchy. Lastly, cache-oblivious algorithms are usually more intricate than cache-aware algorithms.

**Contribution.** We present the first I/O-optimal solution for the non-overlapping indexing problem. To the best of our knowledge, this is the first of its kind over both cache-oblivious and cache-aware models of computation. The main result is summarized below.

► **Theorem 1.** *There exists an  $O(n \log n)$  space data structure for the non-overlapping indexing problem in the cache oblivious model, where  $n$  is the length of the input text  $\mathbb{T}$ . It can report the largest set of non-overlapping occurrences of an input pattern  $P[1, p]$  in their **sorted order** in optimal  $O(\frac{p}{B} + \log_B n + \frac{\text{nocc}}{B})$  I/Os, where  $\text{nocc}$  is the output size.*

The main component of our index is the **suffix tree** data structure and its cache-oblivious counter part [4]. The suffix tree of  $\mathbb{T}$  (denoted by  $\text{ST}$ ) is a compact trie of all  $n$  suffixes of  $\mathbb{T}$ . It has  $n$  leaves and at most  $(n - 1)$  internal nodes (each having at least two children). Corresponding to each leaf in  $\text{ST}$ , there is a unique suffix in  $\mathbb{T}$ . Specifically, the  $i$ th leftmost leaf  $\ell_i$  corresponds to the  $i$ th lexicographically smallest suffix of  $\mathbb{T}$ , denoted by  $\mathbb{T}[\text{SA}[i], n]$ . Edges are labeled and the concatenation of edge labels on the path from root to a node  $u$  is called its path, denoted by  $\text{path}(u)$ . The locus of a pattern  $P$ , denoted by  $\text{locus}(P)$  is the node closest to root, such that  $P$  is a prefix of its path. The array  $\text{SA}$  is called the **suffix array** of  $\mathbb{T}$ . The suffix range of a pattern  $P$ , denoted by  $[\text{sp}(P), \text{ep}(P)]$  is the range of (contiguous) leaves in the subtree of  $\text{locus}(P)$ . Therefore, the set of occurrences of  $P$  is  $\{\text{SA}[i] \mid \text{sp}(P) \leq i \leq \text{ep}(P)\}$  and  $\text{ep}(P) - \text{sp}(P) + 1 = \text{occ}$ . The suffix range can



be computed in  $O(p)$  time. The space is  $O(n)$  words for both suffix array and suffix tree. For our problem, we maintain both the suffix tree and its cache-oblivious equivalent by Brodal and Fagerberg [4], which occupies  $O(n)$  space and can compute  $\text{locus}(P)$  in optimal  $O(p/B + \log_B n)$  I/Os. Moreover, we design a data structure for reporting occurrences in the sorted order (see Theorem 2), which may be of independent interest. We remark that all results in this paper assumes  $M > B^{2+\Theta(1)}$  as in [4].

► **Theorem 2.** *A given text  $\mathbb{T}[1, n]$  can be indexed in  $O(n \log n)$  words in the cache oblivious model, such that we can report all  $\text{occ}$  occurrences of an input pattern  $P[1, p]$  in their **sorted order** in optimal  $O(\frac{p}{B} + \log_B n + \frac{\text{occ}}{B})$  I/Os.*

We arrive at Theorem 1 by exploring the periodicity of query pattern. Let  $Q$  be the shortest prefix of  $P$  such that  $P$  can be written as the concatenation of  $\alpha \geq 1$  copies of  $Q$  and a (possibly empty) prefix  $R$  of  $Q$ . i.e.,  $P = Q^\alpha R$ . Then, the period of  $P$ , denoted by  $\text{period}(P)$  is  $|Q|$ . For example,  $Q = \text{cat}$ ,  $R = \text{ca}$  and  $\alpha = 3$  when  $P = \text{catcatcatca}$ . The period can be computed in  $O(p)$  time [6]. Note that  $\text{nocc} \leq \text{occ} \leq (\alpha + 1)\text{nocc}$ .

## 2 An Overview of Our Non-Overlapping Indexing Framework

In this section, we present a high level description of our query algorithm with some key steps summarized as lemmas (long proofs are deferred to later sections). We maintain a suffix tree  $\text{ST}$  of  $\mathbb{T}$ , however all pattern matching tasks are performed using its cache-oblivious counterpart [4]. The structure in Theorem 2 is also maintained.

We say that the input pattern  $P$  is periodic if  $\text{period}(P) \leq |P|/2$  (equivalently  $\alpha \geq 2$ ), else we say  $P$  is **aperiodic** (i.e.,  $\alpha = 1$ ). The first step of our algorithm is to verify if  $P$  is periodic or not, and we rely on the result in Lemma 3. We handle both cases separately.

► **Lemma 3.** *Given a pattern  $P[1, p]$  which appears at least once in  $\mathbb{T}$ , we can find if  $P$  is periodic or not in  $O(p/B + \log_B n)$  I/Os using an  $O(n \log n)$  space structure. Also, it returns  $\text{period}(P)$  if  $P$  is periodic.*

### 2.1 Handling aperiodic case

When  $P$  is aperiodic,  $\text{occ} = \Theta(\text{nocc})$  and we answer queries using the structure in Theorem 2 as follows. First obtain all occurrences of  $P$  in their sorted order. Then, scan them in the ascending order and do the following: report the first occurrence and report any other occurrence iff it is not overlapping with the last reported occurrence. This step can be implemented in  $\text{occ}/B = \Theta(\text{nocc}/B)$  I/Os. Thus  $O(p/B + \log_B n + \text{nocc}/B)$  I/Os overall.

### 2.2 Handling periodic case

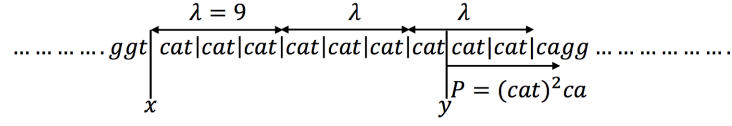
For periodic case, we start with the following simple observation.

► **Observation 4.** *If we list all the occurrences of  $P = Q^\alpha R$  in  $\mathbb{T}$  in the **ascending order**, we can see **clusters** of occurrences holding the following property: two consecutive occurrences*

1. *within a cluster, are exactly  $\text{period}(P)$  distance apart*
2. *not within a cluster cannot have an overlap of length  $\text{period}(P)$  or more.*

► **Lemma 5.** *The number of clusters, denoted by  $\pi$  is  $O(\text{nocc})$ .*

**Proof.** Two occurrences  $i, j$  not within the same cluster overlap only if  $i$  is the last occurrence in a cluster and  $j$  is the first occurrence within the next cluster (follows from Observation 4(2)). Clearly, only one of them can be a part of the final output. Therefore,  $\text{nocc} \geq \pi/2$ . ◀



■ **Figure 1** Here  $P = catcatca$ ,  $x$  is the cluster-head and  $y = x + 21$  is the cluster-tail. Then, the largest set of non-overlapping occurrences with the first occurrence included, and the first occurrence excluded are  $\{x, x + 9, x + 18\}$  and  $\{x + 3, x + 12, x + 21\}$ , respectively.

---

**Algorithm 1** Reports the largest set of non-overlapping occurrences of  $P$  in  $T$ .

---

- 1: report  $S_1$
  - 2: **for**  $(i = 2$  to  $\pi)$  **do**
  - 3:     **if** (the last reported occurrence and  $L'[i]$  are non-overlapping) **then** report  $S_i$
  - 4:     **else** report  $S_i^*$
  - 5: **end for**
- 

► **Definition 6.** An occurrence is a cluster-head (resp., cluster-tail) iff it is the first (resp., last) occurrence within a cluster. Also, let  $L'$  (resp.,  $L''$ ) be the list of all cluster heads (resp., tails) in their *ascending order*.

Observe that the distance between two consecutive non-overlapping occurrences within the same cluster, denoted by  $\lambda$  is  $\text{period}(P) \times \lceil p/\text{period}(P) \rceil$ . Let  $C_i$  be the  $i$ th leftmost cluster and  $S_i$  (resp.,  $S_i^*$ ) be the largest set of non-overlapping occurrences in  $C_i$  including (resp., excluding) the first occurrence  $L'[i]$  in  $C_i$ . Specifically (see Figure 1 for an illustration),

$$S_i = \{L'[i] + k\lambda \mid \text{for } k = 0, 1, 2, 3, \dots \text{ until } L'[i] + k\lambda \leq L''[i]\}$$

$$S_i^* = \{\text{period}(P) + L'[i] + k\lambda \mid \text{for } k = 0, 1, 2, 3, \dots \text{ until } (\text{period}(P) + L'[i] + k\lambda \leq L''[i])\}$$

Then, the final output can be generated by just examining  $L'$  and  $L''$  using the procedure in Algorithm 1. Correctness follows from Observation 4. In short, the periodic case can be handled in  $O(\text{nocc}/B)$  I/Os, given  $L'$  and  $L''$ . What remains to show is, how to obtain the arrays  $L'$  and  $L''$  in optimal I/Os and we rely on the following lemmas for this crucial step.

► **Lemma 7.** *By maintaining an  $O(n \log n)$  space structure, the array  $L''$  corresponding to any query pattern  $P[1, p]$  can be obtained in  $O(p/B + \log_B n + \pi/B)$  I/Os.*

► **Lemma 8.** *By maintaining an  $O(n \log n)$  space structure, the array  $L'$  corresponding to any query pattern  $P[1, p]$  can be obtained in  $O(p/B + \log_B n + \pi/B)$  I/Os.*

By combining all pieces together, we obtain  $O(n \log n)$  total space and  $p/B + \log_B n + \pi/B + \text{nocc}/B = O(p/B + \log_B n + \text{nocc}/B)$  query I/Os. This completes the proof of Theorem 1. The rest of this paper is dedicated to missing proofs.

### 3 Preliminaries for Missing Proofs

#### 3.1 Heavy Path Decomposition

We first categorize the nodes in  $ST$  into light and heavy. The root is light. For each internal node  $u$ , its heaviest child is the one with the maximum number of leaves, denoted by  $\text{size}(\cdot)$ , in its subtree, breaking ties arbitrarily. Therefore, the size of a light node is at most half of the size of its parent. Thus we have the following result.

► **Lemma 9** (Harel and Tarjan [10]). *The number of light nodes on any root to leaf path is at most  $(\log n)$ .*

► **Corollary 10.** *The sum of sub-tree sizes of all light nodes in ST is  $\leq n \log n$ .*

A heavy path is a downward path in the tree, starting from a light node with all other nodes on the path are heavy. Each heavy path ends at a unique leaf node. Also, each node intersect with exactly one heavy path. For brevity, we shall use the following terminologies: for any node  $u$  in ST, let

- $\text{hp\_root}(u)$  be the first light node on the path from  $u$  to root. Equivalently,  $\text{hp\_root}(u)$  is the root of the heavy path that intersects with  $u$ .
- $\text{hp\_leaf}(u) = \ell_j$ , where  $u$  and  $\ell_j$  are on the same heavy path. Note that  $\ell_j$  is unique for  $u$ .

### 3.2 Right-Maximally-Periodic Prefixes

► **Definition 11.** We call a substring  $T[i, i + l - 1]$  *right-maximally-periodic* iff  $T[i, i + l - 1]$  is periodic and  $T[i, i + l]$  is aperiodic.

► **Lemma 12.** *For a fixed suffix  $T[i, n]$ , let  $l_1, l_2, \dots, l_k$  be the length of all right-maximally-periodic prefixes in their ascending order and  $q_1, q_2, \dots, q_k$  be their respective periods. A  $p$ -long prefix of  $T[i, n]$  is periodic (with period  $q_j$ ) iff  $2 \times q_j \leq p \leq l_j$  for some  $j$ .*

► **Lemma 13.** *The number of right-maximally-periodic prefixes of  $T[i, n]$  is  $O(\log n)$ .*

**Proof.** From the definition of periodic and aperiodic,  $q_j \geq l_{j-1}$  and  $l_j \geq 2 \times q_j$ . Therefore,  $l_j \geq 2 \times l_{j-1}$  and  $l_k \geq 2^{k-1} l_1$ . Hence  $k \leq 1 + \log(n - i + 1)$ . ◀

### 3.3 1-Sided Sorted Range Reporting

We now prove two useful results.

► **Lemma 14.** *Let  $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$  be a set of  $m$  points in 2D. A 1-Sided range sorted range reporting query “ $r$ ” asks to return the points in  $S(r, -) = \{(x_i, y_j) \in S \mid x_i \leq r\}$  in the sorted order of their  $y$ -coordinates. The query can be answered in optimal  $O(1 + k/B)$  I/Os using an  $O(m)$  space data structure, where  $k$  is the size of  $S(r, -)$ .*

**Proof.** Without loss of generality, assume  $x_1 \leq x_2 \leq x_3 \leq \dots \leq x_m$ . Let  $A[1, m]$  be an array of length  $m$ , such that  $A[i] = x_i$ . We maintain a Cache-Oblivious B-tree [3] over  $A$ , so that for any query  $r$ , we can find  $k = \max\{i \mid A[i] \leq r\}$  in  $O(\log_B m)$  I/Os. Let  $D$  be an array of points in the ascending order of  $x$ -coordinates (i.e.,  $D[i] = (x_i, y_i)$ ) and  $D^j$  be an array of first  $j$ -points in  $D$  in the ascending order of  $y$ -coordinates. We explicitly maintain  $D^j$  for  $j = 1, 2, 4, 8, \dots, m$ . Total space is  $O(m)$ .

Now to answer a query  $r$ , first find  $k$  using a predecessor search query. Then, simply scan through the array  $D^{k'}$  (in the left to right order) and report only those points  $(x_i, y_j)$  with  $x_i \leq r$ , where  $k' = 2^{\lceil \log k \rceil}$ . I/Os required is  $k'/B = O(k/B)$ . ◀

► **Lemma 15.** *Let  $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$  be a set of  $m$  points in 2D. A 1-Sided range sorted range reporting query “ $r$ ” asks to return the points in  $S(r, +) = \{(x_i, y_j) \in S \mid x_i \geq r\}$  in the sorted order of their  $y$ -coordinates. The query can be answered in optimal  $O(1 + k/B)$  I/Os using an  $O(m)$  space data structure, where  $k$  is the size of  $S(r, +)$ .*

**Proof.** Maintain the data structure in Lemma 14 over the set  $S^* = \{(-x_i, y_j) \mid (x_i, y_j) \in S\}$ . When  $r$  comes as an input, obtain  $S^*(-r, -)$  in  $y$ -sorted order and report them in the same order after replacing each point  $(a, b)$  by  $(-a, b)$ . ◀

#### 4 Proof of Theorem 2

Our data structure is simple. For each light node  $w$  in the suffix tree, define a set  $H_w$  of two-dimensional points, where

$$H_w = \{(\delta(i), \text{SA}[i]) \mid \ell_i \text{ is under } w\}$$

Here  $\delta(i)$  is the string depth of the lowest common ancestor (lca) of  $\ell_i$  and  $\text{hp\_leaf}(w)$ . Note that  $|H_w| = \text{size}(w)$ . For each light node  $w$ , we maintain a 1-sided sorted range reporting structure (in Lemma 15) over the set  $H_w$ . The total space is  $O(n \log n)$  words (from Corollary 10).

To answer a query  $P$ , we first find the locus of  $P$  via searching in the cache-oblivious suffix tree in  $O(p/B + \log_B n)$  I/Os [4]. The remaining task is to report the set  $\{\text{SA}[i] \mid \ell_i \text{ is under } \text{locus}(P)\}$  with its elements sorted. Let  $w$  be the first light node on the path from  $\text{locus}(P)$  to the root of ST. Specifically,  $w = \text{hp\_root}(\text{locus}(P))$ . Then, the following holds.

$$\{\text{SA}[i] \mid \ell_i \text{ is under } \text{locus}(P)\} = \{\text{SA}[i] \mid \text{string depth of } \text{lca}(\ell_i, \text{hp\_leaf}(w)) \text{ is } \geq p\}$$

The remaining part of the query can be completed via a single 1-sided sorted range reporting query “ $p$ ” over the set of points in  $H_w$ . The total number of I/Os required is  $O(p/B + \log_B n + \text{occ}/B)$ .

#### 5 Proof of Lemma 3

The design of our data structure is straightforward from the discussion in Section 3.2. For each  $T[i, n]$ , we maintain the lengths and periods of all its right-maximally-periodic prefixes. The space required is  $O(n \log n)$  words (refer to Lemma 13).

When a pattern  $P$  comes, we first find an occurrence  $i$  of  $P$  in  $T$ . Then examine lengths of all right-maximally-periodic prefixes (and their periods) of  $T[i, n]$  and decide if  $P$  is periodic or not in  $(\log n)/B = O(\log_B n)$  I/Os. Also, retrieve its period if it is periodic.

#### 6 Proof of Lemma 7

We use the following observation [9]: a text position  $y$  is the rightmost occurrence of  $P$  within a cluster (i.e., cluster-tail) iff  $T[y, n]$  is prefixed by  $P = Q^\alpha R$ , but not by  $QP = Q^{1+\alpha} R$ . This means,  $L''$  is the sorted list of all elements in the following set of size  $\pi$  (see Figure 2).

$$\{\text{SA}[i] \mid i \in [\text{sp}(P), \text{ep}(P)] \wedge i \notin [\text{sp}(QP), \text{ep}(QP)]\}$$

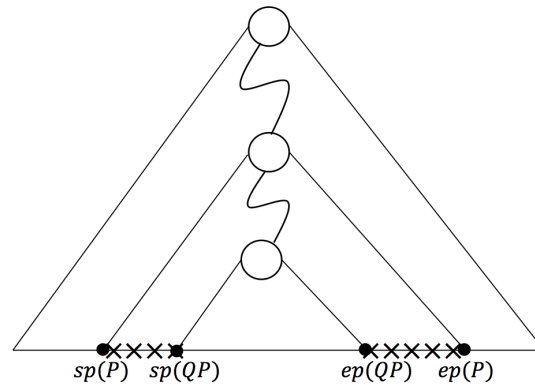
We consider the following two cases separately.

##### 6.1 Case 1: $\text{locus}(P)$ and $\text{locus}(QP)$ are on different heavy paths

Here  $\text{hp\_root}(\text{locus}(P)) \neq \text{hp\_root}(\text{locus}(QP))$  (we perform this check in  $O(1)$  I/Os). The following lemma is the key.

► **Lemma 16.** *When  $\text{locus}(P)$  and  $\text{locus}(QP)$  are on different heavy paths,  $\text{occ} = \Theta(\text{nocc})$ .*

**Proof.** Let  $u$  be the first node on the path from  $\text{locus}(QP)$  to root, such that  $\text{locus}(P)$  and the parent of  $u$  are on the same heavy path and  $u'$  be the heavy sibling of  $u$ . Then, clearly,  $\text{size}(\text{locus}(QP)) \leq \text{size}(u) \leq \text{size}(u') \leq \pi$  and  $\pi + \text{size}(\text{locus}(QP)) = \text{occ}$ . Therefore,  $\pi \geq \text{occ}/2$  and  $\pi \leq \text{nocc}$ , hence  $\text{occ} = \Theta(\text{nocc})$ . ◀



■ **Figure 2** Suffix tree with the region corresponding to  $L'$  highlighted.

In the light of the above lemma, when the query  $P$  falls in this case, we can in fact generate the final output directly instead of generating  $L''$  first and using it. First obtain all occurrences of  $P$  in the sorted order (using the structure in Theorem 2) and extract the largest set of non-overlapping occurrences from it by following the exact same procedure as in aperiodic case. I/Os required is  $p/B + \log_B n + \text{occ}/B = O(p/B + \log_B n + \text{nocc}/B)$ .

## 6.2 Case 2: $\text{locus}(P)$ and $\text{locus}(QP)$ are on the same heavy path

We start with two definitions and a crucial observation based on them.

► **Definition 17.** A pattern  $P$  is a power (or perfectly periodic) iff  $P = Q^\alpha$  for an integer  $\alpha \geq 2$ .

For example,  $aabaabaab$  is a power, whereas  $aabaaba$  is not.

► **Definition 18.** We call a node in the suffix tree *special* if it is the locus of at least one power. Note that a special node can be the locus of a pattern which is not a power.

► **Observation 19.** Let  $P$  be a periodic pattern with  $Q$  being its  $\text{period}(P)$ -long prefix. Then, among all nodes on the path from  $\text{locus}(QP)$  to  $\text{locus}(P)$ , excluding  $\text{locus}(QP)$ , exactly one node is special.

### 6.2.1 The Data Structure

For each light node  $w$  in the suffix tree, we maintain the following structure.

Let  $v_0 = w, v_1, v_2, v_3, \dots, v_h$  be the special nodes on the heavy path corresponding to  $w$  (in the ascending order of pre-order rank) and let  $v_{h+1} = \text{hp\_leaf}(w)$  if it is not special. Define sets  $G_w(v_j)$  for  $j = 0, 1, 2, \dots, h$ , such that  $G_w(v_h) = \{(\delta(i), \text{SA}[i]) \mid \ell_i \text{ is under } v_h\}$  and for  $j < h$ ,

$$G_w(v_j) = \{(\delta(i), \text{SA}[i]) \mid \ell_i \text{ is under } v_j, \text{ but not under } v_{j+1}\}$$

We then maintain the 1-sided sorted range reporting structures (in Lemma 14 and Lemma 15) over the set of points in each  $G_w(v_j)$ . The space required for a fixed  $w$  is  $O(\text{size}(w))$  words. Hence, the space over all lights nodes is  $O(n \log n)$ .

## 6.2.2 The Algorithm

When  $P$  is a power,  $L''$  can be obtained via a single 1-sided sorted range reporting query “ $(p+|Q|-1)$ ” on the structure in Lemma 14 over the set  $G_w(v_j)$ , where  $w = \text{hp\_root}(\text{locus}(P))$  and  $v_j = \text{locus}(P)$ .

For the other case, choose  $w = \text{hp\_root}(\text{locus}(P))$  and  $v_j = \text{locus}(Q^{\alpha+1})$ . Then obtain elements in the following two sets, in the sorted order of  $\text{SA}[\cdot]$  via 1-sided sorted range reported queries.

1.  $\{(\delta(i), \text{SA}[i]) \mid \ell_i \text{ is under } v_j, \text{ but not under } \text{locus}(QP)\}$  via a query “ $(p+|Q|-1)$ ” on the structure in Lemma 14 over  $G_w(v_j)$ .
2.  $\{(\delta(i), \text{SA}[i]) \mid \ell_i \text{ is under } \text{locus}(P), \text{ but not under } v_j\}$  via a 1-sided sorted range reporting query “ $p$ ” on the structure in Lemma 15 maintained over  $G_w(v_{j-1})$ .

By scanning both arrays in linear I/Os, specifically  $O(\pi/B)$  I/Os, we obtain  $L''$ .

## 7 Proof of Lemma 8

For  $L'$ , we use the following observation: for each cluster of  $P$  in  $\mathbb{T}$ , there is a corresponding cluster of  $\overleftarrow{P}$  in  $\overleftarrow{\mathbb{T}}$ . Here  $\overleftarrow{\mathbb{T}}$  (resp.,  $\overleftarrow{P}$ ) is the reverse of  $\mathbb{T}$  (resp.,  $P$ ). Then, we have the following simple observation.

► **Observation 20.** *Suppose  $z$  is the last occurrence of  $\overleftarrow{P}$  within a cluster of  $\overleftarrow{P}$  in  $\overleftarrow{\mathbb{T}}$ , then  $(n+2-p-z)$  is the first occurrence of  $P$  within the corresponding cluster of  $P$  in  $\mathbb{T}$ .*

Therefore, we simply construct and maintain our previous data structure for computing  $L''$ , but on  $\overleftarrow{\mathbb{T}}$ . When  $P$  comes as input to the original problem, we find  $L''$  corresponding to  $\overleftarrow{P}$  in  $\overleftarrow{\mathbb{T}}$ . Then, simply report  $(n+2-p-L''[i])$ 's in the descending order of  $i$ . Note that we need to maintain the suffix tree (and its cache-oblivious version) of  $\overleftarrow{\mathbb{T}}$  as well. However, the total space is still  $O(n \log n)$ .

## 8 Concluding Remarks

We present the first I/O optimal data structure for the non-overlapping indexing problem in both cache-aware and cache-oblivious models of computation. We remark that by combining our framework with standard techniques, we can design an I/O optimal,  $O(n \log^2 n)$  space structure for the *range* non-overlapping problem in the cache-aware model. However, it is not clear if the same is possible in cache-oblivious model. An interesting question is: *Can we improve the space (yet, keeping the query I/Os optimal), at least in the cache-aware model?*

---

## References

- 1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
- 2 Alberto Apostolico and Franco P Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.
- 3 Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005. doi:10.1137/S0097539701389956.
- 4 Gerth Stølting Brodal and Rolf Fagerberg. Cache-oblivious string dictionaries. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 581–590. ACM Press, 2006. URL: <http://dl.acm.org/citation.cfm?id=1109557.1109621>.

- 5 Hagai Cohen and Ely Porat. Range non-overlapping indexing. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 1044–1053. Springer, 2009. doi:10.1007/978-3-642-10631-6\_105.
- 6 Maxime Crochemore. String-matching on ordered alphabets. *Theoretical Computer Science*, 92(1):33–47, 1992.
- 7 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.
- 8 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, 2012. doi:10.1145/2071379.2071383.
- 9 Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. Succinct non-overlapping indexing. In *Annual Symposium on Combinatorial Pattern Matching*, pages 185–195. Springer, 2015.
- 10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 11 Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. Range non-overlapping indexing and successive list indexing. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, volume 4619 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2007. doi:10.1007/978-3-540-73951-7\_54.
- 12 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 13 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In Fedor V. Fomin and Petteri Kaski, editors, *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings*, volume 7357 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2012. doi:10.1007/978-3-642-31155-0\_24.
- 14 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.
- 15 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.





# Faster Online Elastic Degenerate String Matching

**Kotaro Aoyama**


Department of Electrical Engineering and Computer Science, Kyushu University, Japan  
kotaro.aoyama@inf.kyushu-u.ac.jp

**Yuto Nakashima**

Department of Informatics, Kyushu University, Japan  
yuto.nakashima@inf.kyushu-u.ac.jp

**Tomohiro I**

Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan  
tomohiro@ai.kyutech.ac.jp


 <https://orcid.org/0000-0001-9106-6192>

**Shunsuke Inenaga**

Department of Informatics, Kyushu University, Japan  
inenaga@inf.kyushu-u.ac.jp

**Hideo Bannai**

Department of Informatics, Kyushu University, Japan  
bannai@inf.kyushu-u.ac.jp

 <https://orcid.org/0000-0002-6856-5185>

**Masayuki Takeda**

Department of Informatics, Kyushu University, Japan  
takeda@inf.kyushu-u.ac.jp

---

## Abstract

An *Elastic-Degenerate String* [Iliopoulos et al., LATA 2017] is a sequence of sets of strings, which was recently proposed as a way to model a set of similar sequences. We give an online algorithm for the Elastic-Degenerate String Matching (EDSM) problem that runs in  $O(nm\sqrt{m\log m} + N)$  time and  $O(m)$  working space, where  $n$  is the number of elastic degenerate segments of the text,  $N$  is the total length of all strings in the text, and  $m$  is the length of the pattern. This improves the previous algorithm by Grossi et al. [CPM 2017] that runs in  $O(nm^2 + N)$  time.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** elastic degenerate pattern matching, boolean convolution

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.9

**Acknowledgements** This work was supported by JSPS KAKENHI Grant Numbers JP17H06923 (YN), JP17H01697 (SI), JP16H02783 (HB), and JP25240003 (MT).

## 1 Introduction

The *degenerate* string matching problem [7, 1] is a variant of the string matching problem when a position in the text may contain uncertainties; the text string, called a degenerate string, can be regarded as a string over an extended alphabet that consists of non-empty subsets of the original alphabet. A string matches a degenerate string if the subset at each position of the degenerate string contains the character of the pattern at the corresponding position. The *Elastic Degenerate String Matching* (EDSM) problem, first proposed by



© Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda;

licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 9; pp. 9:1–9:10

Leibniz International Proceedings in Informatics



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

Iliopoulos et al. [9], is a further generalization of this setting, where substrings of the text may contain uncertainties; the text string, called an elastic degenerate string (ED string), can be regarded as a sequence of non-empty sets of strings. A string matches an ED string if it is a substring of a string that can be obtained by taking a string from each position of the ED string, and concatenating them. The motivation behind these problems is in bioinformatics, where multiple genomic sequences from individuals of the same species can be obtained. Recently, rather than than considering a single reference sequence, it is increasingly more common to consider the multiple sequences [4], and ED strings is one way to model them.

Iliopoulos et al. [9], gave an (offline) algorithm which runs in  $O(N + \alpha\gamma nm)$  time, where  $m$  is the length of the given pattern,  $n$  and  $N$  are respectively the length and total size of the given elastic-degenerate text,  $\alpha$  and  $\gamma$  respectively represent the maximum number of strings in any elastic degenerate symbols and the largest number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text.

The online version of the problem was considered by Grossi et al. [8], where they gave an algorithm which runs in  $O(nm^2 + N)$  time. They also give an algorithm which runs in  $O(N \lceil \frac{m}{w} \rceil)$  time, where  $w$  is the computer word size. Furthermore, Bernardini et al. [3] consider the EDSM problem with errors, and presented an on-line algorithm that runs in  $O((k+1)^2 mG + (k+1)N)$  time and  $O(m)$  space, where  $k$  is the number of allowed errors (insertion/deletion/substitution), and  $n \leq G \leq N$  is the total size of the sets of subsets (i.e., the total number of strings) in the ED string. They also present a faster  $O((k+1)(mG + N))$  time and  $O(m)$  space algorithm when considering only substitution errors.

In this paper, we improve the first algorithm by Grossi et al., and give a faster on-line algorithm for EDSM that runs in  $O(nm\sqrt{m \log m} + N)$  time and  $O(m)$  working space, assuming that the alphabet size is constant, as in previous work. For the space complexity, we will also assume that the strings at each position of the ED string are given in lexicographically sorted order. We note that the algorithm can be considered better than that of Bernardini et al. (with  $k = 0$ ), when each subset in the ED string may contain many strings, which could be the case as more sequences from many individuals become increasingly available.

## 2 Preliminaries

### 2.1 Strings

For any set  $\Sigma$ , an element of  $\Sigma^*$  is called a string over alphabet  $\Sigma$ . We will assume that  $|\Sigma|$  is constant. The empty string is denoted by  $\varepsilon$ . For any string  $w \in \Sigma^*$ , if  $w = xyz$  for (possibly empty) strings  $x, y, z$ , then,  $x, y$ , and  $z$  are respectively called a prefix, substring, suffix of  $w$ . The length of  $w$  is denoted by  $|w|$ . Let  $Pref(w), Sub(w), Suf(w)$  respectively denote the set of prefixes, substrings, and suffixes of  $w$ . For any integers  $1 \leq i \leq j \leq |w|$ ,  $w[i]$  denotes the  $i$ th symbol of  $w$ , i.e.,  $w = w[1] \cdots w[|w|]$ , and  $w[i..j] = w[i] \cdots w[j]$  denotes a substring of  $w$  that starts at position  $i$  and ends at position  $j$ . For convenience, let  $w[i..j] = \varepsilon$  when  $i > j$ ,  $w[i..j] = w[1..j]$  when  $i < 1$ , and  $w[i..j] = w[i..|w|]$  when  $j > |w|$ .

If  $w = xu = vx$  for non-empty strings  $u, v$  and possibly empty  $x$ , then  $x$  is called a *border* of  $w$ . The *border array* of  $w$  is an array  $B[1..|w|]$  of integers such that  $B[i]$  stores the length of the longest border of  $w[1..i]$ . It is well known that the border array of  $w$  can be computed in linear time in an on-line fashion. Also, given the border array of  $w$ , the length of all borders of  $w$  can be computed in linear time.

## 2.2 Elastic-Degenerate Strings

Let  $\tilde{\Sigma}$  denote the set of all finite non-empty subsets of  $\Sigma^*$  excluding  $\{\varepsilon\}$ . An Elastic-Degenerate string, or ED string, over alphabet  $\Sigma$ , is a string over  $\tilde{\Sigma}$ , i.e., an ED string is an element of  $\tilde{\Sigma}^*$ . Below is an example of an ED string over  $\Sigma = \{A, C, T\}$ .

► **Example 1** (Elastic-Degenerate String).

$$\tilde{T} = \left\{ \begin{array}{c} A, \\ C \end{array} \right\} \cdot \left\{ \begin{array}{c} C, \\ CA, \\ TACA \end{array} \right\} \cdot \left\{ \begin{array}{c} \varepsilon, \\ AC, \\ C \end{array} \right\} \cdot \left\{ \begin{array}{c} AT, \\ C \end{array} \right\}$$

Let  $\tilde{T}$  denote an ED string of length  $n$ , i.e.  $|\tilde{T}| = n$ . We assume that for any  $1 \leq i \leq n$ , the set  $\tilde{T}[i]$  is implemented as an array and can be accessed by an index, i.e.,  $\tilde{T}[i] = \{\tilde{T}[i][k] \mid k = 1, \dots, |\tilde{T}[i]|\}$ . We will also make the assumption that the strings in  $\tilde{T}[i]$ , i.e.,  $\tilde{T}[i][1], \dots, \tilde{T}[i][|\tilde{T}[i]|]$ , are sorted in lexicographic order. For any  $\tilde{c} \in \tilde{\Sigma}$ ,  $\|\tilde{c}\|$  denotes the total length of all strings in  $\tilde{c}$ , and for any ED string  $\tilde{T}$ ,  $\|\tilde{T}\|$  denotes the total length of all strings in all  $\tilde{T}[i]$  for  $i = 1, \dots, |\tilde{T}|$ , i.e.,  $\|\tilde{c}\| = \sum_{s \in \tilde{c}} |s|$  and  $\|\tilde{T}\| = \sum_{i=1}^n \|\tilde{T}[i]\|$ . We note that Grossi et al. define  $|\varepsilon| = 1$  when computing  $\|\tilde{T}\|$ , but this only increases the value of  $\|\tilde{T}\|$  by at most  $n$  (which is less than  $\|\tilde{T}\|$ , in our definition, since  $\|\tilde{c}\| \geq 1$  for any  $\tilde{c} \in \tilde{\Sigma}$ ) and thus does not affect the asymptotic complexities.

An ED string  $\tilde{T}$  can be thought of as a representation of the set of strings  $\mathcal{A}(\tilde{T}) = \tilde{T}[1] \times \dots \times \tilde{T}[n]$ , where  $A \times B = \{xy \mid x \in A, y \in B\}$  for any sets of strings  $A$  and  $B$ . The string in Example 1 can be viewed as a representation of a of  $2 \times 3 \times 3 \times 2 = 36$  strings.

For any ED string  $\tilde{T}$  and string  $P$ , we say that  $P$  matches  $\tilde{T}$  if

1.  $|\tilde{T}| = 1$  and  $P$  is a substring of some  $s \in \tilde{T}[1]$ , or,
2.  $|\tilde{T}| > 1$  and  $P = p_1 \dots p_{|\tilde{T}|}$  where  $p_1$  is a suffix of some string in  $\tilde{T}[1]$ ,  $p_{|\tilde{T}|}$  is a prefix of some string in  $\tilde{T}[|\tilde{T}|]$ , and  $p_i \in \tilde{T}[i]$  for all  $1 < i < |\tilde{T}|$ .

We say that an occurrence of  $P$  in  $\tilde{T}$  starts at position  $i$  and ends at position  $j$ , if  $P$  matches  $\tilde{T}[i..j]$ . Below is the problem we solve.

► **Problem 2** (Elastic-Degenerate String Matching (EDSM) [8]). *Given a string  $P$  of length  $m$ , and an ED string  $\tilde{T}$  of length  $n$  and size  $N \geq m$ , output all positions  $j$  in  $\tilde{T}$  where at least one occurrence of  $P$  ends.*

We will measure space complexity in terms of working space, and exclude the input pattern and ED string, which we assume to be stored in read-only memory, as well as the space for output, which is write-only.

## 3 Tools

### 3.1 Suffix Trees

A *suffix tree* [11]  $ST(w)$  of a string  $w$  is a compacted trie of all suffixes of  $w\$$ , where  $\$$  is a special symbol that does not occur in  $w$ . In other words, the suffix tree of  $w$  is a rooted tree where each edge has string labels, where all and only suffixes of  $w\$$  are represented in the concatenation of all labels on a root to leaf path. Furthermore, each internal node has at least two outgoing edges where the first character of the label of the outgoing edges are distinct. We assume that each leaf is labeled by an integer that represents the starting position of the suffix that corresponds to the root to leaf path. Although the total length of all labels in a suffix tree is not  $O(|w|)$ , each label can be represented in constant space,

i.e., two integers representing positions in  $w$ , since any edge label is a substring of  $w$ . It is well known that the suffix tree of  $w$  can be constructed in  $O(|w|)$  time for constant size alphabets [11] (as well as integer alphabets [6]).

For any node  $u$  in the suffix tree, let  $str(u)$  denote the concatenation of all edge labels on the root to  $u$  path, and let  $len(u) = |str(u)|$ . Let  $parent(u)$  denote the parent of  $u$ , and  $anc(u)$ ,  $desc(u)$  respectively, the set of nodes in the suffix tree that are ancestors and descendants of  $u$ , including  $u$  itself. A position in the suffix tree can be represented a pair  $(u, d)$ , where  $u$  is a node and  $d \geq 0$  is an integer such that  $d \leq |str(u)|$  and  $d > |str(parent(u))|$  (if  $parent(u)$  exists). For any substring  $s$  of  $w$ , the *locus* of  $s$  in  $ST(w)$  is a position  $(u, d)$  in  $ST(w)$  where  $s = str(u)[1..d]$ . For any string  $s$ , the locus of the longest prefix  $s'$  of  $s$  that is a substring of  $w$  can be obtained in  $O(|s'|)$  time by simply traversing the suffix tree from the root.

We denote by  $L(u)$ , the set of integers that are labels on the leaf nodes that are descendants of  $u$ . Let  $Occ(w, s)$  denote the set of occurrences of  $s$  in  $w$ , i.e.  $Occ(w, s) = \{i \mid w[i..i + |s| - 1] = s\}$ . Suffix trees can be used to compute this set efficiently, since  $Occ(w, s) = L(v_s)$ , where  $(v_s, |s|)$  is the locus of  $s$  in  $ST(w)$ , if it exists, and  $Occ(w, s) = \emptyset$  otherwise.

► **Lemma 3** ([11]). *Given the suffix tree  $ST(w)$  of string  $w$ ,  $Occ(w, s)$  for any string  $s$  can be computed in  $O(|s| + Occ(w, s))$  time.*

### 3.2 Sum Set and FFT

For any sets of integers  $A, B$ , we denote by  $A \oplus B = \{a + b \mid a \in A, b \in B\}$  the sum set of  $A$  and  $B$ .

► **Lemma 4** (Efficient Computation of SumSet). *For any integer  $m$  and sets of integers  $A, B \subseteq [1..m]$ ,  $A \oplus B$  can be computed in  $O(m \log m)$  time and  $O(m)$  space.*

**Proof.** For any set  $X \subseteq [1..m]$ , let  $I(X)$  denote an array of Boolean (**true** or **false**) values where  $I(X)[i] = \mathbf{true}$  if and only if  $i \in X$ . For any  $1 \leq i \leq m$ ,  $I(A \oplus B)$  can be computed by the Boolean convolution:

$$I(A \oplus B)[i] = \bigvee_{j=1}^m (I(A)[i - j] \wedge I(B)[j]).$$

It is well known that these values can be done in total  $O(m \log m)$  time and  $O(m)$  space for all  $1 \leq i \leq m$ , using the Fast Fourier Transform [5]. The set  $A \oplus B$  can easily be obtained in  $O(m)$  time by scanning  $I(A \oplus B)[1..m]$ . ◀

## 4 Algorithm

We first give an overview of the algorithm of Grossi et al. [8] which our algorithm is based on, and then describe our improvements to it.

### 4.1 Overview of Algorithm

The algorithm is on-line, i.e., for each ED string position  $i = 1, \dots, n$ , it outputs  $i$  as an answer if there is an occurrence of  $P$  that ends at  $i$ , i.e.,  $P$  matches  $\tilde{T}[l..i]$  for some  $l \leq i$ . Although the total number of strings in  $\mathcal{A}(\tilde{T}[1..i])$ , and thus the total number of occurrences of  $P$  in all these strings, can be exponential in  $i$ , the problem can be solved efficiently since

we only consider whether there is an end of an occurrence of  $P$  in position  $i$  of the ED string. To this end, the key of the algorithm is to compute for each  $i$ , the set

$$\mathcal{S}_i^{\leq} = \{j \mid 1 < j \leq m, \exists s \in \mathcal{A}(\tilde{T}[1..i]) \text{ s.t. } P[1..j-1] \text{ is a suffix of } s\}$$

which represents the positions  $j$  in  $P$  such that  $P[1..j-1]$  occurs as a suffix of some string in  $\mathcal{A}(\tilde{T}[1..i])$ . In other words, the set corresponds to potential positions  $j$  in  $P$  that can result in a match later, if  $P[j..m]$  is a prefix of some string in  $\mathcal{A}(\tilde{T}[i+1..n])$ . For each  $i$ , the computation performs the following steps:

1. Determine whether  $P$  matches  $\tilde{T}[i]$ .
2. If  $i > 1$ , determine whether  $P$  matches  $\tilde{T}[l..i]$  for some  $l < i$ .
3. Compute  $\mathcal{S}_i^{\leq}$ .

Position  $i$  is output as an ending position of an occurrence of  $P$ , if and only if a match is found in either Step 1 or 2. We basically follow previous work for computing Steps 1 and 2, but consider the space usage. Our main contribution is the improvement of Step 3. We note that the set  $\mathcal{S}_i^{\leq}$ , or more generally any subset of  $\{1, \dots, m\}$  can be represented as a bit array of size  $m$ , and determining membership, as well as adding/deleting elements, can be done in  $O(1)$  time. Also, set union can be performed in  $O(m)$  time.

## 4.2 Computing Step 1

In Step 1, we simply determine whether  $P$  is a substring of some  $s \in \tilde{T}[i]$ .

► **Lemma 5.** *Step 1 can be computed in total of  $O(m + \|\tilde{T}\|)$  time for all  $1 \leq i \leq n$ , using  $O(m)$  space.*

**Proof.** For Step 1, we can use a linear time pattern matching algorithm such as KMP [10]. Since the pattern does not change, the preprocessing of the pattern is done once in  $O(m)$  time. The matching can be done in  $O(\|\tilde{T}\|)$  time and  $O(m)$  space. ◀

## 4.3 Computing Step 2

Steps 2 (as well as Step 3) is computed using the suffix tree  $ST(P)$  of  $P$ , and also uses  $\mathcal{S}_{i-1}^{\leq}$ .

► **Lemma 6.** *Given  $\mathcal{S}_{i-1}^{\leq}$ , Step 2 can be computed in total of  $O(m + \|\tilde{T}\|)$  time for all  $1 \leq i \leq n$ , using  $O(m)$  space.*

**Proof.** We first construct the suffix tree  $ST(P)$  of  $P$ , which takes  $O(m)$  time and space. Since, by definition, a value  $j \in \mathcal{S}_{i-1}^{\leq}$  if and only if  $j > 1$  and  $P[1..j-1]$  is a suffix of some  $s \in \mathcal{A}(\tilde{T}[1..i-1])$ , we have, as observed previously, an occurrence of  $P$  that ends at position  $i$  if and only if there exist  $j \in \mathcal{S}_{i-1}^{\leq}$  and  $t \in \tilde{T}[i]$  such that  $P[j..m]$  is a prefix of  $t$ .

This can be checked as follows: For each string  $t \in \tilde{T}[i]$ , traverse the suffix tree from the root with  $t$ . We will detect such an occurrence of  $P$ , if, at any point in the traversal, we reach a position corresponding to a suffix  $P[j..m]$  for some  $j \in \mathcal{S}_{i-1}^{\leq}$ , i.e., when we are at the locus  $(u, d)$  of a prefix  $t' = t[1..d]$  of  $t$  during the traversal, either (1)  $u$  is a leaf that corresponds to the suffix  $P[j..m]$  for some  $j \in \mathcal{S}_{i-1}^{\leq}$ , and  $d = m - j + 1$ , or, (2)  $u$  has an outgoing edge labeled by  $\$$  that leads to a leaf that corresponds to the suffix  $P[j..m]$  for some  $j \in \mathcal{S}_{i-1}^{\leq}$ . Since the traversal can be done in  $O(|t|)$  time for each  $t$ , the total time for all  $1 \leq i \leq n$  is  $O(\|\tilde{T}\|)$ . ◀

#### 4.4 Computing Step 3

To compute  $\mathcal{S}_i^{\leq}$ , we will compute the two sets:

$$\begin{aligned}\mathcal{S}_i^{\bar{=}} &= \{j \mid 1 < j \leq m, \exists s \in \tilde{T}[i] \text{ s.t. } P[1..j-1] \text{ is a suffix of } s\} \\ \mathcal{S}_i^{\leq} &= \{j + |t| : P[j..j + |t| - 1] = t, j \in \mathcal{S}_{i-1}^{\leq}, t \in \tilde{T}[i], j + |t| \leq m\}.\end{aligned}$$

Then, it is clear that  $\mathcal{S}_i^{\leq} = \mathcal{S}_i^{\bar{=}} \cup \mathcal{S}_i^{\leq}$ .

##### 4.4.1 Computing $\mathcal{S}_i^{\bar{=}}$

We first describe how to compute  $\mathcal{S}_i^{\bar{=}}$ .

► **Lemma 7.**  $\mathcal{S}_i^{\bar{=}}$  can be computed in total of  $O(m + \|\tilde{T}\|)$  time for all  $1 \leq i \leq n$ , using  $O(m)$  space.

**Proof.** Consider the string  $P\#\tilde{T}[i][k]$  for each  $k = 1, \dots, |\tilde{T}[i]|$ , where  $\#$  is a character that does not occur in  $P$  or  $\tilde{T}[i][k]$ . Then, it is easy to see that

$$\mathcal{S}_i^{\bar{=}} = \{|b| + 1 \mid 1 \leq b < m, b \text{ is a proper border of } P\#\tilde{T}[i][k]\}.$$

Since the border array of a string can be computed in an on-line fashion and in linear time [10], the border array of  $P\#$  can be computed in  $O(m)$  time once. Furthermore, since the last element of the border array of  $P\#$  is 0, the remaining elements of the border array of  $P\#\tilde{T}[i][k]$ , and thus, the lengths of all borders of  $P\#\tilde{T}[i][k]$  can be computed in  $O(|\tilde{T}[i][k]|)$  time for any  $1 \leq k \leq |\tilde{T}[i]|$ . Thus, the total time for computing  $\mathcal{S}_i^{\bar{=}}$  is  $O(m + \|\tilde{T}\|)$ .

We note that the above description is slightly different from that of Grossi et al. [8], where they describe the computation by computing the border array of the string

$$X = P\#\tilde{T}[i][1]\#_2 \cdots \#_{|\tilde{T}[i]|}\tilde{T}[i][|\tilde{T}[i]|].$$

Although Grossi et al. mention that the time and space complexities for the preprocessing (for computing the border array of  $P$ ) are  $O(m)$ , they do not explicitly mention the space complexity of their matching algorithm. A naive implementation of their description would take  $O(m + \max\{|\tilde{T}[i]| \mid 1 \leq i \leq n\})$  extra space for computing the border array of  $X$ , but this can easily be reduced to  $O(m)$  extra space, since (1) we can compute the borders separately for each  $\tilde{T}[i][k]$  as described above, and (2)  $P\#\tilde{T}[i][k]$  can be replaced with  $P\#Y$ , where  $Y = \tilde{T}[i][k][l - m + 2..l]$  and  $l = |\tilde{T}[i][k]|$ , to obtain the same result. ◀

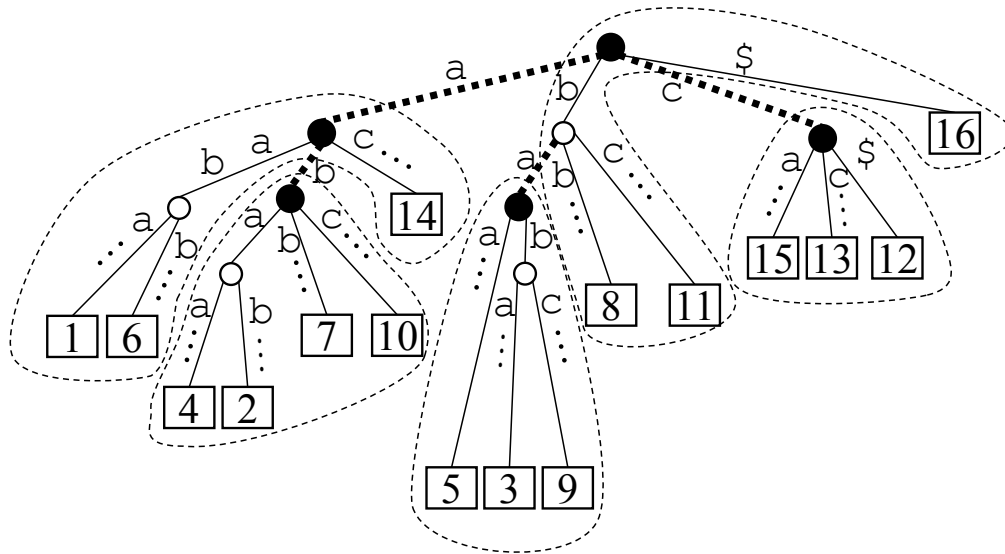
##### 4.4.2 Computing $\mathcal{S}_i^{\leq}$

We first describe how Grossi et al. compute  $\mathcal{S}_i^{\leq}$ . Basically, their algorithm is a fairly straightforward approach that uses  $ST(P)$ . For each  $t \in \tilde{T}[i]$ , compute the set  $\{t\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq})$ . Then,  $\mathcal{S}_i^{\leq}$  is the union of this set for all  $t \in \tilde{T}[i]$ , i.e.,

$$\mathcal{S}_i^{\leq} = \bigcup_{t \in \tilde{T}[i]} \left( \{t\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq}) \right). \quad (1)$$

By Lemma 3, each  $\{t\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq})$  can be computed in  $O(|t| + |Occ(P, t)|)$  time, and thus the total time is  $O(\sum_{t \in \tilde{T}[i]} (|t| + |Occ(P, t)|)) = O(\|\tilde{T}[i]\| + \sum_{t \in \tilde{T}[i]} |Occ(P, t)|)$ . Since all strings in  $\tilde{T}[i]$  are distinct,  $\sum_{t \in \tilde{T}[i]} |Occ(P, t)| = O(m^2)$ , thus, giving an algorithm that computes  $\mathcal{S}_i^{\leq}$  in  $O(\|\tilde{T}[i]\| + m^2)$  time.

Next, we describe how to improve the running time. Our main idea is: rather than compute the sum set independently for each element  $t \in \tilde{T}[i]$ , we appropriately group together elements in  $\tilde{T}[i]$  so that the efficient sum set computation of Lemma 4 can be used.



■ **Figure 1** Example of a partition of  $ST(P)$  for  $P = aababaabbabccac$  with  $\tau = 4$ . There are 25 nodes (including leaves) in total. The black nodes represent the root of each component, and the dotted edges represent boundary edges. By construction, each component contains at least  $\tau = 4$  nodes, so there are  $5 \leq 25/4$  components, and any sub-component rooted at a node that is not a root of a component contains less than  $\tau = 4$  nodes.

► **Lemma 8.**  $S_i^<$  can be computed in  $O(\|\tilde{T}[i]\| + m\sqrt{m \log m})$  time using  $O(m)$  working space.

**Proof.** To improve the running time, we first partition  $ST(P)$  into subtrees as follows, similar to the micro-macro decomposition [2].

Let  $\tau > 1$  be a parameter that will be chosen later. Define the weight  $W(v)$  of a node  $v$  as

$$W(v) = \begin{cases} 1 & v \text{ is a leaf} \\ W'(v) & W'(v) < \tau \\ 0 & W'(v) \geq \tau \text{ or } v \text{ is the root.} \end{cases}$$

where  $W'(v) = 1 + \sum_{u \in \text{chldr}(v)} W(u)$ . The tree is partitioned into subtrees so that all nodes  $v$  such that  $W(v) = 0$  are roots of the subtrees that comprise the partition, i.e., each incoming edge to a node  $v$  with  $W(v) = 0$  is a boundary of the partition. Such an edge will be called a *boundary edge*. For all nodes  $v$ , it is clear that  $W(v) = 0$  — and thus the partition, can be computed in linear time by a post-order traversal on  $ST(P)$ . We will call each such subtree in the partition a *component*. For any node  $v$ , we denote by  $C(v)$ , the set of nodes that are descendants of  $v$ , including  $v$  itself, and are in the same component as  $v$ . Also, let  $Cr(P)$  denote the set of all roots of components of  $ST(P)$ . Figure 1 shows an example of a partition. The important properties of such a partition are:

1. The total number of nodes (including leaves) contained in each component is  $\Omega(\tau)$ .
2. The number of components is bounded by  $O(m/\tau)$ .
3. For any node  $u$  that is not a root of a component,  $|C(u)| = O(\tau)$ .

Now, since  $Occ(P, t) = L(v_t)$ , where  $(v_t, |t|)$  is the locus of  $t$  in  $ST(P)$ , we can rewrite Equation (1) as follows, by partitioning  $L(v_t)$  according to the component that each leaf belongs to:

$$\begin{aligned} & \bigcup_{t \in \tilde{T}[i]} \left( \{|t|\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq}) \right) \\ = & \left( \bigcup_{t \in \tilde{T}[i], v_t \notin Cr(P)} \left( \{|t|\} \oplus (L(v_t) \cap C(v_t) \cap \mathcal{S}_{i-1}^{\leq}) \right) \right) \cup \end{aligned} \quad (2)$$

$$\left( \bigcup_{t \in \tilde{T}[i]} \bigcup_{v \in desc(v_t) \cap Cr(P)} \left( \{|t|\} \oplus (L(v) \cap C(v) \cap \mathcal{S}_{i-1}^{\leq}) \right) \right) \quad (3)$$

Furthermore, by grouping together all  $t$  that correspond to ancestors of each component when computing the sum set, we can rewrite Term (3) as follows:

$$\bigcup_{v \in Cr(P)} \left( \{|t| : t \in \tilde{T}[i] \cap v_t \in anc(v)\} \oplus (L(v) \cap C(v) \cap \mathcal{S}_{i-1}^{\leq}) \right) \quad (4)$$

First, we consider how to compute Term (2). For any  $t \in \tilde{T}[i]$ , its locus  $(v_t, |t|)$  in  $ST(P)$  can be computed in  $O(|t|)$  time, if it exists (we can simply ignore any  $t$  that does not occur in  $P$ ). Since we only consider  $t$  such that  $v_t$  is not a root of a component,  $|C(v_t)| = O(\tau)$  (Property 3). Then, all elements in  $L(v_t) \cap C(v_t) \cap \mathcal{S}_{i-1}^{\leq}$  can be obtained by a simple traversal on  $C(v_t)$ , and thus  $\{|t|\} \oplus (L(v_t) \cap C(v_t) \cap \mathcal{S}_{i-1}^{\leq})$  can be obtained in  $O(\tau)$  time. Note that this can also be bounded by the size of the subtree of  $ST(P)$  rooted at  $v_t$ . Thus, the total time for this traversal for all  $t \in \tilde{T}[i]$  is  $O(\sum_{t \in \tilde{T}[i]} \min\{\tau, |T_{v_t}|\})$ , where  $|T_{v_t}|$  denotes the size of the subtree of  $ST(P)$  rooted at  $v_t$ . Now, let  $X_S = \{t \mid t \in \tilde{T}[i], |t| \leq \tau\}$ ,  $X_L = \tilde{T}[i] \setminus X_S = \{t \mid t \in \tilde{T}[i], |t| > \tau\}$ , i.e.,  $X_S$  is the set of strings in  $\tilde{T}[i]$  shorter than or equal to  $\tau$ , and  $X_L$  is the set of strings in  $\tilde{T}[i]$  longer than  $\tau$ . Then,

$$\begin{aligned} \sum_{t \in \tilde{T}[i]} \min\{\tau, |T_{v_t}|\} &= \sum_{t \in X_S} \min\{\tau, |T_{v_t}|\} + \sum_{t \in X_L} \min\{\tau, |T_{v_t}|\} \\ &\leq \sum_{t \in X_S} |T_{v_t}| + \sum_{t \in X_L} \tau \\ &= \sum_{\ell=1}^{\tau} \sum_{t \in X_S, |t|=\ell} |T_{v_t}| + \sum_{t \in X_L} \tau \\ &= O(\tau m + \|\tilde{T}[i]\|). \end{aligned}$$

Here, the last inequality uses  $\sum_{t \in X_S, |t|=\ell} |T_{v_t}| = O(m)$ , which is true because all substrings in  $X_S$  are distinct, implying that all subtrees rooted at a given depth  $\ell$  of the suffix tree are disjoint and therefore their total size is  $O(m)$ . Also,  $\sum_{t \in X_L} \tau < \sum_{t \in X_L} |t| \leq \|\tilde{T}[i]\|$ .

The total time for computing Term (2) is therefore  $O(\tau m + \|\tilde{T}[i]\|)$ .

Next, we consider how to compute Term (4). Notice that for any  $v \in Cr(P)$ , the size of set  $\{|t| : t \in \tilde{T}[i] \cap v_t \in anc(v)\}$  is  $O(m)$ , since  $len(v) \leq m + 1$ , and can be obtained in  $O(m)$  time provided that all loci of strings in  $\tilde{T}[i]$  are marked on  $ST(P)$ . Also,  $L(v) \cap C(v) \cap \mathcal{S}_{i-1}^{\leq}$  can also be computed in  $O(m)$  time. Since the sets can be computed in  $O(m)$  time, the sum set can be computed in  $O(m \log m)$  time using Lemma 4. The total time for all components is therefore  $O(\frac{m}{\tau} m \log m)$  (Property 2).

From the above arguments, the total time for computing Equation (1) is  $O(\|\tilde{T}[i]\| + \tau m + \frac{m^2}{\tau} \log m)$ . By choosing  $\tau = \sqrt{m \log m}$ , we obtain  $O(\|\tilde{T}[i]\| + m\sqrt{m \log m})$ .

Concerning the space complexity, it is easy to implement the above algorithm in  $O(m + \|\tilde{T}[i]\|)$  space. The term  $\|\tilde{T}[i]\|$  exists because in the above description, we assumed that we had marked the locus of each  $t \in \tilde{T}[i]$  on the suffix tree. If we assume that the strings



---

**Algorithm 1:** Pseudo code of algorithm for computing  $\mathcal{S}_i^<$ .
 

---

```

Input:  $P, \tilde{T}[i], \mathcal{S}_{i-1}^<, ST(P)$ 
Output:  $\mathcal{S}_i^<$ 
// assumes  $\tilde{T}[i]$  is lexicographically sorted.
1  $S = \emptyset;$ 
2 Function  $\text{dfs}(\text{ancl}, k, (v_k, \ell_k), v):$ 
3   while  $v_k = v$  do
4      $\text{ancl.push}(\ell_k);$ 
5      $k \leftarrow k + 1;$ 
6      $(v_k, \ell_k) \leftarrow \text{locus of } \tilde{T}[i][k];$  // (null,0) if  $k > |\tilde{T}[i]|$ 
7   if  $W(v) = 0$  then //  $v$  is a root of a component
8      $S \leftarrow S \cup (\text{ancl} \oplus (L(v) \cap C(v) \cap \mathcal{S}_{i-1}^<));$ 
9   for  $c \in \text{chldr}(v)$  do // in lexicographic order of children
10     $\text{dfs}(\text{ancl}, k, (v_k, \ell_k), c);$  // recurse on child
11    while  $\text{ancl.top}() > \text{len}(c)$  do // discard visited descendants
12     $\text{ancl.pop}();$ 
13  $\text{ancl} \leftarrow \text{empty stack};$ 
14  $r \leftarrow \text{root of } ST(P);$ 
15  $(v_1, \ell_1) \leftarrow \text{locus of } \tilde{T}[i][1];$ 
16  $\text{dfs}(\text{ancl}, 1, (v_1, \ell_1), r);$ 
17 return  $S;$ 

```

---

$\tilde{T}[i][1], \dots, \tilde{T}[i][|\tilde{T}[i]|]$  are lexicographically sorted, we can reduce the space by doing the computation through a depth-first traversal on the suffix tree from left to right, during which we only maintain the loci on the path we are considering. The space requirement follows, since the length of this path is  $O(m)$ . This is illustrated in the pseudo-code shown in Algorithm 1.  $\blacktriangleleft$

**► Theorem 9.** *Problem 2 can be solved in  $O(N + nm^{1.5}\sqrt{\log m})$  time using  $O(m)$  working space.*

**Proof.** For each  $i = 1 \dots n$ , all computations other than Step 3 take  $O(\|\tilde{T}[i]\|)$  time, while Step 3 takes  $O(\|\tilde{T}[i]\| + m\sqrt{m \log m})$  time. Thus, for all  $1 \leq i \leq n$ , the total time is  $O(\sum_{i=1}^n \|\tilde{T}[i]\| + nm\sqrt{m \log m}) = O(N + nm\sqrt{m \log m})$ .  $\blacktriangleleft$

If we cannot assume that the strings in each  $\tilde{T}[i]$  are sorted in lexicographic order, the space complexity becomes  $O(m + \max_{i=1, \dots, n} |\tilde{T}[i]|)$ .

## 5 Conclusion

We present a new algorithm for the elastic degenerate string matching problem which runs in  $O(nm\sqrt{m \log m} + N)$  time using  $O(m)$  working space. While previous algorithms for the EDSM problem are basically applications of now “standard” string matching techniques, our algorithm applies a novel technique combining FFT and the suffix tree.

On a side note, it seems interesting that while Boolean convolution was the key technique that Fischer and Paterson [7] used to solve the degenerate pattern matching problem, we solve a generalized version of the problem with the same tool (Boolean convolution) but applying it in a different way.

---

**References**

---

- 1 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. doi:10.1137/0216067.
- 2 Stephen Alstrup, Jens Peter Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.
- 3 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 74–90, Cham, 2017. Springer International Publishing.
- 4 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.
- 5 James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. URL: <http://www.jstor.org/stable/2003354>.
- 6 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
- 7 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, 1974.
- 8 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPICs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.9.
- 9 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017. doi:10.1007/978-3-319-53733-7\_9.
- 10 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 11 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

# A Simple Linear-Time Algorithm for Computing the Centroid and Canonical Form of a Plane Graph and Its Applications

**Tatsuya Akutsu**

Bioinformatics Center, Institute for Chemical Research, Kyoto University  
Kyoto 611-0011, Japan  
takutsu@kuicr.kyoto-u.ac.jp

**Colin de la Higuera**

LINA, UMR CNRS 6241, Université de Nantes  
44322 Nantes Cedex 03, France  
cdlh@univ-nantes.fr

**Takeyuki Tamura**

Bioinformatics Center, Institute for Chemical Research, Kyoto University  
Kyoto 611-0011, Japan  
tamura@kuicr.kyoto-u.ac.jp

---

## Abstract

We present a simple linear-time algorithm for computing the topological centroid and the canonical form of a plane graph. Although the targets are restricted to plane graphs, it is much simpler than the linear-time algorithm by Hopcroft and Wong for determination of the canonical form and isomorphism of planar graphs. By utilizing a modified centroid for outerplanar graphs, we present a linear-time algorithm for a geometric version of the maximum common connected edge subgraph (MCCES) problem for the special case in which input geometric graphs have outerplanar structures, MCCES can be obtained by deleting at most a constant number of edges from each input graph, and both the maximum degree and the maximum face degree are bounded by constants.

**2012 ACM Subject Classification** Theory of computation → Graph algorithms analysis

**Keywords and phrases** Plane graph, Graph isomorphism, Maximum common subgraph

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.10

**Funding** TA was partially supported by JSPS KAKENHI #18H04113. TT was partially supported by JSPS KAKENHI #25730005. A part of CDLH's work was done when he was a visiting professor at Kyoto University.

## 1 Introduction

Comparison of geometric objects is an important topic in various fields including pattern recognition, computational geometry, and combinatorial pattern matching [7, 8, 17, 18]. In many cases, geometric objects are given as graphs having geometric information. Therefore, comparison of geometric objects are often modeled as pattern matching problems on graphs possibly with geometric information.

Among various problems on graph pattern matching, the most fundamental one is the *graph isomorphism* problem, which asks whether or not two given graphs are isomorphic. Although extensive studies have been done on determining the complexity class of graph



© Tatsuya Akutsu, Colin de la Higuera, and Takeyuki Tamura;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 10; pp. 10:1–10:12

Leibniz International Proceedings in Informatics



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

isomorphism, it is still unclear for general graphs [5]. Polynomial-time algorithms are known for special graph classes, which include graphs of bounded degree [14] and graphs of bounded treewidth [16]. In particular, it is known that *planar graph isomorphism* can be tested in  $O(n \log n)$  time [10] and in linear time [11], where  $n$  is the number of vertices on each given graph, and the former one was modified for testing the congruity of polyhedra [19]. Furthermore, both algorithms can be modified for computation of the *canonical form* of a given graph, where the canonical form is a unique representation of a graph that is invariant under isomorphic transformations. Although the algorithm in [10] is conceptually simple, that in [11] is complicated. In this paper, we focus on *plane graphs*, which is a planar graph with planar embedding, and present a (conceptually) simple linear-time algorithm for computing the canonical form of a plane graph. The algorithm first computes the topological *centroid* of a given graph, then transforms the graph into a *circular string*, and finally computes a canonical form of this circular string [6, 12]. Since it is known that there is a close relationship between planar and related graph isomorphism problems and the circular string problem [11, 15], the approach is reasonable. Of course, it seems possible to modify the algorithm in [11] for computation of the canonical form of a plane graph with keeping the linear time complexity. However, our algorithm is much simpler and, as far as we know, no simple algorithm has been known for determination of the canonical form or isomorphism of plane graphs. In addition, our algorithm constructs a tree representation of an input plane graph. By combining with grammar-based tree compression algorithm [3, 9], plane graphs having many local symmetries might be efficiently compressed.

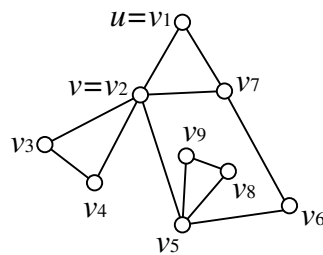
We also apply the (modified) centroid to the maximum common connected edge subgraph (MCCES) problem for two geometric plane graphs, which seeks for a graph with the maximum number of edges that is identical to a subgraph of each input graph under isometric transformations. Note that MCCES for geometric plane graphs is practically important because various kinds of maps (e.g., roadmaps) are often represented as geometric planar graphs. Although the MCCES problem is NP-hard for considerably restricted classes of graphs [1, 2, 4, 13], it can trivially be solved in polynomial time if graphs are restricted to geometric graphs and isomorphism is restricted to those by isometric transformations. We present a linear-time algorithm for the MCCES problem for the special case in which input geometric graphs have outerplanar structures, MCCES can be obtained by deleting at most a constant number of edges from each input graph, and both the maximum degree and the maximum face degree are bounded by constants.

## 2 Preliminaries

Let  $G(V, E)$  be an undirected graph. We assume that  $G$  is connected and its planar embedding is given. Such a graph is called a *plane graph*. We use  $n$  to denote the size of  $V$  (i.e.,  $n = |V|$ ). Since we only consider plane graphs,  $|E|$  is  $\Theta(n)$ . Two plane graphs  $G_1$  and  $G_2$  are called *isomorphic* if there exists an isomorphic mapping from  $G_1$  to  $G_2$  such that outer faces correspond to each other and the circular ordering of edges connected to each vertex is preserved. If  $G_1$  and  $G_2$  are isomorphic, we write  $G_1 \cong G_2$ .

Let  $\phi(G)$  be a function that maps a given undirected graph  $G(V, E)$  to a string over an alphabet of size  $O(n)$ . Then,  $\phi(G)$  is called a *canonical form* of  $G(V, E)$  if the following conditions are satisfied:

- $|\phi(G)|$  (i.e., the length of  $\phi(G)$ ) is  $O(n)$ ,
- $G$  can be reconstructed from  $\phi(G)$ ,
- $\phi(G_1) = \phi(G_2)$  if and only if  $G_1 \cong G_2$ .



■ **Figure 1** Example for  $\text{cano}(G, u, v)$ .

We assume that a plane graph is given in a form of the doubly-connected-edge-list (DCEL) [17] so that deletion of an edge can be done in a constant time and deletion of a face can be done in time proportional to the number of surrounding edges.

For a string  $A = a_0a_1 \dots a_{n-1}$ ,  $a_ia_{i+1} \dots a_{i+n-1}$  is called the *canonical form* of a circular string  $A$  if it is lexicographically smallest among  $i = 0, \dots, n-1$ , where indices are calculated modulo  $n$  [6, 12].

### 3 $O(n^2)$ time canonical form computation

We begin with a simple  $O(n^2)$  time algorithm for computing a canonical form. It is a known fact (e.g., see [18]) but a part of this algorithm is used as a subroutine in the next section.

Let  $G(V, E)$  be a plane graph. Suppose that we are given a pair of vertices  $(u, v)$  such that  $\{u, v\} \in E$ . We construct a string  $\text{cano}(G, u, v)$  by using the following procedure.

1. Perform depth first search (DFS) starting from  $u$  with choosing  $(u, v)$  as the first edge (with the direction from  $u$  to  $v$ ) and regarding it as the leftmost edge from  $u$ . In DFS, we visit edges emanating from each vertex from left to right (i.e., anti-clockwise order).
2. Rename the vertices according to the visited DFS order. Let the resulting vertices be  $v_1, v_2, \dots, v_n$ .
3. Construct the Euler string by concatenating edges in the visited order, where each edge is represented as  $(i, j)$  when  $v_j$  is visited just after  $v_i$ .

► **Example 1.** Consider a plane graph  $G(V, E)$  shown in Figure 1. If DFS is started from  $(u, v)$ , vertices are renamed as in Figure 1. The resulting  $\text{cano}(G, u, v)$  will be

$$(1, 2)(2, 3)(3, 4)(4, 2)(2, 4)(4, 3)(3, 2)(2, 5)(5, 6)(6, 7)(7, 1)(1, 7)(7, 2)(2, 7) \\ (7, 6)(6, 5)(5, 8)(8, 9)(9, 5)(5, 9)(9, 8)(8, 5)(5, 2)(2, 1)$$

It is seen from Example 1 that in  $\text{cano}(G, u, v)$ , each edge appears exactly twice in the opposite directions, which further means that  $\text{cano}(G, u, v)$  is a kind of Euler string. Clearly,  $\text{cano}(G, u, v)$  can be computed in  $O(n)$  time and  $|\text{cano}(G, u, v)|$  is  $O(n)$ . Since the original plane graph is reconstructed from  $\text{cano}(G, u, v)$ , we can see that  $\text{cano}(G_1, u_1, v_1) = \text{cano}(G_2, u_2, v_2)$  holds if and only if there exists an isomorphic mapping from  $G_1$  to  $G_2$  that maps  $u_1$  and  $v_1$  to  $u_2$  and  $v_2$ , respectively.  $\text{cano}(G, u, v)$  is called an *edge-specified canonical form*.

Let  $\text{cano}_0(G)$  be the edge-specified canonical form which is lexicographically smallest among  $\text{cano}(G, u', v')$ s such that  $\{u', v'\} \in E$ . Then, we have

► **Proposition 2.**  $\text{cano}_0(G)$  is a canonical form of a plane graph  $G$  and can be computed in  $O(n^2)$  time.

#### 4 Linear time canonical form computation

As shown in Section 3, we can have a canonical form in  $O(n)$  time if some unique edge is identified. However, it is quite difficult to efficiently identify the unique edge because the canonical form problem intrinsically includes the canonical form problem on circular strings. Therefore, we will reduce the canonical form problem on plane graphs into the canonical form problem on circular strings.

Our idea is to identify the (topological) *centroid* of a given plane graph  $G$ , where the centroid is either a vertex, an edge, or a face. Once the centroid is found, we can create a circular string using the method given in Section 3.

For an unordered tree  $T$ ,  $v$  is called a *centroid* if the longest path from  $v$  to leaves is the shortest. It is known that there exist either one centroid, or two centroids connected by an edge. In the former case, this unique vertex is called the *centroid vertex*. In the latter case, this unique edge is called the *centroid edge*. It is well known that for a tree  $T$ , the centroid vertex/edge can be determined in linear time. We extend this concept for plane graphs.

In the following, we show how to construct a canonical form of a plane graph based on the centroid and the trees connected to the centroid that are constructed in the determination process of the centroid. To this end, we first determine the centroid of a plane graph. It is known that the connected plane graph has the unique outer face. We consider the directed cycle consisting of the edges of the outer face, where edges are visited in the clockwise order. Let  $C$  be this directed cycle (see Figure 2 (A)). An inner face (i.e., a face that is not the outer face) of a plane graph  $G$  is called *exposed* if it includes an edge in  $C$  (with ignoring the direction). Furthermore, an inner face is called *singly exposed* if the outer edges in this face are connected in  $C$ . Similarly, an edge not belonging to an inner face is called *singly exposed* if one of its endpoints is of degree 1. A graph consisting of a vertex, an edge, or a face (including adjunct subgraphs) is not regarded as *singly exposed*. In addition, each maximally connected subgraph  $G_S$  that is surrounded by a singly exposed face with sharing only one vertex  $v$  that is an outer one is called an *adjunct subgraph* (see Figure 2 (B)). Each adjunct subgraph is ignored and removed along with its surrounding face because information on this part can be easily included in the final canonical form as follows. According to the ordering of  $C$ , we can define the parent vertex  $u$  of  $v$ . Then, the leftmost child  $w$  of  $v$  in the subgraph can be uniquely determined and thus  $\text{cano}(G_S, v, w)$  can be computed (see Figure 3). We can insert this  $\text{cano}(G_S, v, w)$  (delimited by a special symbol '&' not appearing in other parts) into a part the canonical form corresponding to removed outer edges.

After identification of the singly exposed faces and edges, all of them will be removed. This removal is done by deleting outer edges included in these faces and edges. However, we keep information about all these edges in order to reconstruct tree structures at the final stage. To this end, we virtually detach the last outer edge from the last endpoint in each singly exposed face, where the last means that in the order of  $C$  (see Figure 2 (B)).

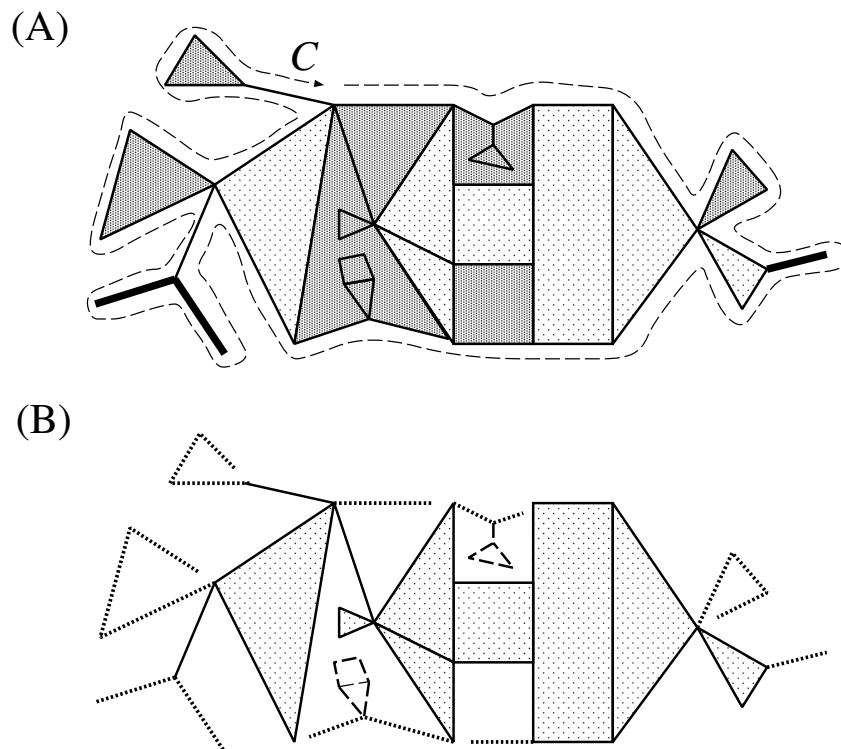
We use the following procedure (denoted as **PEELING**) to identify the centroid vertex/edge/face (see Figure 4).

1. Repeat Steps 2-3 until there does not exist a singly exposed face/edge.
2. Identify all singly exposed faces and edges.
3. Delete outer edges and adjunct subgraphs in these exposed faces and edges.

The correctness of **PEELING** is guaranteed by the following two propositions.

► **Proposition 3.** *After each removal step, the resulting graph is connected.*

**Proof.** Since each face is doubly connected by its surrounding edges and only consecutive outer edges are removed from each face, we do not lose the connectedness. ◀



■ **Figure 2** Example of face removal operations. (A) Directed cycle  $C$  is shown by a dashed curve. Gray regions and bold edges correspond to singly exposed faces and edges, respectively. (B) Deleted outer edges and adjunct subgraphs are shown by dotted lines and dashed lines, respectively.

► **Proposition 4.** *If there does not exist a singly exposed face or edge in  $G$ ,  $G$  is either a vertex, an edge, or a face (possibly including adjunct subgraphs inside).*

**Proof.** Suppose that  $G$  is not a vertex, an edge, or a face. Then, consider the directed cycle  $C$  consisting of outer edges. Each edge in  $C$  belongs to a face or an edge (not in a face). Then,  $C$  must include at least one edge in a singly exposed face or edge. ◀

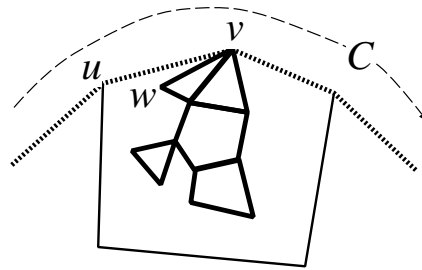
After determining the centroid of a plane graph, we construct a canonical form using the centroid as follows. Since the other cases are easier, we assume w.l.o.g. (without loss of generality) that a single face  $f_c$  (possibly including adjunct subgraphs inside) is finally left. Then, we add trees and adjunct subgraphs deleted by **PEELING** to the centroid  $f_c$ . Let  $G'$  be the resulting graph. As mentioned before, we assume w.l.o.g. that there does not exist any adjunct subgraph. Therefore, the resulting graph consists of the centroid and trees. Let  $v^1, v^2, \dots, v^d$  be the vertices of  $f_c$  arranged in the clockwise order, starting from an arbitrary one (see Figure 5). For each vertex  $v^i$ , let  $v_1^i, \dots, v_{d_v}^i$  be the neighboring vertices (not in  $f_c$ ) in the clockwise order. For each subtree  $T_j^i$  rooted at  $(v^i, v_j^i)$ , we construct  $\text{cano}(T_j^i, v^i, v_j^i)$  and then construct the string  $\text{cano}(v^i)$  by concatenating these as

$$\text{cano}(v^i) = \# \text{cano}(T_1^i, v^i, v_1^i) \# \text{cano}(T_2^i, v^i, v_2^i) \# \dots \# \text{cano}(T_{d_v}^i, v^i, v_{d_v}^i) \#,$$

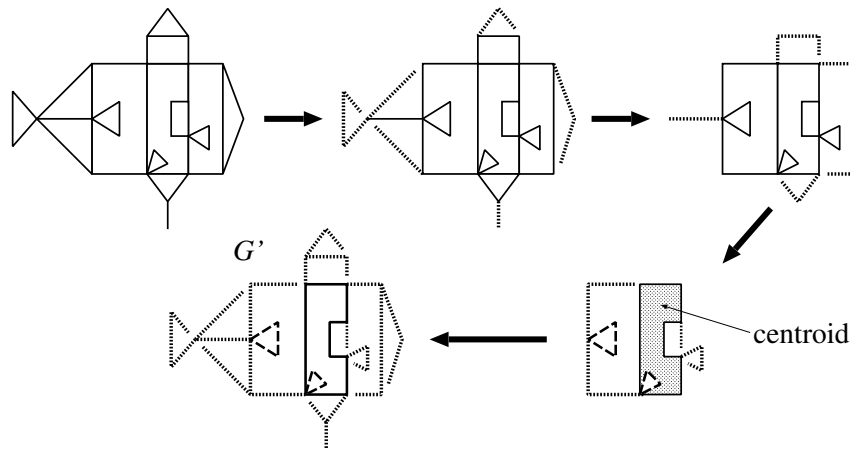
where ‘#’ is a special symbol not appearing in other parts.

Since trees in the canonical form may be obtained by decomposing cycles of the original plane graph, leaves may need information about from which vertices they are detached.





■ **Figure 3** Example of an adjunct subgraph.



■ **Figure 4** Illustration of the **PEELING** procedure. Deleted edges are shown by dotted lines. Adjunct subgraphs are shown by dashed lines.

Therefore, in computation of  $cano(T_j^i, v, v_j^i)$ , we need to add the information about other trees because the disconnected edge shares a vertex in  $T_j^i$ , another tree, or the centroid. In order to cope with this problem, we renumber  $T_j^i$ s to be  $T_1, \dots, T_m$  in the clockwise order starting from an arbitrary tree (we only use the difference of the indices modulo  $m$ ). We consider the following three cases (see Figure 5):

- if the disconnected endpoint  $v_j$  of an edge  $(v_h, v_j)$  in  $T_i$  is actually a vertex  $v_{j'}$  in the same subtree  $T_i$ , we replace  $j$  in  $cano(\dots)$  with  $(T, j')$ ,
- else if the disconnected endpoint  $v_j$  of an edge  $(v_h, v_j)$  in  $T_i$  is actually a vertex  $v_{j'}$  in  $T_{i'}$ , we replace  $j$  in  $cano(\dots)$  with  $(T + (i' - i), j')$ , where  $i' - i$  is computed modulo  $m$ ,
- otherwise (i.e.,  $v_j$  is actually a vertex  $v^{k'}$  in the centroid), we replace  $j$  in  $cano(\dots)$  with  $(C + (k' - k))$ , where  $v^k$  is the root of  $T_i$  and  $k' - k$  is computed modulo  $d$ .

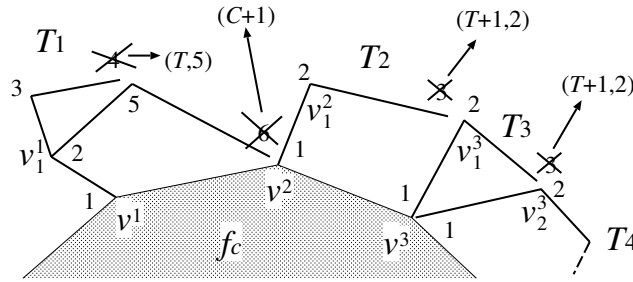
Then, we construct  $cano(F)$  by concatenating  $cano(v_i)$ s by

$$cano(F) = cano(v_1)!cano(v_2)! \dots !cano(v_k)!,$$

where ‘!’ is a special symbol not appearing in other parts. Finally, we regard  $cano(F)$  as a circular string and define  $cano(G)$  to be the canonical form of this circular string.

► **Theorem 5.**  $cano(G)$  is a canonical form of a plane graph  $G$  and can be computed in  $O(n)$  time.





■ **Figure 5** Illustration of replacement of labels for disconnected vertices.

**Proof.** The correctness follows from the following facts:

- The peeling process is invariant under isomorphic transformations, that is, the same set of edges is always deleted at each time step for isomorphic plane graphs.
- After the peeling process, only one face, edge, or vertex remains.
- $cano(v_i)$  is invariant under isomorphic transformations.

Next, we analyze the time complexity. We maintain plane graphs using DCEL data structure with adding information about exposed/not exposed. We also maintain lists of consecutively exposed edges and pointers from each list to the corresponding face and from each face to the corresponding lists. Each list/face also has a flag showing whether or not it is singly exposed one. The peeling process can be done by deleting edges in lists with singly exposed flags. Of course, all data structures must be updated, which can be done in time proportional to the number of deleted edges and the number of newly exposed edges. Since each edge is newly exposed only once and is deleted only once, the total time complexity is proportional to the number of edges (i.e., the total complexity is  $O(n)$ ). It is straightforward to see that  $cano(F)$  can be obtained in  $O(n)$  time. Since the canonical form of a circular string over a general alphabet can be computed in  $O(n)$  time [6, 12], the total time complexity is  $O(n)$ . ◀

## 5 Canonical form of geometric plane graphs

The algorithm in Section 4 can be modified for computing the canonical form of a given geometric plane graph so that the canonical form is invariant under isometric transformations. We say that  $G_1$  and  $G_2$  are isomorphic under isometric transformations if there is an isometric transformation  $T$  (i.e., combination of translation, rotation, and mirror image) such that  $T(shape(G_1)) = shape(G_2)$ , where  $shape(G)$  denotes the set of line segments in  $G$ . We may omit  $shape(\dots)$  and write this relation as  $T(G_1) = G_2$ . Since inclusion of mirror images in isometric transformation can be done by multiplying a constant factor to the time complexity, we ignore them in the following.

In order to include geometric information, it is enough to add geometric information to  $cano(G, u, v)$ . Suppose that  $(u, v)$  and  $(v, w)$  are consecutive edges. Let  $L^2(v, v')$  denote the square of the Euclid distance between  $v$  and  $v'$ , the exact value of which can be computed from the coordinates of  $v$  and  $v'$ . Then, we add the following information to  $(v, w)$ .

- $L^2(u, v), L^2(v, w), L^2(u, w)$ ,
- whether  $w$  is located left or right of  $\vec{uv}$ .

It is straightforward to see the correctness of this modified procedure to define the canonical form. Since it does not increase the order of the size of the canonical form and the time complexity, the following holds.

► **Proposition 6.** *The canonical form of a geometric plane graph be computed in  $O(n)$  time.*

It might be possible to use the geometric centroid (which can be easily computed), in place of the topological centroid, to compute the canonical form in linear time. However, it is unclear whether or not the circular string can be constructed easily.

## 6 Maximum common connected edge subgraph of geometric plane graphs

We consider the problem of finding a maximum common connected edge subgraph (MCCES)  $G_c$  of two geometric plane graphs  $G_1$  and  $G_2$ :  $G$  has the maximum number of edges such that  $G = T(G_1) \cap G_2$  for some isometric transformation  $T$ . For simplicity, we assume that  $G_1$  and  $G_2$  have  $O(n)$  edges. We also assume Real RAM (Random Access Machine) as a computation model in which each arithmetic computation can be done in a constant time. This problem can be solved by the following procedure (**SimpleMCCES**):

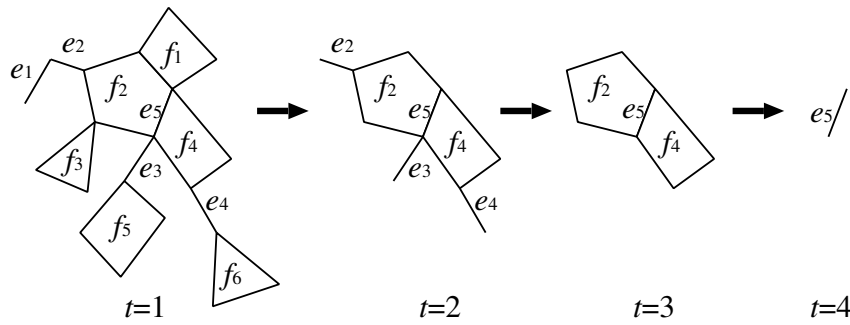
1. Let  $G_0$  be an empty graph.
2. For all directed edge pairs  $(e_1, e_2) \in E(G_1) \times E(G_2)$ , repeat steps 3-6.
3. Determine isometric transformation  $T$  uniquely (except mirror image) that maps  $e_1$  to  $e_2$ .
4. Let  $G \leftarrow T(G_1) \cap G_2$ .
5. Let  $E^c(G)$  be the set of edges in the connected component of  $G$  having the maximum number of edges.
6. If  $|E^c(G)| > |E^c(G_0)|$ , then let  $G_0 \leftarrow G$ .
7. Output  $G_0$ .

In computation of  $T(G_1) \cap G_2$ , edges remain only if two corresponding edges completely overlap. Furthermore,  $T$  is examined only if the lengths of  $e_1$  and  $e_2$  are the same. Then, the correctness of the procedure is obvious.

Next, we analyze the time complexity. Suppose that  $G_c$  has  $O(k)$  edges. Step 4 can be done in  $O(n)$  time by performing DFS using edges common to  $T(G_1)$  and  $G_2$ . If the maximum degree is bounded by a constant, it can be done in  $O(k)$  time. Since we may examine all edge pairs, Steps 3-6 are repeated  $O(n^2)$  times. Therefore, the total time complexity is  $O(n^3)$  in a general case and is  $O(kn^2)$  if the maximum degree is bounded by a constant.

When the maximum degree is bounded by a constant, we can improve the time complexity to  $O(n^2 \log n)$  (it is an improvement when  $k > c \log n$  for some constant  $c$ ) using a *geometric hashing* [20]. For each directed edge pair  $(e_1, e_2) \in E(G_1) \times E(G_2)$  having the same length, we compute the unique isometric transformation  $T$  such that  $T(e_1) = e_2$ , and put this pair into the bin labeled with  $T$ . Then, we find the bin containing the maximum number of pairs connected in both  $G_1$  and  $G_2$ , which corresponds to MCCES. Since  $O(n^2)$  pairs are examined and finding the bin (where a respective edge pair is to be put in) needs  $O(\log n)$  time using binary search, the total computation time to create all bins is  $O(n^2 \log n)$ . Since we assumed that the maximum degree is bounded by a constant, connected components in all bins can be computed in linear time of the total number of edge pairs. Therefore, the total time complexity is  $O(n^2 \log n)$ .

The above results are almost trivial. Here we present a faster algorithm for a special case of geometric MCCES in which graphs are outerplanar, both the maximum degree and the maximum face degree (i.e., maximum number of edges of a face) are bounded by constants, and  $G_c$  is very similar to  $G_1$  and  $G_2$  (precisely,  $G_c$  is obtained by deleting at most  $K$  edges from  $G_1$  and also by deleting at most  $K$  edges from  $G_2$ ). In the following, a plane graph with outerplanar graph structure is called an *outer-plane graph*. Suppose that the maximum



■ **Figure 6** Determination of the centroid for an outer-plane graph. In this case,  $t_G(f_1) = t_G(f_3) = t_G(f_5) = t_G(f_6) = t_G(e_1) = 1$ ,  $t_G(e_2) = G(e_3) = t_G(e_4) = 2$ ,  $t_G(f_2) = G(f_4) = 3$ , and  $e_5$  is the centroid.

degree and the maximum face degree are bounded by constants  $D_v$  and  $D_f$ , respectively. We will show that the position of the centroid changes for a constant amount by addition (or deletion) of an edge.

To this end, we use a simpler definition of the centroid for an outer-plane graph  $G$ , where it can also be applied to an outerplanar graph. We use the following simple procedure (see Figure 6), where the resulting face/edge/vertex is the centroid and is denoted by  $c_O(G)$ .

1. Repeat Step 2 until there remains only one face, edge, or vertex.
2. Identify all faces and edges each of which overlaps with other face(s)/edge(s) at one edge (including its endpoints) or one vertex.
3. Delete all faces and edges identified in Step 2.

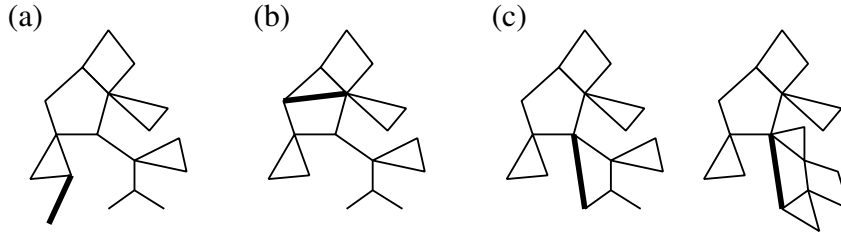
It is straightforward to see that this procedure works in  $O(n)$  time and the centroid is determined uniquely for isomorphic outer-plane graphs. For two edges  $e_1$  and  $e_2$ ,  $d(e_1, e_2)$  denotes the shortest distance between endpoints of  $e_1$  and endpoints of  $e_2$ , where the distance between vertices is defined as the length of the shortest path connecting  $u$  and  $v$ . For the centroid  $C$  in a graph  $G$ , let  $e(x)$  denote the set of edges in  $x$  if  $x$  is a face or an edge, and the set of edges connecting to  $x$  otherwise (i.e.,  $x$  is a vertex). In addition, let  $e(x, d)$  denote the set of edges each of which has an edge in  $x$  within distance  $d$  in  $G$ . Then, we apply **SimpleMCCES** only for the edge pairs (each with two directions) between  $e(c_O(G_1), d_K)$  and  $e(c_O(G_2), d_K)$ , where  $d_K$  is to be determined later so that at least one edge in  $e(c_O(G_c))$  is included in both  $e(c_O(G_1), d_K)$  and  $e(c_O(G_2), d_K)$ .

► **Lemma 7.** *Suppose that  $G_2$  is obtained by adding an edge to  $G_1$  with keeping outerplanarity and connectivity. Then, the minimum distance between  $e(c_O(G_1))$  and  $e(c_O(G_2))$  is at most  $D_f^2/2$ .*

**Proof.** For each face or edge (not in a face), we consider the time step (the number of repeats) at which the face/edge is deleted in the procedure determining the centroid, where the time step for the firstly deleted faces/edges is regarded to be 1. For each face or edge  $x$  in graph  $G$ ,  $t_G(x)$  denotes this deletion time step (see Figure 6).

We classify an addition of an edge into the following three cases (see Figure 7).

- (a) One endpoint is a new vertex.
- (b) An existing face is divided into two faces.
- (c) A new face (not in an existing face) is created.



■ **Figure 7** Classification of edge addition patterns. Added edges are shown by bold lines.

Let  $G_2$  be the graph obtained by addition of an edge to  $G_1$ .

It is straightforward to see that the following properties hold.

- In case (a),  $|t_{G_2}(x) - t_{G_1}(x)| \leq 1$  holds for each face/edge  $x$ .
- In case (b),  $|t_{G_2}(x) - t_{G_1}(x)| \leq 1$  holds for each face/edge  $x$ .
- In case (c),  $|t_{G_2}(x) - t_{G_1}(x)| \leq D_f$  holds for each face/edge  $x$ .

Since the centroid must have an overlap with the lastly deleted face(s)/edge(s) (i.e., face(s)/edge(s) with the maximum  $t_{G_i}(x)$ ) and the distance between two vertices in the same face is at most  $D_f/2$ , the lemma holds. ◀

► **Theorem 8.** *Suppose that both the maximum degree and the maximum face degree of geometric outer-plane graphs  $G_1$  and  $G_2$  are bounded by constants  $D_v$  and  $D_f$ , respectively. Suppose also that a maximum common connected edge subgraph is obtained by deletion of at most  $K$  edges from each of  $G_1$  and  $G_2$ . Then, a maximum common connected edge subgraph can be computed in  $O(f(D_f, D_v, K)n)$  time, where  $f(D_f, D_v, K) = D_f^2 \cdot D_v^{KD_f^2 + D_f + 4}$ .*

**Proof.** From Lemma 7 and the assumption on  $G_c$ , the minimum distance between edges in  $c_O(G_c)$  and  $c_O(G_i)$  is at most  $KD_f^2/2$ . Then, all edges in  $e(c_O(G_c))$  are included in  $e(c_O(G_i), (KD_f^2 + D_f)/2)$  for each  $G_i$ . Therefore, by letting  $d_K = (KD_f^2 + D_f)/2$ , a maximum common connected edge subgraph can be found for two geometric graphs  $G_1$  and  $G_2$ . Since the vertex degree is bounded by  $D_v$ , the number of edges in  $e(c_O(G_i), (KD_f^2 + D_f)/2)$  is at most  $D_f \cdot D_v^{(KD_f^2 + D_f)/2 + 1}$ . Since we examine all pairs in  $e(c_O(G_1), (KD_f^2 + D_f)/2)$  and  $e(c_O(G_2), (KD_f^2 + D_f)/2)$ , the number of directed edge pairs examined is at most  $2D_f^2 \cdot D_v^{KD_f^2 + D_f + 2}$ . For each pair, computation of  $T(G_1) \cap G_2$  can be done in  $O(D_v^2 n)$  time using DFS. Therefore, the theorem holds. ◀

It is unclear whether Lemma 7 or a similar lemma holds for outer-plane graphs if the centroid  $c(G_i)$  defined in Section 3 is used. However, it is easy to see that a similar lemma does not hold for plane graphs by considering a graph including large adjunct subgraphs. Therefore, defining a centroid for plane graphs so that a property similar to Lemma 7 holds is left as an open problem.

---

## References

- 1 Faisal N. Abu-Khzam. Maximum common induced subgraph parameterized by vertex cover. *Inf. Process. Lett.*, 114(3):99–103, 2014. doi:10.1016/j.ipl.2013.11.007.
- 2 Tatsuya Akutsu. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE Transactions*, 76-A(9):1488–1493, 1993.

- 3 Tatsuya Akutsu. A bisection algorithm for grammar-based compression of ordered trees. *Inf. Process. Lett.*, 110(18-19):815–820, 2010. doi:10.1016/j.ipl.2010.07.004.
- 4 Tatsuya Akutsu and Takeyuki Tamura. On the complexity of the maximum common subgraph problem for partial k-trees of bounded degree. In Kun-Mao Chao, Tsan-sheng Hsu, and Der-Tsai Lee, editors, *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, volume 7676 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2012. doi:10.1007/978-3-642-35261-4\_18.
- 5 László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 684–697. ACM, 2016. doi:10.1145/2897518.2897542.
- 6 Kellogg S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980. doi:10.1016/0020-0190(80)90149-0.
- 7 Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3):265–298, 2004. doi:10.1142/S0218001404003228.
- 8 Andreas Fischer, Kaspar Riesen, and Horst Bunke. Improved quadratic time approximation of graph edit distance by combining hausdorff matching and greedy assignment. *Pattern Recognition Letters*, 87:55–62, 2017. doi:10.1016/j.patrec.2016.06.014.
- 9 Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. *Algorithmica*, 80(3):885–917, 2018. doi:10.1007/s00453-017-0279-3.
- 10 John E. Hopcroft and Robert Endre Tarjan. A  $V \log V$  algorithm for isomorphism of triconnected planar graphs. *J. Comput. Syst. Sci.*, 7(3):323–331, 1973. doi:10.1016/S0022-0000(73)80013-3.
- 11 John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, editors, *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 172–184. ACM, 1974. doi:10.1145/800119.803896.
- 12 Costas S. Iliopoulos and William F. Smyth. Optimal algorithms for computing the canonical form of a circular string. *Theor. Comput. Sci.*, 92(1):87–105, 1992. doi:10.1016/0304-3975(92)90137-5.
- 13 Nils Kriege, Florian Kurpicz, and Petra Mutzel. On maximum common subgraph problems in series-parallel graphs. *Eur. J. Comb.*, 68:79–95, 2018. doi:10.1016/j.ejc.2017.07.012.
- 14 Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982. doi:10.1016/0022-0000(82)90009-5.
- 15 Joseph Manning and Mikhail J. Atallah. Fast detection and display of symmetry in outerplanar graphs. *Discrete Applied Mathematics*, 39(1):13–35, 1992. doi:10.1016/0166-218X(92)90112-N.
- 16 Jiří Matoušek and Robin Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, 108(1-3):343–364, 1992. doi:10.1016/0012-365X(92)90687-B.
- 17 Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction*. Texts and Monographs in Computer Science. Springer, 1985. doi:10.1007/978-1-4612-1098-6.
- 18 Christine Solnon, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition*, 48(2):302–316, 2015. doi:10.1016/j.patcog.2014.05.019.

## 10:12 Centroid and Canonical Form of Plane Graph

- 19 Kokichi Sugihara. An  $n \log n$  algorithm for determining the congruity of polyhedra. *J. Comput. Syst. Sci.*, 29(1):36–47, 1984. doi:10.1016/0022-0000(84)90011-4.
- 20 KHaim J. Wolfson and Isidore Rigoutsos. Geometric hashing: An overview. *IEEE Comput. Sci. & Eng.*, 4(4):10–21, 1997. doi:10.1109/99.641604.

# Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms

Amihood Amir<sup>1</sup>

Bar-Ilan University and Johns Hopkins University  
amir@cs.biu.ac.il

Itai Boneh

Bar-Ilan University  
barbunyaboy2@gmail.com

---

## Abstract

There has been recent interest in dynamic string algorithms, i.e. string problems where the input changes dynamically. One such problem is the longest common factor (LCF) problem. It is well known that the LCF of two strings  $S$  and  $D$  of length  $n$  over a fixed constant-sized alphabet  $\Sigma$  can be computed in time linear in  $n$ . Recently, a new challenge was introduced - finding the LCF of two strings in a dynamic setting. The problem is the *fully dynamic one sided LCF (FDOS-LCF) problem*. In the FDOS-LCF problem we get  $q$  consecutive queries of the form  $\langle i, \alpha \rangle$ , where each such query means: “replace  $D[i]$  by  $\alpha$ ,  $\alpha \in \Sigma$  and output the LCF of  $S$  and (the updated)  $D$ . The goal is to initially preprocess  $S$  and  $D$  so that we do not need  $\mathcal{O}(n)$  time to compute an LCF for each such query.

The state-of-the-art is an algorithm that preprocesses the two strings  $S$  and  $D$  in time  $\mathcal{O}(n \log^4 n)$ . Subsequently, the algorithm answers in time  $\mathcal{O}(\log^3 n)$  a **single** query of the form: Given a position  $i$  on  $D$  and a letter  $\alpha$ , return an LCF of  $S$  and  $D'$ , where  $D'$  is the string resulting from  $D$  after substituting  $D[i]$  with  $\alpha$ . That algorithm is not extendable to multiple queries. In this paper we present a tool - Locally Maximal Common Factors (LMCF) - that proves to be quite useful in solving some restricted versions of the FDOS-LCF problem. The versions we solve are the *Decremental FDOS-LCS problem*, where every change  $\langle i, \alpha \rangle$  is of the form  $\langle i, \omega \rangle$ ,  $\omega \notin \Sigma$ , and the *Periodic FDOS-LCS problem*, where  $S$  is a periodic string with period length  $p$ .

For the decremental problem we provide an algorithm with linear time preprocessing and  $\mathcal{O}(\log \log n)$  time per query. For the periodic problem our preprocessing time is linear and the query time is  $\mathcal{O}(p \log \log n)$ .

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Pattern matching, Theory of computation  $\rightarrow$  Dynamic graph algorithms

**Keywords and phrases** Dynamic Algorithms, Periodicity, Longest Common Factor, Priority Queue Data Structures, Suffix Tree, Balanced Search Tree, Range Maximum Queries

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.11

---

<sup>1</sup> Partially supported by ISF grant 571/14





## 1 Introduction

Recently, there has been a growing interest in dynamic pattern matching algorithms. A particular problem that received attention is finding the *longest common factor (LCF)* of two strings. Let  $S$  and  $D$  be strings of lengths  $n_1$  and  $n_2$ , respectively, over a constant size alphabet  $\Sigma$ . Their longest common factor (substring) is the longest string  $F$  that is a sunstring of both  $S$  and  $D$ . The LCF can be computed in  $\mathcal{O}(n_1 + n_2)$  time [10].

As mentioned in [13], the LCF problem is not robust and its solution can vary significantly when the input strings are changed even by one letter. It is, therefore, important to study the dynamic instance of the problem, i.e. finding the LCF between two strings after an arbitrary number of changes in one of the two sequences. Other than the purely theoretical interest, the problem has applications in the field of molecular biology. For instance, when we wish to find the LCFs by incorporating the single nucleotide polymorphisms (SNPs) observed in a population in one of the two sequences. The longest common factor with  $k$ -mismatches problem has also received much attention recently, in particular due to its applications in bioinformatics [11]. We refer the interested reader to [2, 7, 9, 13].

Recently, a solution to the restricted case, where a single edit operation (substitution, insertion or deletion) is allowed, was presented [1]. In that paper, two strings,  $S$  and  $D$ , of length  $n$  over a finite fixed alphabet  $\Sigma$ , are given. The strings are preprocessed in  $\mathcal{O}(n \log^4 n)$  time and  $\mathcal{O}(n \log^3 n)$  space. After preprocessing, the answer to a query replacing the symbol in index  $i$  of string  $D$  by  $\alpha$  is computed in  $\mathcal{O}(\log^3 n)$  time.

The solution in [1] is not extendable to more substitutions. The goal is solving the **fully dynamic one sided LCF (FDOS-LCF) problem**. Specifically, suppose we get  $q$  consecutive queries of the form  $\langle i, \alpha \rangle$ , where each such query means: “replace  $D[i]$  by  $\alpha$ ,  $\alpha \in \Sigma$  and output the LCF of  $S$  and (the updated)  $D$ . The goal is to initially preprocess  $S$  and  $D$  so that we do not need  $\mathcal{O}(n)$  time to compute an LCF for each such query.

This problem, as well as many other dynamic string problems, would be efficiently solvable in a straight-forward manner if one could efficiently maintain a suffix tree of a changing string. Alas, a fully dynamic suffix tree seems like a very difficult challenge. In this paper we present a tool – Locally Maximal Common Factors (LMCF) – that proves to be quite useful in solving some restricted versions of the FDOS-LCF problem .

The first problem we solve is the dynamic LCF problem for the *decremental* dynamic model. In this model, every substitution in  $D$  is of a symbol  $\omega \notin \Sigma$ . We provide an algorithm with two implementations. The first does a linear time preprocessing and answers a query in time  $\mathcal{O}(\log n)$ . It is presented for pedagogical reasons, to describe the power of the LMCF idea in a simple manner. The second implementation uses data structures such as the van Emde Boaz tree, or y-fast tries that enable solving the problem in time  $\mathcal{O}(\log \log n)$  per query, following a linear time preprocessing.

The second problem we solve is the FDOS-LCF problem in the special case where  $S$  is periodic. Our solution has a linear time preprocessing and subsequent  $\mathcal{O}(p \log \log n)$  time queries.

## 2 Preliminaries

We begin with basic definitions and notation generally following [4]. Let  $S = S[1]S[2]\dots S[n]$  be a *string* of length  $|S| = n$  over a finite ordered alphabet  $\Sigma$  of size  $|\Sigma| = \mathcal{O}(1)$ . By  $\varepsilon$  we denote an empty string. For two positions  $i$  and  $j$  on  $S$ , we denote by  $S[i..j] = S[i]..S[j]$  the *factor* (sometimes called *substring*) of  $S$  that starts at position  $i$  and ends at position  $j$  (it equals  $\varepsilon$  if  $j < i$ ). We recall that a prefix of  $S$  is a factor that starts at position 1



( $S[1..j]$ ) and a suffix is a factor that ends at position  $n$  ( $S[i..n]$ ). We denote the reverse string of  $S$  by  $S^R$ , i.e.  $S^R = S[n]S[n-1]\dots S[1]$ . We denote the concatenation of two strings,  $S_1$  and  $S_2$ , by  $S_1S_2$ .

Let  $Y$  be a string of length  $m$  with  $0 < m \leq n$ . We say that there exists an *occurrence* of  $Y$  in  $S$ , or, more simply, that  $Y$  *occurs in*  $S$ , when  $Y$  is a factor of  $S$ . Every occurrence of  $Y$  can be characterised by a starting position in  $S$ . Thus we say that  $Y$  occurs at the *starting position*  $i$  in  $S$  when  $Y = S[i..i+m-1]$ .

Given two strings  $S$  and  $D$ , a string  $Y$  that occurs in both is a longest common factor (LCF) of  $S$  and  $D$  if there is no longer factor of  $D$  that is also a factor of  $S$ ; note that  $S$  and  $D$  can have multiple LCF strings. We introduce a natural representation of an LCF of  $S$  and  $T$  as a triple  $(m, p, q)$  such that  $S[p..p+m-1] = T[q..q+m-1]$  is an LCF of  $S$  and  $T$ . The *decremental dynamic LCF* problem is formally defined as follows;

DECREMENTAL DYNAMIC LCF

**Input:** Two strings  $S$  and  $D$  of length  $n$  over an alphabet  $\Sigma$ , symbol  $\omega \notin \Sigma$ .

Let  $\langle i_1, \omega \rangle, \langle i_2, \omega \rangle, \dots, \langle i_k, \omega \rangle$  be a sequence of substitution operations in  $D$ , and let  $D'$  be the result of these  $k$  substitutions.

**Output:** An LCF of  $S$  and  $D'$ .

Clearly, the problem can be solved by computing an LCF after every change. We will see that such a computation can be done in linear time. We show an algorithm whose preprocessing time is linear, and where an LCF computation after each substitution can be done more efficiently. In Section 4 we present an implementation whose query complexity is logarithmic. In Section 5 we present an implementation whose query processing time is  $\mathcal{O}(\log \log n)$ .

### 3 Algorithm's Idea

We need some additional tools and definitions:

► **Definition 1.** Let  $S$  and  $D$  be two strings of length  $n$  over fixed finite alphabet  $\Sigma$ . A *locally maximal common factor (LMCF)* of  $S$  in  $D$  is a factor  $D[i..j]$  of  $D$  that satisfies the following two conditions:

1.  $D[i..j]$  is a factor of  $S$ .
2. Neither  $D[i..j+1]$  nor  $D[i-1..j]$  are factors of  $S$ .

The following observations are crucial.

► **Observation 2.** An LCF of two strings  $S$  and  $D$  is an LMCF of  $S$  in  $D$ . Moreover, a longest LMCF of  $S$  in  $D$  is an LCF.

► **Observation 3.** There are at most  $n$  LMCF's of  $S$  in  $D$ .

**Proof.** It is clear that only one LMCF can start at any index  $i$  of  $D$ . Otherwise, let  $i$  be an index such that both  $D[i..j]$  and  $D[i..l]$  are LMCF's, and wlog assume  $j < l$ . Then  $D[i..j+1]$  is a factor of  $S$ , contradicting  $D[i..j]$ 's maximality.

Since  $D$  is of length  $n$  there are no more than  $n$  indices where an LMCF can start. ◀

► **Observation 4.** Let  $D[i_1..j_1]$  and  $D[i_2..j_2]$  be LMCF's of  $S$  in  $D$ . Then  $i_1 < i_2$  iff  $j_1 < j_2$ .

**Proof.** Otherwise, one is contained in the other contradicting the fact that both are LMCF's. ◀

To achieve our goal, we will show that we can update and maintain a sorted list of LMCF's in logarithmic time per substitution. One more tool is needed.

### 3.1 The Suffix Tree

The *suffix tree*  $\mathcal{T}(S)$  of a non-empty string  $S$  of length  $n$  is a compact trie representing all suffixes of  $S$ . The branching nodes of the trie as well as the terminal nodes, that correspond to suffixes of  $S$ , become *explicit* nodes of the suffix tree, while the other nodes are *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let  $\mathcal{L}(v)$  denote the *path-label* of a node  $v$ , i.e., the concatenation of the edge labels along the path from the root to  $v$ . We say that  $v$  is path-labelled  $\mathcal{L}(v)$ . Additionally,  $\mathcal{D}(v) = |\mathcal{L}(v)|$  is used to denote the *string-depth* of node  $v$ . Node  $v$  is a *terminal* node if its path-label is a suffix of  $S$ , that is,  $\mathcal{L}(v) = S[i..n]$  for some  $1 \leq i \leq n$ ; here  $v$  is also labelled with index  $i$ . It should be clear that each factor of  $S$  is uniquely represented by either an explicit or an implicit node of  $\mathcal{T}(S)$ , called its *locus*. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once  $\mathcal{T}(S)$  is constructed, it can be traversed in a depth-first manner to compute the string-depth  $\mathcal{D}(v)$  for each node  $v$ . For every two nodes,  $v, u$ , where  $\mathcal{L}(u) = \alpha\mathcal{L}(v)$ ,  $\alpha \in \Sigma$ , there is a pointer from  $v$  to  $u$  called a *suffix link*. It is known that the suffix tree, with the suffix links, of a string of length  $n$ , over a fixed-sized ordered alphabet, can be computed in time and space  $\mathcal{O}(n)$  [16, 12, 14]. The suffix tree of an integer alphabet can also be built in linear time [5].

### 3.2 Preprocessing

Our algorithm has two parts: a preprocessing part, to set up the data structures; and a query part that maintains the data structures upon every substitution in a manner allowing to efficiently pull a maximum LMCF at every instant.

#### Finding All LMCF's:

The first preprocessing step is finding all LMCF's. This is easily done via the suffix tree. Algorithm *AllLMCF(S, D)* computes, for each index  $i$ , the longest factor starting at  $D[i]$  that occurs in  $S$ . Then, starting from  $D[1]$ , output every factor whose ending position is farther than the previous one.

---

#### Algorithm 1: Algorithm AllLMCF(S, D) – Finding all LMCF's

---

1. Construct suffix tree  $\mathcal{T}(S\$D)$  for the string  $S\$D$ , where  $\$ \notin \Sigma$ .
  2. For every leaf, indicate whether it represents a suffix starting in  $S$  or a suffix starting in  $D$ .
  3. Traverse  $\mathcal{T}(S\$D)$  (using, e.g., DFS) and mark as *blue* every node that has in its subtree at least one leaf from  $D$  and one leaf from  $S$ .
  4. For every leaf representing suffix  $D[i]$ , mark the lowest blue node on its path from the root, i.e. the ending index of the longest factor that starts at  $D[i]$  and appears in  $S$ . Denote it by  $LD[i]$ .
  5. Set  $F$  to be  $LD[1]$ .  $\{F$  will be the farthest common factor so far.  $\}$   
 $\langle 1, LD[1] \rangle$  is an LMCF.
  6. For  $i=2$  to  $n$  do:  
 If  $LD[i] > F$  then  $\langle i, LD[i] \rangle$  is an LMCF and  $LD[i]$  is the new value of  $F$ .  
 endFor
-

**Correctness:** The algorithm simply follows the definition of LMCF.

**Time:** The construction and size of the suffix tree, as well as the tree traversals, are done in time  $\mathcal{O}(n)$ .

Let us now examine the effects on the LMCF's of a substitution of  $\omega$  for the symbol in index  $k$  of  $D$ . Clearly, all LMCF's that end before index  $k$  and all those that start after index  $k$  are not affected. Let  $CM_k$  be the set of LMCF's  $D[i..j]$  such that  $i \leq k \leq j$ . Since  $\omega \notin S$  then each of the LMCF's in  $CM_k$  is cut at index  $k$ . But because of local maximality, for any two LMCF's  $D[i_1..j_1]$  and  $D[i_2..j_2]$  where  $i_1 < i_2 \leq k \leq j_1 < j_2$ , only  $D[i_1..k-1]$  and  $D[k+1..j_2]$  are potentially LMCF's after the substitution in  $k$ . We can conclude:

► **Observation 5.** *After a replacement of  $\omega$  in index  $k$  of  $D$ , if  $CM_k$  is empty, there is no change to the LMCF's. Otherwise, all LMCF's that are elements in  $CM_k$  should be deleted and the following two strings should be inserted to the set of LMCF's:*

$$L_1 = D[i'..k-1], \text{ where } i' = \min\{i \mid D[i, j] \in CM_k\}$$

$$L_2 = D[k+1..jw], \text{ where } j' = \max\{j \mid D[i, j] \in CM_k\}$$

We now have an idea of what our algorithm should look like. We need a data structure that allows us to efficiently delete the appropriate sets of LMCF's, to add the two new LMCF's, if necessary, and to efficiently find the maximum. In particular, let LMCF  $D[i..j]$  be represented by the pair  $\langle i, j \rangle$ . We need a data structure that supports the following operations:

1. Construct LMCF structure
2. Given index  $k$ , Delete  $CM_k$
3. Insert LMCF  $\langle i, j \rangle$
4. Find maximum length LMCF.

#### 4 Implementation 1: $\mathcal{O}(\log n)$ Query Processing

Consider a balanced binary search tree  $T$  of the pairs  $\langle i, j \rangle$  that represent LMCF's, sorted by the smaller index in every pair. Given a substitution index  $k$ , we can efficiently find the subtree of all indices  $i$  for which  $i \leq k$ . The problem is that some of the LMCF's in this subtree are not cut by  $k$ , i.e. their second index  $j$  has  $j < k$ , thus they should not be deleted. However, Observation 4 guarantees that the second indices of all LMCF's are *sorted by the same order as the first indices*. This means that the binary search tree allows us to find  $i_1$ , the smallest  $i$  for which  $D[i, j]$  is cut by  $k$ . Similarly, we can find  $i_2$ , the largest  $i$  for which  $D[i, j]$  is cut by  $k$ . We delete all the LMCF's starting between  $i_1$  and  $i_2$  and add  $L_1$  and  $L_2$ . We will say a few words about maintaining the tree balanced.

As for the maximum. For each node in the tree write the maximum length LMCF in its subtree. Updating the maximum after every change means running up the tree, thus the time is  $\mathcal{O}(\log n)$ . Answering a query means starting from the root and running down the tree towards the maximum, again having time  $\mathcal{O}(\log n)$ .

**Implementation Details:** There are  $\mathcal{O}(n)$  LMCF's in total, and each one is a pair  $\langle i, j \rangle$  starting at a unique  $i$ ,  $1 \leq i \leq n$ . Therefore, we use the balanced binary tree of  $\{1, \dots, n\}$ , but for every node we indicate (a) whether the node is "full", i.e., whether there is an LMCF that starts at the node of index  $i$ , and (b) whether the subtree rooted at node  $i$  is empty. Since the height of the tree is  $\mathcal{O}(\log n)$ , the search is bounded by the tree height even in the tree with "empty" nodes.

**Time:**

1. **Preprocessing:** The balanced tree for  $\{1, \dots, n\}$  can be constructed in linear time. The LMCF's can be found on the suffix tree as shown in Subsection 3.2 in linear time. The appropriate nodes are marked and denoted “full” and the rest of the nodes as well as the appropriate subtrees are marked “empty” in linear time using DFS. Every LMCF node also has a “length” field and every subtree maintains in its root the maximum length in the subtree. This is also done by DFS in linear time.
2. **Deleting  $CM_k$ :** The  $j$ 's of the  $\langle i, j \rangle$  LMCF's are sorted in the same order as the  $i$ 's. Thus finding the largest and smallest  $i$  for which  $D[i, j]$  includes  $k$  is done in time  $\mathcal{O}(\log n)$ . The appropriate subtrees are marked “empty”. Because the tree is balanced this requires only  $\mathcal{O}(\log n)$  nodes to be marked. The appropriate maximum length fields along the modified paths are also updated appropriately, also in time  $\mathcal{O}(\log n)$ .
3. **Inserting LMCF  $\langle i, j \rangle$ :** simply turn on the appropriate “full” field of node  $i$  in the tree, as well as the appropriate  $j$  and length. Walk up the path updating the subtree emptiness indicators and the maximum subtree length.
4. **Finding Maximum Length LMCF:** The maximum length appears in the subtree max length field of the root, and can be output in constant time. All LCF's can be found by going down the tree. The time is then  $\mathcal{O}(tocc \log n)$ , where  $tocc$  is the number of LCF's. A single representative LCF can be found in time  $\mathcal{O}(\log n)$ .

**5 Implementation 2:  $\mathcal{O}(\log \log n)$  Query Processing**

The implementation we suggested to our algorithm was based on a balanced search tree. The height of such a tree is generally  $\mathcal{O}(\log n)$ . However, there are data structures that allow searching in time  $\mathcal{O}(\log \log u)$ , where  $u$  is the size of the universe of keys [15, 17]. Our set of keys is  $\{1, \dots, n\}$ , which would make the search time  $\mathcal{O}(\log \log n)$ . In this section we describe an implementation that uses a data structure that supports the following operations on a set  $S \subset \{1, \dots, n\}$ : *insert*, *delete*, *lookup*, *findnext*, *findprev*. Using such a data structure we show how to implement the four operations described in Section 3.

Constructing the tree, searching for a key, and inserting a constant number of LMCF's, can be done naturally on the vEB tree as well as the y-fast trie. The challenge is implementing the deletion of  $CM_k$  and the maintenance of the maximum lengths. The difficulty arises in the fact that these trees don't have a constant number of children per node, as does a balanced search tree, thus these updates are more complex.

**5.1 Data Structures**

As mentioned, we assume a data structure that supports the following operations on a set  $S \subset \{1, \dots, n\}$ : *insert*, *delete*, *lookup*, *findnext*, *findprev*. We call this a *fast tree*. The vEB tree [15] and the y-fast trie [17] can achieve this in space  $\mathcal{O}(n)$  and time per operation  $\mathcal{O}(\log \log n)$ . Nevertheless, our algorithm can use any dynamic priority queue with predecessor and successor query capability.

In order to implement our four operations, we will need a number of fast trees. We define them below.

**► Definition 6.**

1. The *LMCF tree* is a fast tree of the pairs  $\langle i, j \rangle$  that represent LMCF's, sorted by the smaller index in every pair.

2. The *Interval tree* is a fast tree sorted by the starting indices of *valid* intervals in the range  $\{1, \dots, n\}$ . Each such element also holds its length. Initially, the entire range is valid, so there is a single element starting at the first index and whose length is  $n$ , the length of the entire range. As our algorithm progresses, we may need to delete entire intervals of the range, in which no LMCF's start. We implement it by "cutting" them out of the interval tree.
3. The *max length* tree is a fast tree whose entries are the maximum length of the LMCF's in the valid intervals. There is also a link between each valid interval entry in the interval tree and the entry of its length in the max length tree. The max length tree is sorted by the length.

Our algorithm makes use of the *range maximum query* algorithm. The problem is defined as follows:

► **Definition 7.** The *Range Maximum Query (RMQ)* problem has as its input an array  $A[1..n]$  of natural numbers. We wish to preprocess the array in a manner that will enable efficient solution to:

**Query:** Given  $[i, j]$ , where  $1 \leq i \leq j \leq n$ . Return index  $k$ ,  $i \leq k \leq j$  such that  $A[k] \geq A[\ell]$ ,  $\forall \ell$  s.t.  $i \leq \ell \leq j$ .

**Time:** It was shown [8, 3, 6] that the RMQ problem can be solved using linear time and space preprocessing and constant query time.

**Space:** Since both the fast tree, and the RMQ preprocessing are done in linear space, our constructions uses linear space.

## 5.2 The Algorithm

We show the implementation of each of our operations using the fast trees.

**Preprocessing - Construct the fast trees:** We find the LMCF pairs as in Implementation 1 in Section 4. We construct two sets of fast trees: (a) *MFT1*: sorted by the starting position of the LMCF's, along with the length of each LMCF. The lengths are entered into an  $M$  array which is preprocessed for RMQ queries. The length of the  $m$  array is  $n$ .  $M[i]$  gets the value of the LMCF that starts in  $D[i]$ , if such an LMCF exists, otherwise  $M[i]$  gets the value  $-\infty$ . The Interval and the max length fast trees are initialized. (b) *MFT2* Sorted by the ending positions of the LMCF's. Each entry has its length. It comes with its accompanying interval and max length fast trees. *MFT2* is symmetric to *MFT1* and thus in the rest of the paper we will describe operations on *MFT1*. Similar operations are symmetrically done in *MFT2*.

**Time:** Construction time of the fast trees, and RMQ preprocessing:  $\mathcal{O}(n)$ .

**Given Index  $k$ , Delete  $CM_k$ :** Computing  $CM_k$  is straightforward in the LMCF fast trees. For any  $k$ , let  $j_1$  be the smallest entry larger than  $k$  in the *MFT2* set of trees. Initially, it is found in the LMCF tree. Subsequently, it is checked in the LMCF tree in the closest valid interval, as found in the interval fast tree.  $j$  represents an LMCF  $\langle i_1, j_1 \rangle$ . If  $i_1 < k$  then  $i_1$  is the starting position of the LMCF with the smallest index that is cut by  $k$ . Therefore

## 11:8 LMCF's for Dynamic String Algorithms

all LMCF's starting between indices  $i_1$  and  $k - 1$  should be removed and replaced by the new LMCF  $\langle i_1, k - 1 \rangle$ . Now, let  $i_2$  be the largest entry smaller than  $k$  in  $MFT1$ . It represents an LMCF  $\langle i_2, j_2 \rangle$ . If  $j_2 > k$  then a new LMCF  $\langle k + 1, j_2 \rangle$  needs to be added. Deleting an entry from a fast tree is simple. However, we need to efficiently delete many entries, as well as maintain the maximum. As mentioned before, we describe how to handle  $MFT1$ , the operations on  $MFT2$  are symmetrical.

Recall that the interval fast tree is initialized to the entire range. Assume that  $[i_1, k]$  is the first interval to be deleted, then the interval fast tree will have a node starting at the beginning of the interval and ending at  $i_1 - 1$ , a node starting at  $k + 1$  and ending at the end of the interval, and a node of length 1 at location  $i_1$ . In general, the interval fast tree only has non-overlapping intervals. Additionally, since LMCF's are cut at the insertion point, the following holds.

► **Observation 8.** *An interval is deleted from the interval fast tree only if it is entirely contained in a previous valid interval.*

Another crucial observation is the following:

► **Observation 9.** *Only intervals of length 1 (points) may be added and deleted in an interval that was declared "invalid".*

From the above two observations we get:

► **Conclusion 10.** *The interval tree is composed of intervals and points. Once an interval is deleted, the activity in the entire deleted interval consists only of adding and deleting single points.*

The deletion of an interval requires updating the maximum lengths of the remaining LMCF's. If the interval was a point, this is a single operation on the max length fast tree on the path of the change. If  $CM_k$  is an interval, then it caused a range change in the interval fast tree. We need to delete from the max length fast tree the maximum length of the range that is cut, and add the max length LMCF in the shortened range, as well as  $k - i_1$ , the length of the new LMCF. Note that because of Conclusion 10 the only non-point interval changes are the results of cuts in the initial ranges. But the initial ranges were preprocessed for RMQ queries. Consequently, we can, in time  $\mathcal{O}(1)$  update the max length fast tree.

**Time:** Handling points clearly takes time  $\mathcal{O}(\log \log n)$  since these are regular fast tree operations. Deleting an interval of LMCF's consists of a fast tree operation. which takes time  $\mathcal{O}(\log \log n)$ . Similarly, updating the max length fast tree takes time  $\mathcal{O}(\log \log n)$ , for a total of  $\mathcal{O}(\log \log n)$  time.

**Insert LMCF:** Done at the LMCF tree and each of the interval and max length trees.

**Time:**  $\mathcal{O}(\log \log n)$  on the fast trees.

**Find Maximum Length LMCF:** Find the maximum element in the root max length fast tree.

**Time:**  $\mathcal{O}(\log \log n)$ .

## 6 Dynamic LCF for a Static Periodic String

The LMCF's are an efficient tool for handling other dynamic versions of the problem. Our next result is an algorithm for the fully dynamic case. The changes to the text may replace a character in index  $i$  of  $D$  with some other character in  $\Sigma$ . We still assume that  $S$  is static and  $D$  is dynamic, however, we assume that  $S$  is a periodic string whose period length is  $p$ .

► **Definition 11.** Let  $S$  be a string of length  $n$ .  $S$  is called *periodic* if  $S = P^i \text{pref}(P)$ , where  $i \in \mathbb{N}$ ,  $i \geq 1$ ,  $P$  is a substring of  $S$  such that  $|P| < n$ ,  $P^i$  is the concatenation of  $P$  to itself  $i$  times, and  $\text{pref}(P)$  is a prefix of  $P$ . The smallest such substring  $P$  is called the *period* of  $S$ . If  $S$  is not periodic it is called *aperiodic*.

► **Remark.** Throughout the paper we use  $p$  to denote a period length and  $P$  the period string, i.e.,  $p = |P|$ .

Formally our problem is:

PERIODIC DYNAMIC LCF

**Input:** Two strings  $S$  and  $D$  of length  $n$  over an alphabet  $\Sigma$ ,  $S$  is periodic with period length  $p$ .

Let  $\langle i_1, \sigma_1 \rangle, \langle i_2, \sigma_2 \rangle, \dots, \langle i_k, \sigma_k \rangle$  be a sequence of substitution operations in  $D$ , where the symbol  $D[i_j]$  is replaced by  $\sigma_j \in \Sigma$ ,  $j = 1, \dots, k$ , and let  $D'$  be the result of these  $k$  substitutions.

**Output:** An LCF of  $S$  and  $D'$ .

Our algorithm has linear preprocessing time and takes time  $\mathcal{O}(p \log \log n)$  for a substitution and LCF query.

### 6.1 Algorithm's Idea

The periodic static string algorithm also maintain the LMCF's and their maximum length, as the deremental algorithm did. In order to limit the maintenance time at every substitution, we need to prove some properties of LMCF's in a periodic string.

► **Observation 12.** *Every substring of  $S$  whose length is larger than  $p$  also has a period of size  $p$ . In particular this, of course, applies to the LMCF's of  $S$  in  $D$ .*

► **Lemma 13** (Periodicity unity). *Let  $D_1 = \langle i_1, j_1 \rangle$  and  $D_2 = \langle i_2, j_2 \rangle$  be two LMCFs of  $S$  in  $D$ . If the length of the overlap of  $D_1$  and  $D_2$  is at least  $p$  then  $D_1 = D_2$  ( $i_1 = i_2$  and  $j_1 = j_2$ ).*

**Proof.** Let  $D_1 = D[i_1..j_1]$  and  $D_2 = D[i_2..j_2]$  be two LMCF's of  $S$  in  $D$  s.t the overlap of  $D_1$  and  $D_2$  is an interval of at least  $p$  characters. We assume, wlog, that  $i_1 \leq i_2$ . That means that  $i_1 \leq j_1 - p$ ,  $i_2 \leq j_2 - p$  and, because the overlap is of length at least  $p$ ,  $i_2 \leq j_1 - p$ . As substrings of  $S$ , both  $D_1$  and  $D_2$  have a period of size  $p$ . Let  $S[i_3..j_3]$  be an instance of  $D_1$  in  $S$ . According to the local maximum property of  $D_1$  - it must be satisfied that  $D[j_1 + 1] \neq S[j_3 + 1]$ . Otherwise  $D_1$  could have been extended.  $S$  has a period of size  $p$  so  $S[j_3 + 1] = S[j_3 + 1 - p]$ . The index  $j_1 + 1 - p$  is still within the range of  $D_1$  because  $i_1 \leq j_1 - p$ . so  $S[j_3 + 1 - p] = D[j_1 + 1 - p]$ . The index  $j_1 + 1 - p$  is also within the Range of  $D_2$  because  $i_2 \leq j_1 - p$ . **Assuming that  $D_1 \neq D_2$ ,** The index  $j_1 + 1$  is within the range of  $D_2$  as well because  $j_2 > j_1$  (Otherwise,  $D_2$  is fully contained in  $D_1$ ). On top of all,  $D_2$  is a common factor of  $S$  that is larger than  $p$ . so it has a period of size  $p$  as well and it satisfies:  $D[j_1 + 1 - p] = D[j_1 + 1]$ . According to transitivity :  $D[j_1 + 1] = S[j_3 + 1]$ , in contradiction to  $D_1$ 's local maximum property. ◀



► **Lemma 14** (Periodicity singular extension). *Let  $D_1 = D[i..j]$  be a common factor of  $D$  and  $S$  of length greater than  $p$ . Then  $D[i..j+1]$  is also a factor of  $S$  iff  $D[j+1] = D[j+1-p]$ .*

**Proof.**  $\Rightarrow$  Assume  $D[i..j+1]$  is a factor of  $S$ . Its length is greater than  $p$ , therefore it has a period of size  $p$ .  $D_1$  has length greater than  $p$  so  $j-p+1$  is within its range. From the indexes presence in the interval and the period we get :  $D[j+1] = D[j+1-p]$ .

$\Leftarrow$  Assume  $D[j+1] = D[j+1-p]$ . Let  $S[i_3..j_3]$  be an instance of  $D_1$  in  $S$ . With the same reasoning as in the proof of Lemma 13 we get that  $D[j+1-p] = S[j_3+1-p] = S[j_3+1]$ . The final conclusion is derived from transitivity. ◀

**Note:** Both proofs assume that there is an index  $j_3+1$  in  $S$ , which is not necessarily true. However, every substring starting after the first  $p$  letters of  $S$  is equal to a substring that begins in the first  $p$  symbols, and thus there is an instance that can be extended to  $j_3+1$ . There are only at most  $p$  possible substrings where this shift can not be done - those that already start within the first  $p$  symbols of  $S$  and extend all the way to the end. The lemmas don't hold for these strings, but there are only at most  $p$  of them and they are handled separately by the algorithm.

The above lemmas indicate that given a change in index  $x$  in  $D$ , the number of LMCF's that are affected by this change and are "far" from  $x$  is small. "Far" means starting or ending in an index whose distance from  $x$  exceeds  $\mathcal{O}(p)$ . The algorithm will handle "far" LMCF's and "close" LMCF's separately. There are only  $\mathcal{O}(p)$  "close" LMCF's, thus they can be handled in a brute force manner and still cost only  $\mathcal{O}(p)$  per query. The two lemmas guarantee a constant number of affected "far" LMCF's, so they also are handled efficiently.

## 7 The algorithm

**Preprocessing:** The preprocessing stage consists of finding all of the LMCF's, and putting them in a convenient data structure. The LMCF's are found using algorithm  $AllLMCF(S, D)$ , as presented in Subsection 3.2. The LMCF's are put in an efficient dynamic priority queue data structure, (e.g. the fast tree mentioned above) containing the indexes, sorted by the  $i$  value. Additionally, each node contains extra information about the maximum value of  $j-i$  in the subtree rooted in this node. Denote this priority queue by  $T$ .

**Handling an Edit Operation:** Assume a change is made at location  $x$  of  $D$ . We can apply algorithm  $AllLMCF$  on the area  $D[x-p..x+p]$  and, in time  $\mathcal{O}(p)$  get all the LMCF's starting in that area and ending in  $D[x+p]$ . This is almost all we need. The only corrections necessary are: (1) LMCF's that started before  $D[x-p]$  and extended past  $D[x]$ ; (2) LMCF's that started before  $D[x-p]$  and ended at  $D[x-1]$  (maybe they need to be extended); and (3) new LMCF's that start at the interval  $D[x-p..x]$  and extend past  $D[x+p]$ .

Because of the Periodicity Unity Lemma, we know that there is at most one LMCF that starts before  $D[x-p]$  and reaches to or past  $D[x-1]$ . If it passed  $D[x]$  (Case (1) above), its endpoint should be replaced by  $x-1$ . If it ends at  $x-1$  (Case (2) above), then its endpoint should be extended. We can spend  $p$  time to see how much ahead it can be extended. If it can be extended by more than  $p$  positions, then by the Periodicity Unity Lemma, we can merge it with the previous LMCF that started at  $D[x+1]$ . This leaves us with Case (3) - all LMCF's that start at the interval  $D[x-p..x]$  and extend past  $D[x+p]$ . Again, due to the Periodicity Unity Lemma, there is at most one such LMCF. We merge it with the old LMCF that started at  $D[x+1]$ . All old LMCF's that started in  $D[x-p..x]$  are deleted. The total number of changes is  $\mathcal{O}(p)$  and, for each change the maximum length in the subtree is



---

**Algorithm 2:** pseudocode for algorithm UpdateLMCF
 

---

**Algorithm UpdateLMCF( $x, \alpha$ ) – substitute location  $D[x]$  with  $\alpha$** 

1. Update  $D : D[x] \leftarrow \alpha$ .
2. Find a pair  $\pi_1 = \langle i_1, j_1 \rangle$  in  $T$  that satisfies  $j_1 \geq x - 1$  and  $i_1 \leq x - p$ . If there isn't one:  $\pi_1 \leftarrow \text{null}$ .
3. Find a pair  $\pi_2 = \langle i_2, j_2 \rangle$  in  $T$  that satisfies  $i_2 \leq x + 1$  and  $j_2 \geq x + p$ . If there isn't one:  $\pi_2 \leftarrow \text{null}$ .
4. Set two binary flags  $f_i, i \in \{1, 2\}$ .  $f_i = 1 \iff \pi_i = \text{null}$ .
5. For  $c = 0, 1, 2 \dots p$  do:
  - a. If  $D[x + c] \neq D[x + c - p]$  and  $f_1 = 0$  : Remove  $\pi_1$  from  $T$  and add the pair  $\langle i_1, x + c - 1 \rangle$ . Then set  $f_1 = 1$ .
  - b. If  $D[x - c] \neq D[x - c + p]$  and  $f_2 = 0$  : Remove  $\pi_2$  from  $T$  and Add the pair  $\langle x - c + 1, j_2 \rangle$ . Then set  $f_2 = 1$ .
6. If both flags are 0 in the end of the loop: Remove both  $\pi_1$  and  $\pi_2$  from  $T$  and add  $\langle i_1, j_2 \rangle$ .
7. Remove from  $T$  all the pairs  $\langle i, j \rangle$  s.t  $i \geq x - p$  and  $j \leq x + p$ .
8. Use  $ALLLMCF(S[1..p]^3, D[x - p..x + p])$  to get all the LMCF's contained in this interval. Add all the new LMCF's found to  $T$ . Except the one with the minimal  $i$  value and the one with the maximal  $j$  value, denoted as  $\pi_{left}$  and  $\pi_{right}$  respectively.
9. Check if there is an LMCFs that contains  $\pi_{right}$  in  $T$  (smaller  $i$  value and greater  $j$  value). If there is not then add  $\pi_{right}$  to  $T$ . Do the same for  $\pi_{left}$ .

**end Algorithm**


---

maintained in  $\mathcal{O}(\log \log n)$  time, for the fast tree data structure used. A pseudocode of the algorithm can be found in Algorithm 2.

## 7.1 Correctness

As previously indicated, the algorithm handles two types of LMCF's separately - "far" LMCFs and "close" ones. A far LMCF is an LMCF that is cut (or touched) by  $x$ , the location of the edit operation, but the difference between  $x$  and either  $i$  or  $j$  is at least  $p$ . An important observation is that according to the Periodicity Unity Lemma, there is no more than one such possible LMCF from each side of  $x$  (left and right).

Consequently, our algorithm finds the, possibly, single "far" LMCF from each side of  $x$  and figures out how it should be modified after  $D$  has changed.

Another useful property of the "far" LMCF's is that even if the edit operation cuts them - they are still at least of size  $p$  in the updated  $D$ . That property enables the use of the Singular Extension Lemma to check how far they extend in the modified  $D$ .

The final observation to be made in dealing with the "far" LMCF's is that checking  $p + 1$  matches after (or before)  $x$  is enough. If a mismatch was found - that's as far as the LMCF can extend (Singular Extension Lemma). If the  $p$  first letters after  $x$  match, that means that there were "far" LMCFs from both sides of  $x$ , and that after the edit operation, they overlap within an interval greater than  $p$ . That makes them the same LMCF due to the Periodicity Unity Lemma. At that point the algorithm will stop checking symbol by symbol. Rather, it will combine the two "far" LMCF's.

Handling the close LMCF's is done in straightforward way - using  $ALLLMCF$  on the small interval in which "close" LMCF's can be found. It is easy to observe that they must be a substring of  $S[1..p]^3$ .

## 11:12 LMCF's for Dynamic String Algorithms

The algorithm, without any modifications, actually answers a slightly different question from the one asked. It finds the LCF of the dynamic string  $D$  and some infinite period of the first  $p$  letters of  $S$ . If  $S$  is in size  $|D| + p$ , the questions are equivalent. Any other size of  $S$  will bring into play the issue mentioned in the note that follows the proof of the two lemmas in Subsection 6.1. For simplicity's sake we presented the algorithm with the assumption of the appropriate length of  $S$ . However, a slight adjustment, that can be done without changing the time complexity, can solve this problem. Every time we add to  $T$  an LMCF larger than  $n - p$ , which can only happen twice in a single change, we should check if it is an actual factor of  $S$  and fix it if it is not. This can be done by locating the first instance in  $S$  of the new LMCF and use its known size to detect if it is actually a factor. If it is not - then the starting index of  $S$  in the LMCF and its length can be used to deduce the way it should be partitioned, in  $\mathcal{O}(1)$  time.

### 7.2 Complexity

Lines 2 and 3 are standard priority queue searches. Since the size of the LMCF collection is bounded by  $n$ , they take  $\mathcal{O}(\log \log n)$  time.

The loop in line 5 repeats at most  $p$  times. The operations in every repeat are a constant amount of symbol comparison or priority queue manipulations, for a total of  $\mathcal{O}(p + \log \log n)$ .

Line 6 is also a priority queue manipulation.

Line 7 requires a few priority queue manipulations. There are no more than  $\mathcal{O}(p)$  LMCF's starting at that area so the total time complexity is  $\mathcal{O}(p \log \log n)$ .

Line 8 uses *AllLMCF* on two inputs of size  $\mathcal{O}(p)$ . that has  $\mathcal{O}(p)$  time complexity. Adding the  $\mathcal{O}(p)$  new LMCF's to the priority queue takes time  $\mathcal{O}(p \log \log n)$ .

Line 9 makes a constant amount of balanced tree searches and additions for a total of  $\mathcal{O}(\log \log n)$ .

Every change or addition causes a percolation up of the changes in the maximum LMCF length in the subtree. In a balanced search tree, this is  $\mathcal{O}(\log \log n)$  time per change.

**Total Query Time:**  $\mathcal{O}(p \log \log n)$ .

## 8 Conclusions

We have presented a tool - the LMCF - that enables efficient solutions to two variants of the dynamic longest common factor problem. In both variants we have a static string  $S$  and a dynamic string  $D$ . For the decremental case, i.e. where symbols of  $D$  are substituted by a new symbol not in the alphabet, the update time is  $\mathcal{O}(\log \log n)$ , and for the case where  $S$  is periodic with period  $p$ , the update time is  $\mathcal{O}(p \log \log n)$ . In both cases the preprocessing is linear.

Our algorithm is designed for strings over a constant-sized alphabet. However, with a  $\mathcal{O}(n \log n)$  pre-sorting of the strings, and converting to the alphabet  $\{1, \dots, n\}$ , the same algorithms will apply.

An open question is to extend this result to a fully dynamic case, that is, to propose a data structure that allows subsequent edit operations on one or both of the strings  $S$  and  $D$  for a general  $S$ , and reports the LCF after each operation in an efficient time complexity. Of course the ultimate challenge is a fully dynamic suffix tree algorithm. That problem seems hard. In the meanwhile it is important to consider dynamic versions of specific pattern matching problems. We believe that the LMCF idea can prove useful in other dynamic string algorithms as well.

---

**References**

---

- 1 A. Amir, P. Charalampopoulos, C.S. Iliopoulos, S.P. Pissis, and J. Radoszewski. Longest common factor after one edit operation. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS, pages 14–26. Springer, 2017. best paper award.
- 2 M.A. Babenko and T.A. Starikovskaya. Computing the longest common substring with one mismatch. *Probl. Inf. Transm.*, 47(1):28–33, 2011.
- 3 O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–229, 1994.
- 4 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
- 5 M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
- 6 J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 4009 in LNCS, pages 36–48. Springer-Verlag, 2006.
- 7 T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen. Longest common substrings with  $k$  mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015.
- 8 H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.
- 9 S. Grabowski. A note on the longest common substrings with  $k$  mismatches problem. *Information Processing Letters*, 115(6-8):640–642, 2015.
- 10 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 11 C.-A. Leimeister and B. Morgenstern. KMACS: the  $k$ -mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.
- 12 E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.
- 13 T. Starikovskaya. Longest common substrings with approximately  $k$  mismatches. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPICs*, pages 21:1–21:11, 2016.
- 14 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
- 15 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
- 16 P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
- 17 D. E. Willard. Log-logarithmic worst-case range queries are possible in space  $\theta(n)$ . *Information Processing Letters*, 17(2):81–84, 1983.



# Longest substring palindrome after edit

**Mitsuru Funakoshi**

Department of Physics, Kyushu University, Japan  
mitsuru.funakoshi@inf.kyushu-u.ac.jp


**Yuto Nakashima**

Department of Informatics, Kyushu University, Japan  
yuto.nakashima@inf.kyushu-u.ac.jp

**Shunsuke Inenaga**

Department of Informatics, Kyushu University, Japan  
inenaga@inf.kyushu-u.ac.jp

**Hideo Bannai**

Department of Informatics, Kyushu University, Japan  
bannai@inf.kyushu-u.ac.jp  
 <https://orcid.org/0000-0002-6856-5185>

**Masayuki Takeda**

Department of Informatics, Kyushu University, Japan  
takeda@inf.kyushu-u.ac.jp

---

## Abstract

It is known that the length of the longest substring palindromes (LSPals) of a given string  $T$  of length  $n$  can be computed in  $O(n)$  time by Manacher's algorithm [J. ACM '75]. In this paper, we consider the problem of finding the LSPal after the string is edited. We present an algorithm that uses  $O(n)$  time and space for preprocessing, and answers the length of the LSPals in  $O(\log(\min\{\sigma, \log n\}))$  time after single character substitution, insertion, or deletion, where  $\sigma$  denotes the number of distinct characters appearing in  $T$ . We also propose an algorithm that uses  $O(n)$  time and space for preprocessing, and answers the length of the LSPals in  $O(\ell + \log n)$  time, after an existing substring in  $T$  is replaced by a string of arbitrary length  $\ell$ .

**2012 ACM Subject Classification** Mathematics of computing → Combinatorial algorithms

**Keywords and phrases** maximal palindromes, edit operations, periodicity, suffix trees

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.12

## 1 Introduction

*Palindromes* are strings that read the same forward and backward. The problems of finding palindromes or palindrome-like structures in a given string are fundamental tasks in string processing, and thus have been extensively studied (e.g., see [2, 14, 8, 12, 16, 11, 15, 6] and references therein).

One of the earliest problems regarding palindromes is the *longest substring palindrome* (LSPal) problem, which asks to find (the length) of the longest palindromes that appear in a given string. This problem dates back to 1970's [13], and since then it has been popular as a good algorithmic exercise. Observe that the longest substring palindrome is also a maximal (non-extensible) palindrome in the string, whose center is an integer position if its length is odd, or a half-integer position if its length is even. Since one can compute the maximal palindromes for all such centers in  $O(n^2)$  total time by naïve character comparisons, the LSPal problem can also be easily solved in  $O(n^2)$  time.



© Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 12; pp. 12:1–12:14

Leibniz International Proceedings in Informatics



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

Manacher [13] gave an elegant  $O(n)$ -time solution to the LSPal problem. Manacher's algorithm uses symmetry of palindromes and character equality comparisons only, and therefore works in  $O(n)$  time for any alphabet. It was pointed out in [2] that Manacher's algorithm actually computes all the maximal palindromes in the string. In case where the input string is drawn from a constant size alphabet or an integer alphabet of size polynomial in  $n$ , there is an alternative suffix tree [19] based algorithm which takes  $O(n)$  time [9]. This algorithm also computes all maximal palindromes.

There is a simple  $O(n)$ -space data structure representing all of these computed maximal palindromes; simply store their lengths in an array of length  $2n - 1$  together with the input string  $T$ . However, this data structure is apparently not flexible for string edits, since even a single character substitution, insertion, or deletion can significantly break palindromic structures of the string. Indeed,  $\Omega(n^2)$  substring palindromes and  $\Omega(n)$  maximal palindromes can be affected by a single edit operation (E.g., consider to replace the middle character of string  $a^n$  with another character  $b$ ). Hence, an intriguing question is whether there exists a space-efficient data structure for the input string  $T$  which can quickly answer the following query: What is the length of the longest substring palindrome(s), if single character substitution, insertion, or deletion is performed? We call this as a *1-ELSPal query*.

In this paper, we present an algorithm which uses  $O(n)$  time and space for preprocessing and  $O(\log(\min\{\sigma, \log n\}))$  time for 1-ELSPal queries, where  $\sigma$  is the number of distinct characters appearing in  $T$ . We also consider a more general variant of 1-ELSPal queries, where an existing substring in the input string  $T$  can be replaced with a string of arbitrary length  $\ell$ , called an  *$\ell$ -ELSPal queries*. We present an algorithm which uses  $O(n)$  time and space for preprocessing and  $O(\ell + \log n)$  time for  $\ell$ -ELSPal queries. Our results are valid for string of length  $n$  over an integer alphabet of size polynomial in  $n$ . All bounds in this paper are in the worst case unless otherwise stated.

## Related work

This line of research was recently initiated by Amir et al. [1] for the *longest common factor (LCF)* of two strings. For two strings  $S$  and  $T$  of length at most  $n$ , they proposed a data structure of  $O(n \log^3 n)$  space which answers in  $O(\log^3 n)$  time the length of the LCF of  $S$  and the string  $T'$  obtained by a single character edit operation on  $T$ . Their data structure can be constructed in  $O(n \log^4 n)$  expected time.

## 2 Preliminaries

Let  $\Sigma$  be the *alphabet*. An element of  $\Sigma^*$  is called a *string*. The length of a string  $T$  is denoted by  $|T|$ . The empty string  $\varepsilon$  is a string of length 0, namely,  $|\varepsilon| = 0$ . For a string  $T = xyz$ ,  $x$ ,  $y$  and  $z$  are called a *prefix*, *substring*, and *suffix* of  $T$ , respectively. For two strings  $X$  and  $Y$ , let  $lcp(X, Y)$  denote the length of the longest common prefix of  $X$  and  $Y$ .

For a string  $T$  and an integer  $1 \leq i \leq |T|$ ,  $T[i]$  denotes the  $i$ -th character of  $T$ , and 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  $i > j$ . An integer  $p \geq 1$  is said to be a *period* of a string  $T$  iff  $T[i] = T[i + p]$  for all  $1 \leq i \leq |T| - p$ .

The *run length (RL) factorization* of a string  $T$  is a sequence  $f_1, \dots, f_m$  of maximal runs of the same characters such that  $T = f_1 \cdots f_m$  (namely, each RL factor  $f_j$  is a repetition of the same character  $a_j$  with  $a_j \neq a_{j+1}$ ). For each position  $1 \leq i \leq n$  in  $T$ , let  $RLFBeg(i)$  and  $RLFEnd(i)$  denote the beginning and ending positions of the RL factor that contains the position  $i$ , respectively. One can easily compute in  $O(n)$  time the RL factorization of string  $T$  of length  $n$  together with  $RLFBeg(i)$  and  $RLFEnd(i)$  for all positions  $1 \leq i \leq n$ .

Let  $T^R$  denote the reversed string of  $T$ , i.e.,  $T^R = T[|T|] \cdots T[1]$ . A string  $T$  is called a *palindrome* if  $T = T^R$ . For any non-empty substring palindrome  $T[i..j]$  in  $T$ ,  $\frac{i+j}{2}$  is called its *center*. It is clear that for each center  $q = 1, 1.5, \dots, n - 0.5, n$ , we can identify the maximal palindrome  $T[i..j]$  whose center is  $q$  (namely,  $q = \frac{i+j}{2}$ ). Thus, there are exactly  $2n - 1$  maximal palindromes in a string of length  $n$ .

Let  $PrePals(T)$  and  $SufPals(T)$  denote the sets of prefix palindromes and suffix palindromes of  $T$ , respectively. A non-empty substring palindrome  $T[i..j]$  is said to be a *maximal palindrome* of  $T$  if  $T[i - 1] \neq T[j + 1]$ ,  $i = 1$ , or  $j = |T|$ . Clearly, prefix palindromes and suffix palindromes of  $T$  are maximal palindromes of  $T$ .

A *rightward longest common extension (rightward LCE)* query on a string  $T$  is to compute  $lcp(T[i..|T|], T[j..|T|])$  for given two positions  $1 \leq i \neq j \leq |T|$ . Similarly, a *leftward LCE* query is to compute  $lcp(T[1..i]^R, T[1..j]^R)$ . We denote by  $RightLCE_T(i, j)$  and  $LeftLCE_T(i, j)$  rightward and leftward LCE queries for positions  $1 \leq i \neq j \leq |T|$ , respectively. An *outward LCE* query is, given two positions  $1 \leq i < j \leq |T|$ , to compute  $lcp((T[1..i])^R, T[j..|T|])$ . We denote by  $OutLCE_T(i, j)$  an outward LCE query for positions  $i < j$  in the string  $T$ .

Manacher [13] showed an elegant online algorithm which computes all maximal palindromes of a given string  $T$  of length  $n$  in  $O(n)$  time. An alternative offline approach is to use outward LCE queries for  $2n - 1$  pairs of positions in  $T$ . Using the suffix tree [19] for string  $T\$T^R\#$  enhanced with a lowest common ancestor data structure [10, 17, 3], where  $\$$  and  $\#$  are special characters which do not appear in  $T$ , each outward LCE query can be answered in  $O(1)$  time. For any integer alphabet of size polynomial in  $n$ , preprocessing for this approach takes  $O(n)$  time and space [5, 9]. Let  $\mathcal{M}$  be an array of length  $2n - 1$  storing the lengths of maximal palindromes in increasing order of centers. For convenience, we allow the index for  $\mathcal{M}$  to be an integer or a half-integer from 1 to  $n$ , so that  $\mathcal{M}[i]$  stores the length of the maximal palindrome of  $T$  centered at  $i$ .

A palindromic substring  $P$  of a string  $T$  is called a *longest substring palindrome (LSPal)* if there are no palindromic substrings of  $T$  which are longer than  $P$ . Since any LSPal of  $T$  is always a maximal palindrome of  $T$ , we can find all LSPals and their lengths in  $O(n)$  time.

In this paper, we consider the three standard edit operations, i.e., insertion, deletion, and substitution of a character in the input string  $T$  of length  $n$ . Let  $T'$  denote the string after one of the above edit position was performed at a given position. A *1-edit longest substring palindrome* query (*1-ELSPal* query) is to answer (the length of) a longest palindromic substring of  $T'$ . In the next section, we will present an  $O(n)$ -time and space preprocessing scheme such that subsequent *1-ELSPal* queries can be answered in  $O(\log(\min\{\sigma, \log n\}))$  time. For any integer  $\ell \geq 0$ , an  *$\ell$ -block edit longest substring palindrome* query ( *$\ell$ -ELSPal* query), which is a generalization of the *1-ELSPal* query, asks (the length of) a longest palindromic substring of  $T''$ , where  $T''$  denotes the string after an interval (substring) of  $T$  is replaced by a string of length  $\ell$ . In the following section, we will propose an  $O(n)$ -time and space preprocessing scheme such that subsequent  *$\ell$ -ELSPal* queries can be answered in  $O(\ell + \log n)$  time. We remark that in both problems string edits are only given as *queries*, i.e., we do not explicitly rewrite the original string  $T$  into  $T'$  nor  $T''$  and  $T$  remains unchanged for further queries.

### 3 Algorithm for 1-ELSPal

In this section, we will show the following result:

► **Theorem 1.** *There is an algorithm for the 1-ELSPal problem which uses  $O(n)$  time and space for preprocessing, and answers each query in  $O(\log(\min\{\sigma, \log n\}))$  time for single character substitution and insertion, and in  $O(1)$  time for single character deletion.*



### 3.1 Periodic structures of maximal palindromes

Let  $T$  be a string of length  $n$ . For each  $1 \leq i \leq n$ , let  $MaxPalEnd_T(i)$  denote the set of maximal palindromes of  $T$  that end at position  $i$ . Let  $S_i = s_1, \dots, s_k$  be the sequence of lengths of maximal palindromes in  $MaxPalEnd_T(i)$  sorted in increasing order, where  $k = |MaxPalEnd_T(i)|$ . Let  $d_j$  be the progression difference for  $s_j$ , i.e.,  $d_j = s_{j+1} - s_j$  for  $1 \leq j < k$ . We use the following lemma which is based on periodic properties of maximal palindromes ending at the same position.

► **Lemma 2.**

- (i) For any  $1 \leq j < k$ ,  $d_{j+1} \geq d_j$ .
- (ii) For any  $1 < j < k$ , if  $d_{j+1} \neq d_j$ , then  $d_{j+1} \geq d_j + d_{j-1}$ .
- (iii)  $S_i$  can be represented by  $O(\log i)$  arithmetic progressions, where each arithmetic progression is a tuple  $\langle s, d, t \rangle$  representing the sequence  $s, s + d, \dots, s + (t - 1)d$  with common difference  $d$ .
- (iv) If  $t \geq 2$ , then the common difference  $d$  is a period of every maximal palindrome which end at position  $i$  in  $T$  and whose length belongs to the arithmetic progression  $\langle s, d, t \rangle$ .

Each arithmetic progression  $\langle s, d, t \rangle$  is called a *group* of maximal palindromes. Similar arguments hold for the set  $MaxPalBeg_T(i)$  of maximal palindromes of  $T$  that begin at position  $i$ .

To prove Lemma 2, we use arguments from the literature [2, 7, 14]. Let us for now consider any string  $W$  of length  $m$ . In what follows we will focus on suffix palindromes in  $SufPals(W)$  and discuss their useful properties. We remark that symmetric arguments hold for prefix palindromes in  $PrePals(W)$  as well. Let  $S' = s'_1, \dots, s'_{k'}$  be the sequence of lengths of suffix palindromes of  $S'$  sorted in increasing order, where  $k' = |SufPals(W)|$ . Let  $d'_j$  be the progression difference for  $s'_j$ , i.e.,  $d'_j = s'_{j+1} - s'_j$  for  $1 \leq j < k'$ . Then, the following results are known:

► **Lemma 3** ([2, 7, 14]).

- (A) For any  $1 \leq j' < k'$ ,  $d'_{j'+1} \geq d'_{j'}$ .
- (B) For any  $1 < j' < k'$ , if  $d'_{j'+1} \neq d'_{j'}$ , then  $d'_{j'+1} \geq d'_{j'} + d'_{j'-1}$ .
- (C)  $S'$  can be represented by  $O(\log m)$  arithmetic progressions, where each arithmetic progression is a tuple  $\langle s', d', t' \rangle$  representing the sequence  $s', s' + d', \dots, s' + (t' - 1)d'$  of lengths of  $t'$  suffix palindromes with common difference  $d'$ .
- (D) If  $t' \geq 2$ , then the common difference  $d'$  is a period of every suffix palindrome of  $W$  whose length belongs to the arithmetic progression  $\langle s', d', t' \rangle$ .

The set of suffix palindromes of  $W$  whose lengths belong to the same arithmetic progression  $\langle s', d', t' \rangle$  is also called a *group* of suffix palindromes. Clearly, every suffix palindrome in the same group has period  $d'$ , and this periodicity will play a central role in our algorithms.

We are ready to prove Lemma 2.

**Proof.** It is clear that  $MaxPalEnd_T(i) \subseteq SufPals(T[1..i])$ , namely,

$$MaxPalEnd_T(i) = \{s' \in SufPals(T[1..i]) \mid T[i - s'] \neq T[i + 1], i - s' = 1, \text{ or } i = n\}.$$

The case where  $i = n$  is trivial, and hence in what follows suppose that  $i < n$ . Let  $c = T[i + 1]$ , and for a group  $\langle s', d', t' \rangle$  of suffix palindromes let  $a = T[i - s']$  and  $b = T[i - s' - (t' - 1)d']$ , namely,  $a$  (resp.  $b$ ) is the character that immediately precedes the shortest (resp. longest) palindrome in the group (notice that  $a = b$  when  $t' = 1$ ). Then, it follows from Lemma 3 (D) that  $s', s' + d', \dots, s' + (t' - 2)d' \in MaxPalEnd_T(i)$  iff  $a \neq c$ . Also,



$s' + (t' - 1)d' \in \text{MaxPalEnd}_T(i)$  iff  $b \neq c$ . Therefore, for each group of suffix palindromes of  $T[1..i]$ , there are only four possible cases: (1) all members of the group are in  $\text{MaxPalEnd}_T(i)$ , (2) all members but the longest one are in  $\text{MaxPalEnd}_T(i)$ , (3) only the longest member is in  $\text{MaxPalEnd}_T(i)$ , or (4) none of the members is in  $\text{MaxPalEnd}_T(i)$ .

Now, it immediately follows from Lemma 3 that (i)  $d_{j+1} \geq d_j$  for  $1 \leq j < k$  and (ii)  $d_{j+1} \geq d_j + d_{j-1}$  holds for  $1 < j < k$ . Properties (iii) and (iv) also follow from the above arguments and Lemma 3.  $\blacktriangleleft$

For all  $1 \leq i \leq n$  we can compute  $\text{MaxPalEnd}_T(i)$  and  $\text{MaxPalBeg}_T(i)$  in total  $O(n)$  time: After computing all maximal palindromes of  $T$  in  $O(n)$  time, we can bucket sort all the maximal palindromes with their ending positions and with their beginning positions in  $O(n)$  time each.

### 3.2 Algorithm for substitutions

In what follows, we will present our algorithm to compute the length of the LSPals after single character substitution. Our algorithm can also return the occurrence of an LSPal.

Let  $i$  be any position in the string  $T$  of length  $n$  and let  $c = T[i]$ . Also, let  $T' = T[1..i-1]c'T[i+1..n]$ , i.e.,  $T'$  is the string obtained by substituting character  $c'$  for the original character  $c = T[i]$  at position  $i$ . To compute the length of the LSPals of  $T'$ , it suffices to consider maximal palindromes of  $T'$ . Those maximal palindromes of  $T'$  will be computed from the maximal palindromes of  $T$ .

The following observation shows that some maximal palindromes of  $T$  remain unchanged after character substitution at position  $i$ .

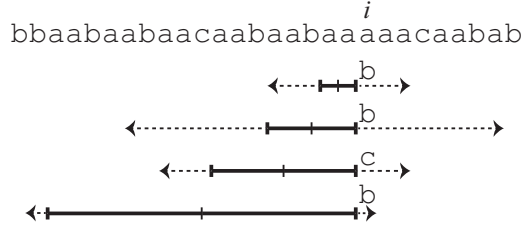
► **Observation 4** (Unchanged maximal palindromes after single character substitution). *For any position  $1 \leq j < i$ ,  $\text{MaxPalEnd}_{T'}(j) = \text{MaxPalEnd}_T(j)$ . For any position  $i < j \leq n$ ,  $\text{MaxPalBeg}_{T'}(j) = \text{MaxPalBeg}_T(j)$ .*

By Observation 4, for each position  $i$  ( $1 \leq i \leq n$ ) of  $T$ , we precompute the largest element of  $\bigcup_{1 \leq j < i} \text{MaxPalEnd}_T(j)$  and that of  $\bigcup_{i < j \leq n} \text{MaxPalBeg}_T(j)$ , and store the larger one in the  $i$ th position of an array  $\mathcal{U}$  of length  $n$ .  $\mathcal{U}[i]$  is a candidate for the solution after the substitution at position  $i$ . For each position  $i$ ,  $\bigcup_{1 \leq j < i} \text{MaxPalEnd}_T(j)$  contains the lengths of all maximal palindromes which end to the left of  $i$ , and  $\bigcup_{i < j \leq n} \text{MaxPalBeg}_T(j)$  contains the lengths of all maximal palindromes which begin to the right of  $i$ . Thus, by simply scanning  $\text{MaxPalEnd}_T(j)$  for increasing  $j = 1, \dots, n$  and  $\text{MaxPalBeg}_T(j)$  for decreasing  $j = n, \dots, 1$ , we can compute  $\mathcal{U}[i]$  for every position  $1 \leq i \leq n$ . Since there are only  $2n - 1$  maximal palindromes in string  $T$ , it takes  $O(n)$  time to compute the whole array  $\mathcal{U}$ .

Next, we consider maximal palindromes of the original string  $T$  whose lengths are extended in the edited string  $T'$ . As above, let  $i$  be the position where a new character  $c'$  is substituted for the original character  $c = T[i]$ . In what follows, let  $\sigma$  denote the number of distinct characters appearing in  $T$ .

► **Observation 5** (Extended maximal palindromes after single character substitution). *For any  $s \in \text{MaxPalEnd}_T(i-1)$ , the corresponding maximal palindrome  $T[i-s..i-1]$  centered at  $\frac{2i-s-1}{2}$  gets extended in  $T'$  iff  $T[i-s-1] = c'$ . Similarly, for any  $p \in \text{MaxPalBeg}_T(i+1)$ , the corresponding maximal palindrome  $T[i+1..i+p]$  centered at  $\frac{2i+p+1}{2}$  gets extended in  $T'$  iff  $T[i+p+1] = c'$ .*

► **Lemma 6.** *Let  $T$  be a string of length  $n$  over an integer alphabet of size polynomial in  $n$ . It is possible to preprocess  $T$  in  $O(n)$  time and space so that later we can compute in  $O(\log(\min\{\sigma, \log n\}))$  time the length of the longest maximal palindromes in  $T'$  that are extended after substitution of a character.*



■ **Figure 1** Example for Lemma 6, with string `bbaabaabaacaabaabaaaaacaabab` where the character `a` at position  $i = 20$  is to be substituted. There are four maximal palindromes ending at position 19, whose lengths are represented by two groups  $\langle 2, 3, 3 \rangle$  and  $\langle 17, 9, 1 \rangle$ . For the first group, `c` precedes the longest maximal palindrome and `b` precedes all the other maximal palindromes. The second group contains only one maximal palindrome and `b` precedes it. The largest extended lengths are 21 for `b`, and 14 for `c`. Thus we have  $\mathcal{E}_i = [(b, 21), (c, 14), (\hat{c}, 17)]$ , where 17 is the length of the longest maximal palindrome ending at position 19 in the original string.

**Proof.** By Observation 5, we consider maximal palindromes corresponding to  $MaxPalEnd_T(i - 1)$ . Those corresponding to  $MaxPalBeg_T(i + 1)$  can be treated similarly. Let  $\langle s, d, t \rangle$  be an arithmetic progression representing a group of maximal palindromes in  $MaxPalEnd_T(i - 1)$ . Let us assume that the group contains more than 1 member (i.e.,  $t \geq 2$ ) and that  $i - s \geq 2$ , since the case where  $t = 1$  or  $i - s = 1$  is easier to deal with. Let  $P_j$  denote the  $j$ th shortest member of the group, i.e.,  $P_1 = T[i - s..i - 1]$  and  $P_t = T[i - s - (t - 1)d..i - 1]$ . Then, it follows from Lemma 2 (iv) that if  $a$  is the character immediately preceding the occurrence of  $P_1$  (i.e.,  $a = T[i - s - 1]$ ), then  $a$  also immediately precedes the occurrences of  $P_2, \dots, P_{t-1}$ . Hence, by Observation 5,  $P_j$  ( $2 \leq j < t$ ) gets extended in the edited text  $T'$  iff  $c' = a$ . Similarly,  $P_t$  gets extended iff  $c' = b$ , where  $b$  is the character immediately preceding the occurrence of  $P_t$ . For each  $1 \leq j \leq t$  the final length of the extended maximal palindrome can be computed in  $O(1)$  time by a single outward LCE query  $OutLCE(i - s - (j - 1)d - 2, i + 1)$ . Let  $P'_j$  denote the extended maximal palindrome for each  $1 \leq j \leq t$ .

The above arguments suggest that for each group of maximal palindromes, there are at most two distinct characters that can extend those palindromes after single character substitution. For each position  $i$  in  $T$ , let  $\Sigma_i$  denote the set of characters which can extend maximal palindromes w.r.t.  $MaxPalEnd_T(i - 1)$  after character substitution at position  $i$ . It now follows from Lemma 2 and from the above arguments that  $|\Sigma_i| = O(\min\{\sigma, \log i\})$ . Also, when any character in  $\Sigma \setminus \Sigma_i$  is given for character substitution at position  $i$ , then no maximal palindromes w.r.t.  $MaxPalEnd_T(i - 1)$  are extended.

For each maximal palindrome  $P$  of  $T$ , let  $(i, c, l)$  be a tuple such that  $i$  is the ending position of  $P$ , and  $l$  is the length of the extended maximal palindrome  $P'$  after the immediately following character  $T[i + 1]$  is substituted for the character  $c = T[i - |P| - 1]$  which immediately precedes the occurrence of  $P$  in  $T$ . We then radix-sort the tuples  $(i, c, l)$  for all maximal palindromes in  $T$  as 3-digit numbers. This can be done in  $O(n)$  time since  $T$  is over an integer alphabet of size polynomial in  $n$ . Then, for each position  $i$ , we compute the maximum value  $l_c$  for each character  $c$ . Since we have sorted the tuples  $(i, c, l)$ , this can also be done in total  $O(n)$  time for all positions and characters. See Figure 1 for a concrete example.

Let  $\hat{c}$  be a special character which represents any character in  $\Sigma \setminus \Sigma_i$  (if  $\Sigma \setminus \Sigma_i \neq \emptyset$ ). Since no maximal palindromes w.r.t.  $MaxPalEnd_T(i - 1)$  are extended by  $\hat{c}$ , we associate  $\hat{c}$  with the length  $l_{\hat{c}}$  of the longest maximal palindrome w.r.t.  $MaxPalEnd_T(i - 1)$ . We assume that  $\hat{c}$  is lexicographically larger than any characters in  $\Sigma_i$ . For each position  $i$  we store pairs  $(c, l_c)$  in an array  $\mathcal{E}_i$  of size  $|\Sigma_i| + 1 = O(\min\{\sigma, \log i\})$  in lexicographical order of  $c$ . Then,

given a character  $c'$  to substitute for the character at position  $i$  ( $1 \leq i \leq n$ ), we can binary search  $\mathcal{E}_i$  for  $(c', l_{c'})$  in  $O(\log(\min\{\sigma, \log n\}))$  time. If  $c'$  is not found in the array, then we take the pair  $(\hat{c}, l_{\hat{c}})$  from the last entry of  $\mathcal{E}_i$ . We remark that  $\sum_{i=1}^n |\mathcal{E}_i| = O(n)$  since there are  $2n - 1$  maximal palindromes in  $T$  and for each of them at most two distinct characters contribute to  $\sum_{i=1}^n |\mathcal{E}_i|$ . ◀

Finally, we consider maximal palindromes of the original string  $T$  whose lengths are shortened in the edited string  $T'$  after substituting a character  $c'$  for the original character at position  $i$ .

► **Observation 7** (Shortened maximal palindromes after single character substitution). *A maximal palindrome  $T[b..e]$  of  $T$  gets shortened in  $T'$  iff  $b \leq i \leq e$ ,  $T[b + e - i] \neq c'$ , and  $i \neq \frac{b+e}{2}$ .*

► **Lemma 8.** *It is possible to preprocess a string  $T$  of length  $n$  in  $O(n)$  time and space so that later we can compute in  $O(1)$  time the length of the longest maximal palindromes of  $T'$  that are shortened after substitution of a character.*

**Proof.** Let  $\mathcal{S}$  be an array of length  $n$  such that  $\mathcal{S}[i]$  stores the length of the longest maximal palindrome that is shortened by the character substitution at position  $i$ . To compute  $\mathcal{S}$ , we preprocess  $T$  by scanning it from left to right. Suppose that we have computed  $\mathcal{S}[i]$ . By Observation 7, we have that  $\mathcal{S}[i] = 2(i - \frac{b+e+1}{2})$  where  $T[b..e]$  is the longest maximal palindrome of  $T$  satisfying the conditions of Observation 7. In other words,  $T[b..e]$  is the maximal palindrome of  $T$  of which the center  $\frac{b+e}{2}$  is the smallest possible under the conditions.

For any position  $i < i' \leq e$ , we have that  $\mathcal{S}[i'] = \mathcal{S}[i]$ . For the next position  $e + 1$ , we can compute  $\mathcal{S}[e + 1]$  in amortized  $O(1)$  time by simply scanning the array  $\mathcal{M}$  from position  $\frac{b+e+1}{2}$  to the right until finding the first (i.e., leftmost) entry of  $\mathcal{M}$  which stores the length of a maximal palindrome whose ending position is at least  $e + 1$ . Hence, we can compute  $\mathcal{S}$  in  $O(n)$  total time and space. ◀

Remark that maximal palindromes of  $T$  which do not satisfy the conditions of Observations 5 and 7 are also unchanged in  $T'$ . The following lemma summarizes this subsection:

► **Lemma 9.** *Let  $T$  be a string of length  $n$  over an integer alphabet of size polynomial in  $n$ . It is possible to preprocess  $T$  of length  $n$  in  $O(n)$  time and space so that later we can compute in  $O(\log(\min\{\sigma, \log n\}))$  time the length of the LSPals of the edited string  $T'$  after substitution of a character.*

### 3.3 Algorithm for deletions

Suppose the character at position  $i$  is deleted from the string  $T$ , and let  $T'_i$  denote the resulting string, namely  $T'_i = T[1..i - 1]T[i + 1..n]$ . Now the RL factorization of  $T$  comes into play: Observe that for any  $1 \leq i \leq n$ ,  $T'_i = T'_{RLFBeg(i)} = T'_{RLFEnd(i)}$ . Thus, it suffices for us to consider only the boundaries of the RL factors for  $T$ .

It is easy to see that an analogue of Observation 4 for unchanged maximal palindromes holds, as follows.

► **Observation 10** (Unchanged maximal palindromes after single character deletion). *For any position  $1 \leq j < RLFEnd(i)$ ,  $MaxPalEnd_{T'}(j) = MaxPalEnd_T(j)$ . For any position  $RLFBeg(i) < j \leq n$ ,  $MaxPalBeg_{T'}(j) = MaxPalBeg_T(j)$ .*



■ **Figure 2** Example for Observation 10. The maximal palindrome `aaaaabaaa` do not change if the character `a` at position  $i$  is deleted. The result is the same if the character `a` at position  $RLFEnd(i)$  is deleted.



■ **Figure 3** Example for Observation 11. The maximal palindrome `aaaaabaaa` gets extended to `bcaaaaabaaaaac` if the character `a` at position  $i$  is deleted. The result is the same if the character `a` at position  $RLFEnd(i)$  is deleted.



■ **Figure 4** Example for Observation 12. The maximal palindrome `caaaaabaaaaac` gets shortened to `aaaaabaaa` if the character `a` at position  $i$  is deleted. The result is the same if the character `a` at position  $RLFEnd(i)$  is deleted.

See Figure 2 for a concrete example of Observation 10.

By the above observation, we can compute the lengths of the longest unchanged maximal palindromes for the boundaries of all RL factors in  $O(n)$  time, in a similar way to the case of substitution.

Clearly the new character at position  $RLFEnd(i)$  in the string  $T'$  after deletion is always  $T[RLFEnd(i) + 1]$ , and a similar argument holds for  $RLFBeg(i)$ . Thus, we have the following observation for extended maximal palindromes after deletion, which is an analogue of Observation 5.

► **Observation 11** (Extended maximal palindromes after single character deletion). *For any  $s \in \text{MaxPalEnd}_T(RLFEnd(i) - 1)$ , the corresponding maximal palindrome  $T[RLFEnd(i) - s..RLFEnd(i) - 1]$  centered at  $\frac{2RLFEnd(i) - s - 1}{2}$  gets extended in  $T'$  iff  $T[RLFEnd(i) - s - 1] = T[RLFEnd(i) + 1]$ . Similarly, for any  $p \in \text{MaxPalBeg}_T(RLFBeg(i) + 1)$ , the corresponding maximal palindrome  $T[RLFBeg(i) + 1..RLFBeg(i) + p]$  centered at  $\frac{2RLFBeg(i) + p + 1}{2}$  gets extended in  $T'$  iff  $T[RLFBeg(i) + p + 1] = T[RLFBeg(i) - 1]$ .*

See Figure 3 for a concrete example for Observation 11.

Since the new characters that come from the left and the right of each deleted position are always unique, for each  $RLFEnd(i)$  and  $RLFBeg(i)$ , the longest maximal palindrome that gets extended after deletion is also unique. Overall, we can precompute their lengths for all positions  $1 \leq i \leq n$  in  $O(n)$  total time by using  $O(n)$  outward LCE queries in the original string  $T$ .

Next, we consider those maximal palindromes which get shortened after single character deletion. We have the following observation which is analogue to Observation 7.

► **Observation 12** (Shortened maximal palindromes after deletion). *A maximal palindrome  $T[b..e]$  of  $T$  gets shortened in  $T'$  iff  $b \leq RLFBeg(i)$  and  $RLFEnd(i) \leq e$ .*

See Figure 4 for a concrete example for Observation 12.

By Observation 12, we can precompute the length of the longest maximal palindrome after deleting the characters at the beginning and ending positions of each RL factors in  $O(n)$  total time, using an analogous way to Lemma 8.

Summing up all the above discussions, we obtain the following lemma:

► **Lemma 13.** *It is possible to preprocess a string  $T$  of length  $n$  in  $O(n)$  time and space so that later we can compute in  $O(1)$  time the length of the LSPals of the edited string  $T'$  after deletion of a character.*

### 3.4 Algorithm for insertion

Consider to insert a new character  $c'$  between the  $i$ th and  $(i + 1)$ th positions in  $T$ , and let  $T' = T[1..i]c'T[i + 1..n]$ . If  $c' \neq T[i]$  and  $c' \neq T[i + 1]$ , we can find the length of the LSPals in  $T'$  in a similar way to substitution. Otherwise (if  $c' = T[i]$  or  $c' = T[i + 1]$ ), then we can find the length of the LSPals in  $T'$  in a similar way to deletion since  $c'$  is merged to an adjacent RL factor. Thus, we have the following.

► **Lemma 14.** *Let  $T$  be a string of length  $n$  over an integer alphabet of size polynomial in  $n$ . It is possible to preprocess in  $O(n)$  time and space string  $T$  so that later we can compute in  $O(\log(\min\{\sigma, \log n\}))$  time the length of the LSPals of the edited string  $T'$  after insertion of a character.*

### 3.5 Hashing

By using hashing instead of binary searches on arrays, the following corollary is immediately obtained from Theorem 1.

► **Corollary 15.** *There is an algorithm for the 1-ELSPal problem which uses  $O(n)$  expected time and  $O(n)$  space for preprocessing, and answers each query in  $O(1)$  time for single character substitution, insertion, and deletion.*

## 4 Algorithm for $\ell$ -ELSPal

In this section, we consider the  $\ell$ -ELSPal problem where an existing block of length  $\ell'$  in the string  $T$  is replaced with a new block of length  $\ell$ . This generalizes substitution when  $\ell' > 0$  and  $\ell > 0$ , insertion when  $\ell' = 0$  and  $\ell > 0$ , and deletion when  $\ell' > 0$  and  $\ell = 0$ .

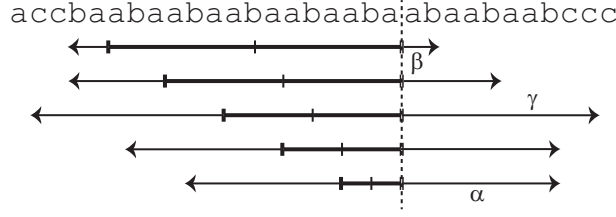
This section presents the following result:

► **Theorem 16.** *There is an  $O(n)$ -time and space preprocessing for the  $\ell$ -ELSPal problem such that each query can be answered in  $O(\ell + \log n)$  time, where  $\ell$  denotes the length of the block after edit.*

Note that the time complexity for our algorithm is independent of the length  $\ell'$  of the original block to edit. Also, the length  $\ell$  of a new block can be arbitrary.

Consider to substitute a substring  $X$  of length  $\ell$  for the substring  $T[i_b..i_e]$  beginning at position  $i_b$  and ending at position  $i_e$ , where  $i_e - i_b + 1 = \ell'$  and  $X \neq T[i_b..i_e]$ . Let  $T'' = T[1..i_b - 1]XT[i_e + 1..n]$  be the string after edit. For ease of explanation, we assume that there exist two positions  $j_1 < j_2$  in  $X$  such that  $j_1$  is the smallest position with  $T[i_b + j_1 - 1] \neq X[j_1]$  and  $j_2$  is the greatest position with  $T[i_e - \ell + j_2] \neq X[j_2]$ . The other cases (e.g.,  $X$  or  $T[i_b..i_e]$  is the empty string,  $j_1$  and  $j_2$  do not exist, or  $j_1 = j_2$ ) can be treated similarly. Given the above assumption, we can restrict ourselves to the case where the first and last characters

12:10 Longest substring palindrome after edit



■ **Figure 5** Example for Lemma 19, where  $Y = \text{accbaabaabaabaabaaba}$  and  $Z = \text{abaabaabccc}$ . Here we have  $\alpha = 8$ ,  $\beta = 2$ , and  $\gamma = 10$ .

of  $T[i_b..i_e]$  differ from those of  $X$ : Otherwise, then let  $p_b = \text{lcp}(T[i_b..i_e], X) = j_1 - 1$  and  $p_e = \text{lcp}((T[i_b..i_e])^R, X^R) = \ell - j_2$ . We can compute  $p_b$  and  $p_e$  in  $O(\ell - \hat{\ell} + 1)$  time by naive character comparisons, where  $\hat{\ell} = \ell - p_b - p_e = j_2 - j_1 + 1$ . Then, the above  $\ell$ -ELSPal query reduces to an  $\hat{\ell}$ -ELSPal query with edited string  $T[1..i_b + p_b]X[p_b + 1..\ell - p_e]T[i_e - p_e..n]$ .

We have the following observation for those of maximal palindromes in  $T$  whose lengths do not change, which is a generalization of Observation 4.

► **Observation 17** (Unchanged maximal palindromes after block edit). *For any position  $1 \leq j < i_b$ ,  $\text{MaxPalEnd}_{T''}(j) = \text{MaxPalEnd}_T(j)$ . For any position  $i_e < j \leq n$ ,  $\text{MaxPalBeg}_{T''}(j) = \text{MaxPalBeg}_T(j)$ .*

Hence, we can use the same  $O(n)$ -time preprocessing and  $O(1)$  queries as the 1-ELSPal problem: When we consider substitution for an existing block  $T[i_b..i_e]$ , we take the length of the longest maximal palindrome ending before  $i_b$  and that of the longest maximal palindrome beginning after  $i_e$  as candidates for a solution to the  $\ell$ -ELSPal query.

Next, we consider the maximal palindromes of  $T$  that get extended after block edit.

► **Observation 18** (Extended maximal palindromes after block edit). *For any  $s \in \text{MaxPalEnd}_T(i_b - 1)$ , the corresponding maximal palindrome  $T[i_b - s..i_b - 1]$  centered at  $\frac{2i_b - s - 1}{2}$  gets extended in  $T''$  iff  $\text{OutLCE}_{T''}(i_b - s - 1, i_b) \geq 1$ . Similarly, for any  $p \in \text{MaxPalBeg}_T(i_e + 1)$ , the corresponding maximal palindrome  $T[i_e + 1..i_e + p]$  centered at  $\frac{2i_e + p + 1}{2}$  gets extended in  $T''$  iff  $\text{OutLCE}_{T''}(i_e, i_e + p + 1) \geq 1$ .*

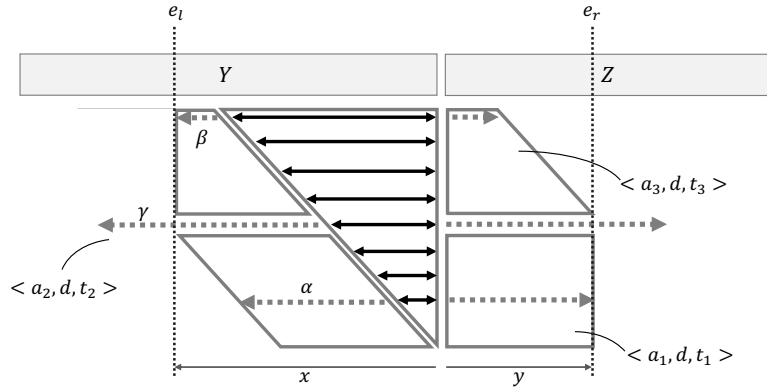
It follows from Observation 18 that it suffices to compute outward LCE queries efficiently for all maximal palindromes which end at position  $i_b - 1$  or begin at position  $i_e + 1$  in the edited string  $T''$ . However, there can be  $\Omega(n)$  maximal palindromes beginning or ending at each position of a string of length  $n$ . Yet, we can compute the length of the longest maximal palindromes that get extended after edit using periodic structures of maximal palindromes.

Let  $\langle s, d, t \rangle$  be an arithmetic progression representing a group of maximal palindromes ending at position  $i_b - 1$ . For each  $1 \leq j \leq t$ , let  $s_j$  denote the  $j$ th shortest element for  $\langle s, d, t \rangle$ , namely,  $s_j = s + (j - 1)d$ . For simplicity, let  $Y = T[1..i_b - 1]$  and  $Z = XT[i_e + 1..n]$ . Let  $\text{Ext}(s_j)$  denote the length of the maximal palindrome that is obtained by extending  $s_j$  in  $YZ$ .

► **Lemma 19.** *Let  $\alpha = \text{lcp}((Y[1..|Y| - s_1])^R, Z)$  and  $\beta = \text{lcp}((Y[1..|Y| - s_t])^R, Z)$ . If there exists  $s_h \in \langle s, d, t \rangle$  such that  $s_h + \alpha = s_t + \beta$ , then let  $\gamma = \text{lcp}((Y[1..|Y| - s_h])^R, Z)$ . Then, for any  $s_j \in \langle s, d, t \rangle \setminus \{s_h\}$ ,  $\text{Ext}(s_j) = s_j + 2 \min\{\alpha, \beta + (t - j)d\}$ . Also, if  $s_h$  exists, then  $\text{Ext}(s_h) = s_h + 2\gamma \geq \text{Ext}(s_j)$  for any  $j \neq h$ .*

See Figure 5 for a concrete example of Lemma 19.

Lemma 19 can be proven immediately from Lemma 12 of [14]. However, for the sake of completeness we here provide a proof. We use the following known result:



■ **Figure 6** Illustration for the proof of Lemma 19, where  $a_1 = s$ ,  $a_2 = s+t_1d$ , and  $a_3 = s+(t_1+t_2)d$ .

► **Lemma 20** ([14]). *For any string  $Y$  and  $\{s_j \mid s_j \in \langle s, d, t \rangle\} \subseteq \text{SufPals}(Y)$ , there exist palindromes  $u, v$  and a non-negative integer  $k$ , such that  $(uv)^{t+k-1}u$  is a suffix of  $Y$ ,  $|uv| = d$  and  $|(uv)^k u| = s$ .*

Now we are ready to prove Lemma 19 (see also Figure 6).

**Proof.** Let us consider  $\text{Ext}(s_j)$ , such that  $s_j \in \langle s, d, t \rangle$ . By Lemma 20,  $Y[|Y| - s_1 - (t - 1)d + 1] = (uv)^{t+k-1}u$ , where  $|uv| = d$  and  $|(uv)^k u| = s$ .

Let  $x$  be the largest integer such that  $(Y[|Y| - x + 1..|Y|])^R$  has a period  $|uv|$ . Namely,  $(Y[|Y| - x + 1..|Y|])^R$  is the longest prefix of  $Y^R$  that has a period  $|uv|$ . Then  $x$  is given as  $x = \text{lcp}(Y^R, (Y[1..|Y| - d])^R) + d$ . Let  $y$  be largest integer such that  $(uv)^{y/d}$  is a prefix of  $Z$ . Then  $y$  is given as  $y = \text{lcp}(Y^R, Z)$ .

Let  $e_l = |Y| - x + 1$  and  $e_r = |Y| + y$ . Then, clearly string  $T''[e_l..e_r]$  has a period  $d$ . We divide  $\langle s, d, t \rangle$  into three disjoint subsets as

$$\langle s, d, t \rangle = \langle s, d, t_1 \rangle \cup \langle s + t_1d, d, t_2 \rangle \cup \langle s + (t_1 + t_2)d, d, t_3 \rangle,$$

such that

- $|Y| - e_l - s_j + 1 > e_r - |Y|$  for any  $s_j \in \langle s, d, t_1 \rangle$ ,
  - $|Y| - e_l - s_j + 1 = e_r - |Y|$  for any  $s_j \in \langle s + t_1d, d, t_2 \rangle$ ,
  - $|Y| - e_l - s_j + 1 < e_r - |Y|$  for any  $s_j \in \langle s + (t_1 + t_2)d, d, t_3 \rangle$ ,
- and  $t_1 + t_2 + t_3 = t$ .

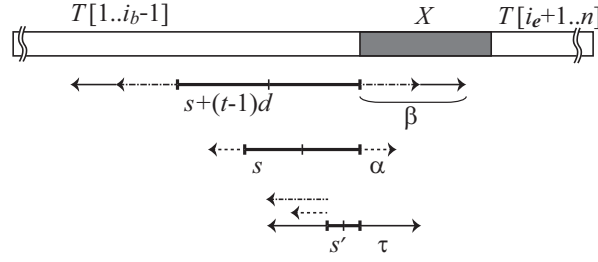
Then, for any  $s_j$  in the first sub-group  $\langle s, d, t_1 \rangle$ ,  $\text{Ext}(s_j) = s_j + 2(e_r - |Y|) = s_j + 2y$ . Also, for any  $s_j$  in the third sub-group  $\langle s + (t_1 + t_2)d, d, t_3 \rangle$ ,  $\text{Ext}(s_j) = s_j + 2(|Y| - e_l - s_j + 1) = s_j + 2(x - s_j)$ . Now let us consider  $s_j \in \langle a_2, d, t_2 \rangle$ , in which case  $s_j = s_h$  (see the statement of Lemma 19). Note that  $0 \leq t_2 \leq 1$ , and here we consider the interesting case where  $t_2 = 1$ . Since the palindrome  $s_h$  can be extended beyond the periodicity w.r.t.  $uv$ , we have  $\text{Ext}(s_h) = s_h + 2\gamma$ , where  $\gamma = \text{lcp}((Y[1..|Y| - s_h])^R, Z)$ .

Additionally, we have that  $y = \text{lcp}(Y^R, Z) = \text{lcp}((Y[1..|Y| - s_1])^R, Z) = \alpha$  where the second equality comes from the periodicity w.r.t.  $uv$ , and that  $x - s_j = \text{lcp}((Y[1..|Y| - s_t])^R, Z) + (t - j)d = \beta + (t - j)d$ . Therefore, for any  $s_j \in \langle s, d, t \rangle$ ,  $\text{Ext}(s_j)$  can be represented as follows:

$$\text{Ext}(s_j) = \begin{cases} s_j + 2\alpha & (\alpha < \beta + (t - j)d) \\ s_j + 2(\beta + (t - j)d) & (\alpha > \beta + (t - j)d) \\ s_j + 2\gamma & (\alpha = \beta + (t - j)d) \end{cases}$$

This completes the proof. ◀





■ **Figure 7** Illustration for Lemma 21, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. Here we consider the case where  $0 < \tau < \ell$ . To compute  $\alpha$ , we first perform a leftward LCE query. Here, the LCE value is less than  $\tau$  and thus it is  $\alpha$ . To compute  $\beta$ , we also perform a leftward LCE query. Here, the LCE value is at least  $\tau$ , and thus we perform naïve character comparisons to determine the remainder of  $\beta$ . Other cases can be treated similarly.

Due to Lemma 19, provided that  $\alpha$ ,  $\beta$ , and  $\gamma$  (if  $s_h$  exists) are already computed, then it is a simple arithmetic to calculate the length of the longest extended maximal palindrome from  $\langle s, d, t \rangle$  in  $T'' = YZ$ .

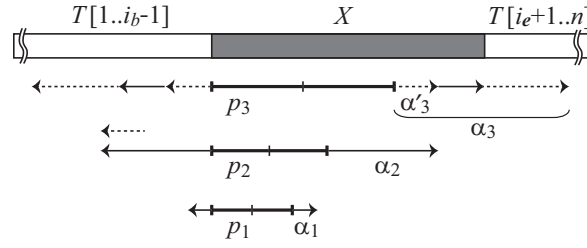
► **Lemma 21.** *Let  $T$  be a string of length  $n$  over an integer alphabet of size polynomially bounded in  $n$ . It is possible to preprocess  $T$  in  $O(n)$  time and space so that later we can compute in  $O(\ell + \log n)$  time the length of the longest maximal palindromes of  $T''$  that are extended after replacing an existing block with a new block of length  $\ell$ .*

**Proof.** Let  $\langle s, d, t \rangle$  be any arithmetic progression representing a group of  $\text{MaxPalEnd}_T(i_b - 1)$ , and  $\alpha$ ,  $\beta$ , and  $\gamma$  be the lcp values for this group as defined in Lemma 19. Suppose that we have already processed all groups of shorter maximal palindromes. Let  $s'$  be one of the already processed maximal palindromes which has the longest extension of length  $\tau$  (i.e.,  $s' + 2\tau$  is the length of the extended maximal palindrome for  $s'$ ). See also Figure 7. There are three cases: (1) If  $\tau = 0$ , then we compute  $\alpha$  by naïve character comparisons between  $(T[1..i_b - s - 1])^R$  and  $X$ . (2) If  $0 < \tau < \ell$ , then we first compute  $\delta = \text{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$ . (2-a) If  $\delta < \tau$ , then  $\alpha = \delta$ . (2-b) Otherwise ( $\delta \geq \tau$ ), then we know that  $\alpha$  is at least as large as  $\tau$ . We then compute the remainder of  $\alpha$  by naïve character comparisons. If the character comparison reaches the end of  $X$ , then the remainder of  $\alpha$  can be computed by  $\text{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$ . Then we update  $\tau$  with  $\alpha$ . (3) If  $\tau \geq \ell$ , then we can compute  $\alpha$  by  $\text{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$ , and if this value is at least  $\ell$ , then by  $\text{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$ .  $\beta$  and  $\gamma$  (if it exists) can also be computed similarly.

After processing all arithmetic progressions representing the groups for  $\text{MaxPalEnd}_T(i_b - 1)$ , the total number of matching character comparisons is at most  $\ell$  since each position of  $X$  is involved in at most one matching character comparison. Also, the total number of mismatching character comparisons is  $O(\log n)$  since for each arithmetic progression there are at most three mismatching character comparisons (those for  $\alpha$ ,  $\beta$ , and  $\gamma$ ). The total number of LCE queries in the original text  $T$  is  $O(\log n)$ , each of which can be answered in  $O(1)$  time. Thus, together with Lemma 19, it takes  $O(\ell + \log n)$  time to compute the length of the longest maximal palindromes of  $T''$  that are extended after block edit. ◀

► **Remark.** An alternative method to Lemma 21 would be to first build the suffix tree of  $T\#T^R\$$  enhanced with a dynamic lowest common ancestor data structure [4] using  $O(n)$  time and space [5], and then to update the suffix tree with string  $T\#T^R\$X\#'X^R\$'$  using Ukkonen's online algorithm [18], where  $\#'$  and  $\$'$  are special characters not appearing in  $T$





■ **Figure 8** Illustration for Lemma 24, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. Here are three prefix palindromes of  $X$  of length  $p_1$ ,  $p_2$ , and  $p_3$ . We compute  $\alpha_1$  naïvely. Here, since  $p_1 + \alpha_1 < p_2$ , we compute  $p_2$  naïvely. Since  $p_2 + \alpha_2 > p_3$ , we compute  $\text{LeftLCE}_T(i_b - 1, i_b - \alpha_2 + \alpha'_3 - 1)$ . Here, since its value reached  $\alpha'_3$ , we perform naïve character comparison for  $X[p_3 + \alpha'_3 + 1..\ell]$  and  $(T[1..i_b - \alpha'_3 - 1])^R$ . Here, since there was no mismatch, we perform  $\text{OutLCE}_T(i_b - \ell + p_3 - 1, i_e + 1)$  and finally obtain  $\alpha_3$ . Other cases can be treated similarly.

nor  $X$ . This way, one can answer LCE queries between any position of the original string  $T$  and any position of the new block  $X$  in  $O(1)$  time. Since we need  $O(\log n)$  LCE queries, it takes  $O(\log n)$  total time for all LCE queries. However, Ukkonen’s algorithm requires  $O(\ell \log \sigma)$  time to insert  $X\#X^R\$\ell$  into the existing suffix tree, where  $\ell = |X|$ . Thus, this method requires us  $O(\ell \log \sigma + \log n)$  time and thus is slower by a factor of  $\log \sigma$  than the method of Lemma 21.

Finally, we consider the maximal palindromes that get shortened after block edit.

► **Observation 22** (Shortened maximal palindromes after block edit). *A maximal palindrome  $T[b..e]$  of  $T$  gets shortened in  $T''$  iff  $b \leq i_b \leq e$  and  $i_b \neq \frac{b+e}{2}$ , or  $b \leq i_e \leq e$  and  $i_e \neq \frac{b+e}{2}$ .*

The difference between Observation 7 and this one is only in that here we need to consider two positions  $i_b$  and  $i_e$ . Hence, we obtain the next lemma using a similar method to Lemma 8:

► **Lemma 23.** *We can preprocess a string  $T$  of length  $n$  in  $O(n)$  time and space so that later we can compute in  $O(1)$  time the length of the longest maximal palindromes of  $T''$  that are shortened after block edit.*

Finally, we consider those maximal palindromes whose centers exist in the new block  $X$  of length  $\ell$ . By symmetric arguments to Observation 18, we only need to consider the prefix palindromes and suffix palindromes of  $X$ . Using a similar technique to Lemma 21, we obtain:

► **Lemma 24.** *We can compute the length of the longest maximal palindromes whose centers are inside  $X$  in  $O(\ell)$  time and space.*

**Proof.** First, we compute all maximal palindromes in  $X$  in  $O(\ell)$  time. Let  $p_1, \dots, p_u$  be a sequence of the lengths of the prefix palindromes of  $X$  sorted in increasing order. For each  $1 \leq j \leq u$ , let  $\alpha_j = \text{lcp}(X[p_j + 1..\ell], (T[1..i_b - 1])^R)$ , namely,  $p_j + 2\alpha_j$  is the length of the extended maximal palindrome for each  $p_j$ . Suppose we have computed  $\alpha_{j-1}$ , and we are to compute  $\alpha_j$ . See also Figure 8. If  $p_{j-1} + \alpha_{j-1} \leq p_j$ , then we compute  $p_j$  by naïve character comparisons. Otherwise, then let  $\alpha'_j = p_{j-1} + \alpha_{j-1} - p_j$ . Then, we can compute  $\text{lcp}(X[p_j + 1..p_j + \alpha'_j], (T[1..i_b - 1])^R)$  by a leftward LCE query in the original string  $T$ . If this value is less than  $\alpha'_j$ , then it equals to  $\alpha_j$ . Otherwise, then we compute  $\text{lcp}(X[p_j + \alpha'_j + 1..\ell], (T[1..i_b - 1])^R)$  by naïve character comparisons. The total number of matching character comparisons is at most  $\ell$  since each position in  $X$  can be involved in at most one matching character comparison. The total number of mismatching character

comparisons is also  $\ell$ , since there are at most  $\ell$  prefix palindromes of  $X$  and for each of them there is at most one mismatching character comparison. Hence, it takes  $O(\ell)$  time to compute the length of the longest maximal palindromes whose centers are inside  $X$ . ◀

---


## References

- 1 Amihoud Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *SPIRE 2017*, pages 14–26, 2017.
- 2 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141:163–173, 1995.
- 3 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000*, pages 88–94, 2000.
- 4 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.
- 5 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
- 6 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter bounds and optimal algorithms for all maximal  $\alpha$ -gapped repeats and palindromes - finding all maximal  $\alpha$ -gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018.
- 7 Leszek Gąsieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT1996)*, volume 1097 of *LNCS*, pages 392–403. Springer, 1996.
- 8 Richard Groult, Élise Prieur, and Gwénaél Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010.
- 9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- 10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- 11 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.
- 12 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Finding distinct subpalindromes online. In *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 63–69, 2013.
- 13 Glenn Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346–351, 1975.
- 14 W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8–10):900–913, 2009.
- 15 Shintaro Narisada, Diptarama, Kazuyuki Narisawa, Shunsuke Inenaga, and Ayumi Shinohara. Computing longest single-arm-gapped palindromes in a string. In *SOFSEM 2017*, pages 375–386, 2017.
- 16 Alexandre H. L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581–2591, 2002.
- 17 Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.
- 18 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- 19 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

# A Succinct Four Russians Speedup for Edit Distance Computation and One-against-many Banded Alignment


Brian Brubach<sup>1</sup>

Department of Computer Science, University of Maryland, College Park, MD 20742, USA  
bbrubach@cs.umd.edu

 <https://orcid.org/0000-0003-1520-2812>

Jay Ghurye<sup>2</sup>

Department of Computer Science, University of Maryland, College Park, MD 20742, USA  
jayg@cs.umd.edu

 <https://orcid.org/0000-0003-1381-4081>

---

## Abstract

The classical Four Russians speedup for computing edit distance (a.k.a. Levenshtein distance), due to Masek and Paterson [15], involves partitioning the dynamic programming table into  $k$ -by- $k$  square blocks and generating a lookup table in  $O(\psi^{2k}k^2|\Sigma|^{2k})$  time and  $O(\psi^{2k}k|\Sigma|^{2k})$  space for block size  $k$ , where  $\psi$  depends on the cost function (for unit costs  $\psi = 3$ ) and  $|\Sigma|$  is the size of the alphabet. We show that the  $O(\psi^{2k}k^2)$  and  $O(\psi^{2k}k)$  factors can be improved to  $O(k^2 \lg k)$  time and  $O(k^2)$  space. Thus, we improve the time and space complexity of that aspect compared to Masek and Paterson [15] and remove the dependence on  $\psi$ .

We further show that for certain problems the  $O(|\Sigma|^{2k})$  factor can also be reduced. Using this technique, we show a new algorithm for the fundamental problem of one-against-many banded alignment. In particular, comparing one string of length  $m$  to  $n$  other strings of length  $m$  with maximum distance  $d$  can be performed in  $O(nm + md^2 \lg d + nd^3)$  time. When  $d$  is reasonably small, this approaches or meets the current best theoretic result of  $O(nm + nd^2)$  achieved by using the best known pairwise algorithm running in  $O(m + d^2)$  time [17, 22] while potentially being more practical. It also improves on the standard practical approach which requires  $O(nmd)$  time to iteratively run an  $O(md)$  time pairwise banded alignment algorithm.

Regarding pairwise comparison, we extend the classic result of Masek and Paterson [15] which computes the edit distance between two strings in  $O(m^2/\log m)$  time to remove the dependence on  $\psi$  even when edits have arbitrary costs from a penalty matrix. Crochemore, Landau, and Ziv-Ukelson [8] achieved a similar result, also allowing for unrestricted scoring matrices, but with variable-sized blocks. In practical applications of the Four Russians speedup wherein space efficiency is important and smaller block sizes  $k$  are used (notably  $k < |\Sigma|$ ), Kim, Na, Park, and Sim [13] showed how to remove the dependence on the alphabet size for the unit cost version, generating a lookup table in  $O(3^{2k}(2k)!k^2)$  time and  $O(3^{2k}(2k)!k)$  space. Combining their work with our result yields an improvement to  $O((2k)!k^2 \lg k)$  time and  $O((2k)!k^2)$  space.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** edit distance, banded alignment, one-against-many alignment, genomics, method of the Four Russians

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.13

---

<sup>1</sup> Supported in part by NSF awards CCF-1422569 and CCF-1749864 as well as the NIH, grant R01-AI-100947 to Mihai Pop.

<sup>2</sup> Supported in part by the NRL, award N00173-16-2-C001 to Mihai Pop.



© Brian Brubach and Jay Ghurye;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 13; pp. 13:1–13:12

Leibniz International Proceedings in Informatics



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

**Acknowledgements** The authors wish to thank advisors Mihai Pop and Aravind Srinivasan.

## 1 Introduction

Edit distance (a.k.a. Levenshtein distance) is one of the most natural and ubiquitous measures of similarity between two strings. In the most common variant, *unit cost*, it counts the minimum number of edits needed to transform one string into another. Here, we use the Levenshtein definition of *edits* which include insertions, deletions, or substitutions of a single character. However, in some cases edit operations may be assigned differing costs from a penalty matrix and additional operations (e.g. inversions or transpositions) may be considered. Computing this distance is a fundamental problem with applications in many areas such as computation biology, natural language processing, and information theory.

The most well known algorithms use dynamic programming to solve the problem in  $O(m^2)$  time where  $m$  is the length of the strings. The only improvement to this has been the Four Russians algorithm [15], running in  $O(m^2/\log m)$  time. While the conditional hardness results, such as [3], suggest this is unlikely to be improved further for arbitrary strings even on small alphabets [5].

The problem of comparing a string against a large set of sequences is of central importance in domains such as computational biology, information retrieval, and databases. The banded alignment variant (a.k.a. the  $d$  differences approximate string matching problem), in which we only report the distance when it is at most some parameter  $d$  is also highly relevant. It's useful in numerous settings wherein we only care about finding small distances or the maximum distance between any two strings is known to be small. In gene clustering for example, solving this problem is a key subroutine in many *greedy* clustering heuristics wherein we iteratively choose a cluster center and form a cluster by recruiting all strings which are within some small maximum distance  $d$  of the center [6]. With the development of faster and cheaper DNA sequencing technologies, metagenomic sequencing datasets can contain over 1 billion sequences [7].

Another area of research surrounding the Four Russians speedup is how to apply it in practice. While the theoretical result uses a block size of  $\log m$ , such a large block size is impractical due the size of the lookup table exceeding hardware constraints. For the unit cost version, [13] showed how to drastically reduce the required space, especially for large alphabets, by avoiding redundant string comparisons. We show that our approach can be combined with theirs to reduce the space (and preprocessing time) requirement even further.

### 1.1 Related Work

The edit distance problem is extremely well-studied and the following related work is by no means exhaustive. We focus primarily on the aspects most related to this paper: pairwise comparison, the Four Russians speedup, and one-against-many comparison. For simplicity, we describe all results in the context wherein all strings have length exactly equal to  $m$ .

The most well-known approach for computing the edit distance between a pair of strings of length  $m$  uses dynamic programming and requires  $O(m^2)$ . This was later improved to  $O(m^2/\log m)$  in 1980 using the Four Russians speedup [15, 14] and [8] achieved  $O(m^2/\log m)$  for unrestricted scoring matrices. The Four Russians speedup, originally proposed for matrix multiplication, has been adapted to many problems besides edit distance including: RNA folding [10], transitive closure of graphs [20], and matrix inversion [4]. On the negative side, [3] recently showed that no algorithm for edit distance can do better than  $m^{2-\epsilon}$  time unless the Strong Exponential Time Hypothesis (SETH) is false and [5] extended this to

include strings on a binary alphabet. They accomplished this by reducing a satisfiability problem to edit distance and showing that a subquadratic algorithm for edit distance implies a subexponential algorithm for satisfiability. However, if we fix a maximum distance  $d$  and only care about reporting the exact distance when it's less than  $d$ , we call this the *banded alignment* problem. This problem has seen improvements to  $O(md)$  time [9] and the current best algorithm takes only  $O(m + d^2)$  time [17, 22].

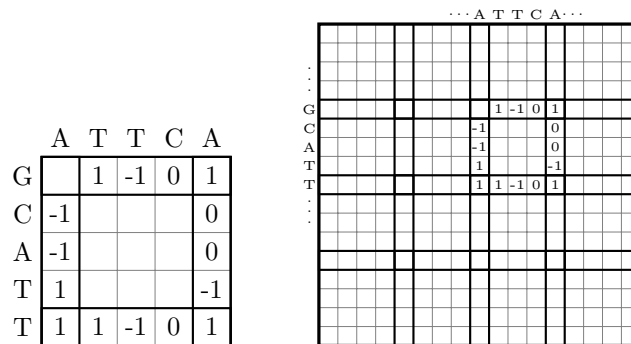
One-against-many edit distance comparison involves comparing a single string to a set of  $n$  other strings. Here, we consider only the banded alignment version of the problem wherein we seek to find the distance to all strings within maximum distance  $d$ . This problem can be solved in  $O(nm + nd^2)$  or  $O(nmd)$  time by iteratively applying the pairwise banded alignment algorithms discussed above. Heuristic approaches may run much faster in practice by exploiting properties of the input strings such as prefix similarity and storing the set of strings in a clever data structure such as a trie or BK-tree [9]. However, little theoretical progress has been made. A popular approach to this problem in the context of spell checkers employs Levenshtein automata and/or transducers [21, 16, 12]. Assuming  $d$  is a fixed constant, these algorithms run in  $O(nm)$  time. However, in practice they consider extremely small values of  $d$  (at most 3 or 4) and their runtime appears to grow exponentially in  $d$ . In the context of gene clustering in computational biology, [6] show that all pairs banded alignment can be performed in  $O(n^2m)$  time under the assumption that all strings are extremely similar. They also use an extension of the Four Russians speedup to one-against-many banded alignment, but our approach to this problem requires no assumptions on the input strings.

The Four Russian speedup is well-studied in context of the regular expression membership problem where the goal is to determine if a particular string matches a given regular expression. Myers[18] showed that for a regular expression of length  $P$  and a string of length  $m$ , the exact regular expression membership problem (no mismatches are allowed) can be solved in  $O(mP/\log m)$  time using the Four Russian speedup compared to the naive  $O(mP)$  runtime. Wu, Manber, and Myers [23] extended this result for approximate regular expression membership problem where the goal is to check if a string is within an edit distance  $d$  from the given regular expression. They showed that approximate regular expression matching problem can be solved in  $O(mP/\log_{d+2} m)$  time.

Space efficiency is also a major concern in practical applications of the Four Russians speedup since the entire lookup table must be stored in main memory. Thus, block sizes as small as  $k = 4$  or  $5$  may be used. The classical approach for the unit cost variant uses  $O(3^{2k}k|\Sigma|^{2k})$  space. Kim, Na, Park, and Sim [13] showed how to remove the dependence on the alphabet size, generating a lookup table in  $O(3^{2k}(2k)!k^2)$  time and  $O(3^{2k}(2k)!k)$  space. This offers a significant improvement, for example, when  $|\Sigma| = 20$  for protein sequences or  $|\Sigma| = 26$  for the English language.

## 1.2 Preliminaries

For simplicity of presentation, we assume all strings have equal length  $m$ . However, the results extend easily to the case where strings have different lengths. We assume the lookup table is any data structure that can perform lookups and insertions in  $O(k)$  time for blocks which are identified by distinct keys of length  $O(k)$ .



■ **Figure 1** Example of classic Four Russians. **Left:** a single block. Notice that for any input in the upper left corner, we can sum that value with one path along the edges of the block to recover the value in the lower right corner. Note that the offset value in the lower right corner may be different for the row and column vectors overlapping at that cell. In this case, the lower right cell is one more than its left neighbor and one less than its above neighbor. **Right:** the full dynamic programming table divided into sixteen  $5 \times 5$  blocks. Note that the offset values in the example block may not correspond to the optimal alignment of the two substrings shown since they depend on the global alignment between the two full length strings.

### 1.2.1 The classical Four Russians speedup

In the classical Four Russians speedup of edit distance computation due to [15, 14], the dynamic programming table is broken up into square *blocks* of size  $k$ -by- $k$  as shown in the right of Fig. 1. These blocks are tiled such that they overlap by one column/row on each side (for a thorough description see [11]).

The high level idea of the Four Russians speedup is to precompute all possible solutions to a *block function* and store them in a lookup table. The block function takes as input the two substrings to be compared in that block and the first row and column of the block itself in the dynamic programming table. It outputs the last row and column of the block. We can see in Figure 1 that given the two strings and the first row and column of the table, such a function could be applied repeatedly to compute the lower right cell of the table and therefore, the edit distance.

There are several tricks that reduce the number of inputs to the block function to bound the time and space requirements of the lookup table. For example, when the edits have unit cost, the input row and column for each block can be reduced to vectors in  $\{-1, 0, 1\}^k$ . These *offset vectors* encode only the difference between one cell and the next (see Fig. 1) which is known to be in  $\{-1, 0, 1\}$ . It has also been shown that the upper left corner does not need to be included in the offset vectors. This bounds the number of possible row and column inputs at  $3^k$  each [15]. More generally, when edit costs are derived from a penalty matrix, the number of row/column inputs is bounded by  $\psi^k$  where  $\psi$  is the number of possible offset values and depends on the penalty matrix.

## 1.3 Our Contributions

We show a new way to store and query block functions. For a given pair of strings corresponding to a  $k$ -by- $k$  block in the dynamic programming table, we store an entry in the lookup table using only  $O(k^2 \lg k)$  time and  $O(k^2)$  space. We show how to query this entry in  $O(k)$  time. By contrast, the classical approach requires  $O(\psi^{2k} k^2)$  time and  $O(\psi^{2k} k)$  space, where  $\psi$  is the number of possible offset values and depends on the costs of edits, to store a



lookup entry for a pair of strings since it computes the function for all possible row/column offset vectors and  $O(k)$  time per query. Thus, we improve the time and space complexity of that aspect by a factor of at least  $\psi^{2k}/k$  and remove the dependence on  $\psi$ . This result is stated in Theorem 1.

► **Theorem 1.** *Given two strings corresponding to a  $k$ -by- $k$  block, we can store a lookup entry using  $O(k^2 \lg k)$  time and  $O(k^2)$  space such that given any values for the first row and column of the block, we can compute the last row and column of the block in  $O(k)$  time.*

We demonstrate the power of our technique for block functions by designing an algorithm for the fundamental problem of one-against-many banded alignment. In particular, comparing one string of length  $m$  to  $n$  other strings of length  $m$  where we only need to report distances within a maximum distance threshold  $d$  can be performed in  $O(nm + md^2 \lg d + nd^3)$  time. When  $d$  is reasonably small, this improves on the common, naive approach which requires  $O(nmd)$  time to iteratively run an  $O(md)$  time pairwise banded alignment algorithm. It also approaches the best theoretic result of  $O(nm + nd^2)$  achieved by using the best known pairwise algorithm running in  $O(m + d^2)$  time [17, 22]. We note that the author of [17], describes the  $O(m + d^2)$  time algorithm as “impractical” and “primarily of theoretical interest”. We are somewhat more optimistic, observing that our algorithm blends neatly with approaches such as [6] for comparing genetic sequences and as discussed in Section 4.3 can be implemented in a way that exploits the prefix similarity occurring in practice.

► **Theorem 2.** *Performing banded alignment with maximum distance  $d$  between a string of length  $m$  and  $n$  other strings also of length  $m$  can be done in  $O(nm + md^2 \lg d + nd^3)$  time.*

We extend the classic result of [15] which computes the edit distance between two strings in  $O(m^2/\log m)$  time to remove the dependence on  $\psi$  even when edits have costs derived from a penalty matrix. Here, the number of entries in the lookup table does not depend on the penalty matrix. We acknowledge that [8] also achieves the same  $O(m^2/\log m)$  running time on unrestricted scoring matrices. However, there are some differences between our approach and theirs which may make one or the other more advantageous in different settings. Most notably our approach adheres more closely to the classic Four Russians speedup and uses a uniform block size which is necessary for our one-against-many algorithm. Uniform block sizes also allow our technique to be combined easily with the space-efficient approach in [13] and the gene clustering technique in [6] since both rely on splitting the dynamic programming table into uniform size blocks. In the case of [6], this is crucial to exploiting the prefix similarity among highly conserved genomic sequences. On the other hand, the blocks in [8] vary in size in a clever way to take advantage of the compressibility of the strings being compared. This yields a faster running time for pairwise comparison of strings with small entropy,  $O(hn^2/\log n)$ , where  $h \leq 1$  is the entropy of the text.

► **Theorem 3.** *Given a penalty matrix for edit operations, the edit distance between two strings can be computed in  $O(m^2/\log m)$  time.*

In practical applications of the Four Russians speedup wherein space efficiency is important and smaller block sizes  $k$  are used (notably  $k < |\Sigma|$ ), [13] showed how to remove the dependence on the alphabet size for the unit cost version, generating a lookup table in  $O(3^{2k}(2k)!k^2)$  time and  $O(3^{2k}(2k)!k)$  space. Combining their work with our result yields an improvement to  $O((2k)!k^2 \lg k)$  time and  $O((2k)!k^2)$  space.

► **Theorem 4.** *For a block size  $k$ , a lookup table can be generated in  $O((2k)!k^2 \lg k)$  time and  $O((2k)!k^2)$  space such that we can find the unit cost edit distance between two strings of length  $m$  in  $O(m^2/k)$  time.*

## 2 Storing and querying the block function

Here, we consider the crucial subroutine in our algorithms and prove Theorem 1. For a block size  $k$ , we first show how to store a lookup entry for any two strings of length  $k$  in  $O(k^2 \lg k)$  time and  $O(k^2)$  space. Then, we show how, given two strings of length  $k$  and the first row and column of the block, we can compute the last row and column in  $O(k)$  time by querying the corresponding lookup entry. Notice that in contrast to the classical Four Russians speedup, the information we precompute and store for a block function is based only on the two strings being compared. Thus, we avoid having to store an entry for each of the  $3^{2k}$  possible input vectors considered in [15] (For unit costs, they encode rows/columns as offset vectors in  $\{-1, 0, 1\}$  since the values in adjacent cells differ by at most 1, yielding  $3^k$  possible inputs each for the row and column vectors).

### 2.1 Notation

We start by defining some notation, illustrated in Figure 2. Let  $U = \{u_1, u_2, \dots, u_{2k-1}\}$  be an ordered set of the cells in the first row and column of the block and let  $V = \{v_1, v_2, \dots, v_{2k-1}\}$  be an ordered set of the cells in the last row and column of the block. For both sets, the ordering starts with the upper right corner and ends in the lower left corner. Thus, both  $u_1$  and  $v_1$  correspond to the upper right corner,  $u_k$  corresponds to the upper left corner,  $v_k$  corresponds to the lower right corner, and both  $u_{2k-1}$  and  $v_{2k-1}$  correspond to the lower left corner.

For each pair of cells  $(u, v)$ , we store the least cost  $c_{u,v}$  of any path through the block from  $u$  to  $v$ . If no such path exists, we set  $c_{u,v} = \infty$  and if  $u$  and  $v$  correspond to the same cell, we set  $c_{u,v} = 0$ . Note that  $c_{u,v}$  is not necessarily based on the optimal alignment within the entire block. It corresponds to an alignment of the subset of the block with  $u$  as the upper left corner and  $v$  as the lower right. Also, recall that this block will be part of a larger dynamic programming table and the path through the block corresponding to the best global alignment may not be the same as the path corresponding to the best local alignment within the block.

We can think of this set of costs as a complete, weighted bipartite graph  $G = \{U, V, U \times V\}$  with weights  $c_{u,v}$  on the edges. We also use  $c_u$  and  $c_v$  to denote the values stored in the corresponding cells of the block within the dynamic programming table. When we query a block function for two strings, the  $c_u$  values (input row and column) will be given as input and our goal will be to compute the  $c_v$  values (output row and column). Thus, if we consider the values stored in the cells after the full dynamic programming table has been computed, we have that  $c_v = \min_{u \in U} (c_u + c_{u,v})$ .

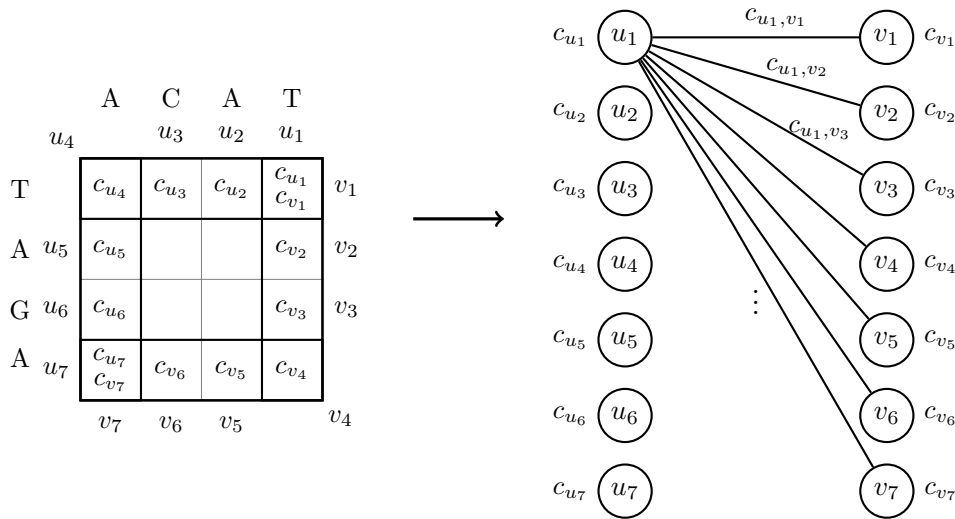
### 2.2 Storing lookup entries

For every pair of substrings we wish to query eventually, our lookup table will simply store the cost  $c_{uv}$  for every edge in the graph  $G$  defined by comparing those substrings. These cost values will be stored in a  $|V| \times |U|$  matrix  $M$  with a row for each  $v \in V$  and a column for each  $u \in U$ . Cell  $M_{ji}$  will contain  $c_{u_i v_j}$ . We now show that computing  $G$  and storing  $M$  for any pair of substrings of length  $k$  can be done in  $O(k^2 \lg k)$  time.

► **Lemma 5.** *Given a pair of strings of length  $k$ , we can compute all  $c_{u,v}$  in  $O(k^2 \lg k)$  time.*

**Proof.** Note that each  $c_{u,v}$  can be seen as the weight of the shortest path in a grid graph of dimension  $k \times k$ . Thus the algorithm of [19] can be applied. That algorithm requires  $O(k^2 \lg k)$  preprocessing time and can then compute each of the  $O(k^2)$   $c_{u,v}$  entries in  $O(\lg k)$  time. This leads to an overall running time of  $O(k^2 \lg k)$ . ◀





■ **Figure 2** Illustration of how the dynamic programming table is represented as a bipartite graph of least cost paths. **Left:** The dynamic programming table for a block comparing the strings “ACAT” and “TAGA” with all  $u, v, c_u$ , and  $c_v$  labeled. **Right:** The bipartite graph representation. Note that this will be a complete, weighted bipartite graph with costs  $c_{u,v}$  for all pairs in  $U \times V$ .

For completeness, we also state the simple fact that the space requirement for an entry is  $O(k^2)$ .

► **Lemma 6.** *Given a pair of strings of length  $k$ , storing the entry requires  $O(k^2)$  space.*

**Proof.** The proof follows directly from the fact that we are simply storing the edges of a complete, weighted bipartite graph  $G = \{U, V, U \times V\}$  with  $|U| = |V| = 2k - 1$ . ◀

### 2.3 Querying a block function

Given the two substrings and the input row and column vectors, we now show how to use our lookup entry matrix  $M$  to compute the output row and column (a.k.a all  $c_v$  for  $v \in V$ ) in  $O(k)$  time.

► **Lemma 7.** *Given the input row and column vectors and the  $O(k) \times O(k)$  lookup entry matrix  $M$ , we can compute the output row and column in  $O(k)$  time using the SMAWK algorithm [2].*

**Proof.** Let  $\vec{w}$  be the vector of all  $c_u$  values generated from the input row and column vectors. Scaling each column of  $M$  by the corresponding cell in  $\vec{w}$  gives us a new matrix  $M'$  wherein the minimum value in each row  $j$  is our desired output value  $c_{v_j} = \min_{u \in U} (c_u + c_{u,v_j})$ . It is known that  $M'$  is totally monotone [1, 19] and thus we can find row minima in  $O(|U|) = O(k)$  time using the classic SMAWK algorithm [2]. Note that we need not explicitly generate  $M'$  since the value of any cell we wish to query can be computed from  $M$  and  $\vec{w}$  as  $M'_{ji} = M_{ji} + \vec{w}_i$ . ◀

The proof of Theorem 1 follows from Lemmas 5, 6, and 7.

## 2.4 Alternatives to query a block function without SMAWK

While our algorithm for banded alignment in Section 3 uses larger block sizes than the typical pairwise Four Russians approach, in many cases, the blocks will be small enough for SMAWK to be inefficient in practice. As such, we introduce a simpler query algorithm here and briefly discuss the potential for future work to speed up the query function in practice.

This simpler query algorithm achieves a slightly worse asymptotic running time of  $O(k \lg k)$  and can be described as follows. Recall that our goal is to find the minimum value of each row in the totally monotone matrix  $M'$  with  $|U|$  columns and  $|V|$  rows. We first find the minimum value in row  $|V|/2$  and let *mincol* be the column containing that cell. We then perform the same operation recursively on two submatrices of  $M'$ . The first submatrix includes all rows up to  $|V|/2$  and all columns up to (and including) *mincol*. The second includes the rows after  $|V|/2$  and columns from *mincol* to  $|U|$ . We do not claim this simpler algorithm is a novel approach to finding row minima and include it merely to illustrate possible alternatives to SMAWK.

► **Lemma 8.** *The algorithm described here runs in  $O(k \lg k)$  time and outputs the correct result.*

**Proof.** For the running time, note that each recursive call nearly partitions the columns of  $M'$  (pairs of submatrices overlap at single columns), resulting in  $O(|U|) = O(k)$  time spent at each level of recursion. Since we split the rows in half at each level, there will be  $O(\lg |V|) = O(\lg k)$  levels total, giving a final running time of  $O(k \lg k)$ .

The correctness follows directly from the properties of totally monotone matrices also utilized in the analysis of SMAWK. ◀

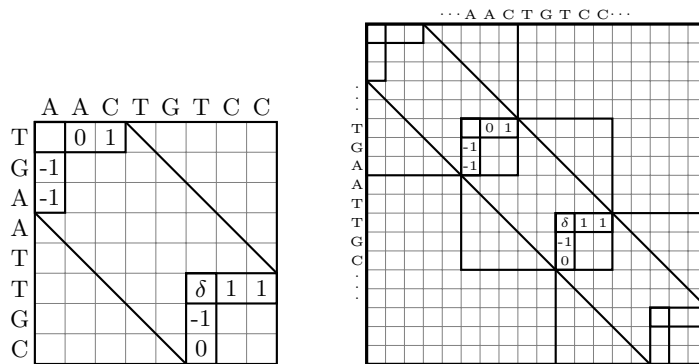
Looking to the future, we note that neither SMAWK nor the algorithm in this section leverage all of the specific properties of the matrix  $M'$ . For example,  $M'$  is not an arbitrary totally monotone matrix. It comes from  $M$ , a matrix which we can afford to spend  $k^2$  time preprocessing, scaled by  $\vec{w}$ , a vector with the property that adjacent cells differ by at most 1 in the unit cost setting.

## 3 One-against-many comparison

### 3.1 Extending the Four Russians approach to banded alignment

For our algorithm for one against many banded alignment, we use the extension to banded alignment from [6] which simplifies both the analysis and practical implementation. The extension uses a slightly different block function and way of tiling blocks to cover the relevant diagonal “band” of the dynamic programming table. The blocks now overlap on a square of size  $d + 1$  at the upper left and lower right corners. We will call these overlapping regions *overlap squares*. The block function still takes as input the two substrings to be compared. The set  $U$  contains only the first row and column of the the upper left overlap square and  $V$  contains only the first row and column of the lower right overlap square as well as the difference between the upper left corners of the two overlap squares.

Thus, we can move directly from one block to the next, storing a sum of the differences between the upper left corners. In this case, reaching the final lower right cell of the table requires an additional  $O(d^2)$  operation to fill in the last overlap square, but this adds only a negligible factor to the running time.



■ **Figure 3** Example of our approach to the Four Russians speedup. **Left:** a block for maximum edit distance  $d = 2$ . The output  $\delta$  represents the offset from the upper left corner of one block to the upper left corner of the next block. Note that we only need to consider a diagonal band of the block itself. **Right:** using these blocks to cover the diagonal band of the dynamic programming table in the context of banded alignment.

### 3.2 Our algorithm

We start with some notation and definitions. For a string  $s$ , let  $s_{i,i+k}$  be a length  $k$  substring starting at index  $i$  of  $s$ . We define two types of block comparisons, *identities* and *differences*, based on the strings being compared. An identity comparison is between the substring  $s_{i,i+k}$  and another substring that is identical to one of the substrings  $s_{j,j+k}$  for  $j \in \{i - d, i - d + 1, \dots, i, \dots, i + d\}$ . All other comparisons are difference comparisons. In other words, identity comparisons between two strings will come from long common subsequences between the two strings. Difference comparisons will come from the locations where an edit occurs. Note that we can stop comparing two strings once we've encountered more than  $d$  differences among their prefixes. Let  $S$  be a set of strings and let  $p$  be the single string we wish to compare to all strings in  $S$ .

The algorithm can be summarized as follows. We first compute and store lookup entries for all possible identity comparisons for each block in  $p$ . We then perform pairwise comparisons between  $p$  and each string in  $S$ . A pairwise comparison is computed as follows. For each block, we first query the lookup table using the corresponding substrings. If we find an entry (similarity comparison), we query it as described in Section 2. Otherwise (difference comparison), we perform standard banded alignment on the two strings with the first row and column of the table initialized to the values of the input row and column of the block. If at any time during a pairwise comparison the distance accumulated exceeds  $d$ , then we immediately halt and move on to the next pair.

We divide the analysis into three parts: the time to compute and store the lookup table, the time to query the lookup table during pairwise comparison, and the time to compute the block function for difference comparisons.

► **Lemma 9.** *The time to compute and store the lookup table for all block identity comparisons in a single string  $p$  of length  $m$  and max distance  $d$  is  $O(md^2 \lg d)$ .*

**Proof.** Let the block size  $k = 2d$ . Then  $p$  will be divided into  $m/d - 1$  blocks. For any given block, let  $p_{i,i+k}$  be the substring of  $p$  corresponding to that block. Then, for every  $j \in \{i - d, i - d + 1, \dots, i, \dots, i + d\}$ , we need to store the comparison between  $p_{i,i+k}$  and  $p_{j,j+k}$ . We need not compare  $p_{i,i+k}$  to any substrings outside this range since that would imply an alignment of distance greater than  $d$ . Thus, for each block we need to store lookups

## 13:10 A Succinct Four Russians Speedup for Edit Distance and Banded Alignment

for at most  $2d + 1 = O(d)$  different identity comparisons. Computing the lookup entry for each comparison takes  $O(k^2 \lg k) = O(d^2 \lg d)$  time by Theorem 1. Putting it all together, we have  $O(m/d \cdot d \cdot d^2 \lg d) = O(md^2 \lg d)$ . ◀

► **Lemma 10.** *Excluding the time to compute block functions for difference comparisons, the time to compare a string  $p$  of length  $m$  to  $n$  other strings using the precomputed lookup table is  $O(nm)$ .*

**Proof.** Each pairwise comparison involves computing  $m/d - 1$  block functions. If a block corresponds to an identity comparison querying the block function takes  $O(k) = O(d)$  time by Theorem 1. Otherwise, if it's a difference comparison block, the only time will come from checking the lookup table which we've assumed takes  $O(d)$  time. It follows that the running time for each pairwise comparison is  $O(m)$  and comparing  $p$  to all  $n$  strings requires  $O(nm)$  time. ◀

► **Lemma 11.** *The time needed to compute block functions for difference comparisons between  $p$  and all  $n$  other strings is  $O(nd^3)$ .*

**Proof.** Notice that each edit is present in at most two overlapping blocks. It follows that for a given pair of strings, the number of block queries corresponding to differences can be at most  $2(d + 1) = O(d)$  since we will halt a comparison if the distance ever reaches  $d + 1$  or more. Thus, the running time to compute the full dynamic programming for difference blocks for all  $n$  pairwise comparisons is  $O(n \cdot d \cdot d^2) = O(nd^3)$ . ◀

The proof of Theorem 2 follows from combining Lemmas 9, 10, and 11.

### 4 Extensions and applications

In this section, we briefly show how the results of Section 2 can be applied to other settings in which the Four Russians speedup is used for computing string edit distance.

#### 4.1 Comparing two arbitrary strings with a penalty matrix

As with the classical Four Russians, when the alphabet size is constant, we can choose the block length  $k$  to be an appropriate logarithmic function of the string length  $m$  such that the lookup table can be computed efficiently. For an alphabet  $\Sigma$ , there are  $|\Sigma|^{2k}$  pairs of string of length  $k$ . By Theorem 1, each pair requires  $O(k^2 \lg k)$  time to compute the lookup entry regardless of the costs of the edits. Thus, the preprocessing for  $k = (\log_{|\Sigma|} m)/2$  takes  $O(m \log^2 m \log \log m)$  time. Since the total number of blocks in the dynamic programming table is  $O(m^2/k^2)$  and computing each block function from the lookup table takes  $O(k)$  time by Theorem 1, the running time to compute the distance using the lookup table is  $O(m^2/\log m)$ . This completes the proof of Theorem 3.

#### 4.2 Improved space-efficiency

The approach in Section 2 can be combined with the work of [13] to achieve the improved time and space bound in Theorem 4 for computing the lookup table. Notice that Theorem 1 gives a time and space bound for each pair of substrings for which we need to compute a block function. Specifically, each pair of strings contributes  $O(k^2 \lg k)$  time and  $O(k^2)$  space. As a complement, [13] showed how to encode strings in such a way that we reduce the number of redundant string comparisons. There, the number of strings compared is reduced to  $O((2k)!)$ . Theorem 4 follows from these simple observations.

### 4.3 Exploiting prefix similarity in one-against-many comparison

Since the one-against-many banded alignment algorithm in Section 3 uses the same extension to banded alignment as [6], it can be combined with other techniques from that paper. In particular, they divide all of the strings in the database  $S$  into blocks and store the blocks in a trie-like data structure. This allows them to exploit prefix similarity of the strings of  $S$  and further improve the running time in practice. Additionally, that uses *lazy computation*, the technique of computing and storing the lookup table on-the-fly rather than precomputing it to heuristically avoid comparing substrings which don't actually appear in the dataset. In the context of Theorem 2, that could potentially reduce the  $md^3$  factor.

## 5 Conclusion and future directions

In this paper, we provided an approach to storing and querying block functions in the Four Russians speedup for edit distance computation using less time and space than the original method. We demonstrated how this approach can lead to an algorithm for the one-against-many banded alignment problem. Finally, we showed how our approach can easily be combined with prior work to gain additional improvements such as space-efficiency.

The problems of comparing two similar strings and one-against-many comparison of highly similar strings have applications in variety of domains. For example, searching a query sequence against the database of multiple sequence within a certain similarity threshold is one of the basic tasks in designing database management systems. In the case of document plagiarism detection, the task is to compare two documents which are assumed to be highly similar to each other. In the case of computational biology, sequence similarity detection is a ubiquitous task in most analysis. Although there have been efficient algorithms proposed in literature, they are not very easy or practical to implement on a routine basis. Our algorithm may bridge this gap and be easier to implement while yielding similar theoretical bounds.

There are many questions and potential future directions following this work. One natural question is whether the techniques in this paper can be applied to other problems yielding a Four Russians speedup. In many cases, such as boolean matrix multiplication, the answer is no. However, problems more closely related to edit distance may yield some improvement. Regarding the specific problems in this paper, the  $O(nd^3)$  term in the one-against-many result can likely be improved to  $O(nd^2)$  to match [17] and doing so using practical techniques would be a nice addition to this work. Similarly, improving the constant factors in the query by using a more specialized algorithm than SMAWK (even an asymptotically worse algorithm) could enhance the practical applications of our approach. On the hardness side, which of these results are tight?

---

### References

- 1 Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 497–512, Oct 1988.
- 2 Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, Nov 1987.
- 3 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing, STOC '15*, pages 51–58, New York, NY, USA, 2015. ACM.

- 4 Gregory Bard. Matrix inversion (or lup-factorization) via the method of four russians, in  $\theta(n^3/\log n)$  time. *LMS J. Comput. Math.*, 1:14, 2008.
- 5 Karl Bringmann and Marvin Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, FOCS '15, pages 79–97, Washington, DC, USA, 2015. IEEE Computer Society.
- 6 Brian Brubach, Jay Ghurye, Mihai Pop, and Aravind Srinivasan. Better greedy sequence clustering with fast banded alignment. In Russell Schwartz and Knut Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 7 J Gregory Caporaso, Christian L Lauber, William A Walters, Donna Berg-Lyons, James Huntley, Noah Fierer, Sarah M Owens, Jason Betley, Louise Fraser, Markus Bauer, et al. Ultra-high-throughput microbial community analysis on the illumina hiseq and miseq platforms. *The ISME journal*, 6(8):1621–1624, 2012.
- 8 Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.
- 9 J. W. Fickett. Fast optimal alignment. *Nucleic Acids Res.*, 12(1 Pt 1):175–179, Jan 1984.
- 10 Yelena Frid and Dan Gusfield. A simple, practical and complete o-time algorithm for rna folding using the four-russians speedup. *Algorithms for Molecular Biology*, 5(1):13, 2010.
- 11 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.
- 12 Ahmed Hassan, Sara Noeman, and Hany Hassan. Language independent text correction using finite state automata. In *In Proceedings of the Third International Joint Conference on Natural Language Processing*, pages 913–918, 2008.
- 13 Youngho Kim, Joong Chae Na, Heejin Park, and Jeong Seop Sim. A space-efficient alphabet-independent four-russians' lookup table and a multithreaded four-russians' edit distance algorithm. *Theor. Comput. Sci.*, 656:173–179, 2016. doi:10.1016/j.tcs.2016.04.028.
- 14 William J. Masek and Michael S. Paterson. How to compute string-edit distances quickly. In *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, pages 337–349. Addison-Wesley Publ. Co., Mass., 1983.
- 15 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.
- 16 Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Comput. Linguist.*, 30(4):451–477, 2004.
- 17 Eugene W. Myers. An  $O(ND)$  difference algorithm and its variations. *Algorithmica*, 1(1):251–266, Nov 1986.
- 18 Gene Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM (JACM)*, 39(2):432–448, 1992.
- 19 Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27(4):972–992, 1998.
- 20 Claus-Peter Schnorr. An algorithm for transitive closure with linear expected time. *SIAM Journal on Computing*, 7(2):127–133, 1978.
- 21 Klaus Schulz and Stoyan Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85, 2002.
- 22 Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, 1985.
- 23 Sun Wu, Udi Manber, and Eugene Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of algorithms*, 19(3):346–360, 1995.

# Can a permutation be sorted by best short swaps?

## Shu Zhang

Department of Computer Science and Technology, Shandong University  
Jinan, China  
zhangshu365@163.com

## Daming Zhu

Department of Computer Science and Technology, Shandong University  
Jinan, China  
dmzhu@sdu.edu.cn

## Haitao Jiang

Department of Computer Science and Technology, Shandong University  
Jinan, China  
htjiang@sdu.edu.cn

## Jingjing Ma

Department of Computer Science and Technology, Shandong University  
Jinan, China  
majingjing.sdu@gmail.com

## Jiong Guo

Department of Computer Science and Technology, Shandong University  
Jinan, China

## Haodi Feng

Department of Computer Science and Technology, Shandong University  
Jinan, China

---

### Abstract

A short swap switches two elements with at most one element caught between them. Sorting permutation by short swaps asks to find a shortest short swap sequence to transform a permutation into another. A short swap can eliminate at most three inversions. It is still open for whether a permutation can be sorted by short swaps each of which can eliminate three inversions. In this paper, we present a polynomial time algorithm to solve the problem, which can decide whether a permutation can be sorted by short swaps each of which can eliminate 3 inversions in  $O(n)$  time, and if so, sort the permutation by such short swaps in  $O(n^2)$  time, where  $n$  is the number of elements in the permutation.

A short swap can cause the total length of two element vectors to decrease by at most 4. We further propose an algorithm to recognize a permutation which can be sorted by short swaps each of which can cause the element vector length sum to decrease by 4 in  $O(n)$  time, and if so, sort the permutation by such short swaps in  $O(n^2)$  time. This improves upon the  $O(n^2)$  algorithm proposed by Heath and Vergara to decide whether a permutation is so called lucky.

**2012 ACM Subject Classification** Mathematics of computing

**Keywords and phrases** Algorithm, Complexity, Short Swap, Permutation, Reversal

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.14

**Acknowledgements** This paper is supported by national natural science foundation of China, No. 61472222, 61732009, 61761136017, 61672325.



© Shu Zhang, Daming Zhu, Haitao Jiang, Jingjing Ma, Jiong Guo, and Haodi Feng; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 14; pp. 14:1–14:12

Leibniz International Proceedings in Informatics



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



## 1 Introduction

A short swap on a permutation represents an operation which switches two elements with at most one element caught between them in the permutation. Sorting by short swaps asks to find a shortest sequence of short swaps which can transform a given permutation into another. This problem was first proposed by Heath and Vergara, who also proposed an approximation algorithm which can achieve a performance ratio 2 for this problem [9].

Short swap can be thought of as a kind of rearrangement operations on permutations, where a rearrangement has been being used to account for the gene order variations in a genome [3], and can be formalized as some basic operations such as reversal, translocation, and transposition [15]. Sorting permutation by rearrangements can be used to trace the evolutionary path between genomes [14], and plays important roles in computational biology and bioinformatics [13][8].

A short swap can be thought of as a two or three element consecutive subsequence reversal on a permutation [9]. Sorting a signed permutation by reversals was introduced by Bafna and Pevzner[1]. Hannenhalli and Pevzner proposed a polynomial time algorithm for this problem [8]. Other algorithmic progresses can be looked up in [11][6][7]. Sorting unsigned permutation by reversals turns to be NP-hard [4]. Thus people have been engaging in designing approximation algorithms for this problem [16][12][2].

Moreover, a short swap can be thought of as a swap of length 2 to 3 on a permutation. Jerrum has shown that minimum sorting by swaps can be solved in polynomial time [10]. The complexity of sorting by short swaps remains open up to now. Heath and Vergara proposed an upper bound  $(\frac{n^2}{4}) + O(n \log n)$  for the minimum number of short swaps to sort an  $n$ -element permutation [9]. Feng *et. al.* improved the bound to  $(\frac{3}{16})n^2 + O(n \log n)$  later [5].

In fact, the time complexity of deciding whether a permutation can be sorted by short swaps which eliminate three inversions, is still open. In this paper, we present a sufficient and necessary condition for a permutation to be sorted by short swaps which eliminate three inversions, based on which, we can propose an algorithm to recognize a permutation which can be sorted by short swaps which eliminate three inversions in  $O(n)$  time, and if so, sort the permutation by short swaps to eliminate three inversions, in  $O(n^2)$  time.

In the 2-approximation algorithm for sorting by short swaps [9], Heath and Vergara proposed to use an element vector to indicate how long a distance the element is from that element position it aims to be moved to, and showed that a short swap can cause two element vector's length sum to decrease by at most 4. Thus a so-called best cancellation refers to a short swap which can cause two element vector's length sum to decrease by 4. Heath and Vergara also presented an  $O(n^2)$  algorithm to decide whether a permutation can be sorted by best cancellations. In this paper, we further propose a sufficient and necessary condition for a permutation to be sorted by best cancellations. Based on this observation, we propose an algorithm to recognize a permutation which can be sorted by best cancellations in  $O(n)$  time, and if so, sort the permutation by best cancellations, in  $O(n^2)$  time.

## 2 Preliminaries

Let  $\pi = [\pi_1, \pi_2, \dots, \pi_n]$  be a permutation of  $\{1, 2, \dots, n\}$ . A *swap* on  $\pi$  switches  $\pi_i$  with  $\pi_j$ , where  $\pi_i$  and  $\pi_j$  are two elements in  $\pi$ . The swap is short, if there is at most one element between  $\pi_i$  and  $\pi_j$  in  $\pi$ . Let  $\rho$  be an arbitrary swap on  $\pi$ . We denote by  $\pi \cdot \rho$  the permutation  $\rho$  transforms  $\pi$  into. For example, let  $\rho$  be a swap which switches 7 with 4 in  $\pi = [5, 3, 1, 7, 6, 4, 2]$ . Then  $\pi \cdot \rho = [5, 3, 1, 4, 6, 7, 2]$ . The problem of sorting a permutation by short swaps can be formulated as follows.



**Instance:** A permutation  $\pi$

**Solution:** A sequence of short swaps  $\rho_1, \rho_2, \dots, \rho_k$ , such that  $\pi \cdot \rho_1 \cdot \rho_2 \cdot \dots \cdot \rho_k = [1, 2, \dots, n]$  and  $k$  is minimized.

As usually used, let  $\iota$  denote the identity permutation  $[1, 2, \dots, n]$ . The minimum number of short swaps which transform  $\pi$  into  $\iota$  is referred to as the short swap distance of  $\pi$ , and denoted by  $sw^3(\pi)$ .

## 2.1 Happy permutation

An *inversion* in  $\pi$  refers to a pair of elements that are not in their correct relative order. Formally, the pair composed of  $\pi_i$  and  $\pi_j$  is an inversion of  $\pi_i$  and  $\pi_j$  in  $\pi$ , if  $i < j$  and  $\pi_i > \pi_j$ . Let  $inv_\pi$  be the set of inversions in  $\pi$ . A short swap  $\rho$  is said to *eliminate*  $|inv_\pi| - |inv_{\pi \cdot \rho}|$  inversions (of  $\pi$ ), if  $|inv_\pi| \geq |inv_{\pi \cdot \rho}|$ , and *add*  $|inv_{\pi \cdot \rho}| - |inv_\pi|$  inversions (of  $\pi$ ) otherwise.

A short swap can eliminate at most 3 inversions of  $\pi$ . If  $\pi \neq \iota$ , at least 1 inversion of two adjacent elements occurs in  $\pi$ , which can be eliminated by a short swap. Thus the short swap distance of  $\pi$  can be bounded by,

► **Lemma 1.**  $\lceil \frac{|inv_\pi|}{3} \rceil \leq sw^3(\pi) \leq |inv_\pi|$

**Proof.** See Theorem 3 in [9]. ◀

Due to Lemma 1, a short swap is referred to as *best* (resp. *worst*), if it can eliminate (resp. add) 3 inversions of  $\pi$ . A permutation, say  $\pi$  is referred to as *happy*, if  $sw^3(\pi) = \frac{|inv_\pi|}{3}$ . A permutation is happy, if and only if it can be transformed into  $\iota$  by none other than best short swaps.

A consecutive sub sequence  $\pi[x \rightarrow y] \equiv [\pi_x, \dots, \pi_y]$  of  $\pi$  is referred to as an *independent sub-permutation* (abbr. ISP) in  $\pi$ , if for  $1 \leq l < x \leq i \leq y < h \leq n$ ,  $\pi_l < \pi_i < \pi_h$ . An ISP is referred to as *minimal*, if none of its sub sequence, other than itself, is an ISP. A minimal ISP in  $\pi$  is abbreviated as an MISP. Since no inversion happens between two distinct ISPs, it suffices to pay attention to sorting an MISP by best short swaps.

For an element  $\pi_i$  in  $\pi$ , we refer to the integer interval  $[i, \pi_i]$  as the vector of  $\pi_i$  in  $\pi$  and denote it as  $v_\pi(\pi_i)$ , where  $|v_\pi(\pi_i)| = |\pi_i - i|$  is referred to as the *length* of  $v_\pi(\pi_i)$ . The element vector length indicates the difference between the element index and its correct index. The element  $\pi_i$  is referred to as *vector-right*, if  $\pi_i - i > 0$ ; *vector-left*, if  $\pi_i - i < 0$ ; and *vector-zero*, if  $\pi_i - i = 0$ . An MISP is *isolated*, if it contains just one element. An isolated MISP must admit one and only one vector-zero element. Let  $\pi[x \rightarrow y]$  be an arbitrary MISP. If  $\pi[x \rightarrow y]$  is not isolated, then  $\pi_x$  must be vector-right, and  $\pi_y$  vector-left.

## 2.2 Lucky permutation

Let  $V_\pi = \{v_\pi(\pi_i) \mid 1 \leq i \leq n\}$ . We denote by  $L(V_\pi)$  the length sum of all those vectors in  $V_\pi$ . A short swap always involves two element vectors. An element can be caused by one short swap to change its vector's length by at most 2. Thus a short swap can cause  $L(V_\pi)$  to decrease by at most 4. If  $\pi \neq \iota$ , Heath *et. al.* have shown in [9] that it can always find two elements in  $\pi$  and a sequence of short swaps to switch them, such that if switching the two elements uses  $m$  short swaps which transform  $\pi$  into  $\pi'$ , then  $L(V_\pi) - L(V_{\pi'}) \geq 2m$ . This leads to another short swap distance bound of  $\pi$ , which can be described as,

► **Lemma 2.**  $\frac{L(V_\pi)}{4} \leq sw^3(\pi) \leq \frac{L(V_\pi)}{2}$

**Proof.** See Theorem 10 in [9]. ◀

A permutation  $\pi$  is referred to as *lucky*, if  $sw^3(\pi) = \frac{L(V_\pi)}{4}$ .

### 3 How to recognize a happy permutation

We denote by  $\rho\langle i, j \rangle$  ( $i < j$ ) a swap on  $\pi$ , which switches  $\pi_i$  with  $\pi_j$ . If  $\rho\langle i, j \rangle$  is short, then  $i + 1 \leq j \leq i + 2$ . The short swap  $\rho\langle i, j \rangle$  affects an ISP in  $\pi$ , if at least one of  $\pi_i, \pi_{i+1}, \pi_j$  occurs in the ISP. The short swap  $\rho\langle i, j \rangle$  acts on an ISP, if all of  $\pi_i, \pi_{i+1}, \pi_j$  occur in the ISP. To check if a permutation is happy, we present a sufficient and necessary condition for a short swap to be worst. A best or worst short swap must switch two elements with another element caught between them. Thus  $\rho\langle i, i + 2 \rangle$  will usually be used to represent a best or worst short swap.

► **Lemma 3.** *A short swap, say  $\rho\langle i, i + 2 \rangle$  on  $\pi$  is worst, if and only if  $\pi_i < \pi_{i+1} < \pi_{i+2}$ .*

Let  $\pi[x \rightarrow y]$  be an ISP in  $\pi$ . If a short swap  $\rho\langle i, j \rangle$  which acts on  $\pi[x \rightarrow y]$  transforms  $\pi$  into  $\pi'$ , then  $\pi'[x \rightarrow y]$  must be an ISP in  $\pi'$ .

► **Lemma 4.** *If a worst short swap acts on an MISIP, it must transform the MISIP into an ISP which remains an MISIP.*

For an arbitrary ISP  $\pi[x \rightarrow y]$  in  $\pi$ , an element  $\pi_j$  in  $\pi[x \rightarrow y]$  is referred to as *position-odd*, if  $j - x$  is zero or even; *position-even*, otherwise. An ISP is referred to as *sorted* if no inversion occurs in the ISP; *unsorted*, otherwise. An ISP  $\pi[x \rightarrow y]$  in  $\pi$  is referred to as *happy*, if it can be transformed into  $\iota[x \rightarrow y]$  by none other than best short swaps. By the following theorem, we present a sufficient and necessary condition for an MISIP to be happy.

► **Theorem 5.** *An unsorted MISIP is happy if and only if, (1) an element in the MISIP is vector-zero if it is position-even; not vector-zero otherwise; and (2) for any two vector-left (resp. vector-right) elements, say  $\pi_i, \pi_j$  in the MISIP, if  $i > j$ , then  $\pi_i > \pi_j$ .*

To prove Theorem 5, let's start with a couple of lemmas. Although in Theorem 5, those two properties are mentioned for an MISIP to meet, it cannot refuse an ISP in  $\pi$  to meet those two properties. Thus an ISP is said to meet the Theorem-5 property (1), if all position-even elements are vector-zero, while all position-odd elements in the ISP are not; and said to meet the Theorem-5 property (2), if all those vector-left as well as vector-right elements increase monotonously. To show Theorem 5, we insist to show that a worst short swap can always transform a sorted ISP or an ISP which meets those two Theorem-5 properties into an ISP which meets those two Theorem-5 properties. This asks to observe on if a worst short swap acts on an ISP which meets those two Theorem-5 properties, and transform it into an MISIP, whether this MISIP meets those two Theorem-5 properties. No matter how many MISIPs a short swap affects, we always treat those MISIPs a short swap affects as an ISP.

► **Lemma 6.** *If a worst short swap acts on an ISP which meets those two Theorem-5 properties, it must transform the ISP into an ISP which meets those two Theorem-5 properties.*

If the ISP the worst short swap acts on is an MISIP, Lemma 6 can be redescribed as:

► **Corollary 7.** *If a worst short swap acts on an MISIP with those two Theorem-5 properties, it must transform the MISIP into an MISIP with those two Theorem-5 properties.*

► **Lemma 8.** *A short swap cannot be worst, if it affects just two MISIPs each of which is isolated or meets those two Theorem-5 properties.*

► **Lemma 9.** *If a worst short swap affects three MISIPs, each of which is isolated or meets those two Theorem-5 properties, it must transform the ISP which consists only these three MISIPs into an MISIP with those two Theorem-5 properties.*

**Proof.** Let  $\rho\langle i, i+2 \rangle$  be a worst short swap which affects three MISPs in  $\pi$ , each of which is isolated or meets those two Theorem-5 properties. Then  $\pi_i < \pi_{i+1} < \pi_{i+2}$ . That MISP caught between the other two MISPs in  $\pi$  must be isolated. Thus without loss of generality, let  $\pi[x \rightarrow i]$ ,  $[i+1]$  and  $\pi[i+2 \rightarrow y]$  be those three MISPs  $\rho\langle i, i+2 \rangle$  affects. Let  $\pi' = \pi \cdot \rho\langle i, i+2 \rangle$ .

**Proof for  $\pi'[x \rightarrow y]$  to be an MISP.** Note that  $\pi'_i = \pi_{i+2}$ ,  $\pi'_{i+2} = \pi_i$  and  $\pi'_j = \pi_j$  for  $j \neq i$  and  $j \neq i+2$ . We show that if  $\pi'[x_1 \rightarrow y_1]$  is an MISP with  $x \leq x_1 \leq y_1 \leq y$ , then  $x = x_1$  and  $y = y_1$ .

Otherwise, let on one hand,  $x \neq x_1$ . (1) If  $x < x_1 < i+1$ , then in  $\pi'[x \rightarrow x_1 - 1]$ , an arbitrary element is less than an arbitrary element in  $\pi'[x_1 \rightarrow y]$ . Since  $\pi[x \rightarrow x_1 - 1] = \pi'[x \rightarrow x_1 - 1]$ ,  $\pi[x \rightarrow x_1 - 1]$  must be an ISP. The assumption for  $\pi[x \rightarrow i]$  to be an MISP is contracted. (2) If  $i+2 < x_1 \leq y$ , it can follow (1) to show that  $\pi[i+2 \rightarrow x_1 - 1]$  must be an ISP. The assumption for  $\pi[i+2 \rightarrow y]$  to be an MISP is contracted. (3) If  $x_1 = i+1$  or  $x_1 = i+2$ , then  $\pi'[x_1 \rightarrow y_1]$  cannot be an MISP because  $\pi'_i > \pi'_{i+1} > \pi'_{i+2}$ . That is the proof for  $x = x_1$ . For the same reason,  $y = y_1$ .

**Proof for  $\pi'[x \rightarrow y]$  to meet those two Theorem-5 properties.** Since  $[i+1]$  is isolated,  $\pi_{i+1} = i+1$ , and for  $x \leq l \leq i$  and  $i+2 \leq h \leq y$ ,  $\pi_l < \pi_{i+1} < \pi_h$ .

(1) If  $\pi[x \rightarrow i]$  and  $\pi[i+2 \rightarrow y]$  are both isolated, then  $i = x$  and  $y = i+2$ , and  $\pi'[x \rightarrow y] = [i+2, i+1, i]$  meets those two Theorem-5 properties trivially.

(2) If one of  $\pi[x \rightarrow i]$  and  $\pi[i+2 \rightarrow y]$  is isolated, then  $i = x$  and  $y \neq i+2$  or  $i \neq x$  and  $y = i+2$ . We only focus on the former subcase, where  $i = x$  and  $y \neq i+2$ , to present the proof. In this subcase,  $\pi_i = \pi'_{i+2} = i < i+2$ ,  $\pi_{i+1} = \pi'_{i+1} = i+1$ , which means  $\pi'_{i+1}$  is vector-zero and  $\pi'_{i+2}$  vector-left. Since  $\pi[i+2 \rightarrow y]$  is not isolated,  $\pi'_i$  and  $\pi_{i+2}$  are vector-right. All position-odd (resp. position-even) elements in  $\pi[i+2 \rightarrow y]$  remain position-odd and not vector-zero (resp. position-even and vector-zero) in  $\pi'[x \rightarrow y]$ . The proof for  $\pi'[x \rightarrow y]$  to meet Theorem-5 property (1), is done.

The vector-zero element  $\pi_i$  in  $\pi[x \rightarrow y]$  turns into the vector-left element  $\pi'_{i+2}$  in  $\pi'[x \rightarrow y]$ , and all elements in  $\pi[i+2 \rightarrow y]$  turn into elements in  $\pi'[x \rightarrow y]$  in the the same relative order as they are in  $\pi[i+2 \rightarrow y]$ . Thus to show that  $\pi'[x \rightarrow y]$  meets Theorem-5, it suffices to show that  $\pi'_{i+2}$  is the leftmost vector-left element in  $\pi'[x \rightarrow y]$ , and less than any other vector-left element in  $\pi'[x \rightarrow y]$ . Of course this is true, because  $\pi'_i$  is vector-right,  $\pi'_{i+1}$  is vector-zero and  $\pi'_{i+2} = \pi_i < \pi_{i+1} < \pi_h$  for  $h > i+1$ . The proof for  $\pi'[x \rightarrow y]$  to meet Theorem-5 property (2), is done.

(3) If none of  $\pi[x \rightarrow i]$  and  $\pi[i+2 \rightarrow y]$  is isolated, then  $i \neq x$  and  $y \neq i+2$ . By Lemma 6, to make sure for  $\pi'[x \rightarrow y]$  to meet those two Theorem-5 properties, it suffices to show that  $\pi[x \rightarrow y]$  meets those two Theorem-5 properties.

Since  $\pi[x \rightarrow i]$  and  $\pi[i+2 \rightarrow y]$  meet Theorem-5 property (2), and  $\pi_l < \pi_{i+1} < \pi_h$  for  $x \leq l \leq i$  and  $i+2 \leq h \leq y$ ,  $\pi[x \rightarrow y]$  meets the Theorem-5 property (2).

Since  $\pi[x \rightarrow i]$  meets the Theorem-5 property (1),  $i-x$  is even. Then, (1)the vector-zero element  $\pi_{i+1}$  is position-even in  $\pi[x \rightarrow y]$ ; (2)each position-odd (resp. position-even) element in  $\pi[x \rightarrow i]$  and  $\pi[i+2 \rightarrow y]$ , remains position-odd (resp. position-even) in  $\pi[x \rightarrow y]$ . This implies that  $\pi[x \rightarrow y]$  meets the Theorem-5 property (1). ◀

The proof of Theorem 5 can be given by Corollary 7 and Lemma 8, 9.

**Proof. Only if:** Let  $\pi[x \rightarrow y]$  be an unsorted and happy MISP, which can be transformed into  $\iota[x \rightarrow y]$  by  $m$  best short swaps, say  $\rho_1, \rho_2, \dots, \rho_m$ . Then  $(\pi \cdot \rho_1 \cdot \rho_2 \dots \rho_{m-1} \cdot \rho_m)[x \rightarrow y] = \iota[x \rightarrow y]$ . Let  $\pi^k[x \rightarrow y] = (\iota \cdot \rho_m \cdot \rho_{m-1} \dots \rho_{m+2-k} \cdot \rho_{m+1-k})[x \rightarrow y]$  for

$1 \leq k \leq m$ . Then  $\pi^m[x \rightarrow y] = \pi[x \rightarrow y]$ . By induction for  $k$ , we show every unsorted MISIP in  $\pi^k[x \rightarrow y]$  meets those two Theorem-5 properties.

- (1) Without loss of generality, let  $\rho_m = \rho\langle i, i+2 \rangle (1 \leq i \leq n-2)$ . Then  $\rho\langle i, i+2 \rangle$  must be a worst short swap which acts on  $\iota$ . It follows that  $\pi^1[x \rightarrow y] = (\iota \cdot \rho_m)[x \rightarrow y] = [x, x+1, \dots, i-1, i+2, i+1, i, i+3, \dots, y]$ , where  $[x], \dots, [i-1], [i+3], \dots, [y]$  are isolated MISIPs and  $[i+2, i+1, i]$  is an unsorted MISIP, which meets those two Theorem-5 properties trivially.
- (2) By inductive assumption, let all unsorted MISIPs in  $\pi^{k-1}[x \rightarrow y]$  meet those two Theorem-5 properties. Assume again  $\rho_{m+1-k} = \rho\langle i, i+2 \rangle (x \leq i \leq y-2)$  with  $\pi^k[x \rightarrow y] = (\pi^{k-1} \cdot \rho\langle i, i+2 \rangle)[x \rightarrow y]$ . Note that  $\rho\langle i, i+2 \rangle$  must be a worst short swap which acts on  $\pi^{k-1}[x \rightarrow y]$ . By Lemma 8,  $\rho\langle i, i+2 \rangle$  cannot affect two MISIPs. By Corollary 7 and Lemma 9, all unsorted MISIPs in  $\pi^k[x \rightarrow y]$  must meet those two Theorem-5 properties.

**If:** Let  $\pi[x \rightarrow y]$  be an MISIP in  $\pi$  which meets those two Theorem-5 properties. The proof for  $\pi[x \rightarrow y]$  to be happy, is to show that one can find a best short swap which can act on  $\pi[x \rightarrow y]$  and transform it into an ISP in which each MISIP either is isolated or meets those two Theorem-5 properties.

**Identify a best short swap:** Let  $\pi_i$  be the biggest element in  $\pi[x \rightarrow y]$ . Then  $\rho\langle i, i+2 \rangle$  can be shown to be a best short swap which acts on  $\pi[x \rightarrow y]$ . The proof can be stated as:

- (1) Since  $\pi[x \rightarrow y]$  meets those two Theorem-5 properties and  $\pi_i$  is the biggest in  $\pi[x \rightarrow y]$ ,  $\pi_i$  must be vector-right and position-odd in  $\pi[x \rightarrow y]$  and no vector-right element can occur on the right side of  $\pi_i$ , which implies  $\pi_{i+1}$  is position-even and equal to  $i+1$ .
- (2) Then  $\pi_i \geq i+2$  follows from that  $\pi_i$  is vector-right,  $\pi_{i+2} \leq i$  follows from that no vector-right element can occur on the right side of  $\pi_i$ . Thus  $\pi_i > \pi_{i+1} > \pi_{i+2}$ .

Let  $\pi'[x \rightarrow y] = (\pi \cdot \rho\langle i, i+2 \rangle)[x \rightarrow y]$ . We devote to show that all unsorted MISIPs in  $\pi'[x \rightarrow y]$  must meet those two Theorem-5 properties.

**The proof to meet the Theorem-5 property (2):** Since  $\pi_i \geq i+2$  is vector-right,  $\pi_{i+2} \leq i$  is vector-left,  $\pi'_i = \pi_{i+2} \leq i$  is either vector-zero or vector-left,  $\pi'_{i+2} = \pi_i \geq i+2$  is either vector-zero or vector-right. This indicates that no vector-left (resp. vector-right) element in  $\pi[x \rightarrow y]$  can turn into vector-right (resp. vector-left) in  $\pi'[x \rightarrow y]$ . Moreover, no two vector-left (resp. vector-right) elements in  $\pi[x \rightarrow y]$  can occur in  $\pi'[x \rightarrow y]$  in the other order than they are in  $\pi[x \rightarrow y]$ . It follows that all unsorted MISIPs in  $\pi'[x \rightarrow y]$  meet the Theorem-5 property (2).

**The proof to meet the Theorem-5 property (1):** All position-even elements in  $\pi'[x \rightarrow y]$  are vector-zero because  $\rho\langle i, i+2 \rangle$  switches only  $\pi_i$  with  $\pi_{i+2}$ . The first element in an unsorted MISIP in  $\pi'[x \rightarrow y]$  must be vector-right, then must be position-odd in  $\pi'[x \rightarrow y]$ . Thus to make sure for all unsorted MISIPs in  $\pi'[x \rightarrow y]$  to meet the Theorem-5 property (1), it suffices to show that for all  $\pi'_j$  in  $\pi'[x \rightarrow y]$ , if  $\pi'_j$  is position-odd and vector-zero, then  $[\pi'_j]$  is an isolated MISIP. Since  $\pi[x \rightarrow y]$  meets the Theorem-5 property (1), only  $\pi'_i$  and  $\pi'_{i+2}$  can be position-odd and vector-zero in  $\pi'[x \rightarrow y]$ .

If  $\pi'_{i+2}$  is vector-zero,  $[\pi'_{i+2}]$  must be an isolated MISIP, because  $\pi'_{i+2}$  is the biggest element in  $\pi'[x \rightarrow y]$ .

If  $\pi'_i$  is vector-zero, it must be the smallest in  $\pi'[i \rightarrow y]$ . The reason is, (1) since  $\pi[x \rightarrow y]$  meets the Theorem-5 property (1) and  $\pi_{i+2} = i$ , an element in  $\pi[i \rightarrow y]$  is bigger than  $\pi_{i+2} = \pi'_i$ , if it is position-even in  $\pi[x \rightarrow y]$ ; (2) since  $\pi[x \rightarrow y]$  meets the Theorem-5 property (2) and  $\pi_{i+2}$  is vector-left, an element in  $\pi[i+3 \rightarrow y]$  is bigger than  $\pi_{i+2} = \pi'_i$ , if it is vector-left in  $\pi[x \rightarrow y]$ ; (3)  $\pi_i$  is the unique vector-right element in  $\pi[i \rightarrow y]$  and bigger than  $\pi_{i+2} = \pi'_i$ . It follows that  $[\pi'_i]$  is an isolated MISIP. ◀

**Algorithm 1:** How to recognize a happy permutation.

---

Algorithm *Happy permutation*  
Input: A permutation  $\pi$ .  
Output: The best short swap sequence  $\rho$  if  $\pi$  is happy; *no*, otherwise.

- 1  $lb \leftarrow 0; rb \leftarrow 0; x \leftarrow 1; b \leftarrow 0;$
- 2 For  $i$  from 1 to  $n$  do
- 3   if  $(i > b)$  then  $x \leftarrow i$ ; (an MISPP starts with  $\pi_x$ )
- 4   if  $(i - x \bmod 2 = 1$  and  $\pi_i = i)$  then  $i \leftarrow i + 1$ ; ( $\pi_i$  is position-even, vector-zero.)
- 5   if  $(i - x \bmod 2 = 0$  and  $\pi_i < i$  and  $\pi_i > lb)$
- 6     then  $lb \leftarrow \pi_i; i \leftarrow i + 1$ ; ( $\pi_i$  is position-odd, vector-left.)
- 7   if  $(i - x \bmod 2 = 0$  and  $\pi_i > i$  and  $\pi_i > rb)$
- 8     then  $rb \leftarrow \pi_i; i \leftarrow i + 1; b \leftarrow \pi_i$ ; ( $\pi_i$  is position-odd, vector-right.)
- 9   if  $(i = x$  and  $\pi_i = i)$  then  $b \leftarrow \pi_i, i \leftarrow i + 1$ ; ( $[\pi_i]$  is isolated.)
- 10   else return *no*;
- 11 end for
- 12 Return Sort( $\pi$ );

---

In fact, an MISPP in  $\pi$  can be recognized by,

► **Lemma 10.** *An MISPP in  $\pi$  starts with  $\pi_i$ , if and only if  $i = 1$  or for  $1 \leq j \leq i - 1$ ,  $i > \pi_j$ .*

To decide if  $\pi$  is happy, it suffices to check if all MISPPs in  $\pi$ , if unsorted, meet those two Theorem-5 properties.

An element in an MISPP can be decided to be position-odd or position-even by the first element index of the MISPP and its index. Then an MISPP can be decided to meet the Theorem-5 property (1) by the value of  $|\pi_i - i|$  for all  $\pi_i$  in this MISPP.

An element in  $\pi$  can be decided to be vector-right, vector-left or vector-zero by the value of  $\pi_i - i$ . To check if all unsorted MISPPs in  $\pi$  meet the Theorem-5 property (2), it suffices to check if  $\pi$  meets the Theorem-5 property (2). Fortunately,  $\pi$  can be decided to meet the Theorem-5 property (2) by checking if all those vector-left (resp. vector-right) elements increase monotonously in the order from  $\pi_1$  to  $\pi_n$ .

We present an algorithm to recognize and sort a happy permutation  $\pi$  in Algorithm 1. If  $\pi$  is happy, the algorithm returns a best short swap sequence which can transform  $\pi$  into  $\iota$  by invoking a subroutine named as Sort( $\pi$ ); returns *no*, otherwise. In the algorithm description, we use the integer parameter  $lb$  (resp.  $rb$ ) to maintain the biggest vector-left (resp. vector-right) element in  $\pi[1 \rightarrow i - 1]$ ,  $b$  the biggest element in  $\pi[1 \rightarrow i - 1]$ ,  $x$  the starting index of the MISPP in which  $\pi_i$  is an element.

Running the algorithm from Step 1 to Step 11 can decide if  $\pi$  is happy or not. This can take  $O(n)$  time, where  $n$  is the number of elements in  $\pi$ . Later, let  $\pi$  be happy. We present on how to find a sequence of best short swaps to transform  $\pi$  into  $\iota$ . To identify a best short swap which switches  $\pi_i$  with  $\pi_{i+2}$ , it suffices to record the integer  $i$ . Thus in Sort( $\pi$ ), we will employ a linear integer array  $\rho[1 \sim X]$  to maintain the best short swap sequence to sort  $\pi$ , where  $X \leq \frac{n(n-1)}{6}$ ,  $\rho[j]$  indicates to switch  $\pi_{\rho[j]}$  with  $\pi_{\rho[j]+2}$ .

The rightmost vector-right element in  $\pi$  must be the rightmost vector-right element in an MISPP in  $\pi$ . Let  $\pi_i$  be the rightmost vector-right element in  $\pi$ . Then it follows the proof of the Theorem 5 sufficient condition that the short swap which switches  $\pi_i$  with  $\pi_{i+2}$  is best. By Theorem 5 again, this operation must transform  $\pi$  into a happy permutation. Thus the trick for finding the rightmost vector-right element in  $\pi$  to identify a best short swap can be done repeatedly until  $\pi$  is transformed into  $\iota$ . The algorithm Sort( $\pi$ ) is depicted in Figure 2.

---

**Algorithm 2:** How to sort a happy permutation.
 

---

```

Algorithm  $Sort(\pi)$ 
1   $x \leftarrow 0$ ;
2  while  $\pi \neq \iota$ 
3    find the rightmost vector-right element  $\pi_i$ ;
4    while  $\pi_i > i$ 
5       $\rho[x] \leftarrow i$ ;  $\pi \leftarrow \pi \cdot \rho[x]$ ;  $x \leftarrow x + 1$ ;
6       $i \leftarrow i + 2$ ;
7    end while
8  end while
9  Return  $\rho$ .
```

---

A rightmost vector-right element, say  $\pi_i$ , remains rightmost and vector-right in the permutation the short swap which switches  $\pi_i$  with  $\pi_{i+2}$  transforms  $\pi$  into, until it turns into vector-zero. So it takes  $O(n)$  time to find all the rightmost vector-right elements. On the other hand, each best short swap can eliminate 3 inversions, the total inversion number is  $O(n^2)$ . Thus the time complexity of  $Sort(\pi)$  is  $O(n^2)$ . It follows that the time complexity of recognizing a happy permutation is  $O(n^2)$ .

#### 4 How to recognize a lucky permutation

A short swap on  $\pi$  is referred to as a *best cancellation*, if it cause  $L(V_\pi)$  to decrease by 4 [9]. The permutation  $\pi$  is referred to as *lucky*, if it can be transformed into  $\iota$  by none other than best cancellations. A short swap is referred to as a *promising cancellation* (resp. *promising addition*), if it switches two adjacent elements in  $\pi$  and causes  $L(V_\pi)$  to decrease (resp. increase) by 2.

An ISP  $\pi[x \rightarrow y]$  is referred to as *sub-lucky*, if it can be transformed into  $\iota[x \rightarrow y]$  by none other than promising cancellations. To check if a permutation is lucky, we set about to check if an ISP is sub-lucky. This asks us to observe what kind of a short swap is a promising addition or cancellation.

► **Lemma 11.** *The short swap  $\rho\langle i, i + 1 \rangle$  on  $\pi$  is a promising addition, if and only if  $\pi_i \leq i$  and  $\pi_{i+1} \geq i + 1$ .*

Following Lemma 11, a promising cancellation can be identified by,

► **Corollary 12.** *The short swap  $\rho\langle i, i + 1 \rangle$  on  $\pi$  is a promising cancellation, if and only if  $\pi_i \geq i + 1$  and  $\pi_{i+1} \leq i$ .*

By the following theorem, we state for what an MISP is sub-lucky.

► **Theorem 13.** *An unsorted MISP is sub-lucky if and only if, (1) all elements in the MISP are not vector-zero; and (2) for any two vector-left (resp. vector-right) elements, say  $\pi_i, \pi_j$  in the MISP, if  $i > j$ , then  $\pi_i > \pi_j$ .*

The second property of the theorem implies that those vector-left as well as vector-right elements increase monotonously. In fact, we can use the same way as used to show Theorem 5 to show the theorem. Although in Theorem 13, those two properties are mentioned for an MISP to meet, it cannot refuse an ISP in  $\pi$  to meet those two properties. Thus an ISP is said to meet the Theorem-13 property (1), if all elements in the ISP are not vector-zero; and said to meet the Theorem-13 property (2), if all those vector-left as well as vector-right elements increase monotonously. The following lemma, although seems trivial, deserves to be stated.

► **Lemma 14.** *If an ISP meets those two Theorem-13 properties, then all MISPs in the ISP meet those two Theorem-13 properties.*

To show Theorem 13, we show that an ISP, if meets those two Theorem-13 properties, cannot be transformed by a promising addition into one out of those two Theorem-13 properties. That is,

► **Lemma 15.** *If a promising addition acts on an ISP which meets those two Theorem-13 properties, it must transform the ISP into one which meets those two Theorem-13 properties.*

An ISP with two or more MISPs does not always meet those two Theorem-13 properties. However, Lemma 15 can be extended to fit for some situation where a promising addition affects two MISPs.

► **Lemma 16.** *If a promising addition affects two MISPs, each of which is isolated or meets those two Theorem-13 properties, it must transform the two MISPs into an ISP which meets those two Theorem-13 properties.*

To show Theorem 13, we need to observe on what kind of an ISP a promising cancellation can transform an MISP with those two Theorem-13 properties into.

► **Lemma 17.** *If a promising cancellation acts on an MISP with those two Theorem-13 properties, it must transform the MISP into an ISP in which all unsorted MISPs meets those two Theorem-13 properties.*

Similar to Theorem 5, Theorem 13 can be proved with Lemma 14, 15, 16 and 17.

A best cancellation must switch two elements between which another element has been caught. Thus we will usually denote by  $\rho\langle i, i+2 \rangle$  a best cancellation on  $\pi$ . A best cancellation can be identified by,

► **Lemma 18.** *A short swap, say  $\rho\langle i, i+2 \rangle$  on  $\pi$  is a best cancellation, if and only if  $\pi_i \geq i+2$  and  $\pi_{i+2} \leq i$ .*

In  $\pi$ , there exist  $\lfloor \frac{n}{2} \rfloor$  even elements and  $\lceil \frac{n}{2} \rceil$  odd elements. Thus those even elements in  $\pi$  can be extracted into a subsequence of  $\pi$  as  $[\pi_{x[1]}, \pi_{x[2]}, \dots, \pi_{x[\lfloor \frac{n}{2} \rfloor]}]$  where, (1)  $x[i] < x[i+1]$  for  $1 \leq i \leq \lfloor \frac{n}{2} \rfloor - 1$ ; (2)  $\pi_{x[i]}$  is even in  $\pi$ ,  $1 \leq x[i] \leq n$ . Likewise, those odd elements in  $\pi$  can be extracted into  $[\pi_{y[1]}, \dots, \pi_{y[\lceil \frac{n}{2} \rceil]}]$  where, (1)  $y[i] < y[i+1]$  for  $1 \leq i \leq \lceil \frac{n}{2} \rceil - 1$ ; (2)  $\pi_{y[i]}$  is odd in  $\pi$ ,  $1 \leq y[i] \leq n$ . Moreover, let  $Even[\pi] \equiv [e_1, e_2 \dots e_{\lfloor \frac{n}{2} \rfloor}]$  with  $e_i = \frac{\pi_{x[i]}}{2}$ ,  $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$ ,  $Odd[\pi] \equiv [o_1, o_2 \dots o_{\lceil \frac{n}{2} \rceil}]$  with  $o_i = \frac{\pi_{y[i]}+1}{2}$ ,  $1 \leq i \leq \lceil \frac{n}{2} \rceil$ . Then  $Even[\pi]$  must be a permutation of  $\{1, 2, \dots, \lfloor \frac{n}{2} \rfloor\}$ ,  $Odd[\pi]$  a permutation of  $\{1, 2, \dots, \lceil \frac{n}{2} \rceil\}$ . A sufficient and necessary condition for a permutation to be lucky can be announced by,

► **Theorem 19.** *The permutation  $\pi$  is lucky if and only if, (1) each of its elements admits a vector with zero or even absolute value; (2) each unsorted MISP in  $Even[\pi]$  and  $Odd[\pi]$  is sub-lucky.*

**Proof. Only if:** Let  $\pi$  be lucky and unsorted,  $\rho\langle i, i+2 \rangle$  a best cancellation on  $\pi$ . Then  $\rho\langle i, i+2 \rangle$  must cause  $|v_\pi(\pi_i)|$  as well as  $|v_\pi(\pi_{i+2})|$  to decrease by 2. Since  $\pi$  can be transformed into  $\iota$  by none other than best cancellations,  $|\pi_j - j| \bmod 2 = 0$  for  $1 \leq j \leq n$ . The proof for  $\pi$  to meet the Theorem-19 property (1), is done.

A position-even (resp. position-odd) element in  $\pi$  remains position-even (resp. position-odd) in  $\pi \cdot \rho\langle i, i+2 \rangle$ . Since  $\pi$  meets the Theorem-19 property (1), an even (resp. odd) element in  $\pi$  must be position-even (resp. position-odd). This implies  $Even[\pi] = [\frac{\pi_2}{2}, \frac{\pi_4}{2}, \dots, \frac{\pi_{2\lfloor \frac{n}{2} \rfloor}}{2}]$ ,  $Odd[\pi] = [\frac{\pi_1}{2}, \frac{\pi_3}{2}, \dots, \frac{\pi_{2\lceil \frac{n}{2} \rceil-1}}{2}]$ .



## 14:10 Can a permutation be sorted by best short swaps?

Let  $i$  be even. By Lemma 18,  $\pi_i \geq i + 2$  and  $\pi_{i+2} \leq i$ . Thus  $\frac{\pi_i}{2} \geq \frac{i}{2} + 1$  and  $\frac{\pi_{i+2}}{2} \leq \frac{i}{2}$ . By Corollary 12,  $\rho\langle \frac{i}{2}, \frac{i}{2} + 1 \rangle$  can be viewed as a promising cancellation which acts on an MISP in  $Even[\pi]$ . Thus, if one can use best cancellations to transform  $\pi$  into a permutation, say  $\pi'$  with  $Even[\pi'] = Even[i]$ , then all unsorted MISPs in  $Even[\pi]$  are sub-lucky. The same argument can be employed to show that all unsorted MISPs in  $Odd[\pi]$  are sub-lucky. The proof for  $\pi$  to meet the Theorem-19 property (2), is done.

**If:** Let  $\pi$  be unsorted and meet those two Theorem-19 properties. The proof for  $\pi$  to be lucky, is to show that one can find a best cancellation  $\rho$  on  $\pi$  which transforms  $\pi$  into a permutation which meets those two Theorem-19 properties. Firstly, the Theorem-19 property (1) implies that  $Even[\pi] = [\frac{\pi_2}{2}, \frac{\pi_4}{2}, \dots, \frac{\pi_{2\lfloor \frac{n}{2} \rfloor}}{2}]$ ,  $Odd[\pi] = [\frac{\pi_1}{2}, \frac{\pi_3}{2}, \dots, \frac{\pi_{2\lceil \frac{n}{2} \rceil - 1}}{2}]$ .

Let  $\pi_i$  be the rightmost vector-right element in  $\pi$ . Then  $\pi_{i+2} \leq i + 2$  because  $\pi_{i+2}$  is either vector-zero or vector-left. We argue that if  $i$  is even,  $\rho\langle i, i + 2 \rangle$  must be a best cancellation on  $\pi$ .

(1) Since  $i$  is even,  $\pi_i \geq i + 2$ , and  $\frac{\pi_i}{2}$  and  $\frac{\pi_{i+2}}{2}$  must occur in  $Even[\pi]$ .

(2) To get to  $\pi_{i+2} \leq i$ , we argue that  $\frac{\pi_i}{2}$  and  $\frac{\pi_{i+2}}{2}$  must occur in one unsorted MISP in  $Even[\pi]$ .

It follows  $\pi_{i+2} \leq i + 2$  and  $\pi_i \geq i + 2$  that  $\frac{\pi_i}{2} \geq \frac{i}{2} + 1$  and  $\frac{\pi_{i+2}}{2} \leq \frac{i}{2} + 1$ . Thus  $\frac{\pi_i}{2} > \frac{\pi_{i+2}}{2}$ . Thus an inversion of  $\frac{\pi_i}{2}$  and  $\frac{\pi_{i+2}}{2}$  occurs in  $Even[\pi]$ , which means  $\frac{\pi_i}{2}$  and  $\frac{\pi_{i+2}}{2}$  occur in one MISP. By the Theorem-19 property (2), the MISP in  $Even[\pi]$  with  $\frac{\pi_i}{2}$  and  $\frac{\pi_{i+2}}{2}$  must be sub-lucky. Thus by the Theorem-13 property (1),  $\frac{\pi_{i+2}}{2}$  in  $Even[\pi]$  is not vector-zero. It follows that  $\frac{\pi_{i+2}}{2} \leq \frac{i}{2}$ , and equivalently,  $\pi_{i+2} \leq i$ .

The same argument can be employed to show that if  $i$  is odd,  $\rho\langle i, i + 2 \rangle$  is a best cancellation.

Let  $\pi' = \pi \cdot \rho\langle i, i + 2 \rangle$ . It remains to show that  $\pi'$ , if unsorted, must meet those two Theorem-19 properties.

Since  $\rho\langle i, i + 2 \rangle$  is a best cancellation, it must cause  $|v_\pi(\pi_i)|$  and  $|v_\pi(\pi_{i+2})|$  each to decrease by 2. Since  $\pi$  meet the Theorem-19 property (1),  $\pi'$  must meet the Theorem-19 property (1).

If  $i$  is even, since  $\pi$  meets the Theorem-19 property (1), then  $\frac{\pi_{i+2}}{2}$  must occur on the right side next to  $\frac{\pi_i}{2}$  in  $Even[\pi]$ . Since  $\rho\langle i, i + 2 \rangle$  is a best cancellation,  $\rho\langle \frac{i}{2}, \frac{i}{2} + 1 \rangle$  must be a promising cancellation which acts on an MISP in  $Even[\pi]$ . By Lemma 17, all unsorted MISPs in  $Even[\pi']$  meet those two Theorem-13 properties. That is, all unsorted MISPs in  $Even[\pi']$  are sub-lucky by Theorem 13. Moreover, it follows  $Odd[\pi'] = Odd[\pi]$  that all MISPs in  $Odd[\pi']$  are sub-lucky. Thus,  $\pi'$  meets Theorem-19 property (2)

If  $i$  is odd,  $\pi'$  can be shown to meet the Theorem-19 property (2) in the same way as for  $i$  to be even. ◀

To decide if  $\pi$  meets the Theorem 19 property (1), it suffices to check for all  $i$  in  $[1, n]$ , if  $i$  and  $\pi_i$  are both even, or both odd.

Let  $\pi_i$  be an arbitrary element in  $\pi$ . We refer to  $\frac{\pi_i}{2}$  (resp.  $\frac{\pi_i+1}{2}$ ) as the *image* of  $\pi_i$  in  $Even[\pi]$  (resp.  $Odd[\pi]$ ). Then for a lucky permutation  $\pi$ ,  $\pi_i$  is vector-right (resp. vector-left, vector-zero) in  $\pi$ , if and only if its image in  $Even[\pi]$  or  $Odd[\pi]$  is vector-right (resp. vector-left, vector-zero). Thus, to decide if  $\pi$  meets the Theorem-19 property (2), it suffices to check for, (1) if the image of a vector-zero element occurs in an isolated MISP in  $Odd[\pi]$  or  $Even[\pi]$ ; and (2) if those vector-left and even (resp. odd) elements in  $\pi$ , as well as those vector-right and even (resp. odd) elements, always increase monotonously in the order from  $\pi_1$  to  $\pi_n$ .

The image in  $Even[\pi]$  (resp.  $Odd[\pi]$ ) of a vector-zero element, say  $\pi_i$ , can be decided to occur in an isolated MISP in  $Even[\pi]$  (resp.  $Odd[\pi]$ ) by checking if all even (resp. odd) elements in  $\pi[1 \rightarrow i - 1]$  are smaller than  $\pi_i$ . Those vector-left (resp. vector-right) elements



**Algorithm 3:** How to recognize a lucky permutation.Algorithm *lucky permutation*Input: A permutation  $\pi$ .Output: The best short swap sequence  $\rho$  if  $\pi$  is lucky; *no*, otherwise.

```

1   $lo \leftarrow 0; ro \leftarrow 0; le \leftarrow 0; re \leftarrow 0;$ 
2  For  $i \rightarrow 1$  to  $n$  do
3    If ( $i$  and  $\pi_i$  are both even) then
4      If ( $\pi_i \geq i$  and  $\pi_i > re$ )
5        then  $re \leftarrow \pi_i; i \leftarrow i + 1;$  ( $\pi_i$  is vector-right even or  $[\pi_i]$  is isolated)
6      If ( $\pi_i < i$  and  $\pi_i > le$ )
7        then  $le \leftarrow \pi_i; i \leftarrow i + 1;$  ( $\pi_i$  is vector-left even)
8    If ( $i$  and  $\pi_i$  are both odd) then
9      If ( $\pi_i \geq i$  and  $\pi_i > ro$ )
10       then  $ro \leftarrow \pi_i; i \leftarrow i + 1;$  ( $\pi_i$  is vector-right odd or  $[\pi_i]$  is isolated)
11     If ( $\pi_i < i$  and  $\pi_i > lo$ )
12       then  $lo \leftarrow \pi_i; i \leftarrow i + 1;$  ( $\pi_i$  is vector-left odd)
13   Else return no;
14 End for
15 Return Sort( $\pi$ );
```

can be decided to be monotonous increasing by checking for each vector-left (resp. vector-right) even (resp. odd) element, say  $\pi_i$ , if  $\pi_i$  is bigger than the biggest vector-left (resp. vector-right) even (resp. odd) element in  $\pi[1 \rightarrow i - 1]$ . In fact, it is not necessary to pay special attention to check if a vector-zero element occurs in an isolated MIS. This benefits from

► **Lemma 20.** *In  $\pi[1 \rightarrow k]$  for  $k \geq 2$ , the biggest vector-right element must be bigger than the biggest vector-left element.*

We present in Figure 3 the algorithm to decide if  $\pi$  is lucky, and if so, to find a best cancellation sequence to sort  $\pi$ . If  $\pi$  is lucky, the algorithm will return a best cancellation sequence which can transform  $\pi$  into  $\iota$  by invoking the Sort( $\pi$ ); return *no*, otherwise. Since by the sufficiency proof of Theorem 19, one can employ the same way as to find a best short swap in Theorem 5 to find a best cancellation, the subroutine Sort( $\pi$ ) is just so as it has been depicted in Algorithm 2.

In the algorithm description, we use the integer parameter  $le$  (resp.  $lo$ ) to maintain the biggest vector-left even (resp. odd) element in  $\pi[1 \rightarrow i - 1]$ ,  $re$  (resp.  $ro$ ) the biggest even (odd) element in  $\pi[1 \rightarrow i - 1]$ . It follows Lemma 20 that  $le < re, lo < ro$ .

Running the algorithm from Step 1 to Step 14 can inform us if  $\pi$  is lucky or not. This takes  $O(n)$  time, where  $n$  is the number of elements in  $\pi$ . Let  $\pi_i$  be the rightmost vector-right element in a lucky permutation  $\pi$ , by the proof of Theorem 19, the short swap which switches  $\pi_i$  with  $\pi_{i+2}$  is a best cancellation. By Theorem 19 again, this operation must transform  $\pi$  into a lucky permutation. By the complexity analysis for Sort( $\pi$ ) in Section 3, it has been known Sort( $\pi$ ) can run in  $O(n^2)$  time. Thus the time complexity of sorting a lucky permutation is  $O(n^2)$ .

## 5 Conclusion

Sort a happy permutation or a lucky permutation by short swaps is a special case of minimum sorting by short swaps problem. In this paper, we proposed a polynomial-time algorithm

to recognize a happy permutation and sort it with the fewest short swaps. We also gave a new algorithm to recognize a lucky permutation with  $O(n)$  steps, which improves the time complexity of  $O(n^2)$  [9]. The complexity of minimum sorting by short swaps problem remains open. The best known approximation ratio of this problem is 2, which was given by Heath and Vergara [9]. It is interesting that if we can get a smaller approximation ratio for this problem.

---

## References

- 1 V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *Siam Journal on Computing*, 25(2):272–289, 1993.
- 2 Piotr Berman, Sridhar Hannenhalli, and Marek Karpinski. 1.375-approximation algorithm for sorting by reversals. *Lecture Notes in Computer Science*, pages 200–210, 2002.
- 3 Guillaume Bourque and Pavel A. Pevzner. Genome-scale evolution: Reconstructing gene orders in the ancestral species. *Genome Research*, 12(1):26–36, 2002.
- 4 Alberto Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *Siam Journal on Discrete Mathematics*, 12(1):91–110, 1999.
- 5 Xuerong Feng, Ivan Hal Sudborough, and Enyue Lu. A fast algorithm for sorting by short swap. In *Proceeding of the 10th IASTED International Conference on Computational and Systems Biology*, pages 62–67, 2006.
- 6 Gustavo Rodrigues Galvão and Zaroni Dias. Approximation algorithms for sorting by signed short reversals. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB '14, Newport Beach, California, USA, September 20-23, 2014*, pages 360–369. ACM, 2014. doi:10.1145/2649387.2649413.
- 7 Gustavo Rodrigues Galvão, Orlando Lee, and Zaroni Dias. Sorting signed permutations by short operations. *Algorithms for Molecular Biology*, 10(1):1–17, 2015.
- 8 Sridhar Hannenhalli. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the Acm*, 46(1):1–27, 1999.
- 9 L. S. Heath and J. P. Vergara. Sorting by short swaps. *Journal of Computational Biology A Journal of Computational Molecular Cell Biology*, 10(5):775–89, 2003.
- 10 Mark R. Jerrum. The complexity of finding minimum-length generator sequences. In *Colloquium on Automata, Languages and Programming*, pages 270–280, 1984.
- 11 Haim Kaplan, Ron Shamir, and Robert E. Tarjan. *A Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals*. Society for Industrial and Applied Mathematics, 1999.
- 12 J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):180–210, 1995.
- 13 P Pevzner and G Tesler. Genome rearrangements in mammalian evolution: lessons from human and mouse genomes. *Genome Research*, 13(1):37–45, 2003.
- 14 G. P. Pradhan and P. V. Prasad. Evaluation of wheat chromosome translocation lines for high temperature stress tolerance at grain filling stage. *Plos One*, 10(2):1–20, 2015.
- 15 D. Sankoff, G. Leduc, N. Antoine, B. Paquin, B F Lang, and R. Cedergren. Gene order comparisons for phylogenetic inference: evolution of the mitochondrial genome. *Proceedings of the National Academy of Sciences of the United States of America*, 89(14):6575–6579, 1992.
- 16 G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7, 1982.

# Computing longest common square subsequences

**Takafumi Inoue**

Department of Informatics, Kyushu University, Japan

**Shunsuke Inenaga**

Department of Informatics, Kyushu University, Japan

inenaga@inf.kyushu-u.ac.jp

**Heikki Hyyrö**


Faculty of Natural Sciences, University of Tampere, Finland

heikki.hyyro@uta.fi

**Hideo Bannai**

Department of Informatics, Kyushu University, Japan

bannai@inf.kyushu-u.ac.jp

 <https://orcid.org/0000-0002-6856-5185>

**Masayuki Takeda**

Department of Informatics, Kyushu University, Japan

takeda@inf.kyushu-u.ac.jp

---

## Abstract

A *square* is a non-empty string of form  $YY$ . The *longest common square subsequence* (LCSqS) problem is to compute a longest square occurring as a subsequence in two given strings  $A$  and  $B$ . We show that the problem can easily be solved in  $O(n^6)$  time or  $O(|\mathcal{M}|n^4)$  time with  $O(n^4)$  space, where  $n$  is the length of the strings and  $\mathcal{M}$  is the set of matching points between  $A$  and  $B$ . Then, we show that the problem can also be solved in  $O(\sigma|\mathcal{M}|^3 + n)$  time and  $O(|\mathcal{M}|^2 + n)$  space, or in  $O(|\mathcal{M}|^3 \log^2 n \log \log n + n)$  time with  $O(|\mathcal{M}|^3 + n)$  space, where  $\sigma$  is the number of distinct characters occurring in  $A$  and  $B$ . We also study lower bounds for the LCSqS problem for two or more strings.

**2012 ACM Subject Classification** Mathematics of computing → Combinatorial algorithms

**Keywords and phrases** squares, subsequences, matching rectangles, dynamic programming

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.15

**Acknowledgements** The authors thank the anonymous referees for correcting errors involved in an earlier version of this paper.

## 1 Introduction

Computing the *longest common subsequence* (LCS) of given strings is the fundamental way to compare the strings. Given two strings  $A$  and  $B$  of length  $n$  each, the basic dynamic programming solution computes the LCS of  $A$  and  $B$  in  $O(n^2)$  time and space [27]. While faster solutions for the LCS problem exist, such as those running in  $O(n^2/\log^2 n)$  time for constant-size alphabets [22], and in  $O(n^2(\log \log n)^2/\log^2 n)$  time or in  $O(n^2 \log \log n/\log^2 n)$  time for non constant-size alphabets [5, 12]<sup>1</sup>, no strongly sub-quadratic  $O(n^{2-\epsilon})$ -time

---

<sup>1</sup> Grabowski's method [12] works when the length  $m$  of one string is at least  $\log^2 n$ , where  $n$  is the length of the other string.



solutions are known for any constant  $\epsilon > 0$ . Difficulty in breaking this barrier is supported by recent studies on *conditional lower bounds* for string similarity measures: It is shown in [1] that if there is an  $O(n^{2-\epsilon})$ -time solution for the LCS problem with a constant  $\epsilon > 0$ , then the famous *strong exponential time hypothesis* (*SETH*) fails.

To reflect a priori knowledge to the solution to be found, many variants of the LCS problem where some *constraints* are introduced in the solution have been considered (see e.g. [7, 2, 14, 20, 9, 10, 28, 11, 29, 30, 8, 16, 19]).

This paper considers a new variant of the LCS problem where the solution must be a *square* (of form  $YY$  with some string  $Y$ ), called the *longest common square subsequence* (*LCSqS*) problem defined as follows: Given two strings  $A$  and  $B$  of length  $n$ , compute (the length of) a longest square which appears as a subsequence in  $A$  and  $B$ . For instance, for  $A = \text{bacbcbdbaca}$  and  $B = \text{dbcacbbcacd}$ , their LCSqSs are  $\text{bacbac}$  and  $\text{bcabca}$  of length 6.

We propose several solutions for the LCSqS problem. We first show that there is a simple  $O(n^6)$ -time  $O(n^4)$ -space solution for the LCSqS problem. The algorithm is also improved to  $O(|\mathcal{M}|n^4)$ -time by using the set  $\mathcal{M}$  of matching points between the two input strings. Albeit  $\mathcal{M}$  can be as large as  $O(n^2)$  in the worst case, it can be smaller in many cases. We then give two more sophisticated algorithms based on the set  $\mathcal{R}$  of *matching rectangles*: one runs in  $O(\sigma|\mathcal{M}||\mathcal{R}| + n) = O(\sigma|\mathcal{M}|^3 + n)$  time with  $O(|\mathcal{M}|^2 + n)$  space, and the other in  $O(|\mathcal{M}||\mathcal{R}| \log^2 \log \log n + |\mathcal{M}|^3 + n) = O(|\mathcal{M}|^3 \log^2 \log \log n + n)$  time with  $O(|\mathcal{M}|^3 + n)$  space, where  $\sigma$  denotes the number of distinct characters that appear in both strings. These two solutions are faster than the simple  $O(n^6)$ -time or  $O(|\mathcal{M}|n^4)$ -time solutions when  $\mathcal{M}$  is sparse. Note e.g. that under uniformly distributed random text  $|\mathcal{M}| \approx n^2/\sigma$  and  $|\mathcal{R}| \approx |\mathcal{M}|^2/\sigma \approx n^4/\sigma^3$ , in which case the *expected* running times of our three algorithms would be  $O(n^6/\sigma)$ ,  $O(n^6/\sigma^3)$  and  $O(n^6(\log^2 \log \log n + \sigma)/\sigma^4)$  respectively.

The set  $\mathcal{M}$  of matching points can easily be computed in  $O(|\mathcal{M}| + n)$  time under a common assumption that the input strings are over an integer alphabet of size  $n^{O(1)}$ .

We also study hardness of the LCSqS problem for two or more strings. The  $k$ -LCSqS problem is to compute the LCSqS of given  $k \geq 2$  strings. We show that the  $k$ -LCSqS problem is at least as hard as the  $2k$ -LCS problem which asks to compute the LCS of  $2k$  given strings. This implies that for unfixed  $k$  the  $k$ -LCSqS problem is NP-hard, and that for fixed  $k$  it seems hard to solve the  $k$ -LCSqS problem in  $O(n^{k-\epsilon})$  time for any constant  $\epsilon > 0$ .

## Related work

It is known that one can compute (the length of) a *longest square subsequence* (*LSqS*) of a single string of length  $n$  in  $O(n^2)$  time and  $O(n)$  space [18]. Also, it is shown in [1] that if there is an  $O(n^{2-\epsilon})$ -time solution for the LSqS problem with a constant  $\epsilon > 0$ , then the famous *strong exponential time hypothesis* (*SETH*) fails. Our results for the LCSqS problem can be seen as a generalization of these results for the LSqS problem.

Technically speaking, our results for the LCSqS problem are most related to those for the *longest common palindromic subsequence* (*LCPS*) problem, where the task is to find a longest palindrome that appears as a subsequence in both of the two strings  $A$  and  $B$ . Chowdhury et al. [8] were the first to consider the LCPS problem, giving an  $O(n^4)$ -time solution and an  $O(|\mathcal{M}|^2 \log^2 n \log \log n + n)$ -time solution<sup>2</sup>. Inenaga and Hyyrö [16] proposed another

<sup>2</sup> Our careful analysis reveals that Chowdhury et al.'s algorithm [8] uses at least  $\Omega(\min\{|\mathcal{M}|^2 n^2 \log n, n^3\})$  space (and hence time), but it can be fixed to run in  $O(|\mathcal{M}|^2 \log^2 n \log \log n + n)$  time using our technique proposed in Section 3.

algorithm which solves the LCPS problem in  $O(\sigma|\mathcal{M}|^2 + n)$  time and  $O(|\mathcal{M}|^2 + n)$  space. Very recently, Bae and Lee [3] showed how to solve the LCPS problem in  $O(|\mathcal{M}|^2 + n)$  time. Inenaga and Hyvrö [16] also showed that the LCPS problem for two strings is at least as hard as the LCS problem for four strings, implying that it seems hard to solve the LCPS problem in  $O(n^{4-\epsilon})$  time for any constant  $\epsilon > 0$ .

## 2 Preliminaries

Let  $\Sigma$  be the alphabet. An element  $X$  of  $\Sigma^*$  is called a string. The length of string  $X$  is denoted by  $|X|$ . For any  $1 \leq i \leq |X|$ ,  $X[i]$  denotes the  $i$ th character of  $X$ . For any  $1 \leq i \leq j \leq |X|$ ,  $X[i..j]$  denotes the substring of  $X$  beginning at position  $i$  and ending at position  $j$ .

A string  $X$  is said to be a *subsequence* of another string  $Y$  if there exists a sequence  $1 \leq i_1 < \dots < i_{|X|} \leq |Y|$  of increasing positions of  $Y$  such that  $X = Y[i_1] \dots Y[i_{|X|}]$ . In other words, a subsequence of  $Y$  can be obtained by removing zero or more characters from  $Y$ . The  $k$ -LCS problem is to compute the length of a *longest common subsequence* (LCS) of given  $k$  strings, where  $k \geq 2$ . Let  $LCS(A_1, \dots, A_k)$  denote the length of a longest common subsequence of  $k$  strings  $A_1, \dots, A_k$ . A non-empty string  $X$  of length  $2k$  is called a *square* if there exists a string  $Y$  of length  $k$  such that  $X = YY$ . A square  $S$  is called a *square subsequence* of another string  $Y$  if square  $S$  is a subsequence of  $Y$ . Let  $LCSqS(A, B)$  denote the length of a *longest common square subsequence* (LCSqS) of strings  $A$  and  $B$ . This paper deals with the problem of computing  $LCSqS(A, B)$  for two given strings  $A$  and  $B$ . For simplicity, we assume that the input strings  $A$  and  $B$  are of the same length and let  $n = |A| = |B|$ . Our algorithms can easily be extended to the case where  $|A| \neq |B|$  as well as to the case where we wish to compute one longest common square subsequence of  $A$  and  $B$ .

For two strings  $A$  and  $B$ , a pair  $(i, j)$  of positions  $1 \leq i \leq |A|$  and  $1 \leq j \leq |B|$  is said to be a *matching point* if  $A[i] = B[j]$ . The set of all matching positions of  $A$  and  $B$  is denoted by  $\mathcal{M}(A, B)$ , namely,  $\mathcal{M}(A, B) = \{(i, j) \mid 1 \leq i \leq |A|, 1 \leq j \leq |B|, A[i] = B[j]\}$ . We will abbreviate  $\mathcal{M}(A, B)$  as  $\mathcal{M}$  when it is clear from the context.

## 3 Algorithms

In this section, we present several algorithms for computing  $LCSqS(A, B)$ . In order to avoid processing unnecessary characters, we will assume that the input strings  $A$  and  $B$  have been already preprocessed by an alphabet reduction technique [16] as follows: First, we compute the lexicographical ranks of the characters in  $A$  and  $B$ . Assuming that  $A$  and  $B$  are drawn from an integer alphabet of size  $n^{O(1)}$ , this can be done in  $O(n)$  time with radix sort. We then replace each character in  $A$  and  $B$  with its rank, turning  $A$  and  $B$  into strings over the integer alphabet  $[1, 2n]$ . Then we remove every character that appears only either in  $A$  or in  $B$ . It is clear that this preprocessing essentially preserves common subsequences between the original  $A$  and  $B$  and thus has no negative effect on computing  $LCSqS(A, B)$ . Note that  $n \leq \mathcal{M}$  holds after alphabet reduction, while  $\mathcal{M} = O(n^2)$  still also holds.

### 3.1 Simple Algorithm

Our first algorithm considers  $\Theta(n^2)$  pairs of partitioning of  $A$  and  $B$ . Namely, we have that

$$LCSqS(A, B) = \max_{1 \leq i < n, 1 \leq j < n} \{2 \times LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n])\}.$$

This immediately implies an  $O(n^6)$ -time  $O(n^4)$ -space algorithm for computing  $LCSqS(A, B)$ , since the LCS of four strings can be computed in  $O(n^4)$  time and space by standard DP.

The  $O(n^6)$ -time complexity can be improved as follows. For any matching point  $(i, j) \in \mathcal{M}$ , let  $i'$  (resp.  $j'$ ) be the smallest position such that  $i < i'$ ,  $j < j'$ , and  $(i', j') \in \mathcal{M}$ . If such  $(i', j')$  does not exist, then let  $i' = j' = n$ .

► **Observation 1.** For any  $i \leq k < i'$  and  $j \leq h < j'$ ,  $LCS(A[1..k], A[k+1..n], B[1..h], B[h+1..n]) = LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n])$ .

By Observation 1, it is sufficient for us to consider only  $|\mathcal{M}|$  partition points between  $A$  and  $B$ . Hence, we can compute  $LCSqS(A, B)$  in  $O(|\mathcal{M}|n^4)$  time and  $O(n^4)$  space.

### 3.2 $O(\sigma|\mathcal{M}|^3 + n)$ -time algorithm

Here we present our  $O(\sigma|\mathcal{M}|^3 + n)$ -time algorithm for computing  $LCSqS(A, B)$ , where  $\sigma$  is the number of distinct characters occurring in  $A$  and  $B$ . This algorithm is based on Inenaga and Hyyrö's algorithm [16] which computes (the length of) a *longest palindromic common subsequence* of two given strings in  $O(\sigma|\mathcal{M}|^2 + n)$  time. Consider a 2D plain where the string  $A$  corresponds to the vertical axis upward (i.e.,  $A[1]$  is on the bottom and  $A[n]$  is on the top), and the string  $B$  corresponds to the horizontal axis rightward (i.e.,  $B[1]$  is on the left end and  $B[n]$  is on the right end). Our key idea is to represent each common square subsequence of strings  $A$  and  $B$  by matching rectangles defined as follows: For  $1 \leq i < j \leq n$  and  $1 \leq k < l \leq n$ , a tuple  $r = (i, j, k, l)$  is said to be a *matching rectangle* iff  $A[i] = A[j] = B[k] = B[l]$ , and more specifically a  $c$ -matching rectangle iff  $A[i] = A[j] = B[k] = B[l] = c$ . For a matching rectangle  $r = (i, j, k, l)$ ,  $(i, k)$  is said to be the left-bottom corner of  $r$ , and  $(j, l)$  is said to be the right-upper corner of  $r$ . Let  $\mathcal{R}$  denote the set of matching rectangles of  $A$  and  $B$ . Notice  $|\mathcal{R}| = O(|\mathcal{M}|^2)$ . For two matching rectangles  $r = (i, j, k, l)$  and  $r' = (i', j', k', l')$ , let

$$\begin{aligned} r = r' &\iff i = i', j = j', k = k', \text{ and } l = l' \\ r < r' &\iff i < i', j < j', k < k', \text{ and } l < l' \\ r \triangleleft r' &\iff i \leq i', j \leq j', k \leq k', l \leq l', \text{ and } r \neq r'. \end{aligned}$$

For two  $c$ -matching rectangles  $r = (i, j, k, l)$  and  $r' = (i', j', k', l')$ , let

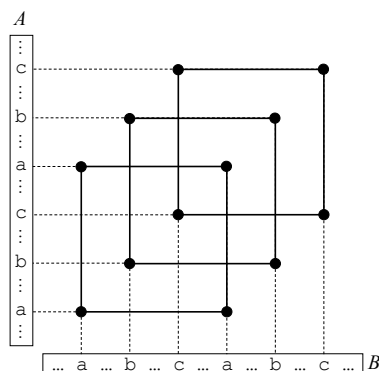
$$r \preceq r' \iff i \leq i', j \leq j', k \leq k' \text{ and } l \leq l'.$$

A sequence  $\langle r_1, \dots, r_m \rangle$  of matching rectangles is said to be a sequence of *diagonally overlapping matching rectangles (DOMRs)* iff  $r_x < r_{x+1}$  for all  $1 \leq x < m$ ,  $i_m < j_1$  and  $k_m < l_1$ , where we use the notation  $r_h = (i_h, j_h, k_h, l_h)$  for all  $h = 1, \dots, m$ . The *size* of a sequence  $\langle r_1, \dots, r_m \rangle$  of DOMRs is the number  $m$  of overlapping rectangles in it.

The following observation lays the foundation to the algorithms of this subsection (and to the one of the following subsection as well):

► **Observation 2.** There is a common square subsequence  $T$  of length  $2m$  of strings  $A$  and  $B$  iff there exists a sequence  $\langle r_1, \dots, r_m \rangle$  of DOMRs of length  $m$ .

See Figure 1 which depicts the relationship between common square subsequences and DOMRs for two strings  $A$  and  $B$ . By Observation 2, the problem of computing  $LCSqS(A, B)$  reduces to the problem of finding a longest sequence of DOMRs.



■ **Figure 1** Illustration of the relationship between common square subsequences and DOMRs.

The basic idea of our algorithm is to extend a given sequence  $S = \langle r_1, \dots, r_m \rangle$  of DOMRs by adding a new matching rectangle to its right-end. We say that a  $c$ -matching rectangle  $r = (i, j, k, l)$  is a  $c$ -extension of  $S$  if  $\langle r_1, \dots, r_m, r \rangle$  is a sequence of DOMRs. A  $c$ -extension  $r$  of  $S$  is *dominant* if the condition  $r \preceq r'$  holds between  $r$  and any  $c$ -extension  $r'$  of  $S$ . The algorithms in this subsection are based on the following lemmas.

► **Lemma 3.** *Let  $S = \langle r_1, \dots, r_m \rangle$  be any sequence of DOMRs. If  $S$  has at least one  $c$ -extension, then  $S$  has a unique dominant  $c$ -extension  $r'$ . It is furthermore possible to compute any such  $r'$  in  $O(1)$  time after initial preprocessing of  $A$  and  $B$  in  $O(\sigma n)$  time and space.*

**Proof.** Consider  $r' = (i', j', k', l')$ , where  $i' = \min(\{i \mid i_m < i < j_1, A[i] = c\} \cup \{n+1\})$ ,  $j' = \min(\{j \mid j_m < j, A[j] = c\} \cup \{n+1\})$ ,  $k' = \min(\{k \mid k_m < k < l_1, B[k] = c\} \cup \{n+1\})$  and  $l' = \min(\{l \mid l > l_m, B[l] = c\} \cup \{n+1\})$ . If any of  $i'$ ,  $j'$ ,  $k'$  and  $l'$  holds the sentinel value  $n+1$  that corresponds to non-existence of a further suitable match with  $c$ , then  $S$  cannot have any  $c$ -extension. Otherwise  $A[i'] = A[j'] = B[k'] = B[l'] = c$  and  $r'$  is a  $c$ -matching rectangle. Furthermore  $i_m < i'$ ,  $j_m < j'$ ,  $k_m < k'$ ,  $l_m < l'$ ,  $i' < j_1$  and  $k' < l_1$ , so  $r'$  is a  $c$ -extension of  $S$ . If we assume the existence of another  $c$ -extension  $r''$  of  $S$  such that  $r'' \preceq r'$  does not hold, then at least one of the definitions of  $i'$ ,  $j'$ ,  $k'$  and  $l'$  above is contradicted. Hence  $r'$  must be dominant. Finally,  $r'$  must clearly be unique: if also  $r'' \neq r'$  is a dominant  $c$ -extension, then both  $r' \preceq r''$  and  $r'' \preceq r'$  must hold, but this is possible only if  $r'' = r'$ .

The values  $i'$  and  $j'$  can be computed in  $O(1)$  time by using a precomputed table  $P_A$  of size  $\sigma \times n$  that holds the values  $P_A[c, h] = \min(\{i \mid h < i, A[i] = c\} \cup \{n+1\})$  for all  $c \in \Sigma$  and  $1 \leq h \leq n$ . The values  $k'$  and  $l'$  can be computed in  $O(1)$  time by using an analogous precomputed table  $P_B$  with values  $P_B[c, h] = \min(\{i \mid h < i, B[i] = c\} \cup \{n+1\})$ . Both tables can be precomputed in  $O(\sigma n)$  time and space in a straight-forward manner. ◀

Note that the proof of Lemma 3 refers only to  $r_1$  and  $r_m$  when determining the unique dominant extension of  $\langle r_1, \dots, r_m \rangle$ : any inner rectangle  $r_i$  for  $1 < i < m$  does not need to be considered. Thus all sequences of DOMRs that begin with the rectangle  $r_1$  and end with the rectangle  $r_m$  share the same unique dominant extensions.

► **Lemma 4.** *Let  $S = \langle r_1, \dots, r_m \rangle$  be any sequence of at least two DOMRs. If any  $c$ -matching rectangle  $r_h$  with  $1 < h \leq m$  is replaced by the dominant  $c$ -extension of  $\langle r_1, \dots, r_{h-1} \rangle$ , also the resulting sequence of matching rectangles is a sequence of DOMRs.*

**Proof.** The lemma clearly holds if  $h = m$ , so consider the case  $1 < h < m$ . Let  $(i', j', k', l')$  be the dominant  $c$ -extension of  $\langle r_1, \dots, r_{h-1} \rangle$ , and let  $S' = \langle r'_1, \dots, r'_m \rangle$  denote the sequence obtained from  $S$  by replacing  $r_h$  with  $(i', j', k', l')$ .  $S'$  is a sequence of DOMRs, and thus



$i'_m = i_m < j_1 = j'_1$ ,  $k'_m = k_m < l_1 = l'_1$ , and  $r_x < r_{x+1}$  for  $1 \leq x < m$ . On the other hand  $r'_{h-1} < r'_h$ , as also  $\langle r'_1, \dots, r'_h \rangle = \langle r_1, \dots, r_{h-1}, (i', j', k', l') \rangle$  is a sequence of DOMRs. Because  $r'_h$  is dominant, we have  $r'_{h-1} < r'_h \preceq r_h < r_{h+1} = r'_{h+1}$ , which in turn implies that  $r'_h < r'_{h+1}$  for  $1 \leq h < m$ , and hence  $S'$  fulfills all conditions of a sequence of DOMRs. ◀

**Basic algorithm.** The basic principle of our first rectangle-based algorithm, Algorithm 1, is to fix the first left-bottom matching rectangle  $r_b$ , and then try to extend it as long as possible to the right-upper direction. For each such starting rectangle  $r_b$ , we compute a dynamic programming table  $DP_{r_b}$  of size  $O(|\mathcal{M}|^2)$  such that  $DP_{r_b}[r_e]$  will finally store the length of the longest sequence of DOMRs beginning with  $r_b$  and ending with  $r_e$ , where  $r_e$  is either  $r_b$  itself or a dominant extension. In more detail, Algorithm 1 works as follows:

---

**Algorithm 1:**

---

**Preprocessing:** Compute a list  $L$  of all matching rectangles sorted according to  $<$  and  $\preceq$  by radix sorting all rectangles  $(i, j, k, l)$  as 4-digit numbers.

**Compute longest sequence of DOMRs:** For each matching rectangle  $r_b$  (in any order), perform the following:

- (1) For each  $r_e (\neq r_b)$ , we initialize  $DP_{r_b}[r_e] \leftarrow 0$ . We let  $DP_{r_b}[r_b] \leftarrow 2$ .
  - (2) Suppose  $r_b$  is the  $i$ th element of  $L$ . For each  $j = i + 1, \dots, |L|$  in increasing order, let  $r \leftarrow L[j]$  and attempt to extend a sequence  $\langle r_b, \dots, r \rangle$  of DOMRs as follows:
    - (a) If  $DP_{r_b}[r] = 0$ , then no sequence of DOMRs of form  $\langle r_b, \dots, r \rangle$  exists.
    - (b) Otherwise, for each character  $c$ , try to compute the unique dominant  $c$ -extension  $r'$  of any sequence  $\langle r_b, \dots, r \rangle$  of DOMRs which begins with  $r_b$  and ends with  $r$ . If such  $r'$  exists, set  $DP_{r_b}[r'] \leftarrow \max\{DP_{r_b}[r'], DP_{r_b}[r] + 2\}$ .
  - (3) If the maximum value in  $DP_{r_b}$  exceeds the current best solution, then update it.
- 

Let us explain the correctness of Algorithm 1. Lemma 4 guarantees that an optimal sequence of DOMRs can be constructed by considering only dominant extensions. Consider any such optimal sequence of DOMRs  $S = \langle r_1, \dots, r_m \rangle$ . The outer loop of Algorithm 1 will at some point select  $r_b = r_1$ . As  $r \leftarrow L[j]$  are processed in increasing order of  $j$ , the sorting order of  $L$  guarantees that rectangles  $r_i$  of  $S$  will be selected as the current  $r$  in the order  $i = 1, \dots, m$ . For each such  $r = r_i$ , the algorithm uses Lemma 3 to consider all possible dominant extensions, including also the extension  $r_{i+1}$  if  $i < m$ . A simple inductive argument shows that the values  $DP_{r_1}[r_i]$  will become correctly computed in the order  $i = 1, \dots, m$ .

Let us analyze the efficiency of Algorithm 1. Constructing the tables  $P_A$  and  $P_B$  takes  $O(\sigma n)$  time and space. Note that alphabet reduction guarantees that  $O(\sigma n) = O(\sigma |\mathcal{M}|)$ . Since  $1 \leq i, j, k, l \leq n$  for each matching rectangle  $(i, j, k, l)$ , we obtain a sorted list  $L$  of all  $O(|\mathcal{M}|^2)$  matching rectangles in  $O(|\mathcal{M}|^2 + n)$  time and space by radix sort. Hence the preprocessing takes  $O(|\mathcal{M}|^2 + n)$  total time and space. We test no more than  $\sigma$  characters for any cell  $DP_{r_b}[r]$  of the dynamic programming table  $DP_{r_b}$ . By Lemma 3, we can compute a unique dominant  $c$ -extension in  $O(1)$  time, if it exists. Since there are  $O(|\mathcal{M}|^2)$  candidates for  $r_b$  and  $O(|\mathcal{R}|) = O(|\mathcal{M}|^2)$  candidates for  $r$ , Algorithm 1 takes overall  $O(\sigma |\mathcal{M}|^4 + n)$  time and  $O(|\mathcal{M}|^2 + n)$  space.

**Improved algorithm.** Now we show how to reduce the number of candidates for the starting rectangle  $r_b$ . We give proof for Lemma 5. Lemmas 6 and 7 can be proven similarly.



► **Lemma 5.** *Let  $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$  and  $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$  be any matching rectangles s.t.  $i_{b_1} < i_{b_2}$ ,  $j_{b_1} = j_{b_2}$ ,  $k_{b_1} = k_{b_2}$ , and  $l_{b_1} = l_{b_2}$ . Let  $\ell_1$  and  $\ell_2$  be the lengths of LCSqS of  $A$  and  $B$  whose corresponding sequences of DOMRs begin with  $r_{b_1}$  and  $r_{b_2}$ , respectively. Then,  $\ell_1 \geq \ell_2$ .*

**Proof.** See Figure 2 for illustration. It follows from  $j_{b_1} = j_{b_2}$ ,  $k_{b_1} = k_{b_2}$ , and  $l_{b_1} = l_{b_2}$  that the two matching rectangles  $r_{b_1}$  and  $r_{b_2}$  correspond to the same character. Let  $\langle r_{b_2,1}, r_{b_2,2}, \dots, r_{b_2,\ell_2} \rangle$  be any sequence of DOMRs which begins with  $r_{b_2}$  and represents a common square subsequence of length  $\ell_2$ , namely  $r_{b_2} = r_{b_2,1}$ . Since  $i_{b_1} < i_{b_2}$ ,  $j_{b_1} = j_{b_2}$ ,  $k_{b_1} = k_{b_2}$ , and  $l_{b_1} = l_{b_2}$ ,  $\langle r_{b_1}, r_{b_2,2}, \dots, r_{b_2,\ell_2} \rangle$  is a sequence of DOMRs which begins with  $r_{b_1}$  and represents a common square subsequence of length  $\ell_2$ . This implies that  $\ell_1 \geq \ell_2$ . ◀

► **Lemma 6.** *Let  $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$  and  $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$  be any matching rectangles s.t.  $i_{b_1} = i_{b_2}$ ,  $j_{b_1} = j_{b_2}$ ,  $k_{b_1} < k_{b_2}$ , and  $l_{b_1} = l_{b_2}$ . Let  $\ell_1$  and  $\ell_2$  be the lengths of LCSqS of  $A$  and  $B$  whose corresponding sequences of DOMRs begin with  $r_{b_1}$  and  $r_{b_2}$ , respectively. Then,  $\ell_1 \geq \ell_2$ .*

► **Lemma 7.** *Let  $r_{b_1} = (i_{b_1}, j_{b_1}, k_{b_1}, l_{b_1})$  and  $r_{b_2} = (i_{b_2}, j_{b_2}, k_{b_2}, l_{b_2})$  be any matching rectangles such that  $i_{b_1} < i_{b_2}$ ,  $j_{b_1} = j_{b_2}$ ,  $k_{b_1} < k_{b_2}$ , and  $l_{b_1} = l_{b_2}$ . Let  $\ell_1$  and  $\ell_2$  be the lengths of longest common square subsequences of  $A$  and  $B$  whose corresponding sequences of DOMRs begin with  $r_{b_1}$  and  $r_{b_2}$ , respectively. Then,  $\ell_1 \geq \ell_2$ .*

It follows from Lemmas 5–7 that it suffices to consider only all right-upper corners  $(j_b, l_b)$  instead of all matching rectangles  $r_b = (i_b, j_b, k_b, l_b)$ . Namely, for each arbitrarily fixed right-upper corner  $(j_b, l_b)$  such that  $A[j_b] = B[l_b] = c$ , we can always use  $(i_{\min}, k_{\min})$  as its left-bottom corner, where  $i_{\min}$  and  $k_{\min}$  are respectively the left-most occurrences of character  $c$  in  $A$  and  $B$ . The following is our improved algorithm.

---

**Algorithm 2:**


---

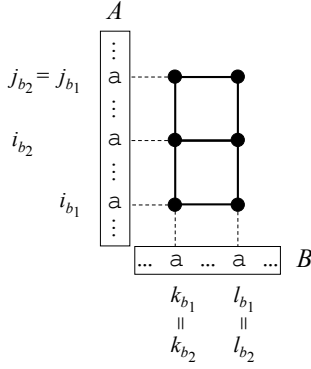
**Preprocessing:** As in Algorithm 1, but now also precompute positions  $i_b = \min\{i \mid A[i] = c\}$  and  $k_b = \min\{k \mid B[k] = c\}$  for each character  $c$  that appears in  $A$  and  $B$ .

**Computing longest sequence of DOMRs:** For each matching point  $p_b = (j_b, l_b) \in \mathcal{M}$  we perform the following:

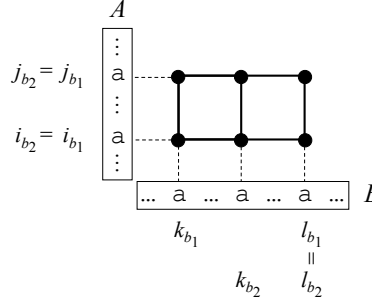
- (i) Let  $c = A[j_b] = B[l_b]$ . We compute  $i_b = \min\{i \mid A[i] = c\}$  and  $k_b = \min\{k \mid B[k] = c\}$ , and let  $r_b \leftarrow (i_b, j_b, k_b, l_b)$ . If  $i_b = j_b$  or  $k_b = l_b$ , then we stop processing the current matching point and proceed to the next matching point in  $\mathcal{M}$ .
  - (ii) Perform the same procedures (1)–(3) as in Algorithm 1.
  - (iii) If the maximum value in  $DP_{r_b}$  exceeds the current best solution, then update it.
- 

The correctness of Algorithm 2 follows from that of Algorithm 1 and Lemmas 5–7.

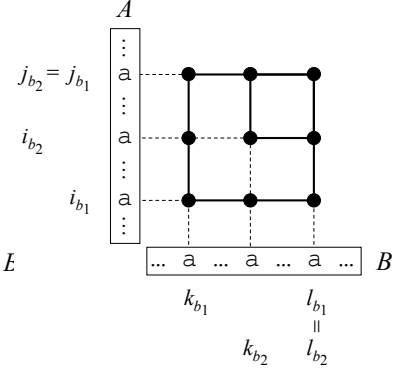
Let us analyze the efficiency of Algorithm 2. For all characters  $c$ , we can precompute  $i_b = \min\{i \mid A[i] = c\}$  and  $k_b = \min\{k \mid B[k] = c\}$  in total  $O(n)$  time and space. The other preprocessing steps are the same as in Algorithm 1 and take  $O(\sigma|\mathcal{M}| + n)$  total time and space. There are  $O(|\mathcal{M}|)$  candidates for the right-upper corner  $p_b = (j_b, l_b)$  of the first matching rectangle from which considered sequences of DOMRs begin. For each  $p_b = (j_b, l_b)$ , its left-bottom corner  $(i_b, k_b)$  can be retrieved in  $O(1)$  time. We again test no more than  $\sigma$  characters for any cell  $DP_{r_b}[r]$ , and Lemma 3 allows to check each unique dominant  $c$ -extension in  $O(1)$  time. Since there are  $O(|\mathcal{M}|)$  candidates for  $r_b$  and  $O(|\mathcal{R}|) = O(|\mathcal{M}|^2)$



■ **Figure 2** Illustration for Lemma 5.



■ **Figure 3** Illustration for Lemma 6.



■ **Figure 4** Illustration for Lemma 7.

candidates for  $r$ , the whole algorithm takes overall  $O(\sigma|\mathcal{M}|^3 + n)$  time and  $O(|\mathcal{M}|^2 + n)$  space. We have shown the following theorem:

► **Theorem 8.** *We can compute  $LCSqS(A, B)$  in  $O(\sigma|\mathcal{M}|^3 + n)$  time and  $O(|\mathcal{M}|^2 + n)$  space.*

### 3.3 $O(|\mathcal{M}|^3 \log^2 n \log \log n + n)$ -time algorithm

In this section we propose an  $O(|\mathcal{M}|^3 \log^2 n \log \log n + n)$ -time and  $O(|\mathcal{M}|^3 + n)$ -space algorithm for computing  $LCSqS(A, B)$ .

For any  $1 \leq i < s \leq j \leq n$  and  $1 \leq k < t \leq l \leq n$ , let  $LCSqS_{s,t}(i, j, k, l) = 2 \times LCS(A[1..i], A[s..j], B[1..k], B[t..l])$ .

By definition,  $LCSqS(A, B) = \max_{1 \leq i < s \leq j \leq n, 1 \leq k < t \leq l \leq n, (s,t) \in \mathcal{M}} \{LCSqS_{s,t}(i, j, k, l)\}$ .

Now, let  $(s, t) \in \mathcal{M}$  be an arbitrarily fixed matching point between  $A$  and  $B$ . This corresponds to Observation 1. A recurrence for computing  $LCSqS_{s,t}(i, j, k, l)$  is given as follows:

$$LCSqS_{s,t}(i, j, k, l) = \begin{cases} \max_{(i',j',k',l') < (i,j,k,l)} \{LCSqS_{s,t}(i', j', k', l')\} + 2 & \begin{aligned} & ((i, j, k, l) \in \mathcal{R}, \\ & 1 \leq i < s \leq j \leq n, \\ & 1 \leq k < t \leq l \leq n) \end{aligned} \\ \max_{(i',j',k',l') < (i,j,k,l)} \{LCSqS_{s,t}(i', j', k', l')\} & \begin{aligned} & ((i, j, k, l) \notin \mathcal{R}, \\ & 1 \leq i < s \leq j \leq n, \\ & 1 \leq k < t \leq l \leq n) \end{aligned} \\ 0 & \text{(otherwise)} \end{cases} \quad (1)$$

Our technique for computing  $LCSqS_{s,t}(i, j, k, l)$  is similar to Chowdhury et al.'s method [8] for computing longest common palindromic subsequences, which uses the following well-known van Emde Boas tree data structure: Let  $\mathcal{S}$  be a set of integers from the universe  $[1, U]$ . The van Emde Boas tree for  $\mathcal{S}$  takes  $\Theta(U)$  space and supports predecessor/successor queries and insertion/deletion operations on  $\mathcal{S}$  in  $O(\log \log U)$  time each [26].

Let  $(s, t) \in \mathcal{M}$  be an arbitrary fixed matching point. We plot a point  $(i, j, k)$  on the 3D grid  $[1..n] \times [1..n] \times [1..n]$  if and only if there is a matching rectangle of form  $(i, j, k, *)$ , namely, one having  $i, j, k$  as its first three coordinates. This 3D point  $(i, j, k)$  will finally be associated with  $\max_{(i,j,k,l) \in \mathcal{R}} \{LCSqS_{s,t}(i, j, k, l)\}$ .

Now we show how to compute those associated values for all the 3D points. We consider the permuted tuples  $(l, i, j, k)$  and sort them as 4-digit numbers, like we did for  $L$  in Section 3.2. We process the permuted tuples in this sorted order. Suppose we are to process a permuted tuple  $(l, i, j, k)$  such that its original tuple  $(i, j, k, l)$  is in  $\mathcal{R}$ . It is now guaranteed that we have processed all tuples  $(l', *, *, *)$  with  $l' < l$ . Therefore, if  $z$  is the maxima among the associated values of all 3D points in the range  $[1..i - 1] \times [1..j - 1] \times [1..k - 1]$ , then we have that  $LCSqS_{s,t}(i, j, k, l) = z + 2$  (see also the recurrence (1) above). We maintain these 3D points with a variant of the 3D range tree [4]. Then, the maxima  $z$  can be efficiently retrieved by querying the point with the maximum associated value in the range  $[1..i - 1] \times [1..j - 1] \times [1..k - 1]$ . If there is no existing 3D point  $(i, j, k)$ , then we insert this point with the associated value  $z + 2$ . Otherwise, we update the associated value of the already existing 3D point  $(i, j, k)$  with  $z + 2$ .

The 3D range tree is a three layered data structure: The top layer tree maintains the first  $i$ -coordinate  $[1..n]$ , and each of its nodes is associated with a middle layer tree. Each middle layer tree maintains the second  $j$ -coordinate  $[1..n]$ , and each of its nodes is associated with a bottom layer tree. Each bottom layer tree maintains the third  $k$ -coordinate  $[1..n]$ . Since each bottom layer tree can contain  $O(n)$  nodes, each middle layer tree can contain at most  $O(n)$  nodes, and the top layer can contain at most  $O(n)$  nodes, the total size of the 3D range tree data structure is trivially bounded by  $O(n^3) = O(|\mathcal{M}|^3)$ . Since at most  $O(|\mathcal{M}|^2)$  points are inserted to the 3D range tree and since  $|\mathcal{M}| = O(n^2)$ , the 3D range tree supports range maxima queries and insertions of new points in  $O(\log^3(|\mathcal{M}|^2)) = O(\log^3 n)$  time.

Next, we improve the query and update times from  $O(\log^3 n)$  to  $O(\log^2 n \log \log n)$ . Chowdhury et al. [8] claimed that using the technique from [15] it is possible to replace each 1D range tree on the bottom layer with a van Emde Boas tree data structure [26], leading to  $O(\log^2 n \log \log n)$  query and update times. However, the way how van Emde Boas trees are used in the approach of [15] indeed requires to maintain a set of integers in the universe of size  $\Theta(n^2)$ . This implies that each van Emde Boas tree requires  $\Theta(n^2)$  space. Since the total size of the top layer tree and the middle layer trees is  $O(n^2)$ , and since each node of a middle layer tree maintains a van Emde Boas tree of size  $O(n^2)$ , it takes  $O(n^4)$  space<sup>3</sup>. This is, however, prohibitive since it can exceed our target time bound  $O(|\mathcal{M}|^3 \log^2 n \log \log n)$  when the set  $\mathcal{M}$  of matching points is sparse (e.g., when  $|\mathcal{M}| = \Theta(n)$ ). Below, we will reduce the space requirement for the van Emde Boas trees used in our data structure.

**Space efficient 3D range tree with van Emde Boas trees.** We briefly recall how the algorithm of [15] computes the maxima in a given range using a van Emde Boas tree. Let  $D[1..n]$  be an array of monotonically non-decreasing non-negative integers from  $[0..n]$ , namely,  $0 \leq D[k] \leq n$  for all  $1 \leq k \leq n$  and  $D[k] \leq D[k + 1]$  for all  $1 \leq k < n$ . We will store in  $D$  the associated values of 3D points in increasing order of positions, and in the sequel we assume that  $D[k + 1] - D[k] \in \{0, 2\}$ . Let  $RMQ_S(1, k)$  denote a query to return the maxima in the sub-array  $D[1..k]$  for  $1 \leq k \leq n$ . For any integer  $\text{val}$  ( $1 \leq \text{val} \leq n$ ), if some entry of  $D$  stores  $\text{val}$ , then we insert the pair  $(\text{pos}, \text{val})$  s.t.  $\text{pos}$  is the rightmost position in  $D$  that stores  $\text{val}$ . For instance, if  $D = [0, 0, 2, 4, 4, 6]$ , then the van Emde Boas tree maintains the set  $\{(2, 0), (3, 2), (5, 4), (6, 6)\}$  of integer pairs. However, since a van Emde Boas tree is an integer data structure, we convert each pair  $(\text{pos}, \text{val})$  to integer  $\text{pos} \times (n + 1) + \text{val}$  and insert

<sup>3</sup> A more careful analysis reveals that the total size of this variant of the 3D range tree with van Emde Boas bottom layer trees is  $O(|\mathcal{M}|^2 n^2 \log n)$ , however, this can also exceed  $O(|\mathcal{M}|^3 \log^2 n \log \log n)$  when  $\mathcal{M}$  is sparse.

it to the van Emde Boas tree. Now, observe that computing  $RMQ_S(1, k)$  reduces to finding the successor for the pair  $(k - 1, n)$ .

The value of  $LCSqS_{s,t}(i, j, k, l)$  is monotonically non-decreasing as  $i, j, k, l$  grow, for fixed  $s$  and  $t$ . Also,  $\text{val}$  in our case is in range  $[0, n]$ . Hence, we can use the above approach in our algorithm. The remaining problem is that the universe size is  $\Theta(n^2)$ , meaning that each van Emde Boas tree above takes  $\Theta(n^2)$  space.

To reduce the space requirement, we maintain only  $\text{pos}$ 's in our van Emde Boas tree, and store  $\text{val}$ 's in an array  $V$  of size  $n$  so that  $V[\text{pos}] = \text{val}$ . We let  $V[i] = -1$  if  $i$  does not exist in the van Emde Boas tree. Let us denote by **Pos\_vEB** and **ValPos\_vEB** the van Emde Boas trees which store  $\text{pos}$ 's only and pairs  $(\text{pos}, \text{val})$ , respectively. Namely, the former is ours and the latter is the method from [15]. It is sufficient for **ValPos\_vEB** to support insertions, deletions, and successor queries. These operations and queries can be simulated by our **Pos\_vEB** as follows: When a pair  $(\text{pos}, \text{val})$  is inserted to **ValPos\_vEB**, then we insert  $\text{pos}$  to **Pos\_vEB** and set  $V[\text{pos}] \leftarrow \text{val}$ . Notice that at any moment **ValPos\_vEB** never maintains two pairs  $(\text{pos}_1, \text{val})$  and  $(\text{pos}_2, \text{val})$  with  $\text{pos}_1 \neq \text{pos}_2$  for the same associated value  $\text{val}$ , since otherwise we get  $\text{argmax}\{i \mid D[i] = \text{val}\} = \text{pos}_1 \neq \text{pos}_2 = \text{argmax}\{i \mid D[i] = \text{val}\}$ , a contradiction. Therefore, we can simulate insertions on **ValPos\_vEB** with **Pos\_vEB** and  $V$  as above. When we delete a pair  $(\text{pos}, \text{val})$  from **ValPos\_vEB**, then we delete  $\text{pos}$  from **Pos\_vEB** and modify the value stored in  $V[\text{pos}]$  accordingly. When we query the successor  $(\text{pos}, \text{val})$  of  $(k - 1, n)$  on **ValPos\_vEB**, then we query the successor  $\text{pos}$  of  $k - 1$  on **Pos\_vEB**, and retrieve  $\text{val} = V[\text{pos}]$ . This way, we can simulate **ValPos\_vEB** with **Pos\_vEB** of  $O(n)$  total space, retaining  $O(\log \log n)$  time efficiency for insertion/deletion operations and successor queries. Since the total number of **ValPos\_vEB**'s is linear in the number of nodes in the top and middle layer trees, our version of 3D range tree, named **New\_vEB\_3DRangeTree**, takes a total of  $O(n^3)$  space and supports range maxima queries in  $O(\log^2 n \log \log n)$  time for query ranges of form  $[1..i] \times [1..j] \times [i..k]$ . The whole algorithm is the following:

---

**Algorithm 3:**


---

**Preprocessing:** For all matching rectangles  $(i, j, k, l) \in \mathcal{R}$ , sort the permuted tuples  $(l, i, j, k)$  as 4-digit numbers. Initialize **New\_vEB\_3DRangeTree**, so that no points are inserted and every entry of array  $V$  in each **Pos\_vEB** stores 0.

**Compute  $LCSqS_{s,t}(i, j, k, l)$ :** For each matching point  $(s, t) \in \mathcal{M}$ , perform the following:

- (1) Process each permuted tuple  $(l, i, j, k)$  in the sorted order. Compute  $LCSqS_{s,t}(i, j, k, l)$  according to recurrence (1): For each different value of  $l$ , let  $\mathcal{PT}_l$  denote the list of permuted tuples whose first elements are  $l$ . For each permuted tuple  $q = (l, i, j, k) \in \mathcal{PT}_l$ , perform the following:
    - If  $i < s < j$  and  $k < t < l$ , then using **New\_vEB\_3DRangeTree** find a 3D point with the maximum associated value  $z_q$  in range  $[1..i - 1] \times [1..j - 1] \times [1..k - 1]$ .
    - After computing  $LCSqS_{s,t}(i, j, k, l)$  for all permuted tuples  $q = (l, i, j, k) \in \mathcal{PT}_l$ , insert  $z_q + 2$  in  $(i, j, k)$  to **New\_vEB\_3DRangeTree** for all such permuted tuples in  $\mathcal{PT}_l$ .
  - (2) If some value  $LCSqS_{s,t}(i, j, k, l)$  exceeds the currently stored maxima, we update it. Then, delete all existing 3D points from **New\_vEB\_3DRangeTree**.
-

Let us recall recurrence (1) to see why Algorithm 3 correctly computes  $LCSqS_{s,t}(i, j, k, l)$ . The rule for the second case (where  $(i, j, k, l) \in \mathcal{R}$ ) requires  $(i', j', k', l') < (i, j, k, l)$ . To reflect this, Algorithm 3 processes all permuted tuples in  $\mathcal{PT}_l$  for each difference value of  $l$  and in increasing order of  $l$ . After processing all permuted tuples  $q = (l, i, j, k) \in \mathcal{PT}_l$ , we can safely insert the value  $z_q + 2$  in the corresponding 3D point  $(i, j, k)$  for all such tuples  $q$ , and can proceed to the permuted tuples with larger first values.

Let us analyze the efficiency of Algorithm 3. For preprocessing, we use  $O(n)$  time and space for alphabet reduction, for sorting the permuted tuples  $(l, i, j, k)$ , and for initializing **New\_vEB\_3DRangeTree**. For each  $(s, t) \in \mathcal{M}$ , we compute  $LCSqS_{s,t}(i, j, k, l)$  with each  $(i, j, k, l) \in \mathcal{R}$ , by querying and updating **New\_vEB\_3DRangeTree**. Each query and update here take  $O(\log^2 n \log \log n)$  time. After computing all  $LCSqS_{s,t}(i, j, k, l)$  for the current matching point  $(s, t)$ , we delete all 3D points from **New\_vEB\_3DRangeTree**. Thus it takes  $O(|\mathcal{R}| \log^2 n \log \log n)$  time for each  $(s, t) \in \mathcal{M}$ . **New\_vEB\_3DRangeTree** uses  $O(n^3) = O(|\mathcal{M}|^3)$  space (recall that  $n \leq |\mathcal{M}|$  holds after alphabet reduction). Since  $|\mathcal{R}| = O(|\mathcal{M}|^2)$ , Algorithm 3 takes a total of  $O(|\mathcal{M}||\mathcal{R}| \log^2 n \log \log n + |\mathcal{M}|^3 + n) = O(|\mathcal{M}|^3 \log^2 n \log \log n + n)$  time and  $O(|\mathcal{M}|^3 + n)$  space.

We have shown the following theorem:

► **Theorem 9.** *We can compute  $LCSqS(A, B)$  in  $O(|\mathcal{M}|^3 \log^2 n \log \log n + n)$  time and  $O(|\mathcal{M}|^3 + n)$  space.*

## 4 Hardness results on the LCSqS problem

The  $k$ -LCSqS problem is to compute an LCSqS of  $k$  given strings. For simplicity, we assume that each given string is of length  $n$ .

► **Lemma 10.** *For any  $k \geq 2$ , the  $k$ -LCS problem can be reduced in linear time to the  $\lfloor k/2 \rfloor$ -LCSqS problem.*

**Proof.** Our proof uses an idea similar to [6] and [16]. We first consider the case where  $k$  is even. Let  $A_1, \dots, A_k$  be the input strings for the  $k$ -LCS problem. For each  $1 \leq i \leq k/2$ , we construct a string  $B_i$  of length  $4n + 2$  such that  $B_i = A_{2i-1} \$^{n+1} A_{2i} \$^{n+1}$ , where  $\$$  is a special character which does not appear in  $A_1, \dots, A_k$ . Let  $Z$  be any LCSqS of  $B_1, \dots, B_{k/2}$ . Since each  $A_j$  ( $1 \leq j \leq k$ ) is of length  $n$ ,  $Z$  must be of form  $X \$^{n+1} X \$^{n+1}$ . Then, clearly the string  $X$  is a longest common subsequence of the original strings  $A_1, \dots, A_k$ .

For odd  $k$ , it suffices to consider the same strings  $B_i$  for  $1 \leq i \leq \lfloor k/2 \rfloor$  and one additional string  $B_{\lfloor k/2 \rfloor} = A_k \$^{n+1} A_k \$^{n+1}$ . This completes the proof. ◀

By Lemma 10, the  $k$ -LCSqS problem is NP-hard for an unfixed  $k$ . For an arbitrarily fixed  $k$ , Abboud et al. [1] showed that if there exist a constant  $\epsilon > 0$ , an integer  $k \geq 2$ , and an algorithm which solves the  $k$ -LCS problem for an alphabet of size  $O(k)$  in  $O(n^{k-\epsilon})$  time, then the famous *strong exponential time hypothesis (SETH)* is false. This suggests that it seems hard to compute  $LCSqS(A, B)$  in  $O(n^{4-\epsilon})$  time for any  $\epsilon > 0$ .

## 5 Discussions

We observe that it seems difficult to shave the  $|\mathcal{M}|^3$  term in the time complexity of any matching-rectangle-based algorithm for computing the LCSqS: For instance, in both Algorithm 2 and Algorithm 3, we first fix a matching point in  $\mathcal{M}$ , and this indeed corresponds to the  $|\mathcal{M}|$  term in the  $O(|\mathcal{M}|n^4)$ -time complexity of the simple solution for computing

$LCSqS(A, B)$ . The rest of all these algorithms exactly computes the LCS of the four strings obtained by partitioning  $A$  and  $B$  at a given matching point using at least  $O(|\mathcal{M}|^2)$  or  $O(n^4)$  time. This seems almost best possible, since it is widely believed that there is no algorithm which computes the LCS of four strings in  $O(n^{4-\epsilon})$  time for any  $\epsilon > 0$  (recall Section 4).

Can we break the  $O(|\mathcal{M}|^3)$  or  $O(n^6)$  barrier? The only hope seems to generalize an *incremental LCS computation* algorithm for two strings ([21, 24, 17, 23, 25, 13]) to the case of four strings. This would help us update a data structure for  $LCS(A[1..i-1], A[i..n], B[1..j-1], B[j..n])$  to that for  $LCS(A[1..i], A[i+1..n], B[1..j], B[j+1..n])$  in faster than  $O(n^4)$  time. However, this seems difficult, too. We investigated whether Kim and Park’s method [17], the simplest incremental LCS algorithm for two strings, can be generalized to more strings. Their algorithm uses the differential encoding of the 2-dimensional DP tables (for two strings) before and after the first character of one string is deleted, and they showed that only  $O(n)$  entries of the differential encoding need to be updated. However, our preliminary experiments for 3-dimensional DP tables (i.e. for three strings) already suggested that there would be more than  $O(n^2)$  entries in the differential encoding that need to be updated.

Overall, it is an intriguing open question how one can close the (almost) quadratic gap between the upper and lower bounds for the LCSqS problem.

---

## References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proc. FOCS 2015*, pages 59–78, 2015.
- 2 Abdullah N. Arslan. Regular expression constrained sequence alignment. *J. Disc. Algo.*, 5(4):647–661, 2007.
- 3 Sang Won Bae and Inbok Lee. On finding a longest common palindromic subsequence. *Theor. Comput. Sci.*, 710:29–34, 2018.
- 4 Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.
- 5 Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.
- 6 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proc. FOCS 2015*, pages 79–97, 2015.
- 7 Francis Y. L. Chin, Alfredo De Santis, Anna Lisa Ferrara, N. L. Ho, and S. K. Kim. A simple algorithm for the constrained sequence problems. *Inf. Process. Lett.*, 90(4):175–179, 2004.
- 8 Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. *Fundam. Inform.*, 129(4):329–340, 2014.
- 9 Sebastian Deorowicz. Quadratic-time algorithm for a string constrained LCS problem. *Inf. Process. Lett.*, 112(11):423–426, 2012.
- 10 Effat Farhana and M. Sohel Rahman. Doubly-constrained LCS and hybrid-constrained LCS problems revisited. *Inf. Process. Lett.*, 112(13):562–565, 2012.
- 11 Effat Farhana and M. Sohel Rahman. Constrained sequence analysis algorithms in computational biology. *Inf. Sci.*, 295:247–257, 2015.
- 12 Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016.
- 13 Heikki Hyrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *J. Disc. Algo.*, 34:2–17, 2015.
- 14 Costas S. Iliopoulos and Mohammad Sohel Rahman. New efficient algorithms for the LCS and constrained LCS problems. *Inf. Process. Lett.*, 106(1):13–18, 2008.



- 15 Costas S. Iliopoulos and Mohammad Sohel Rahman. A new efficient algorithm for computing the longest common subsequence. *Theory Comput. Syst.*, 45(2):355–371, 2009.
- 16 Shunsuke Inenaga and Heikki Hyvrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Inf. Process. Lett.*, 129:11–15, 2018.
- 17 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *J. Disc. Algo.*, 2:302–312, 2004.
- 18 Adrian Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *Proc. SPIRE 2004*, pages 93–100, 2004.
- 19 Keita Kuboi, Yuta Fujishige, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster str-ic-lcs computation via rle. In *Proc. CPM 2017*, page 25:1–25:12, 2017.
- 20 Gregory Kucherov, Tamar Pinhas, and Michal Ziv-Ukelson. Regular language constrained sequence alignment revisited. *J. Computational Biology*, 18(5):771–781, 2011.
- 21 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comp.*, 27(2):557–582, 1998.
- 22 William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.
- 23 Yoshifumi Sakai. An almost quadratic time algorithm for sparse spliced alignment. *Theory Comput. Syst.*, 48(1):189–210, 2011.
- 24 Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application in finding all approximate repeats in strings. *SIAM J. Comp.*, 27(4):972–992, 1998.
- 25 Alexandre Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2007. URL: <http://arxiv.org/abs/0707.3619>.
- 26 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. FOCS 1975*, pages 75–84, 1975.
- 27 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
- 28 Daxin Zhu and Xiaodong Wang. A simple algorithm for solving for the generalized longest common subsequence (LCS) problem with a substring exclusion constraint. *Algorithms*, 6(3):485–493, 2013.
- 29 Daxin Zhu, Yingjie Wu, and Xiaodong Wang. An efficient algorithm for a new constrained LCS problem. In *Proc. ACIIDS 2016*, pages 261–267, 2016.
- 30 Daxin Zhu, Yingjie Wu, and Xiaodong Wang. An efficient dynamic programming algorithm for STR-IC-STR-EC-LCS problem. In *Proc. GPC 2016*, pages 3–17, 2016.





# Slowing Down Top Trees for Better Worst-Case Compression

**Bartłomiej Dudek**

Institute of Computer Science, University of Wrocław, Poland

bartlomiej.dudek@cs.uni.wroc.pl

**Paweł Gawrychowski**

Institute of Computer Science, University of Wrocław, Poland

gawry@cs.uni.wroc.pl

---

## Abstract

We consider the top tree compression scheme introduced by Bille et al. [ICALP 2013] and construct an infinite family of trees on  $n$  nodes labeled from an alphabet of size  $\sigma$ , for which the size of the top DAG is  $\Theta(\frac{n}{\log_\sigma n} \log \log_\sigma n)$ . Our construction matches a previously known upper bound and exhibits a weakness of this scheme, as the information-theoretic lower bound is  $\Omega(\frac{n}{\log_\sigma n})$ . This settles an open problem stated by Lohrey et al. [arXiv 2017], who designed a more involved version achieving the lower bound. We show that this can be also guaranteed by a very minor modification of the original scheme: informally, one only needs to ensure that different parts of the tree are not compressed too quickly. Arguably, our version is more uniform, and in particular, the compression procedure is oblivious to the value of  $\sigma$ .

**2012 ACM Subject Classification** Theory of computation → Data compression

**Keywords and phrases** top trees, compression, tree grammars

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.16

**Funding** Work supported under National Science Centre, Poland, project number 2014/15/B/ST6/00615.

## 1 Introduction

Labeled trees are fundamental data structures in computer science. Generalizing strings, they can be used to compactly represent hierarchical dependencies between objects and have multiple applications. In many of them, such as XML files, we need to operate on very large trees that are in some sense repetitive. Therefore, it is desirable to design compression schemes for trees that are able to exploit this. Known tree compression methods include DAG compression that uses subtree repeats and represents a tree as a Directed Acyclic Graph [3, 7, 13], compression with tree grammars that focuses on the more general tree patterns and represents a tree by a tree grammar [4, 8, 11, 12], and finally succinct data structures [6, 10].

In this paper we analyze tree compression with top trees introduced by Bille et al. [2]. It is able to take advantage of internal repeats in a tree while supporting various navigational queries directly on the compressed representation in logarithmic time. At a high level, the idea is to hierarchically partition the tree into *clusters* containing at most two boundary nodes that are shared between different clusters. A representation of this hierarchical partition is called the top tree. Then, the top DAG is obtained by identifying isomorphic subtrees of the top tree. Bille et al. [2] proved that the size of the top DAG is always  $O(n/\log_\sigma^{0.19} n)$  for a tree on  $n$  nodes labeled with labels from  $\Sigma$  where  $\sigma = \max\{2, |\Sigma|\}$ . Furthermore, they



© Bartłomiej Dudek and Paweł Gawrychowski;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 16; pp. 16:1–16:8

Leibniz International Proceedings in Informatics



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

showed that top DAG compression is always at most logarithmically worse than the classical DAG compression (and Bille et al. [1] constructed a family of trees for which this logarithmic upper bound is tight). Later, Hübschle-Schneider and Raman [9] improved the bound on the size of the top DAG to  $O(\frac{n}{\log_\sigma n} \log \log_\sigma n)$  using a more involved reasoning based on the heavy path decomposition. This should be compared with the information-theoretic lower bound of  $\Omega(\frac{n}{\log_\sigma n})$ .

A natural question is to close the gap between the information-theoretic lower bound of  $\Omega(\frac{n}{\log_\sigma n})$  and the upper bound of  $O(\frac{n}{\log_\sigma n} \log \log_\sigma n)$ . We show that the latter is tight for the top tree construction algorithm of Bille et al. [2].

► **Theorem 1.** *There exists an infinite family of trees on  $n$  nodes labeled from an alphabet  $\Sigma$  for which the size of the top DAG is  $\Omega(\frac{n}{\log_\sigma n} \log \log_\sigma n)$  where  $\sigma = \max\{2, |\Sigma|\}$ .*

This answers an open question explicitly mentioned by Lohrey et al. [14], who developed a different algorithm for constructing a top tree which guarantees that the size of the top DAG matches the information-theoretic lower bound. A crucial ingredient of their algorithm is a partition of the tree  $T$  into  $O(n/k)$  clusters of size at most  $k$ , where  $k = \Theta(\log_\sigma n)$ . As a byproduct, they obtain a top tree of depth  $O(\log n)$  for each cluster. Then they consider a tree  $T'$  obtained by collapsing every cluster of  $T$  and run the algorithm of Bille et al. [2] on  $T'$ . Finally, the edges of  $T'$  are replaced by the top trees of their corresponding clusters of  $T$  constructed in the first phase of the algorithm to obtain the top tree of the whole  $T$ . While this method guarantees that the number of distinct clusters is  $O(\frac{n}{\log_\sigma n})$ , its disadvantage is that the resulting procedure is non-uniform, and in particular needs to be aware of the value of  $\sigma$  and  $n$ .

We show that a slight modification of the algorithm of Bille et al. [2] is, in fact, enough to guarantee that the number of distinct clusters, and so also the size of the top DAG, matches the information-theoretic lower bound. The key insight actually comes from the proof of Theorem 1, where we construct a tree with the property that some of its parts are compressed much faster than the others, resulting in a larger number of different clusters. The original algorithm proceeds in iterations, and in every iteration tries to merge adjacent clusters as long as they meet some additional conditions. Surprisingly, it turns out that the information-theoretic lower bound can be achieved by slowing down this process to avoid some parts of the tree being compressed much faster than the others. Informally, we show that it is enough to require that in the  $t^{\text{th}}$  iteration adjacent clusters are merged only if their size is at most  $\alpha^t$ , for some constant  $\alpha > 1$ . The modified algorithm preserves nice properties of the original method such as the  $O(\log n)$  depth of the obtained top tree.

A detailed description of the original algorithm of Bille et al. [2] can be found in Section 2. In Section 3 we prove Theorem 1 and in Section 4 describe the modification.

## 2 Preliminaries

In this section, we briefly restate the top tree construction algorithm of Bille et al. [2]. To construct trees that can be used to show the lower bound and present our modification of the original algorithm we need to work with exactly the same definitions. Consequently, the following description closely follows the condensed presentation from Bille et al. [1] and can be omitted if the reader is already familiar with the approach.

Let  $T$  be a (rooted) tree on  $n$  nodes. The children of every node are ordered from left to right, and every node has a label from an alphabet  $\Sigma$ .  $T(v)$  denotes the subtree of  $v$ , including  $v$  itself, and  $F(v)$  is the forest of subtrees of all children  $v_1, v_2, \dots, v_k$  of  $v$ , that is,

$F(v) = T(v_1) \cup T(v_2) \cup \dots \cup T(v_k)$ . For  $1 \leq s \leq r \leq k$  we define  $T(v, v_s, v_r)$  to be the tree consisting of  $v$  and a contiguous range of its children starting from the  $s^{\text{th}}$  and ending at the  $r^{\text{th}}$ , that is,  $T(v, v_s, v_r) = \{v\} \cup T(v_s) \cup T(v_{s+1}) \cup \dots \cup T(v_r)$ .

We define two types of *clusters*. A cluster with only a top boundary node  $v$  is of the form  $T(v, v_s, v_r)$ . A cluster with a top boundary node  $v$  and a bottom boundary node  $u$  is of the form  $T(v, v_s, v_r) \setminus F(u)$  for a node  $u \in T(v, v_s, v_r) \setminus \{v\}$ .

If edge-disjoint clusters  $A$  and  $B$  have exactly one common boundary node and  $C = A \cup B$  is a cluster, then  $A$  and  $B$  can be *merged* into  $C$ . Then one of the top boundary nodes of  $A$  and  $B$  becomes the top boundary node of  $C$  and there are various ways of choosing the bottom boundary node of  $C$ . See Figure 2 in [2] for the details of all five possible ways of merging two clusters.

A *top tree*  $\mathcal{T}$  of  $T$  is an ordered and labeled binary tree describing a hierarchical decomposition of  $T$  into clusters.

- The nodes of  $\mathcal{T}$  correspond to the clusters of  $T$ .
- The root of  $\mathcal{T}$  corresponds to the whole  $T$ .
- The leaves of  $\mathcal{T}$  correspond to the edges of  $T$ . The label of each leaf is the pair of labels of the endpoints of its corresponding edge  $(u, v)$  in  $T$ . The two labels are ordered so that the label of the parent appears before the label of the child.
- Each internal node of  $\mathcal{T}$  corresponds to the merged cluster of the clusters corresponding to its two children. The label of each internal node is the type of merge it represents (out of the five merging options). The children are ordered so that the left child is the child cluster visited first in a preorder traversal of  $T$ .

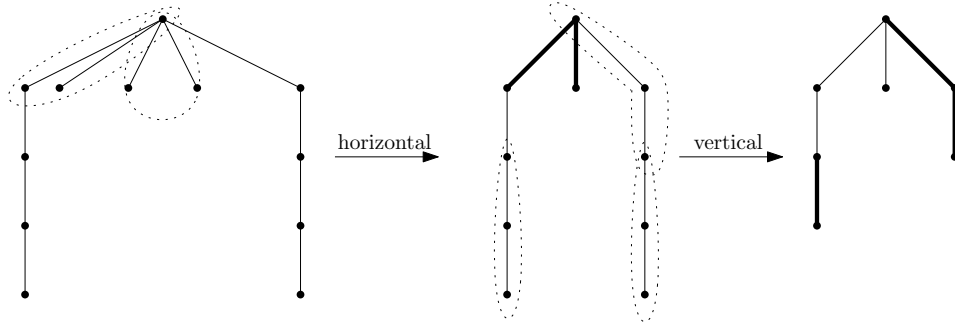
The top tree  $\mathcal{T}$  is constructed bottom-up in iterations, starting with the edges of  $T$  as the leaves of  $\mathcal{T}$ . During the whole process, we maintain an auxiliary ordered tree  $\tilde{T}$ , initially set to  $T$ . The edges of  $\tilde{T}$  correspond to the nodes of  $\mathcal{T}$ , which in turn correspond to the clusters of  $T$ . The internal nodes of  $\tilde{T}$  correspond to the boundary nodes of these clusters and the leaves of  $\tilde{T}$  correspond to a subset of the leaves of  $T$ .

On a high level, the iterations are designed in such a way that each of them merges a constant fraction of edges of  $\tilde{T}$ . This is proved in Lemma 1 of [2], and we describe a slightly stronger property in Lemma 2. This guarantees that the height of the resulting top tree is  $O(\log n)$ . Each iteration consists of two steps:

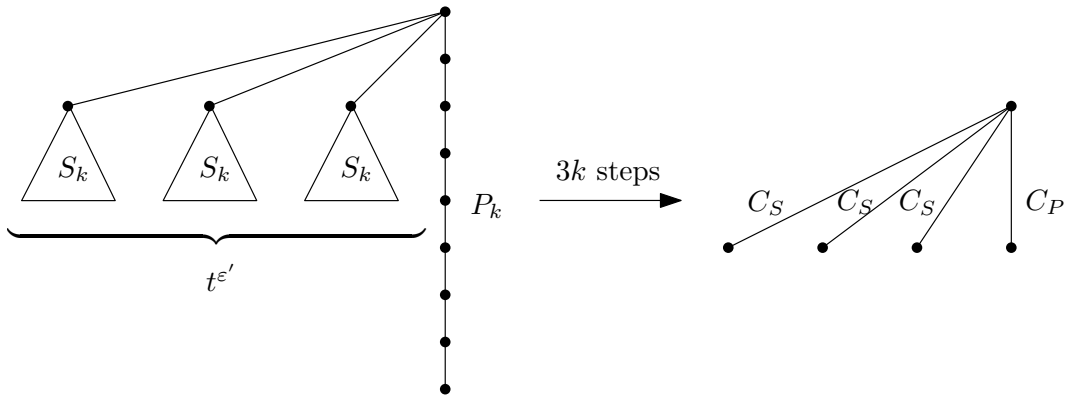
**Horizontal merges.** For each node  $v \in \tilde{T}$  with  $k \geq 2$  children  $v_1, \dots, v_k$ , for  $i = 1$  to  $\lfloor \frac{k}{2} \rfloor$ , merge the edges  $(v, v_{2i-1})$  and  $(v, v_{2i})$  if  $v_{2i-1}$  or  $v_{2i}$  is a leaf. If  $k$  is odd and  $v_k$  is a leaf and both  $v_{k-2}$  and  $v_{k-1}$  are non-leaves then also merge  $(v, v_{k-1})$  and  $(v, v_k)$ .

**Vertical merges.** For each maximal path  $v_1, \dots, v_p$  of nodes in  $\tilde{T}$  such that  $v_{i+1}$  is the parent of  $v_i$  and  $v_2, \dots, v_{p-1}$  have a single child: If  $p$  is even merge the following pairs of edges  $\{(v_1, v_2), (v_2, v_3)\}, \dots, \{(v_{p-3}, v_{p-2}), (v_{p-2}, v_{p-1})\}$ . If  $p$  is odd merge the following pairs of edges  $\{(v_1, v_2), (v_2, v_3)\}, \dots, \{(v_{p-4}, v_{p-3}), (v_{p-3}, v_{p-2})\}$ , and if  $(v_{p-1}, v_p)$  was not merged in the previous step then also merge  $\{(v_{p-2}, v_{p-1}), (v_{p-1}, v_p)\}$ .

See an example of a single iteration in Figure 1. Finally, the compressed representation of  $T$  is the so-called top DAG  $\mathcal{TD}$ , which is the minimal DAG representation of  $\mathcal{T}$  obtained by identifying identical subtrees of  $\mathcal{T}$ . As every iteration shrinks  $\tilde{T}$  by a constant factor,  $\mathcal{T}$  can be computed in  $O(n)$  time, and then  $\mathcal{TD}$  can be computed in  $O(|\mathcal{T}|)$  time [5]. Thus, the entire compression takes  $O(n)$  time.



■ **Figure 1** Tree  $\tilde{T}$  after two steps of a single iteration. Dotted lines denote the merged edges (clusters) and thick edges denote the results of merging. Note that one edge does not participate in the vertical merge due to having been obtained as a result of a horizontal merge.



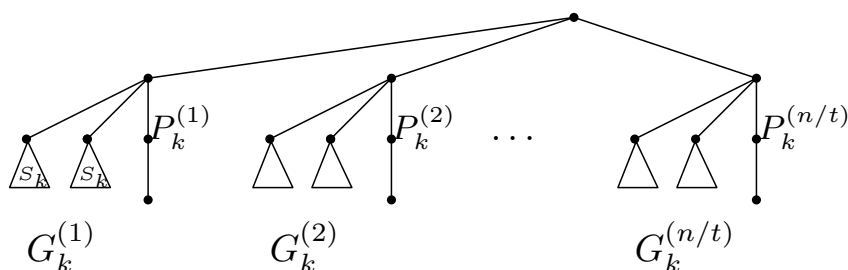
■ **Figure 2** Gadget  $G_k$  consists of  $2^k - 1 = O(t^{\epsilon'})$  trees  $S_k$  and one path  $P_k$ . After  $3k$  iterations it gets compressed to a tree with  $2^k$  nodes connected to the root.

### 3 A lower bound for the approach of Bille et al.

In this section, we prove Theorem 1 and show that the  $O(\frac{n}{\log_\sigma n} \log \log_\sigma n)$  bound from [9] on the number of distinct clusters created by the algorithm described in Section 2 is tight. We first consider labeled trees for which  $|\Sigma| > 1$  and  $\sigma = |\Sigma|$ . Then we show how to modify our construction and apply it to unlabeled trees.

For every  $k \in \mathbb{N}$  we will construct a tree  $T_k$  with  $n = \Theta(\sigma^{8^k})$  nodes for which the corresponding top DAG is of size  $\Theta(\frac{n}{\log_\sigma n} \log \log_\sigma n)$ . Let  $t = 8^k = \Theta(\log_\sigma n)$ . In the beginning, we describe a gadget  $G_k$  that is the main building block of  $T_k$ . It consists of  $O(t)$  nodes: a path of  $t$  nodes and  $2^k - 1 = O(t^{\epsilon'})$  full ternary trees of size  $O(t^\epsilon)$  connected to the root, where  $\epsilon + \epsilon' < 1$ . See Figure 2. The main intuition behind the construction is that full ternary trees are significantly smaller than the path, but they need the same number of iterations to get compressed.

More precisely, let  $P_k$  be the path of length  $8^k = t$ . Clearly, after 3 iterations it gets compressed to  $P_{k-1}$ , and so after  $3k$  iterations becomes a single cluster. Similarly, let  $S_k$  be the full ternary tree of height  $k$  with  $3^k$  leaves, so  $\frac{3^{k+1}-1}{2} = O(3^k) = O(t^{0.53})$  nodes in total. Observe that after 3 iterations  $S_k$  becomes  $S_{k-1}$ , and so after  $3k$  iterations becomes a single cluster. To sum up, the gadget  $G_k$  consists of path  $P_k$  of  $t$  nodes and  $2^k - 1 = O(t^{1/3})$  trees of size  $O(t^{0.53})$ , so in total  $O(t)$  nodes. After  $3k$  iterations  $G_k$  consists of  $2^k - 1$  clusters  $C_S$



■ **Figure 3**  $T_k$  consists of  $\Theta(n/t)$  gadgets  $G_k^{(i)}$ , where the  $i^{\text{th}}$  of them contains a unique path  $P_k^{(i)}$ .

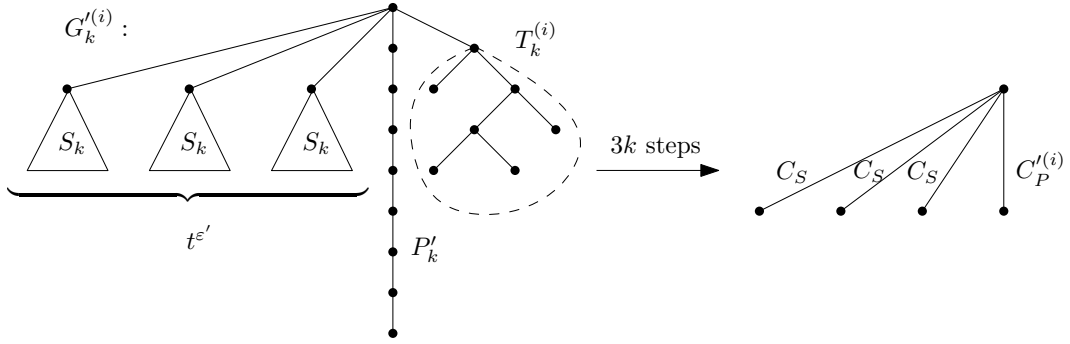
corresponding to  $S_k$  and one cluster  $C_P$  corresponding to  $P_k$ , as shown in Figure 2. In each of the subsequent  $k$  iterations, the remaining clusters are merged in pairs.

Recall that the top DAG contains a node for every distinct subtree of the top tree, and every node of the top tree corresponds to a cluster obtained during the compression process. Our next step will be to create many almost identical gadgets connected to a common root. In order to ensure that they are distinct, we assign labels on the nodes on paths  $P_k$  so that no two paths are equal. Then the cluster  $C_P$  obtained after the first  $3k$  iterations corresponds to a distinct subtree of the top tree. Consequently, so does the cluster obtained from  $C_P$  in each of the subsequent  $k$  iterations. Note that during all the subsequent iterations only horizontal merges are performed and each of them halves the number of clusters.

Finally, the tree  $T_k$  consists of  $\Theta(n/t)$  gadgets connected to a common root as in Figure 3. The  $i^{\text{th}}$  gadget  $G_k^{(i)}$  is a copy of  $G_k$  with the labels of  $P_k^{(i)}$  chosen as to spell out the  $i^{\text{th}}$  (in the lexicographical order) word of length  $t$  over  $\Sigma$ . For all the remaining nodes (nodes of trees  $S_k$ , roots of gadgets and the common root of  $T_k$ ) it is enough to choose the same label, e.g. the smallest in  $\Sigma$ . Note that  $\sigma^t > n/t$ , so there are more possible words of length  $t$  than the number of gadgets that we want to create. Then each  $C_P^{(i)}$  and the clusters obtained from it during the subsequent  $k$  iterations correspond to distinct subtrees of the top tree. Thus, overall the top DAG contains  $\Omega(n/t \cdot k) = \Omega(n/t \cdot \log t) = \Omega(n/\log_\sigma n \cdot \log \log_\sigma n)$  nodes, which concludes the proof of Theorem 1 for labeled trees with non-unary alphabet.

**Unlabeled trees.** Now we modify the above construction so that it works for unary alphabets. Recall that we set  $t = \log n$  and  $k = \log_8 t$ . We cannot use the earlier approach directly, as we cannot distinguish the gadgets  $G_k^{(i)}$  by modifying labels on the path  $P_k^{(i)}$ . To address this we extend the gadgets  $G_k^{(i)}$  with distinct unlabeled binary trees in such a way that after  $3k$  steps the new gadgets get compressed to the same trees as before (shown in Figure 2) that is the  $i^{\text{th}}$  gadget is compressed to  $t^{\varepsilon'}$  clusters  $C_S$  and a cluster  $C_P^{(i)}$ . Again  $C_S$  represents the cluster of full ternary tree  $S_k$  and clusters  $C_P^{(i)}$  correspond to distinct subtrees.

More precisely, now the  $i^{\text{th}}$  gadget  $G_k^{(i)}$  consists of  $t^{\varepsilon'}$  trees  $S_k$  connected to the root, a path  $P_k'$  of length  $4 \cdot 8^{k-1} + 1$  and the  $i^{\text{th}}$  binary tree  $T_k^{(i)}$  on  $t = \log n$  nodes (we consider an arbitrary ordering on all such trees). Intuitively, the construction of path  $P_k'$  guarantees that no matter how fast the tree  $T_k^{(i)}$  gets compressed, during the first  $3k$  steps it does not interact with subtrees  $S_k$ . Without the path  $P_k'$ , it might happen that the single cluster obtained from  $T_k^{(i)}$  participates in a horizontal step with the (partially compressed) rightmost tree  $S_k$  within the first  $3k$  steps. Next, the sizes of each component of  $G_k^{(i)}$  are chosen in such a way that again  $G_k^{(i)}$  consists of  $O(t)$  nodes and after  $3k$  steps the obtained tree is exactly the same as in the case of non-unary alphabets. See Figure 4.



■ **Figure 4** The modified gadget  $G_k^{(i)}$  for unlabeled trees.

Note that path  $P'_k$  gets compressed to path  $P'_{k-1}$  in 3 steps. Furthermore, the first edge of  $P'_k$  does not take part in any vertical merge unless the path consists of only two edges, that is in the  $(3k)^{\text{th}}$  step. Observe that trees  $T_k^{(i)}$  are compressed with different speeds depending on their shape and at some moment they become a single cluster that will be merged in the next horizontal step. As pointed earlier, the first edge of  $P'_k$  does not take part in vertical merges before the  $(3k)^{\text{th}}$  step, so eventually it can participate in a horizontal merge with the cluster of  $T_k^{(i)}$  without affecting the compression of the remaining edges of  $P'_k$ . As all the merges inside  $T_k^{(i)}$  are independent from the rest of the tree, Lemma 1 of [2] guarantees that after every step of the compression the tree  $T_k^{(i)}$  shrinks at least by a factor of  $8/7$ . Thus  $T_k^{(i)}$  becomes a single cluster in at most  $\log_{8/7} \log n < 9 \log \log n = 3k$  steps, and so after  $3k$  steps the gadget  $G_k^{(i)}$  gets compressed to the tree described in Figure 4.

Finally, in order to further apply the reasoning from the case of labeled trees, it remains to show that there are  $\Omega(n/t)$  distinct binary trees on  $t$  nodes. From the folklore properties of Catalan numbers, there are  $\frac{1}{t+1} \binom{2t}{t}$  distinct binary trees on  $t$  nodes. Applying the bound  $\binom{n}{k} \geq (\frac{n}{k})^k$  we obtain that there are at least  $\frac{2^t}{t+1} = \Omega(n/t)$  distinct binary trees  $T_k^{(i)}$ , which is sufficient for our construction. It concludes the case of unlabeled trees and thus ends the proof of Theorem 1.

#### 4 An optimal tree compression algorithm

Let  $\alpha$  be a constant greater than 1 and consider the following modification of algorithm [2]. Our algorithm works in the same way for both labeled and unlabeled trees. As mentioned in the introduction, intuitively we would like to proceed exactly as the original algorithm, except that in the  $t^{\text{th}}$  iteration we do not perform a merge if one of the participating clusters is of size larger than  $\alpha^t$ . However, this would require a slight modification of the original charging argument showing that that after every iteration the tree  $\tilde{T}$  shrinks by a constant factor. To avoid adapting the whole proof of [2] to our new approach, we proceed slightly differently. In each iteration we first generate and list all the merges that would have been performed in both steps of a single iteration of the original algorithm. Then we apply only the merges in which both clusters have size at most  $\alpha^t$ .

We run the algorithm until the tree  $\tilde{T}$  becomes a single edge. Clearly, there are  $O(\log n)$  iterations, because after  $\log_\alpha n$  iterations the algorithm is no longer constrained and can behave not worse than the original one. Thus the depth of the obtained DAG is  $O(\log n)$  as before. In the following lemma we show that even if there are some clusters that cannot be

---

**Algorithm 1** A modified top tree construction algorithm of Bille et al. [2] for a tree  $T$ .

---

- 1:  $\tilde{T} := T$
  - 2: initialize leaves of  $\mathcal{T}$  with edges of  $T$
  - 3: **for**  $t = 1, \dots, \Theta(\log n)$ , as long as  $\tilde{T}$  is not a single edge **do**
  - 4:   list all the merges that would have been made by one iteration of the original algorithm
  - 5:   filter out the merges that use a cluster of size bigger than  $\alpha^t$
  - 6:   modify  $\tilde{T}$  and  $\mathcal{T}$  by applying the remaining merges
  - 7: construct DAG  $\mathcal{TD}$  of  $\mathcal{T}$   $\triangleright \mathcal{TD}$  is the top DAG of  $T$
- 

merged in one step, the tree still shrinks by roughly a constant factor.

► **Lemma 2.** *Suppose that there are  $m = p + q$  clusters in  $\tilde{T}$  after  $t - 1$  iterations of Algorithm 1, where  $q$  is the number of clusters of size larger than  $\alpha^t$ . Then, after  $t$  iterations there are at most  $7/8m + q$  clusters.*

**Proof.** The proof is a generalization of Lemma 1 from [2]. There are  $m + 1$  nodes in  $\tilde{T}$ , so at least  $m/2 + 1$  of them have degree smaller than 2. Consider  $m/2$  edges from these nodes to their parents and denote this set as  $M$ . Then, from a charging argument (see the details in [2]) we obtain that at least half of the edges in  $M$  would have been merged in a single iteration of the original algorithm. Denote these edges by  $M'$ , where  $|M'| \geq m/4$  and observe that at least  $|M'|/2 \geq m/8$  pairs of edges can be merged.

Now,  $q$  clusters (edges) are too large to participate in a merge and in the worst case each of them would have participated in a different merge of the original algorithm. Thus, Algorithm 1 performs at least  $m/8 - q$  merges and after a single iteration the number of clusters decreases to at most  $m - (m/8 - q) = 7/8m + q$ . ◀

Our goal will be to prove the following theorem.

► **Theorem 3.** *Let  $T$  be a tree on  $n$  nodes labeled from an alphabet of size  $\sigma$ . Then the size of the corresponding top DAG obtained by Algorithm 1 with  $\alpha = 10/9$  is  $O(\frac{n}{\log_\sigma n})$ .*

In the following we assume that  $\alpha = 10/9$ , but do not substitute it to avoid clutter.

► **Lemma 4.** *After  $t$  iterations of Algorithm 1 there are  $O(n/\alpha^{t+1})$  clusters in  $\tilde{T}$ .*

**Proof.** We prove by induction on  $t$  that after  $t$  iterations  $\tilde{T}$  contains at most  $cn/\alpha^{t+1}$  clusters, where  $c = 113$ . The case  $t = 0$  is immediate. Let  $t > 0$ . From the induction hypothesis, after  $t - 1$  iterations there are at most  $cn/\alpha^t$  clusters,  $p$  of them having size at most  $\alpha^t$  (call them small) and  $q$  of them having size larger than  $\alpha^t$  that cannot be yet merged in the  $t^{\text{th}}$  iteration (call them big). We know that  $p \leq cn/\alpha^t$  and, as the big clusters are pairwise disjoint,  $q \leq n/\alpha^t$ .

We need to show that the total number of clusters after  $t$  iterations is at most  $cn/\alpha^{t+1}$ . There are two cases to consider:

- $q \leq \frac{1}{100}p$ : We apply Lemma 2 and conclude that the total number of clusters after the  $t^{\text{th}}$  iteration is at most  $7/8(p + q) + q < 9/10p \leq cn/\alpha^{t+1}$ .
- $p < 100q$ : In the worst case no pair of clusters was merged and the total number of clusters after the  $t^{\text{th}}$  iteration is  $p + q < 101q < 101n/\alpha^t \leq 113n/\alpha^{t+1} = cn/\alpha^{t+1}$ . ◀

**Proof of Theorem 3.** Clusters are represented with binary trees labeled either with pairs of labels from the original alphabet or one of the 5 labels representing the type of merging, so in total there are  $|\Sigma|^2 + 5 \leq \sigma^2 + 5$  possible labels of nodes in  $\mathcal{T}$ . From the properties of



Catalan numbers, it follows that the number of different binary trees of size  $x$  is bounded by  $4^x$ . Thus there are at most  $\sum_{i=1}^x (4(\sigma^2 + 5))^i \leq \sum_{i=1}^x (12\sigma^2)^i \leq (12\sigma^2)^{x+1}$  distinct labeled trees of size at most  $x$ , since  $\sigma \geq 2$ . Even if some of them appear many times in  $\tilde{T}$ , they will be represented only once in  $\mathcal{TD}$ .

Consider the situation after  $t - 1$  iterations of the algorithm. Then, from Lemma 4 there are at most  $O(n/\alpha^t)$  clusters in  $\tilde{T}$ . Setting  $t$  to be the maximal integer number such that  $\alpha^t + 1 \leq 3/4 \log_{12\sigma^2} n$  we obtain that there are at most  $n^{3/4}$  distinct subtrees of  $\mathcal{T}$  of size at most  $\alpha^t$ . As identical subtrees of  $\mathcal{T}$  are identified by the same node in the top DAG, all clusters created during the first  $t - 1$  iterations of the algorithm are represented by at most  $n^{3/4}$  nodes in  $\mathcal{TD}$ . Next, the remaining  $O(n/\alpha^t)$  clusters can introduce at most that many new nodes in the DAG.

Finally, the size of the DAG obtained by Algorithm 1 on a tree  $T$  of size  $n$  is bounded by  $n^{3/4} + O(n/\alpha^t) = O(n/\log_{12\sigma^2} n)$ , which is  $O(n/\log_\sigma n)$  as  $\sigma \geq 2$ . ◀

---

## References

- 1 Philip Bille, Finn Fernström, and Inge Li Gørtz. Tight bounds for top tree compression. In *SPIRE*, volume 10508 of *Lecture Notes in Computer Science*, pages 97–102. Springer, 2017.
- 2 Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Inf. Comput.*, 243:166–177, 2015.
- 3 Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB*, pages 141–152. Morgan Kaufmann, 2003.
- 4 Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.
- 5 Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.
- 6 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009.
- 7 Markus Frick, Martin Grohe, and Christoph Koch. Query evaluation on compressed trees (extended abstract). In *LICS*, page 188. IEEE Computer Society, 2003.
- 8 Paweł Gawrychowski and Artur Jeż. LZ77 factorisation of trees. In *FSTTCS*, volume 65 of *LIPICs*, pages 35:1–35:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- 9 Lorenz Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *SEA*, volume 9125 of *Lecture Notes in Computer Science*, pages 15–27. Springer, 2015.
- 10 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989.
- 11 Artur Jeż and Markus Lohrey. Approximation of smallest linear tree grammar. *Inf. Comput.*, 251:215–251, 2016.
- 12 Markus Lohrey and Sebastian Maneth. The complexity of tree automata and xpath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.
- 13 Markus Lohrey, Sebastian Maneth, and Eric Noeth. XML compression via dags. In *ICDT*, pages 69–80. ACM, 2013.
- 14 Markus Lohrey, Carl Philipp Reh, and Kurt Sieber. Optimal top dag compression. *CoRR*, abs/1712.05822, 2017. [arXiv:1712.05822](https://arxiv.org/abs/1712.05822).




# On the Maximum Colorful Arborescence Problem and Color Hierarchy Graph Structure

Guillaume Fertin<sup>1</sup>

LS2N UMR CNRS 6004, Université de Nantes, Nantes, France

guillaume.fertin@univ-nantes.fr

 <https://orcid.org/0000-0002-8251-2012>

Julien Fradin<sup>2</sup>


LS2N UMR CNRS 6004, Université de Nantes, Nantes, France

julien.fradin@univ-nantes.fr

Christian Komusiewicz<sup>3</sup>

Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Marburg, Germany

komusiewicz@informatik.uni-marburg.de

 <https://orcid.org/0000-0003-0829-7032>

---

## Abstract

Let  $G = (V, A)$  be a vertex-colored arc-weighted directed acyclic graph (DAG) rooted in some vertex  $r$ . The color hierarchy graph  $\mathcal{H}(G)$  of  $G$  is defined as follows: the vertex set of  $\mathcal{H}(G)$  is the color set  $\mathcal{C}$  of  $G$ , and  $\mathcal{H}(G)$  has an arc from  $c$  to  $c'$  if  $G$  has an arc from a vertex of color  $c$  to a vertex of color  $c'$ . We study the MAXIMUM COLORFUL ARBORESCENCE (MCA) problem, which takes as input a DAG  $G$  such that  $\mathcal{H}(G)$  is also a DAG, and aims at finding in  $G$  a maximum-weight arborescence rooted in  $r$  in which no color appears more than once. The MCA problem models the *de novo* inference of unknown metabolites by mass spectrometry experiments. Although the problem has been introduced ten years ago (under a different name), it was only recently pointed out that a crucial additional property in the problem definition was missing: by essence,  $\mathcal{H}(G)$  must be a DAG. In this paper, we further investigate MCA under this new light and provide new algorithmic results for this problem, with a focus on fixed-parameter tractability (FPT) issues for different structural parameters of  $\mathcal{H}(G)$ . In particular, we develop an  $\mathcal{O}^*(3^{x_{\mathcal{H}}})$ -time algorithm for solving MCA, where  $x_{\mathcal{H}}$  is the number of vertices of indegree at least two in  $\mathcal{H}(G)$ , thereby improving the  $\mathcal{O}^*(3^{|\mathcal{C}|})$ -time algorithm of Böcker et al. [Proc. ECCB '08]. We also prove that MCA is W[2]-hard with respect to the treewidth  $t_{\mathcal{H}}$  of the underlying undirected graph of  $\mathcal{H}(G)$ , and further show that it is FPT with respect to  $t_{\mathcal{H}} + \ell_{\mathcal{C}}$ , where  $\ell_{\mathcal{C}} := |V| - |\mathcal{C}|$ .

**2012 ACM Subject Classification** Theory of computation → Fixed parameter tractability, Theory of computation → Dynamic programming

**Keywords and phrases** Subgraph problem, computational complexity, algorithms, fixed-parameter tractability, kernelization

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.17

---

<sup>1</sup> GF was partially supported by PHC PROCOPE number 37748TL.

<sup>2</sup> JF was partially supported by PHC PROCOPE number 37748TL.

<sup>3</sup> CK was partially supported by the DFG, project MAGZ (KO 3669/4-1).



© Guillaume Fertin, Julien Fradin, and Christian Komusiewicz; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 17; pp. 17:1–17:15

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Motivated by *de novo* inference of metabolites from mass spectrometry experiments, Böcker et al. [4] introduced the MAXIMUM COLORFUL SUBTREE problem. This optimization problem takes as input a vertex-colored arc-weighted directed acyclic graph  $G = (V, A)$  rooted in some vertex  $r$ , and asks for a maximum-weight colorful arborescence in  $G$  with root  $r$ . Herein, a vertex-colored graph or a vertex set is called *colorful* if the vertices have pairwise different colors and a directed graph  $G$  is an *arborescence* with root  $r$  if the underlying undirected graph of  $G$  is a tree and there is a path from  $r$  to every vertex in  $G$ . In this model, the root  $r$  in  $G$  represents the sought metabolite, any vertex in  $G$  represents a molecule obtained from  $r$  after (possibly several) fragmentation(s), and vertices are colored according to their masses. An arc connects two molecules (vertices)  $u$  and  $v$  when  $v$  can be obtained from  $u$  by fragmentation, and is assigned a weight that indicates the (possibly negative) degree of confidence that the fragmentation from  $u$  to  $v$  actually occurs. A maximum-weight colorful arborescence from  $G$  with root  $r$  thus represents a most plausible fragmentation scenario from  $r$ . Let  $\mathcal{H}(G)$  be the following graph built from  $G$ :  $V(\mathcal{H}(G))$  is the set  $\mathcal{C}$  of colors used to color  $V(G)$ , and there is an arc from  $c$  to  $c'$  in  $\mathcal{H}(G)$  if there is an arc in  $G$  from a vertex of color  $c$  to a vertex of color  $c'$ . We call  $\mathcal{H}(G)$  the *color hierarchy graph* of  $G$ . Observe that  $\mathcal{H}(G)$  must be a DAG since colors represent masses and fragmenting a molecule gives new molecules with lower mass. As recently pointed out [14], the initial definition of MAXIMUM COLORFUL SUBTREE omits this crucial property of  $G$ . This led Fertin et al. [14] to reformulate the initial MAXIMUM COLORFUL SUBTREE problem as follows.

### Maximum Colorful Arborescence (MCA)

**Input:** A DAG  $G = (V, A)$  rooted in some vertex  $r$ , a set  $\mathcal{C}$  of colors, a coloring function  $\text{col} : V \rightarrow \mathcal{C}$  such that  $\mathcal{H}(G)$  is a DAG and an arc weight function  $w : A \rightarrow \mathbb{R}$ .

**Output:** A colorful arborescence  $T = (V_T, A_T)$  rooted in  $r$  of maximum weight  $w(T) := \sum_{a \in A_T} w(a)$ .

The study of MCA initiated in [14] essentially focused on the particular case where  $G$  is an arborescence and showed for example that MCA is NP-hard even for very restricted such instances. This work was also the first one to explicitly exploit that  $\mathcal{H}(G)$  is a DAG. In particular, it was shown that if  $\mathcal{H}(G)$  is an arborescence, then MCA is polynomially solvable. This latter promising result is the starting point of the present paper, in which we aim at better understanding the structural parameters of  $\mathcal{H}(G)$  that could lead to fixed-parameter tractable (FPT), *i.e.* exact and moderately exponential, algorithms. As pointed out in a recent study [12], obtaining exact solutions instead of approximate ones is indeed preferable for MCA. Hence, improved exact algorithms are truly desirable for this problem.

### Related Work

The MCA problem is NP-hard and highly inapproximable even when  $G$  is an arborescence and every arc weight is equal to 1 [14]. Moreover, MCA is NP-hard even if  $\ell_{\mathcal{C}} = 0$  where  $\ell_{\mathcal{C}} := |V(G)| - |\mathcal{C}|$  [14] (a consequence of the proof of [19, Theorem 1]). On the positive side, MCA can be solved in  $\mathcal{O}^*(3^{|\mathcal{C}|})$  time by dynamic programming [4]. Moreover, as previously mentioned, MCA is in P when  $\mathcal{H}(G)$  is an arborescence [14]. This result can be extended to some arborescence-like color hierarchy graphs as MCA can be solved by a branching algorithm in time  $\mathcal{O}^*(2^s)$  where  $s$  is the minimum number of arcs of  $\mathcal{H}$  whose

■ **Table 1** Overview of the results for the MCA problem presented in this paper. Here,  $x_{\mathcal{H}}$  is the number of vertices of indegree at least two in  $\mathcal{H}$ ,  $t_{\mathcal{H}}$  is the treewidth of the underlying undirected graph of  $\mathcal{H}$ ,  $\ell_{\mathcal{C}} := |V(G)| - |\mathcal{C}|$  and  $\ell \geq \ell_{\mathcal{C}}$  is the number of vertices that are not part of the solution.

Parameter	FPT status	Kernel status
$x_{\mathcal{H}}$	$\mathcal{O}^*(3^{x_{\mathcal{H}}})$ (Thm. 2.2)	No poly. kernel (Thm. 2.4)
$\ell$	W[1]-hard (from [19])	
$x_{\mathcal{H}} + \ell_{\mathcal{C}}$	FPT (from Thm. 2.2)	No poly. kernel (Thm. 2.7)
$x_{\mathcal{H}} + \ell$	Poly. kernel (Thm. 2.8)	
$t_{\mathcal{H}}$	W[2]-hard (Thm. 3.3)	
$t_{\mathcal{H}} + \ell_{\mathcal{C}}$	$\mathcal{O}^*(2^{\ell_{\mathcal{C}}} \cdot 4^{t_{\mathcal{H}}})$ (Thm. 3.7)	No poly. kernel (Cor. 3.8)

removal turns  $\mathcal{H}$  into an arborescence [14].<sup>4</sup> Finally, a solution of MCA of order  $k$  can be computed in  $\mathcal{O}^*((3e)^k)$  time using the color-coding technique [1] in combination with dynamic programming [7].

A related pattern matching problem in graphs is GRAPH MOTIF where, in its simplest version, we are given an undirected vertex-colored graph and ask whether there is a connected subgraph containing one vertex of each color [18, 13, 2, 3]. In contrast to MCA, GRAPH MOTIF is fixed-parameter tractable for the parameter  $\ell_{\mathcal{C}}$  [2, 15].

## Our Contribution

Our results are summarized in Table 1. We focus on two parameters from  $\mathcal{H}(G)$ , namely its number  $x_{\mathcal{H}}$  of vertices of indegree at least two, and the treewidth  $t_{\mathcal{H}}$  of its underlying undirected graph. This choice is motivated by the fact that when  $\mathcal{H}(G)$  is an arborescence, each of these two parameters is constant (namely,  $x_{\mathcal{H}} = 0$  and  $t_{\mathcal{H}} = 1$ ) and MCA is in P. Thus, our parameters measure the distance from this trivial case [16]. In addition, we consider the parameter  $\ell_{\mathcal{C}} := |V(G)| - |\mathcal{C}|$  and the parameter  $\ell$  which is the number of vertices that are not part of a solution with a maximum number of vertices. More precisely, whenever we refer to the parameter  $\ell$  we consider the problem variant where we are constrained to report the best arborescence among those with at least  $|V| - \ell$  vertices. Intuitively,  $\ell_{\mathcal{C}}$  is the number of vertices that we need to delete just to obtain a colorful subgraph of  $G$ , and hence  $\ell \geq \ell_{\mathcal{C}}$ . Observe that MCA is W[1]-hard parameterized by  $\ell$  [14]; this is a consequence of the proof of [19, Theorem 1].

Together with FPT issues, we also address the (in)existence of polynomial problem kernels for these parameters. In a nutshell, we provide a complete dichotomy for fixed-parameter tractability and problem kernelization for these parameters.

## Preliminaries

In the following, let  $G = (V, A)$  be the input graph of MCA, with  $n_G := |V(G)|$ . For any integer  $p$ , we let  $[p] := \{1, \dots, p\}$ . For any vertex  $v \in V$ ,  $N^+(v)$  is the set of outneighbors of  $v$ . We say that a vertex  $v$  is *reachable* from another vertex  $v' \in V(G)$  in a directed graph  $G$  if there exists a path from  $v'$  to  $v$  in  $G$ . The color hierarchy graph of  $G$  is denoted  $\mathcal{H}(G) := (\mathcal{C}, A_{\mathcal{C}})$ , or, when clear from the context, simply  $\mathcal{H}$ .

<sup>4</sup> The notation  $\mathcal{O}^*(\cdot)$  does not take polynomial factors into account.

We briefly recall the relevant notions of parameterized algorithmics (see e.g. [8]). A parameterized problem is a subset of  $\Sigma \times \mathbb{N}$  where the second component is the parameter. A parameterized problem is *fixed-parameter tractable* if every instance  $(x, k)$  can be solved in  $f(k) \cdot |x|^{O(1)}$  time. A *reduction to a problem kernel*, or *kernelization*, is an algorithm that takes as input an instance  $(x, k)$  of a parameterized problem  $Q$  and produces in polynomial time an equivalent (*i.e.*, having the same solution) instance  $(x', k')$  of  $Q$  such that (i)  $|x'| \leq g(k)$ , and (ii)  $k' \leq k$ . The instance  $(x', k')$  is called *problem kernel*, and  $g$  is called the *size of the problem kernel*. If  $g$  is a polynomial function, then the problem admits a *polynomial-size kernel*. Classes  $W[1]$  and  $W[2]$  are classes of presumed fixed-parameter intractability: if a parameterized problem is  $W[1]$ -hard or  $W[2]$ -hard, then it is generally assumed that it is not fixed-parameter tractable.

This paper is organized as follows. In Section 2, we study in detail the impact of  $x_{\mathcal{H}}$  on the parameterized complexity of the MCA problem, while in Section 3, the same type of study is realized with parameter  $t_{\mathcal{H}}$ .

## 2 Parameterizing the MCA Problem by $x_{\mathcal{H}}$

Two main reasons lead us to be particularly interested in  $x_{\mathcal{H}}$ , the number of vertices with indegree at least two in  $\mathcal{H}$ . First, MCA is in  $P$  when  $\mathcal{H}$  is an arborescence [14], thus when  $x_{\mathcal{H}} = 0$ . Second, MCA can be solved in  $\mathcal{O}^*(3^{|\mathcal{C}|})$  time [4]. Since by definition  $x_{\mathcal{H}} \leq |\mathcal{C}|$ , determining whether MCA is FPT with respect to  $x_{\mathcal{H}}$  is of particular interest. We answer this question positively in Theorem 2.2. We first need some additional definitions.

Let  $X$  be the set of vertices of indegree at least two in  $\mathcal{H}$  (thus  $|X| = x_{\mathcal{H}}$ ) and call  $X$  the set of *difficult colors*. For any  $V' \subseteq V(G)$ , let  $\text{col}(V')$  denote the set of colors used by  $\text{col}$  on the vertices in  $V'$ . Moreover, for any vertex  $v \in V$  that has at least one outneighbor in  $G$ , assume that  $\text{col}(N^+(v))$  has an arbitrary but fixed ordering. Therefore, for any  $i \in [|\text{col}(N^+(v))|]$ , we may let  $\text{col}^+(v, i)$  denote the  $i$ th color in  $\text{col}(N^+(v))$ . Finally, for any arborescence  $T$  in  $G$  or in  $\mathcal{H}$ , let  $X(T) := X \cap \text{col}(V(T))$  denote the set of difficult colors in  $T$ . We have the following lemma.

► **Lemma 2.1.** *Let  $T_1$  and  $T_2$  be two arborescences in  $\mathcal{H}$  such that  $T_1$  is rooted in  $c_1$ ,  $T_2$  is rooted in  $c_2 \neq c_1$ , and  $c_1, c_2 \in N^+(c)$  for some  $c \in \mathcal{C}$ . If  $X(T_1)$  and  $X(T_2)$  are disjoint, then  $V(T_1)$  and  $V(T_2)$  are disjoint.*

**Proof.** Assume without loss of generality that  $c_1$  is not reachable from  $c_2$  in  $\mathcal{H}$ . If  $V(T_1)$  and  $V(T_2)$  are not disjoint, then there exists a color  $c^* \in \mathcal{C}$  that belongs to  $T_1$  and to  $T_2$ . In order to prove that such a color  $c^*$  cannot exist, let  $\tau_1$  (resp.  $\tau_2$ ) be the set of colors on the path from  $c_1$  (resp.  $c_2$ ) to  $c^*$  including  $c_1$  in  $T_1$  (resp.  $c_2$  in  $T_2$ ). Then, either  $\tau_2 \subset \tau_1$  or  $c_2 \notin \tau_1$ . First, if  $\tau_2 \subset \tau_1$ , then there exists a vertex  $c' \in \tau_1$  such that  $c' \neq c$  with an arc  $(c', c_2)$ . Since  $\mathcal{H}$  contains the arc  $(c, c_2)$ , the color  $c_2$  is thus difficult. This contradicts the assumption that  $X(T_1)$  and  $X(T_2)$  are disjoint. Second, if  $c_2 \notin \tau_1$ , then  $|\tau_1 \cap \tau_2| \geq 1$  since  $c^* \in \tau_1 \cap \tau_2$ . Therefore, let  $\bar{c} \in \tau_1 \cap \tau_2$  such that there exists a path from  $\bar{c}$  to any other color of  $\tau_1 \cap \tau_2$ . By definition, the father of  $\bar{c}$  in  $\tau_1$  is different from the father of  $\bar{c}$  in  $\tau_2$ , which means that  $\bar{c}$  is a difficult color. This contradicts the assumption that  $X(T_1)$  and  $X(T_2)$  are disjoint. ◀

► **Theorem 2.2.** *MCA can be solved in  $\mathcal{O}^*(3^{x_{\mathcal{H}}})$  time and  $\mathcal{O}^*(2^{x_{\mathcal{H}}})$  space.*

**Proof.** We propose a dynamic programming algorithm which makes use of two tables. The first one,  $A[v, X', i]$ , is computed for all  $v \in V(G)$ ,  $X' \subseteq X$  and  $i \in \{0\} \cup [|\text{col}(N^+(v))|]$  and stores the weight of a maximum colorful arborescence  $T_A(v, X', i)$  in  $G$  such that

- $T_A(v, X', i)$  is rooted in  $v$ ,
- $(X(T_A(v, X', i)) \setminus \{\text{col}(v)\}) \subseteq X'$ , and
- $T_A(v, X', i)$  contains an arc  $(v, u)$  only if  $\text{col}(u) = \text{col}^+(v, j)$  for some  $j \leq i$ .

The second one,  $B[v, X', i]$ , is computed for all  $v \in V$ ,  $X' \subseteq X$  and  $i \in [|\text{col}(N^+(v))|]$  and stores the weight of a maximum colorful arborescence  $T_B(v, X', i)$  in  $G$  such that

- $T_B(v, X', i)$  is rooted in  $v$ ,
- $(X(T_B(v, X', i)) \setminus \{\text{col}(v)\}) \subseteq X'$ , and
- $T_B(v, X', i)$  contains an arc  $(v, u)$  only if  $\text{col}(u) = \text{col}^+(v, i)$ .

In a nutshell,  $T_A(v, X', i)$  and  $T_B(v, X', i)$  share the same root  $v$  and the same allowed set of difficult colors  $X'$  (disregarding  $\text{col}(v)$ ), but  $T_A(v, X', i)$  contains outneighbors of  $v$  up to color  $\text{col}^+(i)$  and  $T_B(v, X', i)$  contains at most one outneighbor of  $v$  which is of color  $\text{col}^+(v, i)$ . Hence, there is no  $u \in N^+(v)$  such that  $(v, u) \in T_A(v, X', i-1)$  and  $(v, u) \in T_B(v, X', i)$ . We now show how to compute the two abovementioned tables.

$$A[v, X', i] = \begin{cases} 0 & \text{if } i = 0, \\ \max_{X'' \subseteq X'} \{A[v, X'', i-1] + B[v, X' \setminus X'', i]\} & \text{otherwise.} \end{cases}$$

For an entry  $A[v, X', i]$  with  $i = 0$  note that  $T_A(v, X', i)$  can only contain  $v$ . For  $i > 0$ , by definition there cannot exist any  $u \in N^+(v)$  such that  $u$  belongs both to  $T_A(v, X'', i-1)$  and  $T_B(v, X' \setminus X'', i)$ . Therefore, Lemma 2.1 shows that  $\text{col}(v)$  is the only color occurring in  $T_A(v, X'', i-1)$  and  $T_B(v, X' \setminus X'', i)$ . Thus, the union of  $T_A(v, X'', i-1)$  and  $T_B(v, X' \setminus X'', i)$  is a colorful arborescence. Finally, testing every possible  $X'' \subseteq X'$  ensures the correctness of the formula.

$$B[v, X', i] = \begin{cases} 0 & \text{if } \text{col}^+(v, i) \in X \setminus X', \\ \max_{\substack{u \in N^+(v): \\ \text{col}(u) = \text{col}^+(v, i)}} \{0, w(v, u) + A[u, X', |\text{col}(N^+(u))|]\} & \text{otherwise.} \end{cases}$$

For an entry of type  $B[v, X', i]$ , if  $\text{col}^+(v, i)$  is a difficult color which does not belong to  $X'$ , then  $V(T_B(v, X', i)) = \{v\}$ , and hence  $B[v, X', i] = 0$ . Otherwise, recall that  $B[v, X', i]$  stores the weight of a maximum colorful arborescence rooted in  $v$  containing at most one further vertex  $u \in N^+(v)$  of color  $\text{col}^+(v, i)$ . Therefore, computing the maximum colorful arborescences for any such  $u$  and only keeping the best one if it is positive ensures the correctness of the formula.

Recall that any DAG has a topological ordering of its vertices, *i.e.* a linear ordering of its vertices such that for every arc  $(u, v)$ ,  $u$  appears before  $v$  in this ordering. In Algorithm 1, we show how to compute all the entries of both dynamic programming tables. For this, we consider the entries from last to first according to some topological ordering of  $G$ . The total running time derives from the fact that our algorithm needs at most  $3^{x_{\mathcal{H}}}$  steps to compute  $A[v, X', i]$  since a difficult color can be in  $X''$ ,  $X' \setminus X''$  or in  $X \setminus X'$ . ◀

Recall that a parameterized problem  $Q$  is FPT with respect to a parameter  $k$  if and only if it has a kernelization algorithm for  $k$  [11], but that such a kernel is not necessarily polynomial. In Theorem 2.4, we prove that although MCA parameterized by  $x_{\mathcal{H}}$  is FPT (as proved by Theorem 2.2), MCA is unlikely to admit a polynomial kernel for  $x_{\mathcal{H}}$ . For this, we use the or-composition technique which, roughly speaking, is a reduction that combines many instances of a problem into one instance of the problem  $Q$ . We first recall the definition of or-compositions.

**Algorithm 1** COMPUTING THE ENTRIES IN TABLES  $A$  AND  $B$ .

---

```

for all  $v \in V$  from last to first in some topological ordering of  $G$  do
  for all  $X' \subseteq X$  do
    for all  $i \in \{1, \dots, |\text{col}(N^+(v))|\}$  do
      Compute  $B[v, X', i]$ 
    end for
  end for
  for all  $X' \subseteq X$  do
    for all  $i \in \{0, \dots, |\text{col}(N^+(v))|\}$  do
      Compute  $A[v, X', i]$ 
    end for
  end for
end for

```

---

► **Definition 2.3.** ([5]) An *or-composition* for a parameterized problem  $Q \in \Sigma \times \mathbb{N}$  is an algorithm that receives as input a sequence  $(x_1, k), (x_2, k), \dots, (x_t, k)$  with  $(x_i, k) \in \Sigma \times \mathbb{N}$  for each  $1 \leq i \leq t$ , takes polynomial time in  $\sum_{i=1}^t |x_i| + k$ , and outputs  $(y, k') \in \Sigma \times \mathbb{N}$  with  $(y, k') \in Q$  if and only if  $\exists_{1 \leq i \leq t} (x_i, k) \in Q$  and  $k'$  is polynomial in  $k$ .

If an NP-hard parameterized problem  $Q$  admits an or-composition, then  $Q$  does not admit any polynomial-size problem kernel (unless  $\text{NP} \subseteq \text{coNP/Poly}$ ) [5]. Our or-composition actually shows that MCA is unlikely to admit a polynomial kernel for the parameter  $|\mathcal{C}|$ .

► **Theorem 2.4.** *Unless  $\text{NP} \in \text{coNP/Poly}$ , MCA does not admit a polynomial kernel for parameter  $|\mathcal{C}|$ , and consequently for parameter  $x_{\mathcal{H}}$ , even if  $G$  is an arborescence.*

**Proof.** In the following, let  $t$  be a positive integer. For any  $i \in [t]$ , let  $G_i = (V_i, A_i)$  be the graph of an instance of MCA which is rooted in a vertex  $r_i$  and assume that the  $t$  instances are built on the same color set  $\mathcal{C}' = \{c_1, \dots, c_{|\mathcal{C}'|}\}$ , otherwise colors can be relabeled suitably.

We now compose the  $t$  instances of MCA into a new instance of MCA. Let  $G = (V, A)$  be the graph of such a new instance with  $V = \{r\} \cup \{r'_i : i \in [t]\} \cup \{v \in V_i : i \in [t]\}$  and  $A = \{(r, r'_i) : i \in [t]\} \cup \{(r'_i, r_i) : i \in [t]\} \cup \{(u, v) \in A_i : i \in [t]\}$ . Here,  $r$  is a vertex not contained in any of the  $t$  MCA instances and which has a path of length 2 towards the root  $r_i$  of any graph  $G_i$ ; thus  $G$  is clearly a DAG. Let  $\mathcal{C}$  be the color set of  $G$ , and let us define the coloring function on  $V(G)$  as follows: the root  $r$  is assigned a unique color  $c_r \notin \mathcal{C}'$ ; all vertices of type  $r'_i$  are assigned the same color  $c_{r'} \notin (\mathcal{C}' \cup \{c_r\})$ ; all arcs of type  $(r'_i, r_i)$  and  $(r, r'_i)$  are given a weight of 0; the color (resp. weight) of all other vertices (resp. arcs) is the same in the new instance as in their initial instance. Clearly,  $(G, \mathcal{C}, \text{col}, w, r)$  is a correct instance of MCA and  $|\mathcal{C}| = |\mathcal{C}'| + 2$ . Moreover, if  $G_i$  is an arborescence for every  $i \in [t]$ , then  $G$  is also an arborescence. We now prove that there exists  $i \in [t]$  such that  $G_i$  has a colorful arborescence  $T = (V_T, A_T)$  rooted in  $r_i$  of weight  $W > 0$  if and only if  $G$  has a colorful arborescence  $T' = (V_{T'}, A_{T'})$  rooted in  $r$  and of weight  $W > 0$ .

( $\Rightarrow$ ) If there exists  $i \in [t]$  such that  $G_i$  has a colorful arborescence  $T = (V_T, A_T)$  rooted in  $r_i$  and of weight  $W > 0$ , then let  $T' = (V_{T'}, A_{T'})$  with  $V_{T'} = V_T \cup \{r, r'_i\}$  and  $A_{T'} = A_T \cup \{(r, r'_i), (r'_i, r_i)\}$ . Clearly,  $T'$  is connected, colorful and of weight  $W$ .

( $\Leftarrow$ ) Suppose  $G$  contains a colorful arborescence  $T' = (V_{T'}, A_{T'})$  with root  $r$  and weight  $W > 0$ . Since  $T'$  is colorful and all vertices of type  $r'_i$  share the same color, there cannot exist  $i$  and  $j$  in  $[t]$ ,  $v_i \in V_i$  and  $v_j \in V_j$  such that both  $v_i$  and  $v_j$  belong to  $T'$ . Thus, let  $i^*$  be the only index in  $[t]$  such that  $V_{i^*} \cap V_{T'} \neq \emptyset$  and let  $T = (V_T, A_T)$  with  $V_T = V_{T'} \setminus \{r, r'_{i^*}\}$  and  $A_T = A_{T'} \setminus \{(r, r'_{i^*}), (r'_{i^*}, r_{i^*})\}$ . Clearly,  $T$  is connected, colorful and of weight  $W$ .

Now, recall that  $|\mathcal{C}| = |\mathcal{C}'| + 2$  and thus that we made a correct composition of MCA into MCA. Moreover, recall that MCA is NP-hard [14] and that  $x_{\mathcal{H}} \leq |\mathcal{C}|$ . As a consequence, MCA does not admit a polynomial kernel for the parameter  $|\mathcal{C}|$ , and hence for the parameter  $x_{\mathcal{H}}$ , even in arborescences, unless  $\text{NP} \subseteq \text{coNP/Poly}$ . ◀

Recall that MCA can be solved in time  $\mathcal{O}^*(2^s)$  where  $s$  is the minimum number of arcs needed to turn  $\mathcal{H}$  into an arborescence [14]. Since  $s < |\mathcal{C}|^2$ , we have the following.

► **Corollary 2.5.** *Unless  $\text{NP} \in \text{coNP}/\text{Poly}$ , MCA parameterized by  $s$  does not admit a polynomial kernel, even if  $G$  is an arborescence.*

In the following, we use a different technique, called polynomial parameter transformation [6], to show that MCA is also unlikely to admit a polynomial kernel for the parameter  $x_{\mathcal{H}} + \ell_{\mathcal{C}}$ , where  $\ell_{\mathcal{C}} = n_G - |\mathcal{C}|$ .

► **Definition 2.6.** ([6, 10, 9]) Let  $P$  and  $Q$  be two parameterized problems. We say that  $P$  is *polynomial parameter reducible* to  $Q$  if there exists a polynomial-time computable function  $f : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \times \mathbb{N}$  and a polynomial  $p$ , such that for all  $(x, k) \in \Sigma^* \times \mathbb{N}$  the following holds:  $(x, k) \in P$  if and only if  $(x', k') = f(x, k) \in Q$ , and  $k' \leq p(k)$ . The function  $f$  is called a *polynomial parameter transformation*.

If  $P$  is an NP-hard problem and  $Q$  belongs to NP, then a polynomial parameter transformation from  $P$  parameterized by  $k$  to  $Q$  parameterized by  $k'$  has the following consequence: if  $Q$  parameterized by  $k'$  admits a polynomial kernel, then  $P$  parameterized by  $k$  admits a polynomial kernel [6]. Using such a transformation, we obtain the following result.

► **Theorem 2.7.** *MCA parameterized by  $x_{\mathcal{H}}$  does not admit a polynomial kernel unless  $\text{NP} \subseteq \text{coNP}/\text{Poly}$  even when restricted to the special case where  $\ell_{\mathcal{C}} = 0$ .*

**Proof.** We reduce from SET COVER, which is defined as follows.

**Set Cover**

**Input:** A universe  $\mathcal{U} = \{u_1, u_2, \dots, u_q\}$ , a family  $\mathcal{F} = \{S_1, S_2, \dots, S_p\}$  of subsets of  $\mathcal{U}$ , an integer  $k$ .

**Output:** A  $k$ -sized subfamily  $\mathcal{S} \subseteq \mathcal{F}$  of sets whose union is  $\mathcal{U}$ .

The reduction is as follows: for any instance of SET COVER, we create a three-levels DAG  $G = (V = V_1 \cup V_2 \cup V_3, A)$  with  $V_1 = \{r\}$ ,  $V_2 = \{v_i : i \in [p]\}$  and  $V_3 = \{z_j : j \in [q]\}$ . We call  $V_2$  the *second level* of  $G$  and  $V_3$  the *third level* of  $G$ . Informally, we associate one vertex at the second level to each set of  $\mathcal{F}$  and one vertex at the third level to each element of  $\mathcal{U}$ . There is an arc of weight  $-1$  from  $r$  to each vertex at level 2 and an arc of weight  $p$  from  $v_i$  to  $z_j$ , for all  $i \in [p]$  and  $j \in [q]$  such that the element  $u_j$  is contained in the set  $S_i$ . Now, our coloring function  $\text{col}$  is as follows: give a unique color to each vertex of  $G$ . Notice that  $\mathcal{H}$  is also a three-levels DAG with  $\text{col}(V_1)$ ,  $\text{col}(V_2)$ , and  $\text{col}(V_3)$  at the first, second, and third levels, respectively. Therefore, the above construction is a correct instance of MCA. We now prove that there exists a  $k$ -sized subfamily  $\mathcal{S} \subseteq \mathcal{F}$  of sets whose union is  $\mathcal{U}$  if and only if there exists a colorful arborescence  $T$  in  $G$  of weight  $w(T) = pq - k$ .

( $\Rightarrow$ ) Suppose there exists a  $k$ -sized subfamily  $\mathcal{S} \subseteq \mathcal{F}$  of sets whose union is  $\mathcal{U}$  and let  $\mathbf{True} = \{i \in [p] : S_i \in \mathcal{S}\}$ . Then, we set  $V_T = \{r\} \cup \{v_i : i \in \mathbf{True}\} \cup \{z_j : j \in [q]\}$ . Necessarily,  $G[V_T]$  is connected: first,  $r$  is connected to every level-2 vertex; second, a vertex  $z_j$  corresponds to an element  $u_j$  which is contained in some set  $S_i \in \mathcal{S}$ . Now, let  $T$  be a spanning arborescence of  $G[V_T]$ . Clearly,  $T$  is colorful and of weight  $pq - k$ .

( $\Leftarrow$ ) Suppose there exists a colorful arborescence  $T = (V_T, A_T)$  in  $G$  of weight  $w(T) = pq - k$ . Notice that any arborescence  $T'$  in  $G$  which contains  $r$  and at least one vertex from  $V_3$  must contain at least one vertex from  $V_2$  in order to be connected. Therefore, if such an arborescence  $T'$  does not contain one vertex of type  $z_j$ , then  $w(T') < pq - p - 1$  and  $w(T') < w(T)$ . Hence, if  $w(T) = pq - k$  then  $T$  contains each vertex of the third level and  $T$



contains exactly  $k$  vertices at the second level. Now, let  $\mathcal{S} = \{S_i : i \in [p] \text{ s.t. } v_i \in V_T\}$  and notice that  $\mathcal{S}$  is a  $k$ -sized subfamily of  $\mathcal{F}$  whose union is  $\mathcal{U}$  as all vertices of the third level belong to  $T$ . Our reduction is thus correct.

Now, recall that  $\mathcal{H}$  is a three-levels DAG with  $\text{col}(V_1)$ ,  $\text{col}(V_2)$ , and  $\text{col}(V_3)$  at the first, second and third levels, respectively. By construction of  $G$ , if there exists  $c \in V(\mathcal{H})$  such that  $d^-(c) \geq 1$ , then  $c \in \text{col}(V_3)$ . Moreover, recall that  $|\text{col}(V_3)| = |\mathcal{U}|$  and thus  $x_{\mathcal{H}} \leq |\mathcal{U}|$ . Thus we provided a correct polynomial parameter transformation from SET COVER parameterized by  $|\mathcal{U}|$  to MCA parameterized by  $x_{\mathcal{H}}$ . Now, recall that SET COVER does not admit a polynomial kernel for  $|\mathcal{U}|$  unless  $\text{NP} \subseteq \text{coNP}/\text{Poly}$  [10] and that SET COVER is NP-hard [17]. Moreover, the decision version of MCA, which asks for a solution of weight at least  $k$ , clearly belongs to NP. Finally, observe that  $\ell_{\mathcal{C}} = 0$  as  $G$  is colorful. As a consequence, MCA does not admit any polynomial kernel for  $x_{\mathcal{H}}$  unless  $\text{NP} \subseteq \text{coNP}/\text{Poly}$  even if  $\ell_{\mathcal{C}} = 0$ . ◀

Since  $\ell \geq \ell_{\mathcal{C}}$ , and in light of Theorem 2.7, we aim at determining whether a polynomial kernel exists for MCA parameterized  $x_{\mathcal{H}} + \ell$ . We have the following theorem.

► **Theorem 2.8.** *MCA admits a problem kernel with  $\mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$  vertices.*

To show this result we provide three data reduction rules. To formulate the rules, we introduce some notation first.

For any vertex  $v \in V(G)$ , we define  $G^+(v)$  as the subgraph of  $G$  that is induced by the set of vertices that are reachable from  $v$  in  $G$  (including  $v$ ). Similarly, for any color  $c \in V(\mathcal{H})$ , we define  $\mathcal{H}^+(c)$  as the subgraph of  $\mathcal{H}$  that is induced by the set of vertices that are reachable from  $c$  in  $\mathcal{H}$  (including  $c$ ). We call a color  $c$  *autonomous* if (i)  $\mathcal{H}^+(c)$  is an arborescence, and (ii) there does not exist an arc from a color  $c_1 \notin \mathcal{H}^+(c)$  to a color  $c_2 \in \mathcal{H}^+(c)$  in  $\mathcal{H}$ . For a vertex  $v$ , let  $T_v$  denote a maximum colorful arborescence in  $G$  that is rooted at  $v$ . Finally, for a color  $c \in \mathcal{C}$ , let  $V_c := \{v \in V : \text{col}(v) = c\}$  denote the set of vertices with color  $c$ .

► **Reduction Rule 1.** *If an instance  $(G, \mathcal{C}, \text{col}, w, r)$  of MCA contains an autonomous color  $c$  such that  $\mathcal{H}^+(c)$  contains at least two vertices, then do the following.*

- For each vertex  $v \in V_c$ , compute the value  $w(T_v)$  of  $T_v$ , and add  $w(T_v)$  to the weight of each incoming arc of  $v$ .
- Remove from  $G$  all vertices that are reachable from a vertex in  $V_c$ , except the vertices of  $V_c$ .

► **Lemma 2.9.** *Reduction Rule 1 is correct and can be performed exhaustively in polynomial time.*

**Proof.** Consider a vertex  $v \in V_c$ . Since  $c$  is autonomous,  $\mathcal{H}^+(c)$  is an arborescence and thus we may compute  $T_v$  which contains only colors from  $\mathcal{H}^+(c)$  in polynomial time [14].

Now, we prove the correctness of the rule, that is, the original instance  $(G, \mathcal{C}, \text{col}, w, r)$  has a colorful arborescence  $T = (V_T, A_T)$  of weight at least  $W$  if and only if the new instance  $(G', \mathcal{C}', \text{col}', w', r')$  has a colorful arborescence  $T' = (V_{T'}, A_{T'})$  of weight at least  $W$ . We only show the forward direction of the equivalence; the converse can be seen by symmetric arguments. First, recall that  $c$  is autonomous. Therefore, if  $T$  does not contain any vertex of color  $c$ , then  $T$  does not contain any vertex whose color belongs to  $V(\mathcal{H}^+(c))$  and we can trivially set  $T' = T$ . Otherwise, if  $T$  contains a vertex  $v$  of color  $c$ , then let  $S_c \subseteq V_T$  be the set of vertices that are reachable from  $v$  in  $T$ . We now set  $V_{T'} := (V_T \setminus S_c) \cup \{v\}$  and let  $A_{T'}$  contain all the arcs from  $A_T$  that are not in  $\mathcal{H}^+(c)$ . Now, recall that we computed the weight  $w(T_v)$  of the maximum colorful arborescence in  $G$  that was rooted in  $v$  and that  $w'(v^-, v) = w(v^-, v) + w(T_v)$  where  $v^-$  is the inneighbor of  $v$  in  $T$ . This ensures that  $w(T) \leq w'(T')$ . ◀



In the following, for any vertices  $v, v' \in V(G)$  such that  $v'$  is reachable from  $v$  in  $G$ , we denote  $\pi(v, v')$  as the length of the maximum weighted path from  $v$  to  $v'$  in  $G$ .

► **Reduction Rule 2.** *If an instance  $(G, \mathcal{C}, \text{col}, w, r)$  of MCA contains a triple  $\{c_1, c_2, c_3\} \subseteq \mathcal{C}$  such that (i)  $c_1$  is the unique inneighbor of  $c_2$ , (ii)  $c_2$  is the unique inneighbor of  $c_3$  and (iii)  $c_3$  is the unique outneighbor of  $c_2$ , then do the following.*

- For any  $v_1 \in V_{c_1}$  and  $v_3 \in V_{c_3}$  such that there exists a path from  $v_1$  to  $v_3$  in  $G$ , create an arc  $(v_1, v_3)$  and set  $w(v_1, v_3) := \pi(v_1, v_3)$ .
- Add a vertex  $v^*$  of color  $c_3$  and, for any vertex  $v_1 \in V_{c_1}$  that has at least one outneighbor of color  $c_2$  in  $G$ , add the arc  $(v_1, v^*)$  and set  $w(v_1, v^*)$  to the highest weighted outgoing arc from  $v_1$  to any vertex of color  $c_2$  in  $G$ .
- Remove all vertices of  $V_{c_2}$  from  $G'$ .

► **Lemma 2.10.** *Reduction Rule 2 is correct and can be performed exhaustively in polynomial time.*

**Proof.** We first prove that our transformation is correct. We show only the direction that an arborescence of weight at least  $W$  in the original instance  $(G, \mathcal{C}, \text{col}, w, r)$  implies an arborescence of weight at least  $W$  in the new instance  $(G', \mathcal{C}', \text{col}', w', r')$ ; the converse direction can be shown by symmetric arguments. Let  $T = (V_T, A_T)$  be a colorful arborescence of weight  $W$  in the original instance. First, if  $T$  does not contain a vertex of color  $c_2$ , then  $T$  is an arborescence of the new instance. Second, if  $T$  contains a vertex  $v_2$  of color  $c_2$  whose inneighbor is  $v_1$  in  $T$  and if  $T$  does not contain any vertex of color  $c_3$ , then setting  $V_{T'} := V_T \setminus \{v_2\} \cup \{v^*\}$  and  $A_{T'} := A_T \setminus \{(v_1, v_2)\} \cup \{(v_1, v^*)\}$  gives an arborescence  $T' = (V_{T'}, A_{T'})$  of the new instance. Moreover,  $w(T) = w'(T')$  since  $w(v_1, v_2) = w'(v_1, v^*)$ . Third, if  $T$  contains a vertex  $v_2$  of color  $c_2$  whose inneighbor is  $v_1$  in  $T$  and if  $T$  contains a vertex  $v_3$  of color  $c_3$  (whose inneighbor is necessarily  $v_2$ ), then setting  $V_{T'} := V_T \setminus \{v_2\}$  and  $A_{T'} := A_T \setminus \{(v_1, v_2), (v_2, v_3)\} \cup \{(v_1, v_3)\}$  gives an arborescence  $T' = (V_{T'}, A_{T'})$  of the new instance. Moreover,  $w(T) = w'(T')$  since  $w(v_1, v_2) + w(v_2, v_3) = w'(v_1, v_3)$ .

The polynomial running time follows from the fact that  $\pi(v_1, v_3)$  can be computed in polynomial time. ◀

To describe the final rule, let  $N_{\bar{v}}^-(v)$  denote the set of unique colors in the inneighborhood of  $v$  in  $G$ , where a color  $c$  is *unique* if  $|V_c| = 1$ . Recall also that  $\ell$  is the maximum number of vertices that do not belong to  $T$  in  $G$ .

► **Reduction Rule 3.** *If an instance  $(G, \mathcal{C}, \text{col}, w, r)$  of MCA contains a vertex  $v \in V$  such that  $|N_{\bar{v}}^-(v)| > \ell + 1$ , then delete the  $|N_{\bar{v}}^-(v)| - \ell - 1$  least-weighted arcs from  $N_{\bar{v}}^-(v)$  to  $v$ .*

► **Lemma 2.11.** *Reduction Rule 3 is correct and can be performed exhaustively in polynomial time.*

**Proof.** Since  $|N_{\bar{v}}^-(v)| > \ell + 1$ ,  $T$  has to contain at least two vertices from  $N_{\bar{v}}^-(v)$ . Now, let  $v_1$  be a vertex from  $N_{\bar{v}}^-(v)$  such that  $(v_1, v)$  is the least-weighted incoming arc from a unique color to  $v$  in  $G$ . Even if  $v_1$  belongs to  $T$ , there will always exist at least one other vertex  $v_2$  that will also belong to  $T$  and such that  $w(v_1, v) \leq w(v_2, v)$ . Thus, we may assume that  $T$  does not contain the arc  $(v_1, v)$  and safely delete it. The correctness of the rule now follows from repeated application of this argument. ◀

We are now ready to prove Theorem 2.8.

**Proof.** The kernelization algorithm consists of the exhaustive application of Reduction Rules 1–3 in polynomial time. Let  $(G, \mathcal{C}, \text{col}, w, r)$  denote the resulting equivalent instance and let  $T = (V_T, A_T)$  be a solution of this instance. It remains to show that  $G$  has  $\mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$  vertices. First, we show that the indegree of any color in  $\mathcal{H}$  is at most  $(\ell + 1)^2 + \ell$ . This will allow us to show, subsequently, the claimed bound on  $n_G$ .

Let us first bound the indegree of any color in  $\mathcal{H}$ . Since  $T$  is colorful and since  $|V_T| = n_G - \ell$ , there exist at most  $\ell$  non-unique colors in  $\mathcal{C}$  and hence the inneighborhood of any color  $c \in V(\mathcal{H})$  cannot contain more than  $\ell$  non-unique colors in  $\mathcal{H}$ . Moreover, since the instance is reduced with respect to Reduction Rule 3, the inneighborhood of any vertex  $v \in V(G)$  contains at most  $\ell + 1$  vertices of unique color in  $G$ . Furthermore, we may assume  $|V_c| \leq \ell + 1$  for any any color  $c \in V(\mathcal{H})$  as  $T$  cannot be colorful if there exists more than  $\ell + 1$  occurrences of  $c$  in  $G$ . As a consequence, for any color  $c \in V(\mathcal{H})$ , the inneighborhood of  $c$  cannot contain more than  $|V_c| \cdot (\ell + 1) = (\ell + 1)^2$  unique colors in  $\mathcal{H}$ , and hence  $c$  has at most  $(\ell + 1)^2 + \ell$  inneighbors.

Now, let  $F$  be the forest whose vertex set is  $\mathcal{C}_F = \mathcal{C} \setminus X$  and which contains each arc  $(c, c')$  of  $\mathcal{H}$  such that  $\{c, c'\} \subseteq \mathcal{C}_F$ . In the following, we successively bound the maximum number of leaves of  $F$ , the maximum number of vertices of  $F$ , of  $V(\mathcal{H})$  and finally of  $V(G)$  in a function of  $\ell$  and  $x_{\mathcal{H}}$ . First, recall that there does not exist any autonomous color  $c \in \mathcal{C}$  to which Reduction Rule 1 applies. Thus, each leaf  $c$  of  $\mathcal{H}$  is in fact a difficult color. Consequently, every leaf of  $F$  is in  $\mathcal{H}$  an inneighbor of a difficult color. Since the maximum indegree of any color in  $\mathcal{H}$  is at most  $(\ell + 1)^2 + \ell$ , the number of leaves in  $F$  is at most  $x_{\mathcal{H}}((\ell + 1)^2 + \ell)$ . Now, by Lemma 2.10,  $\mathcal{H}$  does not contain any color which has a unique inneighbor and a unique outneighbor. As a consequence,  $F$  has no internal vertices of degree two that are not inneighbors of a difficult color. Hence, the number of nonleaves of  $F$  that are not inneighbors of a difficult color is  $\mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$ , and thus  $|V(F)| = \mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$ . Moreover, since  $\mathcal{C}_F = \mathcal{C} \setminus X$ , we have that  $|\mathcal{C}| \leq x_{\mathcal{H}} + \mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$ . Finally, the number of vertices in  $G$  can exceed the number of colors in  $\mathcal{H}$  by at most  $\ell$ . Therefore,  $n_G = \mathcal{O}(x_{\mathcal{H}} \cdot \ell^2)$  as claimed. ◀

### 3 Parameterizing the MCA Problem by the Treewidth of the Color Hierarchy Graph

Let  $U(\mathcal{H})$  denote the underlying undirected graph of  $\mathcal{H}$ . In this section, we are interested in parameter  $t_{\mathcal{H}}$ , defined as the treewidth of  $U(\mathcal{H})$ . Indeed, since MCA is in P whenever  $\mathcal{H}$  is an arborescence [14], it is natural to study whether MCA parameterized by  $t_{\mathcal{H}}$  is FPT. To do so, we first introduce some definitions.

► **Definition 3.1.** Let  $G = (V, E)$  be a undirected graph. A tree decomposition of  $G$  is a pair  $\langle \{X_i : i \in I\}, \mathcal{T} \rangle$ , where  $\mathcal{T}$  is a tree whose vertex set is  $I$ , and each  $X_i$  is a subset of  $V$ , called a *bag*. The following three properties must hold:

1.  $\cup_{i \in I} X_i = V$ .
2. For every edge  $(u, v) \in E$ , there is an  $i \in I$  such that  $\{u, v\} \subseteq X_i$ .
3. For all  $i, j, k \in I$ , if  $j$  lies on the path between  $i$  and  $k$  in  $\mathcal{T}$ , then  $X_i \cap X_k \subseteq X_j$ .

The *width* of  $\langle \{X_i : i \in I\}, \mathcal{T} \rangle$  is defined as  $\max\{|X_i| : i \in I\} - 1$ , and the *treewidth* of  $G$  is the minimum  $k$  such that  $G$  admits a tree decomposition of width  $k$ .

► **Definition 3.2.** A tree decomposition  $\langle \{X_i : i \in I\}, \mathcal{T} \rangle$  is called *nice* if the following conditions are satisfied:

1. Every node of  $\mathcal{T}$  has at most two children.

2. If a node  $i$  has two children  $j$  and  $k$ , then  $X_i = X_j = X_k$  and in this case,  $X_i$  is called a JOIN NODE.
3. If a node  $i$  has one child  $j$ , then one of the following situations must hold:
  - a)  $|X_i| = |X_j| + 1$  and  $X_j \subset X_i$  and in this case,  $X_i$  is called an INTRODUCE NODE, or
  - b)  $|X_i| = |X_j| - 1$  and  $X_i \subset X_j$  and in this case,  $X_i$  is called a FORGET NODE.
4. If a node  $i$  has no child, then  $|X_i| = 1$  and in this case,  $X_i$  is called a LEAF NODE.

We first show that MCA is unlikely to be FPT with respect to parameter  $t_{\mathcal{H}}$ .

► **Theorem 3.3.** *MCA parameterized by  $t_{\mathcal{H}}$  is  $W[2]$ -hard.*

**Proof.** We reduce from the  $k$ -MULTICOLORED SET COVER problem, which is defined below.

**$k$ -Multicolored Set Cover**

**Input:** A universe  $\mathcal{U} = \{u_1, u_2, \dots, u_q\}$ , a family  $\mathcal{F} = \{S_1, S_2, \dots, S_p\}$  of subsets of  $\mathcal{U}$ , a set of colors  $\Lambda$  with a coloring function  $\text{col}' : \mathcal{F} \rightarrow \Lambda$ , an integer  $k$ .

**Output:** A subfamily  $\mathcal{S} \subseteq \mathcal{F}$  of sets whose union is  $\mathcal{U}$ , and such that (i)  $|\mathcal{S}| = k$  and (ii)  $\mathcal{S}$  is colorful, i.e.  $\text{col}'(S_i) \neq \text{col}'(S_j)$  for any  $i \neq j$  such that  $S_i, S_j \in \mathcal{S}$ .

The reduction is as follows: for any instance of  $k$ -MULTICOLORED SET COVER, we create a three-level DAG  $G = (V = V_1 \cup V_2 \cup V_3, A)$  with  $V_1 = \{r\}$ ,  $V_2 = \{v_i : i \in [p]\}$  and  $V_3 = \{z_j : j \in [q]\}$ . Informally, we associate a vertex at the second level to each set of  $\mathcal{F}$  and a vertex at the third level to each element of  $\mathcal{U}$ . We then add an arc of weight  $-1$  from  $r$  to each vertex at level 2 and an arc of weight  $p$  from  $v_i$  to  $z_j$ , for all  $i \in [p]$  and  $j \in [q]$  such that  $u_j \in S_i$ . Now, our coloring function  $\text{col}$  is as follows: we give a unique color to each vertex in  $V_1 \cup V_3$ , while at the second level (thus in  $V_2$ ), two vertices of type  $v_i$  are assigned the same color if and only if their two associated sets are assigned the same color by  $\text{col}'$ . Notice that  $\mathcal{H}$  is also a three-levels DAG with  $\text{col}(V_1)$ ,  $\text{col}(V_2)$ , and  $\text{col}(V_3)$  at the first, second and third levels, respectively. Therefore,  $(G, \mathcal{C}, \text{col}, w, r)$  is a correct instance of MCA. We now prove that there exists a colorful set  $\mathcal{S} \in \mathcal{F}$  of size  $k$  whose union is  $\mathcal{U}$  if and only if there exists a colorful arborescence  $T$  in  $G$  of weight  $w(T) = pq - k$ .

( $\Rightarrow$ ) Suppose there exists a colorful set  $\mathcal{S} \in \mathcal{F}$  of size  $k$  whose union is  $\mathcal{U}$  and let  $\text{True} = \{i \in [p] : S_i \in \mathcal{S}\}$ . Let  $V_T = \{r\} \cup \{v_i : i \in \text{True}\} \cup \{z_j : j \in [q]\}$ . Necessarily,  $G[V_T]$  is connected: first,  $r$  is connected to every level-2 vertex; second, a vertex  $z_j$  corresponds to an element  $u_j$  which is contained in some set  $S_i \in \mathcal{S}$ . Now, let  $T$  be a spanning arborescence of  $G[V_T]$ . Clearly,  $T$  is colorful and of weight  $pq - k$ .

( $\Leftarrow$ ) Suppose there exists a colorful arborescence  $T = (V_T, A_T)$  in  $G$  of weight  $w(T) = pq - k$ . Notice that any arborescence  $T'$  in  $G$  which contains  $r$  and at least one vertex from  $V_3$  must contain at least one vertex from  $V_2$  in order to be connected. Therefore, if such an arborescence  $T'$  does not contain one vertex of type  $z_j$ , then  $w(T') < pq - p - 1$  and  $w(T') < w(T)$ . Hence, if  $w(T) = pq - k$  then  $T$  necessarily contains each vertex from  $V_3$ , and thus contains exactly  $k$  vertices from  $V_2$ . Now, let  $\mathcal{S} = \{S_i : i \in [p] \text{ s.t. } v_i \in V_T\}$  and notice that  $\mathcal{S}$  is a colorful subfamily of size  $k$  whose union is  $\mathcal{U}$  as all vertices of the third level belong to  $T$ . Our reduction is thus correct.

Now, recall that  $\mathcal{H}$  is a three-levels DAG with resp.  $\text{col}(V_1)$ ,  $\text{col}(V_2)$  and  $\text{col}(V_3)$  at the first, second and third levels. Thus, there exists a trivial tree decomposition  $\langle \{X_i : i \in [|\text{col}(V_3)| + 2]\}, \mathcal{T} \rangle$  of  $U(\mathcal{H})$  which is as follows: the bag  $X_0 = \{\text{col}(r)\}$  has an arc towards the bag  $X_1 = \{\{\text{col}(r)\} \cup \text{col}(V_2)\}$  and, for any  $i \in [|\text{col}(V_3)|]$ , there exists an arc from  $X_1$  to  $X_i$  where each  $X_i$  contains  $\text{col}(V_2)$  and a different vertex of  $\text{col}(V_3)$ . Consequently, the width of  $\langle \{X_i : i \in [|\text{col}(V_3)| + 2]\}, \mathcal{T} \rangle$  is  $k$ , and hence MCA is  $W[2]$ -hard parameterized by  $t_{\mathcal{H}}$  as  $k$ -MULTICOLORED SET COVER is well-known to be  $W[2]$ -hard parameterized by  $k$ . ◀

We now use the above proof to show that MCA is unlikely to admit FPT algorithms relatively for different further parameters related to  $\mathcal{H}$ . The vertex cover number of  $U(\mathcal{H})$  is the size of a smallest subset  $S \subseteq V(\mathcal{H})$  such that at least one incident vertex of any arc of  $\mathcal{H}$  belongs to  $S$ . Notice that  $\text{col}(V_2)$  is a vertex cover of  $U(\mathcal{H})$  and thus  $U(\mathcal{H}) \leq k$ . The *feedback vertex set* number is the size of a smallest subset  $S \subseteq \mathcal{H}$  whose removal makes  $U(\mathcal{H})$  acyclic. The size of such a subset  $S$  is an interesting parameter as  $x_{\mathcal{H}} = 0$  in  $\mathcal{H}[V(\mathcal{H}) \setminus S]$  and any vertex cover of  $U(\mathcal{H})$  is also a feedback vertex set of  $U(\mathcal{H})$  – hence,  $\text{col}(V_2)$  is also a feedback vertex set of  $U(\mathcal{H})$ . Altogether, we thus obtain the following corollary.

► **Corollary 3.4.** *MCA parameterized by the vertex cover number of  $U(\mathcal{H})$  or the feedback vertex set number of  $U(\mathcal{H})$  is  $W[2]$ -hard.*

Next, recall that in the proof of Theorem 3.3 each color from the third level of  $\mathcal{H}$  is a leaf. Hence, the number of colors of outdegree at least two in  $\mathcal{H}$  is  $|\text{col}(V_1)| + |\text{col}(V_2)| = k + 1$ . Although Theorem 2.2 showed that MCA is FPT relatively to  $x_{\mathcal{H}}$ , we obtain the following.

► **Corollary 3.5.** *MCA parameterized by the number of colors of outdegree at least two in  $\mathcal{H}$  is  $W[2]$ -hard.*

By Theorem 3.3, MCA parameterized by  $t_{\mathcal{H}}$  is  $W[2]$ -hard; thus, one may look for a parameter whose combination with  $t_{\mathcal{H}}$  may lead to MCA being FPT. Here, we focus on parameter  $\ell_{\mathcal{C}} = n_G - |\mathcal{C}|$ . We know that MCA parameterized by  $\ell_{\mathcal{C}}$  is  $W[1]$ -hard, but the problem can be solved in  $\mathcal{O}^*(2^{\ell_{\mathcal{C}}})$  time when  $G$  is an arborescence [14]. Recall also that MCA is in P when  $\mathcal{H}$  is an arborescence [14], and hence when  $t_{\mathcal{H}} = 1$ . In the following, a *fully-colorful subgraph* of  $G$  is a subgraph of  $G$  that contains *exactly* one occurrence of each color  $c \in \mathcal{C}$ .

► **Lemma 3.6.** *Any graph  $G$  with  $|\mathcal{C}|$  colors has at most  $2^{\ell_{\mathcal{C}}}$  fully-colorful subgraphs.*

**Proof.** Let  $n_c$  be the number of vertices of color  $c \in \mathcal{C}$  and notice that  $\prod_{c \in \mathcal{C}} n_c$  is the number of fully-colorful subgraphs of  $G$ . Then, observe that  $n_c \leq 2^{n_c - 1}$  for all  $n_c \in \mathbb{N}$ , which implies  $\prod_{c \in \mathcal{C}} n_c \leq 2^{\sum_{c \in \mathcal{C}} n_c - 1}$  and thus  $\prod_{c \in \mathcal{C}} n_c \leq 2^{\ell_{\mathcal{C}}}$ . ◀

► **Theorem 3.7.** *MCA can be solved in  $\mathcal{O}^*(2^{\ell_{\mathcal{C}}} \cdot 4^{t_{\mathcal{H}}})$  time and  $\mathcal{O}^*(3^{t_{\mathcal{H}}})$  space.*

**Proof.** In the following, let  $(\{X_i : i \in I\}, \mathcal{T})$  be a nice tree decomposition of  $U(\mathcal{H})$ . In this proof, we provide a dynamic programming algorithm that makes use of  $(\{X_i : i \in I\}, \mathcal{T})$  in order to compute a solution to MCA in any fully-colorful subgraph  $G' \subseteq G$ , to which we remove all vertices that are not accessible from  $r$ . First, observe that  $(\{X_i : i \in I\}, \mathcal{T})$  is also a correct nice tree decomposition for the (undirected) color hierarchy graph of any subgraph of  $G$ . Second, as any colorful graph is equivalent to its color hierarchy graph, notice that  $(\{X_i : i \in I\}, \mathcal{T})$  is also a correct nice tree decomposition of any fully-colorful subgraph  $G' \subseteq G$ . Therefore, we assume without loss of generality that any bag  $X_i$  contains vertices of such graph  $G'$  instead of colors, and that  $X_0 = \{r\}$  is the root of  $(\{X_i : i \in I\}, \mathcal{T})$ .

Now, for any  $i \in I$  and for any subsets  $L_1, L_2, L_3$  that belong to  $X_i$  such that  $L_1 \oplus L_2 \oplus L_3 = X_i$ , let  $T_i[L_1, L_2, L_3]$  store the weight of a *partial solution* of MCA in  $G'$ , which is a collection of  $|L_1|$  disjoint arborescences such that:

- each  $v \in L_1$  is the root of exactly one such arborescence,
- each  $v \in L_2$  is contained in exactly one such arborescence,
- no vertex  $v \in L_3$  belongs to any of these arborescences,
- any vertex  $v \in V$  whose color is forgotten below  $X_i$  can belong to any such arborescence,

- there does not exist another collection of arborescences with a larger sum of weights under the same constraints.

Besides, let us define an entry of type  $D_i[L_1, L_2, L_3]$  which stores the same partial solution as entry  $T_i[L_1, L_2, L_3]$ , except for the vertices  $v \in V$  whose colors are forgotten below  $X_i$  which cannot belong to any arborescence of the partial solution. We now detail how to compute each entry of  $T_i[L_1, L_2, L_3]$ . We stress that each entry of  $D_i[L_1, L_2, L_3]$  is filled exactly as an entry of type  $T_i[L_1, L_2, L_3]$ , apart from the case of forget nodes which we detail below.

- *If  $X_i$  is a leaf node:*  $T_i[L_1, L_2, L_3] = 0$   
Notice that leaf nodes are base cases of the dynamic programming algorithm as  $\langle \{X_i : i \in I\}, \mathcal{T} \rangle$  is a nice tree decomposition. Moreover, recall that leaf nodes have size 1 and thus that the only partial solution for such nodes has a weight of zero.
- *If  $X_i$  is an introduce node having a child  $X_j$  and if  $v^*$  is the introduced vertex:*

$$T_i[L_1, L_2, L_3] = \begin{cases} A) \max_{\substack{S \subseteq L_2 \\ v \in S}} \{ \sum_{v \in S} w(v^*, v) + T_j[L_1 \cup S \setminus \{v^*\}, L_2 \setminus S, L_3] \} & \text{if } v^* \in L_1 \\ B) \max_{u \in (L_1 \cup L_2)} \{ w(u, v^*) + \max_{\substack{S \subseteq (L_2 \setminus \{u\}) \\ v \in S}} \{ \sum_{v \in S} w(v^*, v) + T_j[L_1 \cup S \setminus \{v^*\}, L_2 \setminus S, L_3] \} \} & \text{if } v^* \in L_2 \\ C) T_j[L_1, L_2, L_3 \setminus \{v^*\}] & \text{if } v^* \in L_3 \end{cases}$$

where we set  $w(u, v) := -\infty$  when there is no arc from  $u$  to  $v$  in  $G'$ . There are three cases:  $v^*$  is the root of an arborescence in a partial solution (case A)), an internal vertex of such a solution (case B)) or  $v^*$  does not belong to such a solution (case C)). In case A),  $S$  corresponds to the set of outneighbors of  $v^*$  in the partial solution, thus the vertices of  $S$  do not have any other inneighbor in the partial solution. Therefore, in the corresponding entry  $T_j$ , the vertices of  $S$  are roots. Now, notice that B) is very similar to A). In addition to a given set  $S$  of outneighbors,  $v^*$  being in  $L_2$  implies that  $v^*$  has an inneighbor  $u \in (L_1 \cup L_2)$  in the partial solution. Since the inneighbor  $u$  cannot be an outneighbor at the same time,  $u$  is not contained in  $S$ . Exhaustively trying all possibilities for both  $S$  and  $u$  ensures the correctness of the solution. Finally, by definition of  $L_3$ , observe that  $v^*$  does not belong to the partial solution of  $T_i[L_1, L_2, L_3]$  if  $v^* \in L_3$ .

- *If  $X_i$  is a forget node having a child  $X_j$  and if  $v^*$  is the forgotten vertex:*

$$T_i[L_1, L_2, L_3] = \max\{T_j[L_1, L_2 \cup \{v^*\}, L_3], T_j[L_1, L_2, L_3 \cup \{v^*\}]\}$$

Informally, the above formula determines whether the collection of arborescences that is stored in  $T_i[L_1, L_2, L_3]$  had a higher weight with or without  $v^*$  as an internal vertex. Observe that we do not consider the case where  $v^*$  is the root of an arborescence as such an arborescence could not be connected to the rest of the partial solution via an introduced vertex afterwards. Besides, notice that  $D_i[L_1, L_2, L_3] = D_j[L_1, L_2, L_3 \cup \{v^*\}]$  as the partial solution in  $D_i[L_1, L_2, L_3]$  does not contain any forgotten vertex by definition.

- *If  $X_i$  is a join node having two children  $X_j$  and  $X_k$ :*

$$T_i[L_1, L_2, L_3] = T_j[L_1, L_2, L_3] + T_k[L_1, L_2, L_3] - D_i[L_1, L_2, L_3]$$

Informally, the partial solution in  $T_i[L_1, L_2, L_3]$  can contain both the forgotten vertices of the partial solution in  $T_j[L_1, L_2, L_3]$  and those of the partial solution in  $T_k[L_1, L_2, L_3]$ . Recall that the partial solution in  $D_i[L_1, L_2, L_3]$  does not contain any forgotten vertices and therefore that any arc of the partial solution in  $T_i[L_1, L_2, L_3]$  is only counted once.

We fill the tables from the leaves to the root for all  $i \in I$  until  $T_0$  and any entry of type  $T_i[L_1, L_2, L_3]$  is directly computed after the entry of type  $D_i[L_1, L_2, L_3]$ . If  $T' = (V_{T'}, A_{T'})$  is a solution of MCA in a fully-colorful subgraph  $G' \subseteq G$ , then  $w(T') = T_0[\{r\}, \emptyset, \emptyset]$ . Thus, for each fully-colorful subgraph we can compute the solution by filling the tables  $T$  and  $D$ . The table has  $3^{t_{\mathcal{H}}}$  entries which implies the upper bound on the space consumption. The most expensive recurrences in terms of running time are the one of cases A) and B) for introduce nodes  $X_i$  where we consider altogether  $\mathcal{O}(4^{t_{\mathcal{H}}})$  cases: each term corresponds to a partition of  $X_i$  into four sets  $L_1$ ,  $L_2 \setminus S$ ,  $L_2 \cap S$ , and  $L_3$ . Finally, the solution of MCA in  $G$  is also the solution of at least one fully-colorful subgraph  $G' \subseteq G$ . Therefore, computing the solution of MCA for any such subgraph  $G'$  ensures the correctness of the algorithm and hence, by Lemma 3.6, adding a factor  $\mathcal{O}(2^{\ell_C})$  to the complexity of the above algorithm proves our theorem.  $\blacktriangleleft$

We now use the proof of Theorem 2.7 to show that MCA parameterized by  $t_{\mathcal{H}} + \ell_C$  is unlikely to admit a polynomial kernel. Recall that the proof shows a polynomial parameter transformation from SET COVER to MCA and notice that  $(\text{col}(V_1) \cup \text{col}(V_3))$  is a vertex cover of  $U(\mathcal{H})$  that is of size  $x_{\mathcal{H}} + 1$ . Moreover, recall that the size of a minimum vertex cover of a graph is lower-bounded by its treewidth. As a consequence, MCA does not admit any polynomial kernel for  $t_{\mathcal{H}}$  unless  $\text{NP} \subseteq \text{coNP}/\text{Poly}$  even if  $\ell_C = 0$ .

► **Corollary 3.8.** *MCA parameterized by  $t_{\mathcal{H}}$  does not admit a polynomial kernel unless  $\text{NP} \subseteq \text{coNP}/\text{Poly}$ , even when restricted to the special case where  $\ell_C = 0$ .*

## 4 Conclusion

In this paper, we obtained an  $\mathcal{O}^*(3^{x_{\mathcal{H}}})$  time algorithm for MCA, which improves upon the  $\mathcal{O}^*(3^{|C|})$  of Böcker *et al.* [4]. We also showed that MCA parameterized by  $x_{\mathcal{H}} + \ell_C$  is unlikely to admit a polynomial kernel and then that the problem admits such a kernel for the parameter  $x_{\mathcal{H}} + \ell$ . Furthermore, we proposed an FPT algorithm for MCA relatively to  $t_{\mathcal{H}} + \ell_C$  and showed that MCA is W[2]-hard relatively to  $t_{\mathcal{H}}$ . Moreover, we showed that MCA parameterized by  $\ell_C + t_{\mathcal{H}}$  does not admit a polynomial kernel. In light of these results, we ask the following question: does MCA parameterized by the larger parameter  $\ell + t_{\mathcal{H}}$  admit a polynomial kernel?

A further issue that is not addressed by our algorithm and previous algorithms is that parameterization by  $\ell$  or  $k$  essentially constrains the cardinality of the arborescences that are considered to be solutions. In other words, to make use of these parameters we need to know the number of vertices in an optimal solution in advance. Can we obtain fixed-parameter algorithms also when we do not know the number of vertices in the optimal solution?

---

## References

- 1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.
- 2 Nadja Betzler, René van Bevern, Michael R. Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 8(5):1296–1308, 2011.
- 3 Andreas Björklund, Petteri Kaski, and Lukasz Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.
- 4 Sebastian Böcker and Florian Rasche. Towards *de novo* identification of metabolites by analyzing tandem mass spectra. In *Proceedings of the 7th European Conference on Computational Biology (ECCB '08)*, volume 24(16), pages i49–i55, 2008.



- 5 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009. doi:10.1016/j.jcss.2009.04.001.
- 6 Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo. Kernel bounds for disjoint cycles and disjoint paths. *Theor. Comput. Sci.*, 412(35):4570–4578, 2011. doi:10.1016/j.tcs.2011.04.039.
- 7 Sharon Bruckner, Falk Hüffner, Richard M. Karp, Ron Shamir, and Roded Sharan. Topology-free querying of protein interaction networks. *J. Comput. Biol.*, 17(3):237–252, 2010.
- 8 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- 9 Marek Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Jakub Onufry Wojtaszczyk. Kernelization hardness of connectivity problems in  $d$ -degenerate graphs. *Discr. Appl. Math.*, 160(15):2131–2141, 2012.
- 10 Michael Dom, Daniel Lokshtanov, and Saket Saurabh. Kernelization lower bounds through colors and ids. *ACM Trans. Algorithms*, 11(2):13:1–13:20, 2014. doi:10.1145/2650261.
- 11 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. doi:10.1007/978-1-4471-5559-1.
- 12 Kai Dührkop, Marie Anne Lataretu, W. Timothy J. White, and Sebastian Böcker. Heuristic algorithms for the maximum colorful subtree problem. *arXiv*, 2018. URL: <https://arxiv.org/abs/1801.07456>.
- 13 Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Upper and lower bounds for finding connected motifs in vertex-colored graphs. *J. Comput. Syst. Sci.*, 77(4):799–811, 2011.
- 14 Guillaume Fertin, Julien Fradin, and Géraldine Jean. Algorithmic aspects of the maximum colorful arborescence problem. In T. V. Gopal, Gerhard Jäger, and Silvia Steila, editors, *Theory and Applications of Models of Computation - 14th Annual Conference, TAMC 2017, Bern, Switzerland, April 20-22, 2017, Proceedings*, volume 10185 of *Lecture Notes in Computer Science*, pages 216–230, 2017. doi:10.1007/978-3-319-55911-7\_16.
- 15 Guillaume Fertin and Christian Komusiewicz. Graph motif problems parameterized by dual. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM '16)*, volume 54 of *LIPICs*, pages 7:1–7:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- 16 Jiong Guo, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of the First International Workshop on Parameterized and Exact Computation (IWPEC '04)*, volume 3162 of *LNCS*, pages 162–173. Springer, 2004.
- 17 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. URL: <http://www.cs.berkeley.edu/~luca/cs172/karp.pdf>.
- 18 Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):360–368, 2006.
- 19 Imran Rauf, Florian Rasche, Francois Nicolas, and Sebastian Böcker. Finding maximum colorful subtrees in practice. *J. Comput. Biol.*, 20(4):311–321, 2013.





# Dualities in Tree Representations

**Rayan Chikhi**

CNRS, Université de Lille, CRIStAL, Lille, France  
rayan.chikhi@univ-lille1.fr

**Alexander Schönhuth**

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands  
alexander.schoenhuth@cwi.nl

---

## Abstract

A characterization of the tree  $T^*$  such that  $\text{BP}(T^*) = \overleftarrow{\text{DFUDS}(T)}$ , the reversal of  $\text{DFUDS}(T)$  is given. An immediate consequence is a rigorous characterization of the tree  $\hat{T}$  such that  $\text{BP}(\hat{T}) = \text{DFUDS}(T)$ . In summary, BP and DFUDS are unified within an encompassing framework, which might have the potential to imply future simplifications with regard to queries in BP and/or DFUDS. Immediate benefits displayed here are to identify so far unnoted commonalities in most recent work on the Range Minimum Query problem, and to provide improvements for the Minimum Length Interval Query problem.

**2012 ACM Subject Classification** Mathematics of computing → Trees

**Keywords and phrases** Data Structures, Succinct Tree Representation, Balanced Parenthesis Representation, Isomorphisms

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.18

**Related Version** A full version with proofs is available at [4], <https://arxiv.org/abs/1804.04263>.

**Acknowledgements** The authors are grateful to H el ene Touzet for helpful discussions, and to CPM reviewers for insightful comments, providing Figure 2 and the reference to Davoodi *et al* [5].

## 1 Motivation

Given an array  $A[1, n]$  with elements from a totally ordered set, the Range Minimum Query (RMQ) problem is to provide a data structure that on input positions  $1 \leq i \leq j \leq n$  returns

$$\text{rmq}_A(i, j) := \min\{A[k] \mid i \leq k \leq j\}. \quad (1)$$

In [9], Fischer and Heun presented the first data structure that uses  $2n + o(n)$  bits and answers queries in  $O(1)$  time (in fact, without accessing  $A$ ). They first construct a tree  $T[A]$  (the 2D-Min-Heap of  $A$ ). Then they observe that in a certain parenthesis representation of  $T[A]$  (DFUDS), the following query leads to success for computing  $\text{rmq}_A(i, j)$  (where 0 and 1 refer to closing and opening parentheses in  $\text{DFUDS}(T[A])$ , respectively):

$$w_1 \leftarrow \text{rmq}_D(\text{select}_0(i+1), \text{select}_0(j)) \quad (2)$$

$$\text{if } \text{rank}_0(\text{open}(w_1)) = i \text{ then return } i \quad (3)$$

$$\text{else return } \text{rank}_0(w_1) \quad (4)$$

where  $\text{rmq}_D$  refers to performing a range minimum query on the array  $D[x] := \text{rank}_1(x) - \text{rank}_0(x)$  where  $x$  indexes parentheses in  $\text{DFUDS}(T[A])$ , and 1 and 0 represent opening and



  Rayan Chikhi and Alexander Sch onhuth;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 18; pp. 18:1–18:12

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum f ur Informatik, Dagstuhl Publishing, Germany

closing parentheses, respectively.  $\text{open}(w_1)$  returns the position of the opening parenthesis matching the one closing at position  $w_1$ . Note that  $D[x] - D[x - 1] \in \{-1, +1\}$  for all  $x \in \{2, \dots, 2N\}$ , which turns  $\text{rmq}_D$  into an easier problem ( $\pm 1$ -RMQ), as was shown in [1].

Most recently, Ferrada and Navarro suggested an alternative approach which leads to a shorter, hence faster query procedure [8]. They construct a tree  $\widehat{T[A]}$  that results from a systematic while non-trivial transformation of the edges of  $T[A]$  (the number of non-root nodes  $N$  remains the same). They observed that in  $\text{BP}(\widehat{T[A]})$  the following simpler query computes  $\text{rmq}_A(i, j)$ :

$$w_2 \leftarrow \text{rmq}_D(\text{select}_0(i), \text{select}_0(j)) \quad (5)$$

$$\text{return rank}_0(w_2) \quad (6)$$

The *major motivation of our treatment* is the observation – which passes unnoted in both [8, 9] – that

$$\text{DFUDS}(T[A]) = \text{BP}(\widehat{T[A]}) \quad (7)$$

So, the shorter query raised by Ferrada and Gonzalez would have worked for Fischer and Heun as well. It further raises the question whether there are principles by which to transform trees  $T$  into trees  $\widehat{T}$  such that

$$\text{DFUDS}(T) = \text{BP}(\widehat{T}) \quad (8)$$

and, if so, what these principles look like. Here, we thoroughly investigate related questions so as to obtain conclusive insight. We will show that the respective trees and their possible representations can be juxtaposed in terms of a *new duality for tree representations*. In doing so, we will obtain a proof for (7) as an easy corollary (to consolidate our findings, we also give a direct proof that [8]’s query also would have worked for [9] in the Appendix of the full version [4]). In summary, our treatment puts BP and DFUDS into a unifying context.

## 1.1 Related Work

**RMQ’s.** The RMQ problem has originally been anchored in the study of Cartesian trees [21], because it is related to computing the least common ancestor (LCA) of two nodes in a Cartesian tree derived from  $A$  [10], further complemented by the realization that any LCA computation can be cast as an  $\pm 1$ -RMQ problem [3] for which subsequently further improvements were raised [15, 19]. Fischer and Heun finally established the first structure that requires  $2n + o(n)$  space and  $O(1)$  time (without accessing  $A$ ) [9], establishing an anchor point for many related topics (e.g. [16, 17]), which justified to strive for further improvements [8, 11].

**Isomorphisms.** For their latest (and likely conclusive) improvements, [8] made use of an isomorphism between binary and general ordinary trees, presented in [15], and successfully experiment with certain variations on the ground theme of this isomorphism, to finally obtain the above-mentioned  $\widehat{T[A]}$ . Here, we provide an explicit treatment of these trees, which [8] are implicitly making use of. From this point of view, we provide a rigorous re-interpretation of the treatments [8, 9] and the links drawn with [15] therein. Finally, note that [5] further expands on [15].

**BP and DFUDS.** The BP representation was first presented in [13] and developed further in many ways (e.g. [15]). Since neither the BP nor the LOUDS [6, 13] representations allow for a few basic operations relating to children and subtrees, the DFUDS representation was presented as an improvement in this regard [2, 14]. A tree-unifying approach different to ours was proposed by Farzan *et al* [7]. [5] observes relationships between BP and DFUDS and proves them via the (above-mentioned) isomorphism by [15]. Since our treatment avoids binary trees altogether, it establishes a more direct approach to identifying dualities between ordinal trees than [5].

## 1.2 Notation

**Trees.** Throughout, we consider rooted, ordered trees  $T = (V, E)$  (with nodes  $V = V[T]$  and (directed) edges  $E = E[T]$ ) with root  $r$ . For the sake of notational convenience (following standard abuse of tree notation), we will write  $v \in T$  instead of  $v \in V[T]$  and  $T_1 \subset T_2$  for  $V[T_1] \subset V[T_2]$ ; note that induced subgraphs do not play a relevant role in this treatment. By definition of ordered trees, *siblings*, that is nodes sharing their parent node are ordered, implying the notions of left, right, immediate right, immediate left siblings. By  $rmc_T(v)$ , we denote the rightmost child of a node  $v$  in  $T$  if it exists (if  $T$  is understood, we write  $rmc(v)$ ). Similarly, we denote by  $ils_T(v)$  (or  $ils(v)$  if  $T$  is understood) the immediate left sibling of  $v$  in  $T$  if it exists. For two siblings,  $u < v$  means that  $u$  is left of  $v$ . As usual, the partial order on siblings can be extended to a full order, ordering all  $v \in T$ , by depth-first-traversal (or breadth-first-traversal) logic, for example; here, by default, we write  $u <_T v$  (or  $u < v$  if  $T$  is understood) if  $u$  comes before  $v$  in the depth-first traversal of  $T$ . We write  $u = \text{pa}(v)$  indicating that  $u$  is the parent of  $v$ , that is  $(u, v)$  is a directed edge in  $T$ .

**Parenthesis Based Tree Representations.** In the following, we will deal with parenthesis based representations for trees, which are vectors of opening parentheses '(' and closing parentheses ')'. The number of opening parentheses will match the number of closing parentheses, thereby for a tree  $T$ , each node  $v \in T$  will be represented by a pair of opening and closing parentheses, for which we write  $\text{OP}(v)$  and  $\text{CP}(v)$ , respectively.

The *Balanced Parenthesis (BP)* representation  $\text{BP}(T)$  (e.g. [13, 15]) is built by traversing  $T$  in depth-first order, writing an opening parenthesis when reaching a node for the first time, and writing a closing parenthesis when reaching a node for the second time. By depth-first order logic, this yields a balanced representation, meaning that the number of opening matches the number of closing parentheses (see Figure 1). By default, a node is identified with its opening parenthesis  $\text{OP}(v)$ .

The *Depth-First Unary Degree Sequence (DFUDS)* representation  $\text{DFUDS}(T)$  [2] is again obtained by traversing  $T$  in depth-first order, but, when reaching a node with  $d$  children for the first time, writing  $d$  opening parentheses and one closing parenthesis (and writing no parentheses when reaching it for the second time). This sequence of parentheses becomes balanced when appending an opening parenthesis at the beginning. It is further convenient to identify a node with the parenthesis preceding the block of opening parentheses that represent its children<sup>1</sup>, which for all non-root nodes is a closing parenthesis. In other words, in DFUDS, the  $i$ -th closing parenthesis reflects the  $i$ -th non-root node in DFT order. Note that,

<sup>1</sup> Literature references are ambiguous about the exact choice of parenthesis. None of the alternative choices, like the first opening parenthesis or the closing parenthesis following the block of opening parentheses, would lead to any real complications also in our treatment.

according to this definition, when matching opening parentheses with closing parentheses in a balanced manner, the opening parentheses in one block refer to the children of the closing parenthesis preceding the block from right to left.

**Rank/Select/Open/Close.** In the following, we will treat parenthesis vectors as bitvectors, where opening and closing parentheses are identified with 1 and 0. Let  $B \in \{0, 1\}^n$  be a bitvector and  $x \in \{1, \dots, n\}$  (for enhanced exposition, running indices run from 1 to  $n$ ). Then  $\text{rank}_{B,0}(x), \text{rank}_{B,1}(x)$  are defined to be the number of 0's or 1' in  $B$  up to (and including)  $B[x]$ . Further,  $\text{select}_{B,0}(i), \text{select}_{B,1}(i)$  are defined to be the position of the  $i$ -th 0 or 1 in  $B$  (if this exists). We omit the subscript  $B$  and write  $\text{rank}_0(x), \text{rank}_1(x), \text{select}_0(i), \text{select}_1(i)$  if the choice of  $B$  is evident. As a relevant example (see (5)), for  $\text{DFUDS}(T)$  and  $v \in T$ , we have  $\text{CP}(v) = \text{select}_0(i)$  if and only if  $\text{DFT}(v) = i + 1$ , that is  $v$  is the  $i + 1$ -th node in depth-first traversal order, also counting the root. We further write  $\text{open}(x)$  and  $\text{close}(x)$  to identify the matching partner in a (balanced parenthesis) bitvector, that is  $\text{open}(x)$  for a position  $x$  in  $B$  with  $B[x] = 0$  is the position of the 1 matching  $x$  and vice versa for  $\text{close}(x)$ .

### 1.3 Outline of Sections

We will start with the definition of a dual tree  $T^*$  of  $T$  in section 2; according to this definition,  $T^*$  is a directed graph, so we still have to prove that  $T^*$  is a tree, which we will do immediately afterwards. We proceed by proving  $(T^*)^* = T$ , arguably necessary for a well-defined duality. In section 2.1, we then show how to decompose our duality into subdualities by introducing the definition of a reversed tree  $\overleftarrow{T}$ . We conclude by providing the definition of  $\hat{T}$  as the reversed dual tree; without being able to provide a proof at this point, note that  $\hat{T}$  will turn out to be the tree from (8).

In section 3, we provide the definition of a primal-dual ancestor, which is crucial for re-interpreting RMQ's in terms of the notions of duality provided here. Upon having proven the unique existence of the primal-dual ancestor in theorem 13, we re-interpret RMQ's, and beyond that not only re-interpret, but also improve on running minimal length interval queries (MLIQ's) both in terms of space requirements and query counts.

We will finally prove our main theorem in section 4.

► **Theorem 1.** Let  $T$  be a tree and let the reversal  $\overleftarrow{B}$  of a bitvector  $B$  be defined by  $\overleftarrow{B}[x] := 1 - B[n - x + 1], \forall x \in \{1, \dots, n\}$ . Then

$$\text{BP}(T) = \overleftarrow{\text{DFUDS}(T^*)}. \quad (9)$$

Returning to [8], we will finally demonstrate that (7), our motivating insight, indeed holds.

## 2 Tree Duality: Definition

► **Definition 2** (Dual tree). Let  $T$  be a tree. The dual tree  $T^*$  of  $T$  is a directed graph that has the same vertices as  $T$ . Edges and order (among nodes sharing a parent) are given by the following rules, where we write  $\text{pa}^*(v)$  for the parent of  $v$  in  $T^*$ :

- **Rule 1a:** The root  $r$  of  $T$  is also the root of  $T^*$ , that is  $r$  has no parent also in  $T^*$ .
- **Rule 1b:** If  $v = \text{rmc}_T(r)$  then also  $v = \text{rmc}_{T^*}(r)$ , implying in particular that  $\text{pa}^*(v) = r$ .
- **Rule 2:** If  $v = \text{rmc}_T(u)$  with  $u \neq r$ , then  $v = \text{ils}_{T^*}(u)$ , implying that  $\text{pa}^*(v) = \text{pa}^*(u)$ .
- **Rule 3:** If  $v = \text{ils}_T(u)$ , then  $v = \text{rmc}_{T^*}(u)$ , implying that  $\text{pa}^*(v) = u$ .

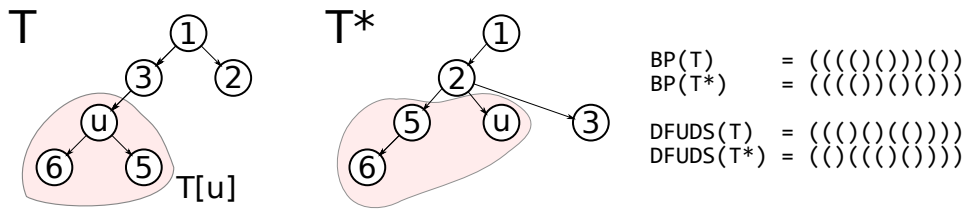


Figure 1 A tree and its dual, along with the BP and DFUDS representations. A subtree  $T[u]$  is also highlighted, along with the corresponding nodes in the dual.

► Remark. Rules 1a, 1b, 2 and 3 immediately imply that  $T^*$  is a directed graph where each node other than  $r$  has one parent. Note that the existence of a parent due to Rule 2 is guaranteed by induction on the depth of a node in  $T$ , where Rule 1b makes the start.

► Remark. It is similarly immediate to observe that there is a well-defined order among nodes that share a parent. It suffices to notice that in  $T^*$  each node either is a rightmost child (Rules 1b, 3), or it is the (unique) immediate left sibling of another node (Rule 2).

All nodes but  $r$  have exactly one (incoming) edge, which implies  $|E| = |V| - 1$ . To conclude that  $T^*$  is a tree, it remains to show that  $T^*$  contains no cycles, which we immediately do:

► **Theorem 3.**  $T^*$  is a well-defined, rooted, ordered tree.

We do this by explicitly specifying the parents of nodes in  $T^*$ , by making use of the depth-first traversal order  $<$  in  $T$ . For this, let  $T[v]$  be the subtree of  $T$  that hangs off (and includes)  $v \in T$ , i.e.  $T[v]$  contains  $v$  and all its descendants in  $T$ . Let further

$$R[v] := \{u \in T \setminus T[v] \mid v < u\}$$

be all nodes “right of”  $v$  according to depth-first traversal order. For two nodes  $u, v$  where  $u$  is an ancestor of  $v$ , we immediately note that

$$T[v] \subset T[u], \quad R[u] \subset R[v] \quad \text{and} \quad R[v] \subset T[u] \cup R[u] \tag{10}$$

For a node  $v \in T \setminus \{r\}$ , we then obtain the following lemma:

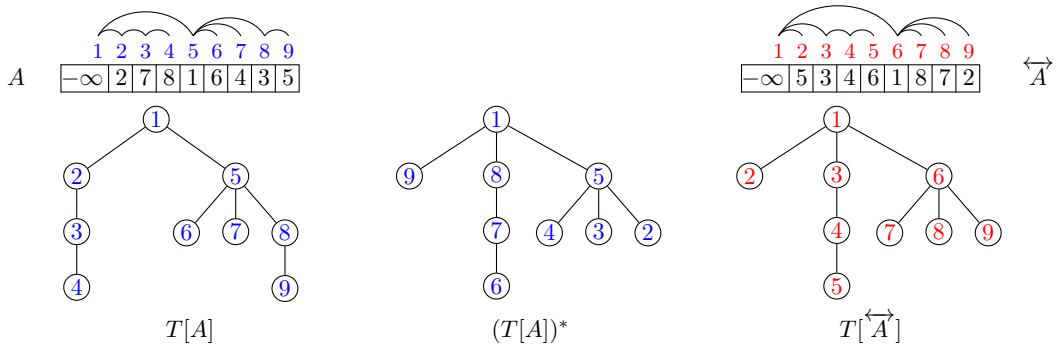
► **Lemma 4.**

$$\text{pa}^*(v) = \begin{cases} \min R[v] & R[v] \neq \emptyset \\ r & R[v] = \emptyset \end{cases}$$

We refer to the Appendix of the full version [4] for the proof of Lemma 4. Using Lemma 4, a proof of theorem 3 can be immediately given:

**Proof of Theorem 3.** Lemma 4 implies that  $v <_T \text{pa}^*(v)$  for all  $v \in T \setminus \{r\}$ . Therefore,  $T^*$  can contain no cycles and we obtain that  $T^*$  is a tree as a corollary. Furthermore, lemma 4 reveals that  $T^*$  is unique. ◀

See again the Appendix of [4] for immediate corollaries which point out how parents and subtrees in  $T^*$  relate with one another.



■ **Figure 2** (left) An array  $A$  along with the 2D-Min-Heap  $T[A]$ . Arcs above array indices indicate tree paths. (middle) The dual tree  $(T[A])^*$ . (right) The reversed array  $\overleftarrow{A}$  along with the 2D-Min-Heap  $T[\overleftarrow{A}]$ .

► **Remark.** An intuitive guideline for describing  $T^*$  in comparison to  $T$  is that parent- and siblinghood, as well as left and right are exchanged. In other words (and as will become clearer explicitly later) the duality describing  $T^*$  can be decomposed into two subdualities, one of which turns parents into siblings and vice versa, and the other one of which exchanges left and right.

This remark had left us with some choices for characterizing tree duality. Our choice is motivated by [9], arguably a cornerstone in RMQ theory development. To understand this, let  $A = A[1, n]$  be the array, on which RMQ's are to be run, and let  $\overleftarrow{A}$  be its reversal, given by  $\overleftarrow{A}[i] = A[n - i + 1]$ . Let  $T[A]$  be the 2D-Min-Heap constructed from  $A$ , as described in [9] (a definition is provided in the Appendix of [4],

to which RMQ's refer (see (2),(3),(4)). An immediate question to ask is what RMQ's would look like when performing RMQ's on  $\overleftarrow{A}$  instead of  $A$ . Here is the answer.

► **Theorem 5.** Let  $A[1, N]$  be an array and let  $\overleftarrow{A} := [A[N], \dots, A[1]]$  its reversal. Then

$$(T[A])^* = T[\overleftarrow{A}] \tag{11}$$

An illustration of the Theorem is provided in Figure 2. See the Appendix of [4] for a more detailed treatment of this motivating example, including proofs. Thanks to theorem 5, the definition of  $T^*$  can arguably be considered a most natural choice, at least when relating tree duality with RMQ's.

Before proceeding with results on succinct tree representations, we provide the following intuitive lemma about the depth-first traversal order of  $T^*$  as a rooted, ordered tree. This lemma, in combination with lemma 4, supports the (intended) intuition that in  $T^*$  up and down, as well as left and right, are exchanged, properties that are characteristic for rooted, ordered tree duality. It also provides motivation beyond theorem 5 in the Introduction why  $T^*$  is the possibly canonical choice of the dual of a tree.

Therefore, let  $<^*$  denote the depth-first traversal order in  $T^*$  (well-defined by theorem 3) while  $<$  denotes the depth-first traversal order in (the primal tree)  $T$ .

► **Lemma 6.** Let  $u, v \in T \setminus \{r\}$ . Then

$$u <^* v \text{ if and only if } v < u$$

The proof of lemma 6 makes use of the following technical lemmata 7 and 8, which are of use also elsewhere. We therefore state these technical lemmata here. The proofs for all lemmata 6, 7 and 8 can finally be found in the Appendix of [4].

► **Lemma 7.** *Let  $w := \text{pa}^*(v)$  and  $v_2 \in T[v] \setminus v$  such that  $\text{pa}^*(v_2) = w$ . Then  $v_2 <^* v$ .*

► **Lemma 8.** *Let  $v_1$  be a sibling left of  $u_1$  in  $T$ . Then  $T[v_1] \subset T^*[u_1]$ .*

With lemma 6 proven, we can conclude with proving a main theorem of this treatment. It states that the dual of the dual is the primal tree, arguably a key property for a sensibly defined duality. Despite all lemmata raised so far, the proof still entails a few technically more demanding arguments.

► **Theorem 9.**  $(T^*)^* = T$

**Proof.** It suffices to show that  $\text{pa}^{**}(v) = \text{pa}(v)$ , since lemma 6 establishes that the order in  $(T^*)^*$  agrees with that of  $T$ . Let  $u = \text{pa}(v)$ . In the Appendix of [4], we provide a (heavily technical) proof that

$$u = \begin{cases} \min_{<^*} R^*[v] & R^*[v] \neq \emptyset \\ r & R^*[v] = \emptyset \end{cases}$$

which completes the proof by applying lemma 4. ◀

## 2.1 Tree Reversal

We bring in another, simpler notion of tree duality, namely that of reversing trees. We will further elucidate what the trees are like when combining tree reversal with the tree duality  $(T^*)$  raised earlier.

► **Definition 10** (Reversed tree). Let  $T$  be a tree. The reversed tree  $\overleftarrow{T}$  of  $T$  is the tree resulting from reversing the order among the children of each node.

► **Proposition 11.** *Let  $\overleftarrow{T}$  be the reversed tree of  $T$  and  $\overleftarrow{T}^*$  be the reversed dual of  $T$ . We define *irs* (immediate right sibling) and *lmc* (left-most child) similarly as in Section 1.2.*

(a) *The root  $r$  of  $T$  is also the root of  $\overleftarrow{T}$ .*

(b) *Let  $u = \text{pa}_T(v)$ . Then also  $u = \text{pa}_{\overleftarrow{T}}(v)$ .*

(c) *Let  $u = \text{ils}_T(v)$ . Then  $u = \text{irs}_{\overleftarrow{T}}(v)$ .*

(d) *The root  $r$  of  $T$  is also the root of  $\overleftarrow{T}^*$ .*

(e) *If  $v = \text{lmc}_T(r)$  then also  $v = \text{lmc}_{\overleftarrow{T}^*}(r)$ , implying in particular that  $\text{pa}_{\overleftarrow{T}^*}(v) = r$ .*

(f) *If  $v = \text{lmc}_T(u)$  with  $u \neq r$ , so  $v = \text{ils}_{\overleftarrow{T}^*}(u)$ , implying that  $\text{pa}_{\overleftarrow{T}^*}(v) = \text{pa}_{\overleftarrow{T}^*}(u)$ .*

(g) *If  $v = \text{ils}_T(u)$ , then  $v = \text{lmc}_{\overleftarrow{T}^*}(u)$ , implying that  $\text{pa}_{\overleftarrow{T}^*}(v) = u$ .*

(h)  $\overleftarrow{T}^* = \overleftarrow{\overleftarrow{T}^*}$ , *that is the reversed dual tree of  $T$  is the dual of the reversed tree of  $T$ .*

All of those are, in comparison with statements referring to the definition of the dual tree, rather obvious observations. See the Appendix of [4] for the proof.

Since  $\overleftarrow{T}^*$  plays a particular role in the context of our introductory motivation, we give it a particular name:  $\hat{T}$ .

► **Definition 12** (Reversed dual tree). Let  $T$  be a tree. The tree  $\hat{T} := \overleftarrow{\overleftarrow{T}^*}$  of  $T$  is the dual of the reversed (or the reversed dual) tree of  $T$ .

Based on proposition 11, we realize that  $\hat{T}$  can be described as turning leftmost children into immediate left siblings.

► **Remark.** Following the arguments provided in [8], it becomes evident that the tree  $T$  in use there, on which  $\text{BP}(T)$  is constructed, turns indeed out to be  $\widehat{T[A]} = \overleftarrow{T[A]}^*$ .

### 3 The Primal-Dual Ancestor

The following theorem points out that pairs of nodes have a unique *primal-dual ancestor*. We will further point out properties of that node.

► **Theorem 13.** *Let  $v_1, v_2 \in T \setminus \{r\}$  be two nodes where  $v_1 \leq v_2$ . Then there is a unique node  $v \in T \setminus \{r\}$  such that  $v_1 \in T^*[v]$  and  $v_2 \in T[v]$ .*

We henceforth refer to this unique node as *primal-dual ancestor* of  $v_1$  and  $v_2$ , written  $\text{pda}(v_1, v_2)$ .

**Proof.** Let

$$v := \max_{<_T} \{v_1 \leq x \leq v_2 \mid v_1 \in T^*[x]\} \quad (12)$$

be, relative to depth-first traversal order in  $T$ , the largest ancestor of  $v_1$  in  $T^*$  that precedes  $v_2$ . We claim that  $v$  is the unique primal-dual ancestor of  $v_1$  and  $v_2$ .

By definition, we immediately obtain that  $v_1 \in T^*[v]$ . To prove  $v_2 \in T[v]$ , consider  $\text{pa}^*(v)$ , for which, by choice of  $v$ , we have that  $v_2 < \text{pa}^*(v)$ . By lemma 4, however,  $\text{pa}^*(v)$  is the first node in  $R[v]$ , relative to depth-first traversal order in  $T$ . Hence, for any  $y$  such that  $v \leq y < \text{pa}^*(v)$ , which includes  $v_2$ , it holds that  $y \in T[v]$ .

It remains to show that  $v$  is the only possible primal-dual ancestor. By definition of the primal-dual ancestor,  $v$  must be an ancestor of  $v_1$  in  $T^*$ .

*First*, consider an ancestor  $y$  of  $v_1$  in  $T^*$  such that  $y < v$ . By choice of  $v$ , it holds that  $\text{pa}^*(y) \leq v_2$ , while  $\text{pa}^*(y) \in R[y]$ . This implies that also  $v_2 \in R[y]$ , and not  $v_2 \in T[y]$ , hence  $y$  cannot be a primal-dual ancestor of  $v_1$  and  $v_2$ .

*Second*, consider an ancestor  $y$  of  $v_1$  in  $T^*$  such that  $v < y$ . Because  $v$  is an ancestor of  $v_1$  in  $T^*$ , and  $y$  is larger than  $v$ ,  $y$  is also an ancestor of  $v$  in  $T^*$ . By lemma 4, we know that  $y \in R[v]$ . This, in combination with  $v_2 \in T[v]$  implies that  $v_2 < y$ , hence,  $y$  cannot be an ancestor of  $v_2$  in  $T$ . ◀

For the following theorem, let

$$\text{depth}_T(v_1, v_2) := \min\{\text{depth}_T(y) \mid v_1 \leq y \leq v_2\}$$

be the minimal depth of nodes between (and including)  $v_1$  and  $v_2$ .

► **Theorem 14.** *Let  $v_1, v_2 \in T \setminus \{r\}$  such that  $v_1 < v_2$ . It holds that*

$$\text{pda}(v_1, v_2) = \max_{<} \{v_1 \leq x \leq v_2 \mid \text{depth}_T(x) = \text{depth}_T(v_1, v_2)\} \quad (13)$$

*That is, according to depth-first traversal order in  $T$ , the primal-dual ancestor is the greatest node whose  $T$ -depth is minimal among all nodes between (and including)  $v_1$  and  $v_2$ .*

The proof is based on the following lemma:

► **Lemma 15.** *Let  $v < w$  such that  $w \in T[\text{pa}^*(v)]$ . Then it holds that*

$$\text{depth}_T(v, w) = \text{depth}_T(\text{pa}^*(v)) \quad (14)$$



See the Appendix of [4] for a proof of lemma 15 and then theorem 14.

Note immediately that theorem 14 implies that  $v$  can be found in  $O(1)$  runtime, by performing a range minimum query on the excess array  $D$  of  $\text{BP}(T)$ , defined by  $D[x] := \text{rank}_1(x) - \text{rank}_0(x)$  where  $\text{rank}$  refers to  $\text{BP}(T)$ . Since  $D[x+1] - D[x] \in \{-1, +1\}$ , an RMQ on  $D$  means performing a  $\pm 1$ -RMQ, for which convenient solutions exist [1].

**Re-interpretation of RMQ's.** Because it was shown [9], that the node in the 2D-Min-Heap  $T[A]$  that corresponds to the solution of  $\text{rmq}_A(i, j)$  is given by the right hand side of (13), theorems 13 and 14 allow for a reinterpretation of an RMQ query  $\text{rmq}_A(i, j)$  on an array  $A$  (without going into details here, because the proof is an easy exercise based on collecting facts from here, [9] and [8]).

1. Determine the node  $v$  in  $T[A]$  corresponding to  $i$ .
2. Determine the node  $w$  in  $T[A]$  corresponding to  $j$ .
3. Determine  $\text{pda}(v, w)$  in  $T[A]$ ; return the corresponding index  $i_o$ .

**Re-interpretation and improvement of Minimal Length Interval Queries (MLIQ).** To illustrate the potential practical benefits of our treatment, we further revisit the problem of *minimal length interval queries (MLIQ)*. The improvements we will be outlining are similar in spirit to the ones delivered in [8]. However, based on our results, they are considerably more convenient to obtain.

► **Problem 16 (MLIQ).** Let  $([a_i, b_i])_{i \in \{1, \dots, n\}}$ ,  $a_i, b_i \in \mathbb{N}$  such that  $a_i \leq b_i$  for all  $i \in \{1, \dots, n\}$  and  $a_i < a_j$  and  $b_i < b_j$  for  $i < j$ .

■ **Input:**  $(a, b)$  such that  $a < b$

■ **Output:** The index  $i_0$  such that  $[a_{i_0}, b_{i_0}]$  is the shortest interval that contains  $[a, b]$ , if such an interval exists.

This problem makes part of other relevant problems, for example the *shortest unique interval* problem. In this context, a solution for the MLIQ problem was presented in [12] that requires  $O(b_n \log b_n)$  space to answer the query in  $O(1)$  time. Therefore, the following strategy was suggested.

Let  $l_i := |b_i - a_i + 1|$  be the length of the  $i$ -th interval,  $A := [l_1, \dots, l_n]$  and  $T[A]$  the corresponding 2D-Min-Heap.

1.  $i_{min} := \min\{i \mid b_i > b\}$ ,  $i_{max} := \max\{i \mid a_i < a\}$ ; if  $i_{max} < i_{min}$  **output** 'None'.
2. Determine nodes  $v, w \in T[A]$  corresponding to  $i_{min}, i_{max}$ .
3. Determine  $\text{pda}(v, w) \in T[A]$ ; **output** its index.

The solution presented in [12] can immediately be improved by employing bitmaps for the first step (which, according to [18], requires  $O(n \log(b_n/n)) + o(b_n)$  space). Steps 2 and 3 then reflect an ordinary RMQ, which can be dealt with following [8]. In terms of query counts, Step 1 reflects two  $\text{rank}$  queries, while the resulting RMQ, following [8], requires two  $\text{select}$ 's, one  $\pm 1$ - $\text{rmq}$ , and one  $\text{rank}$ .

If  $|a_i - a_{i-1}|, |b_i - b_{i-1}|$  are in  $O(\log n)$  (which applies for several important applications), further improvements can be made based on suggestions made in [20] for BP representations of trees with weighted parentheses. For that, we construct  $T_a = T[A]$  and  $T_b = \overleftarrow{T[A]}$ . We then assign weights  $w_{a,i} := |a_i - a_{i-1}|$  to  $i+1$ -st opening parentheses in  $T_a$ , whereas in  $T_b$  we assign  $w_{b,i} := |b_i - b_{i-1}|$  to the  $i$ -th closing parentheses (where  $a_0 = b_0 = 0$ ; we recall that the number of non-root nodes in  $T[A]$  is  $n$ ). When aiming at running queries presented in [20],

## 18:10 Tree Representation Duality

this requires  $2n \log \log n + o(n)$  bits of space, an improvement over  $O(n \log(b_n/n)) + o(b_n)$  for the above, naive approach. Following [20], let  $\text{bpselect}_{w_a,0}(a)$ ,  $\text{bpselect}_{0,w_b}(b)$  be defined by selecting the largest index in the balanced parenthesis vector such that adding up all weights attached to opening parentheses ( $w_a$ ) is at most  $a$ , or adding up all weights attached to closing parentheses ( $w_b$ ) is at most  $b$ . We can then run

1.  $w := \text{bpselect}_{w_a,0}(a)$  in  $T_a$  and  $v := 2n - \text{bpselect}_{0,w_b}(b) + 3$  in  $T_b$ ; if  $v > w$  **output** 'None'
2. Determine  $\text{pda}(v, w) \in T_a$ ; **output** its index.

In comparison to the naive approach from above, this makes two **bpselect** queries, instead of two **rank**'s and two **select**'s. The *decisive trick* is to place  $a$  and  $b$  directly into  $T[A]$ , which avoids determining indices  $i_{min}, i_{max}$  first, which subsequently need to be placed. Beyond the improvements in terms of space and query counts, we argue that this solution reflects all symmetries inherent to the MLIQ problem in a particularly compact manner.

### 4 Relating BP and DFUDS representations

We will use the following construction to set up a tree induction for proving our main theorem.

► **Definition 17** (Tree joining operation). Let  $T_1$  and  $T_2$  be two trees, let  $r_2$  be the root of  $T_2$ ,  $\text{rmc}_{T_2}(r_2)$  needs to exist and be a leaf. The notation  $T_1 \curvearrowright T_2$  will denote a new tree formed by taking  $T_2$  and inserting the children of the root of  $T_1$  as children of the rightmost child of the root of the new tree. Extend this operation to  $n$  trees  $T_1, \dots, T_n$  where  $T_2, \dots, T_n$  all satisfy the same property as  $T_2$  above, in the following way:  $T_1 \curvearrowright T_2 \curvearrowright T_3 = (T_1 \curvearrowright T_2) \curvearrowright T_3$  and so on,

$$T_1 \curvearrowright T_2 \dots \curvearrowright T_n = (((\dots((T_1 \curvearrowright T_2) \curvearrowright T_3) \curvearrowright \dots) \curvearrowright T_n).$$

► **Observation 18.** Let  $T$  be a tree such that its root  $r$  has a single child  $c$  (that may or may not be a leaf). Then in  $T^*$ , by Rule 1b,  $\text{rmc}_{T^*}(r) = c$  and is a leaf.

The following Lemma (proven in the Appendix of [4])

relates the dual tree to the tree joining operation. We will use the  $r \rightarrow T$  notation to denote a new tree formed by adding a new root  $r$  as a parent of the root of  $T$ .

► **Lemma 19.** Let  $T$  be a tree consisting of a root  $r$  and  $n \geq 1$  subtrees  $A_1, A_2, \dots, A_n$  as children. When  $n = 1$ ,  $T^*$  is  $(r \rightarrow A_1)^*$ . When  $n \geq 2$ ,  $T^*$  is  $(r \rightarrow A_1)^* \curvearrowright (r \rightarrow A_2)^* \curvearrowright \dots \curvearrowright (r \rightarrow A_n)^*$ .

We are now ready to prove Theorem 1. Parentheses in BP and DFUDS representations will be denoted by  $\underline{\quad}$  and  $\overline{\quad}$  to avoid confusion with usual mathematical parentheses. Recall that we use  $\overleftarrow{s}$  to mirror a string  $s$  of parentheses, e.g.  $\overleftarrow{()} = ()$  and  $\overleftarrow{()()} = ()()$ .

**Proof of Theorem 1.** Let  $T$  be a tree with  $n$  subtrees  $A_1, \dots, A_n$ . It is clear that  $\text{BP}(T) = (\text{BP}(A_1)\text{BP}(A_2)\dots\text{BP}(A_n))$ . Observe that for two trees  $T_1$  and  $T_2$  with roots  $v_1$  and  $v_2$ , and where  $\text{rmc}_{T_1}(v_1), \text{rmc}_{T_2}(v_2)$  both exist and are leaves,

$$\text{DFUDS}(T_1 \curvearrowright T_2) = \underline{(\text{DFUDS}(T_2 \setminus \text{rmc}_{T_2}(v_2))\text{DFUDS}(T_1 \setminus \text{rmc}_{T_1}(v_1)))}.$$

In fact, one can show recursively that such a decomposition can be extended to  $T_1 \curvearrowright \dots \curvearrowright T_n$ . We will now prove the theorem with a tree structural induction. Observe that for a tree  $T$  of depth 1 (a single root node),

$$\text{BP}(T) = \underline{()} = \text{DFUDS}(T^*) = \overleftarrow{\overline{\text{DFUDS}(T^*)}}.$$

Now, assume the theorem equality is true for trees of depth  $i$  and we will show it for trees of depth  $i + 1$ . A tree  $T$  of depth  $i + 1$  can be decomposed into a root node  $r$  and  $n$  subtrees  $A_1, \dots, A_n$  that are all of of depth  $\leq i$  with roots  $a_1, \dots, a_n$ . Using Lemma 19,

$$\text{DFUDS}(T^*) = \text{DFUDS}((r \rightarrow A_1)^* \curvearrowright (r \rightarrow A_2)^* \curvearrowright \dots \curvearrowright (r \rightarrow A_n)^*).$$

By the recursive decomposition that we observed above, and using Observation 18 stating that the rightmost child of  $r$  in  $(r \rightarrow A_i)^*$  is a leaf,

$$\text{DFUDS}(T^*) = \underline{(\text{DFUDS}((r \rightarrow A_n)^* \setminus \{a_n\}) \dots \text{DFUDS}((r \rightarrow A_1)^* \setminus \{a_1\}))}.$$

Observe that we can take each DFUDS term in the expression above and wrap it around parentheses, i.e.  $\underline{(\text{DFUDS}((r \rightarrow A_i)^* \setminus \{a_i\}))}$  which is equal to  $\text{DFUDS}((r \rightarrow A_i)^*)$ . Furthermore, note the following identity:  $\text{DFUDS}((r \rightarrow A_i)^*) = \underline{(\text{DFUDS}(A_i^*))}$ . And by inductive hypothesis,  $\text{DFUDS}(A_i^*) = \overleftarrow{\text{BP}(A_i)}$ , thus  $\text{DFUDS}((r \rightarrow A_i)^* \setminus \{a_i\}) = \overleftarrow{\text{BP}(A_i)}$ . Hence,

$$\overleftarrow{\text{DFUDS}(T^*)} = \underline{(\text{BP}(A_1) \dots \text{BP}(A_n))} = \text{BP}(T). \quad \blacktriangleleft$$

**Proving (7) from the Introduction.** Eventually, we also realize that  $\text{BP}(\overleftarrow{T}) = \overleftarrow{\text{BP}(T)}$  and also  $\text{DFUDS}(\overleftarrow{T}) = \overleftarrow{\text{DFUDS}(T)}$ , both of which is straightforward  $[\star]$ . Using this in combination with theorems 9 and 1, we obtain

$$\text{DFUDS}(T[A]) \stackrel{[\star]}{=} \overleftarrow{\text{DFUDS}(\overleftarrow{T}[A])} \stackrel{Th.9}{=} \overleftarrow{\text{DFUDS}((\overleftarrow{T}[A])^*)} \stackrel{Th.1}{=} \text{BP}(\overleftarrow{T}[A]^*) \stackrel{D.12}{=} \text{BP}(\widehat{T[A]})$$

which establishes equation (7) from the introduction.

**Conclusive Remarks.** In summary, we have provided a framework that unifies BP and DFUDS. From a certain point of view, we have pointed out that neither should BP based approaches have advantages over DFUDS based approaches, nor vice versa. As an exemplary perspective of our framework, BP based treatments such as [17, 20] might have an easier grasp of the advantages that DFUDS based approaches bring along. Finally, we consider it interesting future work to also characterize trees that put BP and/or DFUDS based representations into context with LOUDS based representations.

---

## References

- 1 M. Bender and G. Farach-Colton. The LCA problem revisited. *Proc. 4th LATIN*, LNCS 1776:88–94, 2000.
- 2 D. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, and S.S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
- 3 O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22(2):221–242, 1993.
- 4 R. Chikhi and A. Schönhuth. Dualities in Tree Representations. *ArXiv e-prints*, 2018. arXiv:1804.04263.
- 5 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Mathematics in Computer Science*, 11(2):177–189, 2017. doi:10.1007/s11786-017-0294-4.
- 6 O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In *Proc. of the WEA*, LNCS 4007, pages 134–145, 2006.

- 7 Arash Farzan, Rajeev Raman, and S. Srinivasa Rao. Universal succinct representations of trees? In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikolettseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2009. doi:10.1007/978-3-642-02927-1\_38.
- 8 H. Ferrada and G. Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.
- 9 J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
- 10 H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th STOC*, pages 135–143, 1984.
- 11 R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proceedings of the 12th SEA*, volume LNCS 7933, pages 5–17, 2013.
- 12 X. Hu, J. Pei, and Y. Tao. Shortest unique queries on strings. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 161–172, 2014.
- 13 G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the FOCS*, pages 549–554, 1989.
- 14 J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. of the SODA*, pages 575–584, 2007.
- 15 J.I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal of Computing*, 31(3):762–776, 2001.
- 16 G. Navarro, Y. Nekrich, and L.M.S. Russo. Space-efficient data analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.
- 17 G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3), 2014. Article 16.
- 18 R. Raman, V. Raman, and S.R. Sattie. Succinct indexable dictionaries with applications to encoding  $k$ -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.
- 19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
- 20 D. Tsur. Succinct representation of labeled trees. Technical Report 1312.6039, ArXiv, 2015.
- 21 J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 4:229–239, 1980.

# Longest Lyndon Substring After Edit

**Yuki Urabe**

Department of Electrical Engineering and Computer Science, Kyushu University, Japan  
yuki.urabe@inf.kyushu-u.ac.jp

**Yuto Nakashima**


Department of Informatics, Kyushu University, Japan  
yuto.nakashima@inf.kyushu-u.ac.jp

**Shunsuke Inenaga**

Department of Informatics, Kyushu University, Japan  
inenaga@inf.kyushu-u.ac.jp

**Hideo Bannai**

Department of Informatics, Kyushu University, Japan  
bannai@inf.kyushu-u.ac.jp

 <https://orcid.org/0000-0002-6856-5185>

**Masayuki Takeda**

Department of Informatics, Kyushu University, Japan  
takeda@inf.kyushu-u.ac.jp

---

## Abstract

The longest Lyndon substring of a string  $T$  is the longest substring of  $T$  which is a Lyndon word.  $LLS(T)$  denotes the length of the longest Lyndon substring of a string  $T$ . In this paper, we consider computing  $LLS(T')$  where  $T'$  is an edited string formed from  $T$ . After  $O(n)$  time and space preprocessing, our algorithm returns  $LLS(T')$  in  $O(\log n)$  time for any single character edit. We also consider a version of the problem with block edits, i.e., a substring of  $T$  is replaced by a given string of length  $l$ . After  $O(n)$  time and space preprocessing, our algorithm returns  $LLS(T')$  in  $O(l \log \sigma + \log n)$  time for any block edit where  $\sigma$  is the number of distinct characters in  $T$ . We can modify our algorithm so as to output all the longest Lyndon substrings of  $T'$  for both problems.

**2012 ACM Subject Classification** Mathematics of computing → Combinatorial algorithms

**Keywords and phrases** Lyndon word, Lyndon factorization, Lyndon tree, Edit operation

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.19

**Funding** This work was supported by JSPS KAKENHI Grant Numbers JP17H06923 (YN), JP17H01697 (SI), JP16H02783 (HB), and JP25240003 (MT).

## 1 Introduction

A string  $w$  is said to be a *Lyndon word* if  $w$  is lexicographically smaller than any of its non-empty proper suffixes. An equivalent definition of Lyndon words is a string  $w$  which is lexicographically smaller than any of its cyclic rotations. For instance, **aab** is a Lyndon word, but its cyclic rotations **aba** and **baa** are not. Lyndon words have many important combinatorial properties in stringology, and have various applications in, e.g., musicology [7], bioinformatics [12], approximation algorithms [21], string matching [9, 6, 23], combinatorics on words [3, 15, 24], and free Lie algebras [20].



© Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 19; pp. 19:1–19:10

Leibniz International Proceedings in Informatics



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

In stringology, Lyndon words are closely related to repetitive structures. A string  $w$  is said to be primitive if there do not exist an integer  $k$  and a string  $x$  such that  $w = x^k$ . For any primitive string  $w$ ,  $ww$  contains one or two Lyndon words of length  $|w|$ . Recently, Bannai et al. showed that the maximum number of *maximal repetitions* in a string of length  $n$ , is less than  $n$  [3]. A key idea of their proof relied on the notion of the longest Lyndon word that starts at each position of the string. There are several recent studies on Lyndon trees and Lyndon arrays [14, 10, 22], which are closely related to longest Lyndon word because they represent all the longest Lyndon words in a given string. Although these structures take linear space and can be computed in linear time for an integer alphabet, they are not easy to maintain when allowing dynamic edit operations, since the structures may change a lot, even for a single character edit operation.

Although fully dynamic data structures are difficult in general, Amir et al. considered a new type of problem concerning the *Longest Common Factor* problem [1]. The goal there was to compute, given strings  $S$  and  $T$ , the longest common factor of strings  $S$  and  $T'$  where  $T'$  is a string which is obtained by a single character edit operation on  $T$ . Their algorithm uses  $O(n \log^4 n)$  expected time and  $O(n \log^3 n)$  space for preprocessing, and then for any single character edit query, the LCF can be answered in  $O(\log^3 n)$  time. The important and interesting aspect of this problem setting is that all edit queries are on the original string  $T$ , and the edited string is not maintained for subsequent edit queries.

In this paper, we consider the problem of computing the longest Lyndon substring after a single (character or block) edit operation. Let  $LLS(T)$  be the length of the longest Lyndon substring of a string  $T$  of length  $n$ . We first consider the problem of computing  $LLS(T')$  for any single character edit (substitution, insertion, deletion) where  $T'$  is the string obtained after the edit operation on  $T$ . We then extend the problem that asks for  $LLS(T')$  for any single *block* edit, where  $T'$  is the string obtained by replacing a substring of  $T$  with a given string of length  $l$  specified in the edit query. For single character edit operations, our algorithm runs in  $O(\log n)$  time for each edit query after  $O(n)$  time and space preprocessing. For block edit operations, our algorithm runs in  $O(l \log \sigma + \log n)$  time for each edit query after  $O(n)$  time and space preprocessing, where  $\sigma$  is the number of distinct characters in  $T$ . We can modify our algorithm so as to output all the longest Lyndon substrings of  $T'$  for both problems.

The rest of this paper is organized as follows. In Section 2, we state some definitions and properties on strings. In Section 3, we propose our algorithm for a version of the problem with single character edits. In Section 4, we show our algorithm for a version of the problem with single block edits. Finally, we conclude in Section 5.

## 2 Preliminaries

### 2.1 Strings and model of computation

Let  $\Sigma$  be an ordered finite *alphabet*. An element of  $\Sigma^*$  is called a *string*. The length of a string  $w$  is denoted by  $|w|$ . The empty string  $\varepsilon$  is a string of length 0. Let  $\Sigma^+$  be the set of non-empty strings, i.e.,  $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ . For a string  $w = xyz$ ,  $x$ ,  $y$  and  $z$  are called a *prefix*, *substring*, and *suffix* of  $w$ , respectively. A prefix  $x$ , a substring  $y$ , and a suffix  $z$  of  $w$  are called a *proper prefix*, a *proper substring*, and a *proper suffix* of  $w$  if  $x \neq w$ ,  $y \neq w$ , and  $z \neq w$ , respectively. The  $i$ -th character of a string  $w$  is denoted by  $w[i]$ , where  $1 \leq i \leq |w|$ . For a string  $w$  and two integers  $1 \leq i \leq j \leq |w|$ , let  $w[i..j]$  denote the substring of  $w$  that begins at position  $i$  and ends at position  $j$ . For convenience, let  $w[i..j] = \varepsilon$  when  $i > j$ . For any string  $w$  let  $w^1 = w$ , and for any integer  $k \geq 2$  let  $w^k = ww^{k-1}$ , i.e.,  $w^k$  is a  $k$ -times repetition of  $w$ .

If character  $a$  is lexicographically smaller than another character  $b$ , then we write  $a \prec b$ . For any strings  $x, y$ , let  $\text{lcp}(x, y)$  be the length of the longest common prefix of  $x$  and  $y$ . We write  $x \prec y$  iff either  $x[\text{lcp}(x, y) + 1] \prec y[\text{lcp}(x, y) + 1]$  or  $x$  is a proper prefix of  $y$ .

Our model of computation is the word RAM. We assume the computer word size is at least  $\lceil \log_2 |w| \rceil$ , and hence, standard operations on values representing lengths and positions of string  $w$  can be manipulated in  $O(1)$  time. Space complexities will be determined by the number of computer words (not bits).

## 2.2 Lyndon words and Lyndon factorization of strings

A string  $w$  is said to be a *Lyndon word*, if  $w$  is lexicographically strictly smaller than all of its non-empty proper suffixes. The *longest Lyndon substring* of a string  $w$  is the longest substring of  $w$  which is a Lyndon word.  $LLS(w)$  denotes the length of the longest Lyndon substring of a string  $w$ .

The *Lyndon factorization* of a string  $w$ , denoted  $LF_w$ , is the factorization  $\ell_1^{p_1}, \dots, \ell_m^{p_m}$  of  $w$ , such that each  $\ell_i \in \Sigma^+$  is a Lyndon word,  $p_i \geq 1$ , and  $\ell_i \succ \ell_{i+1}$  for all  $1 \leq i < m$ . The size of  $LF_w$ , denoted by  $|LF_w|$ , is  $m$ .  $LF_w$  can be represented by the sequence  $(|\ell_1|, p_1), \dots, (|\ell_m|, p_m)$  of integer pairs, where each pair  $(|\ell_i|, p_i)$  represents the  $i$ -th Lyndon factor  $\ell_i^{p_i}$  of  $w$ . Note that this representation requires  $O(m)$  space.

In the literature, the Lyndon factorization is sometimes defined to be a sequence of lexicographically non-increasing Lyndon words, namely, each Lyndon factor  $\ell^p$  is decomposed into a sequence of  $p$   $\ell$ 's. In this paper, each Lyndon word  $\ell$  in the Lyndon factor  $\ell^p$  is called a *decomposed Lyndon factor*. We also refer to the factorization by decomposed Lyndon factors as the *decomposed Lyndon factorization*.

► **Lemma 1** ([13]). *For any string  $w$ , we can compute  $LF_w$  in  $O(|w|)$  time.*

For any string  $w$ , let  $LF_w = \ell_1^{p_1}, \dots, \ell_m^{p_m}$ . Let  $\text{lfb}_w(i)$  denote the position where the  $i$ -th Lyndon factor begins in  $w$ , i.e.,  $\text{lfb}_w(1) = 1$  and  $\text{lfb}_w(i) = \text{lfb}_w(i-1) + |\ell_{i-1}^{p_{i-1}}|$  for any  $2 \leq i \leq m$ . For any  $1 \leq i \leq m$ , let  $\text{lfs}_w(i) = \ell_i^{p_i} \ell_{i+1}^{p_{i+1}} \dots \ell_m^{p_m}$  and  $\text{lfp}_w(i) = \ell_1^{p_1} \ell_2^{p_2} \dots \ell_i^{p_i}$ . For convenience, let  $\text{lfs}_w(m+1) = \text{lfp}_w(0) = \varepsilon$ .

## 2.3 Lyndon tree

Given a Lyndon word  $w$  of length  $|w| > 1$ ,  $(u, v)$  is the *standard factorization* [8, 19] of  $w$ , if  $w = uv$  and  $v$  is the longest proper suffix of  $w$  that is a Lyndon word, or equivalently, the lexicographically smallest proper suffix of  $w$ . It is well known that for the standard factorization  $(u, v)$  of any Lyndon word  $w$ , the factors  $u$  and  $v$  are also Lyndon words (e.g.[4]). The *Lyndon tree* of  $w$  is the full binary tree defined by recursive standard factorization of  $w$ ;  $w$  is the root of the Lyndon tree of  $w$ , its left child is the root of the Lyndon tree of  $u$ , and its right child is the root of the Lyndon tree of  $v$ . The longest Lyndon word that starts at each position can be obtained from the Lyndon tree, due to the following lemma.

► **Lemma 2** (Lemma 5.4 of [3]). *Let  $w$  be a Lyndon word with respect to  $\prec$ .  $w[i..j]$  corresponds to a right node (or possibly the root) of the Lyndon tree with respect to  $\prec$  if and only if  $w[i..j]$  is the longest Lyndon word with respect to  $\prec$  that starts from  $i$ .*



## 2.4 Longest Common Extension

For any string  $w$ , the *longest common extension query* is, given two positions  $1 \leq i, j \leq |w|$ , to answer

$$LCE_w(i, j) = \max\{k \mid w[i..i+k-1] = w[j..j+k-1], i+k-1, j+k-1 \leq |w|\}.$$

By using suffix tree [26] of  $w$  and the *Lowest Common Ancestor query* (also called *Nearest Common Ancestor*) [16] on the suffix tree, we can compute any LCE query in constant time after  $O(|w|)$  time and space preprocessing.

### 3 Longest Lyndon substring after 1-edit

In this paper, we consider three edit operations, i.e., substitution, insertion and deletion. Let  $T'$  be the string which was edited at a given position from a string  $T$  of length  $n$ . A *1-edit longest Lyndon substring query* (1-edit LLS) asks us to return  $LLS(T')$ .

Firstly, we explain a basic property of  $LLS(T)$  and give a naïve solution. The following lemma can be obtained by the definition of Lyndon factorization.

► **Lemma 3.** *For any string  $T$ ,  $LLS(T)$  is the length of the longest decomposed Lyndon factor of  $LF_T$ .*

**Proof.** Let  $x$  be the longest Lyndon substring of  $T$ . Suppose that  $x$  is not a decomposed Lyndon factor of  $LF_T$ . If  $x$  is a proper substring of a decomposed Lyndon factor  $y$ , then  $y$  is a Lyndon substring which is longer than  $x$ . This implies that  $x$  contains a boundary of consecutive decomposed factors. Let  $x = stu$  where  $s$  is a suffix of some decomposed Lyndon factor and  $u$  is a prefix of some Lyndon decomposed factor ( $s, u \in \Sigma^+, t \in \Sigma^*$ ). By the definition of Lyndon factorization,  $s \succeq u$  holds. Thus,  $x$  is not a Lyndon word. ◀

This fact can be obtained by Observation 3 of [14] in a different context. Due to this lemma, computing  $LF_T$  leads to the longest Lyndon substring of  $T$ . Since we can compute  $LF_{T'}$  in  $O(n)$  time by using Duval's algorithm [13], we can compute  $LLS(T')$  in  $O(n)$  time for each query.

► **Example 4** (1-edit LLS). Let  $T = \text{acbabcabcabac}$ . Since  $LF_T = \text{acb}, (\text{abc})^2, \text{abac}$ , the longest Lyndon substring of  $T$  is **abac**. (*substitution*) If the second **c** is replaced by **b**, then the longest Lyndon substring of  $T'$  is **abbabc** since  $LF_{T'} = \text{acb}, \text{abbabc}, \text{abac}$ . (*insertion*) If **a** is inserted at the position preceded by the last **b**, then the longest Lyndon substrings of  $T'$  are **acb, abc, aac** since  $LF_{T'} = \text{acb}, (\text{abc})^2, \text{ab}, \text{aac}$ . (*deletion*) If the second **a** from the last is deleted, then the longest Lyndon substring of  $T'$  is **abcabcabc** since  $LF_{T'} = \text{acb}, \text{abcabcabc}$ .

Our goal of this paper is the following.

► **Theorem 5.** *After constructing an  $O(n)$ -space data structure of a given string  $T$  in  $O(n)$  time, we can compute  $LLS(T')$  in  $O(\log n)$  time for each 1-edit query.*

In this section, we explain our algorithm for substitution operations. We can solve our problem for the other two types of operations in a similar way.

More formally, for substitutions, let  $T' = T[1..e-1] \cdot \alpha \cdot T[e+1..n] = T_p \cdot \alpha \cdot T_s$  where  $\alpha \in \Sigma$ . In our algorithm, we compute  $LF_{T'}$  by concatenating  $LF_{T_p}$ ,  $LF_\alpha$ , and  $LF_{T_s}$ . I et al. [18] showed an efficient algorithm to compute  $LF_{uv}$  from  $LF_u$  and  $LF_v$  for any string  $u, v$  (we will explain in Section 3.1). Hence, we can use this concatenation algorithm to compute  $LF_{T'}$ . The rest of this section is organized as follows. In Section 3.2, we show how to compute  $LF_{T_p}$ . In Section 3.3, we explain how to characterize  $LF_{T_s}$ . Finally, we summarize our algorithm in Section 3.4.



### 3.1 Overview of computing Lyndon factorization by concatenation

Here, we explain an overview of a Lyndon factorization algorithm which was proposed by I et al. [18]. This algorithm computes the Lyndon factorization of the concatenation of two strings by using their Lyndon factorizations.

For any string  $u$  and  $v$ , let  $LF_u = u_1^{p_1}, \dots, u_m^{p_m}$  and  $LF_v = v_1^{q_1}, \dots, v_{m'}^{q_{m'}}$ . Then,  $LF_{uv}$  is characterized as follows.

► **Lemma 6** ([2, 11]).  $LF_{uv} = u_1^{p_1} \dots u_c^{p_c} z^k v_{c'}^{q_{c'}} \dots v_{m'}^{q_{m'}}$  for some  $0 \leq c \leq m$ ,  $1 \leq c' \leq m' + 1$  and  $LF_{lfs_u(c+1)lfp_v(c'-1)} = z^k$ .

This lemma implies that there is at most one new Lyndon factor  $z^k$  (each of the other Lyndon factors of  $LF_{uv}$  is also a Lyndon factor of  $LF_u$  or  $LF_v$ ). By a simple observation, we can consider three cases as follows.

- If  $u_m \succ v_1$ , then  $LF_{uv} = LF_u, LF_v (z = \epsilon)$ .
  - If  $u_m = v_1$ , then  $LF_{uv} = u_1^{p_1}, \dots, u_{m-1}^{p_{m-1}}, u_m^{p_m+q_1}, v_2^{q_2}, \dots, v_{m'}^{q_{m'}}$  ( $z = u_m = v_1$ ).
  - If  $u_m \prec v_1$ , there exists *the medial decomposed factor*  $z$  which begins in  $u$  and ends in  $v$ .
- In the first two cases, we can compute  $LF_{uv}$  by one lexicographic string comparisons. In the third case, computing the medial decomposed Lyndon factor  $z$  leads to computing  $LF_{uv}$ .

► **Lemma 7** (Lemma 16 of [18]). *Assume that  $LF_u$  and  $LF_v$  have been computed. Then, we can compute the medial decomposed Lyndon factor  $z$  by  $O(\log |LF_u| + \log |LF_v|)$  lexicographic string comparisons.*

A key point of that result is that the medial decomposed Lyndon factor  $z$  satisfies the following properties.

- The beginning position of  $z$  is equal to  $lfb_u(i)$  such that  $lfs_u(1)v \succ \dots \succ lfs_u(i)v \prec \dots \prec lfs_u(m+1)v$ .
- The ending position of  $z$  is equal to  $lfb_v(j) - 1$  such that  $lfs_v(1) \succ lfs_v(j-1) \succ lfs_u(i)v \succ lfs_v(j) \succ \dots \succ lfs_v(m'+1)$ .

From these monotonous conditions of suffixes which begin at the beginning position of some Lyndon factor, we can compute the beginning position and the ending position of  $z$ , respectively, by a binary search. After computing  $z$ , we can compute the Lyndon factor  $z^k$  by checking whether  $u_{i-1} (v_j)$  is equal to  $z$  or not, respectively (i.e.,  $u_{i-1}$  and  $v_j$  may be equal to  $z$ ).

► **Example 8.** Let  $LF_u = \text{abb}, (\text{ab})^2, \text{a}$  and  $LF_v = \text{bc}, \text{b}, \text{abababc}, \text{ab}, (\text{a})^2$ . Then, the medial decomposed Lyndon factor is  $z = \text{abababc}$  obtained by Lyndon factors  $(\text{ab})^2, \text{a}, \text{bc}, \text{b}$ . Since the decomposed Lyndon factor succeeding to  $z$  is also  $\text{abababc}$ , we need to pack them. Thus,  $LF_{uv} = \text{abb}, (\text{abababc})^2, \text{ab}, (\text{a})^2$ .

In addition, we can modify the second property for the decomposed Lyndon factorization of  $v$  by using the following property.

► **Lemma 9.** *Let  $z = lfs_u(i)lfp_v(j-1)$  be the medial decomposed factor of  $LF_{uv}$ . Then,  $lfs_v(j-1) \succ v_{j-1}^{q_{j-1}-1} lfs_v(j) \succ \dots \succ v_{j-1} lfs_v(j) \succ lfs_u(i)v \succ lfs_v(j)$  also holds.*

### 3.2 Computing the Lyndon factorization of $T_p$

The following lemma is a well-known property of Lyndon words and Lyndon factorizations.

$\ell =$	a	b	a	b	b	a	b	a	b	b	a	b	b
$ x $	1	2	2	2	5	5	5	5	5	5	5	5	13
$k$	1	1	1	2	1	1	1	1	1	2	2	2	1
$ x' $	0	0	1	0	0	1	2	3	4	0	1	2	0

■ **Figure 1** By Lemma 10, any prefix  $w$  of a Lyndon word  $\ell$  can be represented as  $w = x^k x'$ . For instance,  $\text{ababbababba} = (\text{ababb})^2 \text{a}$ . Thus, we store  $(|x|, k, |x'|) = (5, 2, 1)$  for this prefix of  $\ell$ .

► **Lemma 10.** *For any string  $w$  which is a prefix of some Lyndon word, there exists a unique Lyndon word  $x$  s.t.  $w = x^k x'$  where  $x'$  is a proper prefix of  $x$  and an integer  $k \geq 1$ . Moreover,  $LF_w = x^k, LF_{x'}$ .*

► **Lemma 11.** *We can compute  $LF_{T_p}$  for any  $1 \leq e \leq n$  in  $O(\log n)$  time after  $O(n)$  time and space preprocessing.*

**Proof.** Let  $LF_T = \ell_1^{p_1}, \dots, \ell_m^{p_m}$ . Assume that  $lfb_T(i) + |\ell_i^{k-1}| \leq e < lfb_T(i) + |\ell_i^k|$ , i.e., the edited position  $e$  is in the  $k$ -th decomposed Lyndon factor of the  $i$ -th Lyndon factor. Then,  $T_p = \ell_1^{p_1} \dots \ell_i^{k-1} \ell'_i$  where  $\ell'_i$  is a (possibly empty) proper prefix of  $\ell_i$ . It is easy to see that the Lyndon factorization of  $\ell_1^{p_1} \dots \ell_i^{k-1}$  is  $\ell_1^{p_1}, \dots, \ell_{i-1}^{p_{i-1}}, \ell_i^{k-1}$ . On the other hand, from Lemma 10,  $LF_{\ell'_i} = x^j, x'$  for some integer  $j \geq 1$  and some Lyndon word  $x$ . Since  $x'$  is also a prefix of Lyndon word  $x$ , we can consider  $LF_{x'}$  in the same way. Because  $x'$  must be shorter than half of  $\ell'_i$ , it follows that  $LF_{\ell'_i}$  consists of at most  $\log |\ell'_i|$  Lyndon factors. It is easy to see that  $LF_{T_p} = \ell_1^{p_1}, \dots, \ell_i^{k-1}, LF_{\ell'_i}$ .

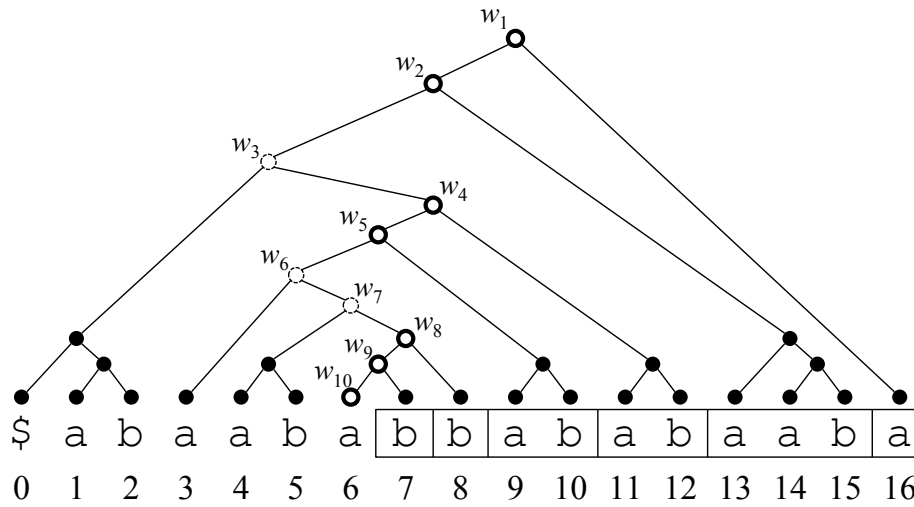
Based on this observation, we show our data structure for computing  $LF_{T_p}$ . We can compute  $LF_T$  in  $O(n)$  time and store it in  $O(|LF_T|)$  space. Let  $\ell$  be a decomposed Lyndon factor of  $T$ . For each prefix of  $\ell$ , we store a triple  $(|x|, k, |x'|)$  based on Lemma 10. An example is shown in Figure 1. We note that all the triples for  $T$  can be computed in  $O(n)$  time by using Duval's Lyndon factorization algorithm [13] (we can compute them together with the Lyndon factorization of  $T$ ).

Now we explain how to compute  $LF_{T_p}$  by using above data structures. The first  $(i-1)$  Lyndon factors of  $LF_{T_p}$  are in  $LF_T$ . The  $i$ -th Lyndon factor of  $LF_{T_p}$  is  $\ell_i^{k-1}$  (changed only the exponent of  $\ell_i$ ). Finally, we have to compute  $LF_{\ell'_i}$ . The form of  $LF_{\ell'_i} = u^j, u'$  is stored as the  $|\ell'_i|$ -th triple of  $\ell_i$ . If  $u' = \varepsilon$  (i.e., the third entry of the triple is 0), then  $u^j$  is a Lyndon factor of  $LF_{T_p}$ . Otherwise, since  $u'$  is a prefix of  $\ell_i$ , the form of  $LF_{u'}$  is stored as the  $|u'|$ -th triple of  $\ell_i$ . By repeating this recursively at most  $\log |\ell'_i|$  times, we can obtain  $LF_{\ell'_i}$ . Therefore, we can compute  $LF_{T_p}$  in  $O(\log n)$  time. ◀

### 3.3 The Lyndon factorization of $T_s$ by Lyndon tree

In the previous subsection, we have computed the Lyndon factorization of a prefix of some Lyndon word, since the number of Lyndon factors of  $T_p$  which are not in  $LF_T$  is bounded by  $\log n$ . We also want to compute  $LF_{T_s}$ , but the size of the factorization can be large. Hence, we cannot compute  $LF_{T_s}$  explicitly for each query in order to achieve an  $O(\log n)$  time bound. To overcome this problem, we use the *Lyndon tree* of  $T$ , which can represent the Lyndon factorization of each suffix of  $T$ .

Let  $LF_T = \ell_1^{p_1}, \dots, \ell_m^{p_m}$ . Assume that  $lfb_T(i) + |\ell_i^{k-1}| \leq e < lfb_T(i) + |\ell_i^k|$ , i.e., the edited position  $e$  is in the  $k$ -th decomposed factor of the  $i$ -th Lyndon factor. Then,  $T_s =$



**Figure 2** This figure shows the Lyndon tree of  $\$abaababbababaaba$  and the path  $P_6 = w_1, \dots, w_{10}$ .  $P'_6$  is the sequence of internal nodes on  $P_6$  which are drawn by circles, namely,  $P'_6 = w_1, w_2, w_4, w_5, w_8, w_9$ . For this example, Lemma 12 shows that  $Rstr(w_9), Rstr(w_8), Rstr(w_5), Rstr(w_4), Rstr(w_2), Rstr(w_1) = b, b, ab, ab, aab, a$  is the decomposed Lyndon factorization of  $T[7..16]$ .

$\ell''_i \ell_i^{p_i-k} \dots \ell_m^{p_m}$  where  $\ell''_i$  is a (possibly empty) proper suffix of  $\ell_i$ . For convenience, we introduce a special character  $\$$  which is lexicographically smaller than any other characters. It is easy to see that the string  $\$T$  is a Lyndon word for any  $T$ . We consider  $LTree(\$T)$ . Let  $P_j = w_1, \dots, w_h$  be the path from the root to the leaf which corresponds to  $T[j]$  in  $LTree(\$T)$  ( $w_1$  is the root and  $w_h$  is the leaf), and  $P'_j = w'_1, \dots, w'_{h'}$  be the sequence of internal nodes on path  $P$  such that the right child of any node on  $P'_j$  is not on  $P_j$ . For any node  $w$ ,  $Rstr(w)$  denotes the string which is represented by the rightchild of  $w$  (see also Figure 2). The following lemma shows that the Lyndon tree of  $\$T$  represents the Lyndon factorization of any of its suffixes.

► **Lemma 12.**  $Rstr(w'_{h'}), \dots, Rstr(w'_1)$  is the decomposed Lyndon factorization of  $LF_{T[j+1..n]}$ .

**Proof.** For any  $1 \leq i < h'$ ,  $Rstr(w'_{i+1})$  is a suffix of the string which is represented by the leftchild of  $w'_i$ . Since  $Rstr(w'_{i+1})$  is the longest Lyndon word which begins at that position by Lemma 2, then  $Rstr(w'_{i+1}) \succeq Rstr(w'_i)$ . It is clear that  $Rstr(w'_{h'}) \dots Rstr(w'_1) = T[j+1..n]$ . Thus  $Rstr(w'_{h'}), \dots, Rstr(w'_1)$  is the decomposed Lyndon factorization of  $T[j+1..n]$ . ◀

It is known that the Lyndon tree of a string can be computed in linear time [17, 3]. We can compute  $LTree(\$T)$  in  $O(n)$  time and space. In addition, for our algorithm, we process the Lyndon tree so as to be able to answer *Level Ancestor Query* (shortly LAQ).

► **Lemma 13** (Level Ancestor Query [5]). *We can pre-process a given rooted tree in linear time and space so that the  $i$ -th node in the path from any node to the root can be found in  $O(1)$  time for any  $i \geq 0$ , if such exists.*

For any node  $w$ , we also compute  $na(w)$  which is the nearest ancestor of  $w$  that has  $w$  in the left subtree. This preprocessing can also be done in  $O(n)$  time and space.

### 3.4 Computing the longest Lyndon substring

In the rest of this section, we summarize our method.

Firstly, we compute  $LF_{T_p}$  based on Lemma 11 in  $O(\log n)$  time. From  $LF_{T_p}$  and  $\alpha$ , we compute  $LF_{T_p \cdot \alpha}$  by  $O(\log |LF_{T_p}|)$  lexicographic string comparisons by using Lemma 6 and 7. After that, we compute  $LF_{T'}$  from  $LF_{T_p \cdot \alpha}$  and  $LF_{T_s}$ . Let  $z$  be the medial decomposed Lyndon factor in this step. Since we know  $LF_{T_p \cdot \alpha}$  and  $T_s$ , we can compute the beginning position of  $z$  by  $O(\log |LF_{T_p \cdot \alpha}|)$  lexicographic string comparisons on  $T'$ . Then we compute the ending position of  $z$ , by using Lemmas 7 and 9.

In order to compute the ending position, we access the necessary suffixes by considering the path  $P_e$ , defined in Section 3.3, in the  $LTree(\$T)$ . The key idea is that we can conduct a binary search on  $P_e$ , and obtain  $z$  by  $O(\log h)$  lexicographic string comparisons on  $T'$ . For any range of depths on  $P_e$ , we can choose the middle node  $w$  in the range in constant time using Lemma 13. If the rightchild of  $w$  is on  $P_e$ , we choose  $na(w)$  as  $w$ . We then compare the suffix of  $T_s$  which begins at the beginning position of  $Rstr(w)$  and the suffix of  $T'$  which begins at the beginning position of  $z$ , and recurse on the upper or lower half of the range depending on the result of the comparison.

Thus we can get  $LF_{T'}$  by  $O(\log n)$  string comparisons in total. The number of Lyndon factors of  $T'$  such that we should have explicitly is  $O(\log n)$  (new  $\log n$  factors in  $T_p$  and a new factor by concatenations). It is easy to see that we can compare lexicographic order between any substrings of  $T'$  by constant number of LCE queries on  $T$ . Thus, we can compute  $LF_{T'}$  in  $O(\log n)$  time.

We have three candidates as the longest Lyndon substrings.

- Unchanged Lyndon factors at prefix.
- $O(\log n)$  new Lyndon factors.
- Unchanged Lyndon factors at suffix.

Since we store  $O(\log n)$  new Lyndon factors explicitly, we can get the longest Lyndon factor in this part in  $O(\log n)$  time. To get the longest decomposed Lyndon factor in the first candidate, we precompute the rightmost longest Lyndon factor for each prefix of  $T$  which is a concatenation of Lyndon factors (i.e., for each  $\ell_1^{p_1}, \dots, \ell_i^{p_i}$ ). This can be computed in  $O(n)$  time and space. By using this information, we can see the length of longest Lyndon factor in the first part in constant time. For suffixes of  $T$ , we precompute the same data structure as prefixes. Therefore, we obtain Theorem 5.

It is easy to see that we can return all the longest Lyndon substrings in unchanged part at prefix and at suffix in linear time w.r.t. the number of such factors. Then, we can get all the longest Lyndon substrings in  $T'$ .

► **Corollary 14.** *After constructing an  $O(n)$ -space data structure of a given string  $T$  in  $O(n)$  time, we can compute all the longest Lyndon substrings of  $T'$  in  $O(\log n + occ)$  time for each 1-edit query where  $occ$  is the number of outputs.*

## 4 Longest Lyndon substring after block edit

Here, we consider more general problem called *1-block-edit longest Lyndon substring query* (1-block-edit LLS). Namely, a substring of  $T$  is replaced by a given string of length  $l$ .

► **Example 15** (1-block-edit LLS). Let  $T = \text{acbabcabcabac}$ . The longest Lyndon substring of  $T$  is  $\text{abac}$  since  $LF_T = \text{acb}, (\text{abc})^2, \text{abac}$ . When we are given an interval  $[2, 3]$  of  $T$  and a string  $\text{bac}$ , the longest Lyndon substring of  $T'$  is  $\text{abacabcabc}$  since  $LF_{T'} = \text{abacabcabc}, \text{abac}$ . When we are given  $[8, 10]$  and an empty string, the longest Lyndon substring of  $T'$  is  $\text{abac}$  since  $LF_{T'} = \text{acb}, \text{abc}, \text{abac}$ .

► **Theorem 16.** *After constructing an  $O(n)$ -space data structure of a given string  $T$  in  $O(n)$  time, we can compute  $LLS(T')$  in  $O(l \log \sigma + \log n)$  time for each 1-block-edit query.*

This algorithm is almost similar to the 1-edit version. Let  $\alpha$  be a given string of length  $l$ . Firstly, we need to compute  $LF_\alpha$  in  $O(l)$  time. After that we can concatenate three parts in the similar way. The key difference is that we conduct an additional  $O(l \log \sigma)$  time and  $O(l)$  space processing in order to compare any two substrings in  $T'$  in constant time. Any comparisons on  $T'$  can be separated to constant number of comparisons between

- a substring in  $T$  and a substring in  $T$ ,
- a substring in  $\alpha$  and a substring in  $\alpha$ ,
- a substring in  $T$  and a substring in  $\alpha$ .

The first one can be done by an LCE query on  $T$ . The second one can be done in constant time after constructing an LCE data structure for  $\alpha$  in  $O(l)$  time and space. Now we explain the last case. Assume that we have computed the suffix tree of  $T$  in  $O(n)$  time preprocessing. For each of suffixes  $\alpha_i$  of  $\alpha$ , we compute the lowest node in the suffix tree which corresponds to some prefix of  $\alpha_i$ . This can be done in  $O(l \log \sigma)$  time by using Ukkonen's suffix tree construction algorithm [25]. Then we can compare a substring in  $T$  and a substring in  $\alpha$  by using LCA queries. Thus we can do any substring comparisons in constant time after constructing  $O(l \log \sigma)$  time and space data structures. Therefore, we obtain Theorem 16.

In the similar way to Section 3, we can get the following.

► **Corollary 17.** *After constructing an  $O(n)$ -space data structure of a given string  $T$  in  $O(n)$  time, we can compute all the longest Lyndon substrings of  $T'$  in  $O(l \log \sigma + \log n + occ)$  time for each 1-block-edit query where  $occ$  is the number of outputs.*

► **Remark.** If  $l$  is constant, we can compare the lexicographic order of any two substrings in  $T'$  in constant time (by using constant number of LCE queries and constant number of character comparisons) without using suffix trees. Then the querying time of Theorem 16 turns out to be  $O(\log n)$  time. Thus, this result includes Theorem 5.

## 5 Conclusion

We considered the problem of computing the longest Lyndon substring after 1-edit operation. We proposed an algorithm which uses  $O(n)$  time and space so that for any single block edit query, the longest Lyndon substring can be answered in  $O(l \log \sigma + \log n)$  time where  $l$  is the length of a given query string and  $\sigma$  is the number of distinct characters in  $T$ .

Our algorithm in this paper is almost the same for single characters edits and single block edits, and one of our interests is whether there is a more efficient solution at least for the case of single character edits.

---

## References

- 1 Amihod Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, pages 14–26, 2017.
- 2 Alberto Apostolico and Maxime Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995.
- 3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

- 4 Frédérique Bassino, Julien Clément, and Cyril Nicaud. The standard factorization of Lyndon words: an average point of view. *Discrete Mathematics*, 290(1):1–25, 2005.
- 5 Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *TCS*, 321(1):5–12, 2004.
- 6 Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. In *Proc. CPM 2011*, pages 173–183, 2011.
- 7 Marc Chemillier. Periodic musical sequences and Lyndon words. *Soft Comput.*, 8(9):611–616, 2004.
- 8 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.
- 9 Maxime Crochemore and Dominique Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.
- 10 Jacqueline W. Daykin, Frantisek Franek, Jan Holub, A. S. M. Sohidull Islam, and W. F. Smyth. Reconstructing a string from its Lyndon arrays. *Theor. Comput. Sci.*, 710:44–51, 2018.
- 11 Jacqueline W. Daykin, Costas S. Iliopoulos, and William F. Smyth. Parallel RAM algorithms for factorizing words. *Theor. Comput. Sci.*, 127(1):53–67, 1994.
- 12 Olivier Delgrange and Eric Rivals. STAR: an algorithm to search for tandem approximate repeats. *Bioinformatics*, 20(16):2812–2820, 2004.
- 13 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
- 14 Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 172–184, 2016.
- 15 Harold Fredricksen and James Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978.
- 16 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- 17 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003. doi:10.1016/S0304-3975(03)00099-9.
- 18 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016.
- 19 M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983.
- 20 R. C. Lyndon. On Burnside’s problem. *Transactions of the American Mathematical Society*, 77:202–215, 1954.
- 21 Marcin Mucha. Lyndon words and short superstrings. In *Proc. SODA’13*, pages 958–972, 2013.
- 22 Yuto Nakashima, Takuya Takagi, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On reverse engineering the Lyndon tree. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 108–117, 2017.
- 23 Shoshana Neuburger and Dina Sokol. Succinct 2D dictionary matching. *Algorithmica*, pages 1–23, 2012. 10.1007/s00453-012-9615-9.
- 24 Xavier Provençal. Minimal non-convex words. *Theor. Comput. Sci.*, 412(27):3002–3009, 2011.
- 25 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- 26 P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11. Institute of Electrical Electronics Engineers, New York, 1973.

# The Heaviest Induced Ancestors Problem Revisited

**Paniz Abedin**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
paniz@cs.ucf.edu

**Sahar Hooshmand**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
sahar@cs.ucf.edu

**Arnab Ganguly**

Dept. of Computer Science, University of Wisconsin - Whitewater, USA  
gangulya@uww.edu

**Sharma V. Thankachan**

Dept. of Computer Science, University of Central Florida - Orlando, USA  
sharma.thankachan@ucf.edu

---

## Abstract

---

We revisit the *heaviest induced ancestors* problem, which has several interesting applications in string matching. Let  $\mathcal{T}_1$  and  $\mathcal{T}_2$  be two weighted trees, where the weight  $W(u)$  of a node  $u$  in either of the two trees is more than the weight of  $u$ 's parent. Additionally, the leaves in both trees are labeled and the labeling of the leaves in  $\mathcal{T}_2$  is a permutation of those in  $\mathcal{T}_1$ . A node  $x \in \mathcal{T}_1$  and a node  $y \in \mathcal{T}_2$  are induced, iff their subtree have at least one common leaf label. A heaviest induced ancestor query  $\text{HIA}(u_1, u_2)$  is: given a node  $u_1 \in \mathcal{T}_1$  and a node  $u_2 \in \mathcal{T}_2$ , output the pair  $(u_1^*, u_2^*)$  of induced nodes with the highest combined weight  $W(u_1^*) + W(u_2^*)$ , such that  $u_1^*$  is an ancestor of  $u_1$  and  $u_2^*$  is an ancestor of  $u_2$ . Let  $n$  be the number of nodes in both trees combined and  $\epsilon > 0$  be an arbitrarily small constant. Gagie et al. [CCCG' 13] introduced this problem and proposed three solutions with the following space-time trade-offs:

- an  $O(n \log^2 n)$ -word data structure with  $O(\log n \log \log n)$  query time
- an  $O(n \log n)$ -word data structure with  $O(\log^2 n)$  query time
- an  $O(n)$ -word data structure with  $O(\log^{3+\epsilon} n)$  query time.

In this paper, we revisit this problem and present new data structures, with improved bounds. Our results are as follows.

- an  $O(n \log n)$ -word data structure with  $O(\log n \log \log n)$  query time
- an  $O(n)$ -word data structure with  $O\left(\frac{\log^2 n}{\log \log n}\right)$  query time.

As a corollary, we also improve the LZ compressed index of Gagie et al. [CCCG' 13] for answering longest common substring (LCS) queries. Additionally, we show that the LCS *after one edit* problem of size  $n$  [Amir et al., SPIRE' 17] can also be reduced to the heaviest induced ancestors problem over two trees of  $n$  nodes in total. This yields a straightforward improvement over its current solution of  $O(n \log^3 n)$  space and  $O(\log^3 n)$  query time.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** Data Structure, String Algorithms, Orthogonal Range Queries

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.20

**Funding** This research is supported in part by the U.S. National Science Foundation under the grant CCF-1703489.



© Paniz Abedin, Sahar Hooshmand, Arnab Ganguly and Sharma V. Thankachan; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 20; pp. 20:1–20:13

Leibniz International Proceedings in Informatics



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



## 1 Introduction

Let  $\mathcal{T}_1$  and  $\mathcal{T}_2$  be two weighted trees, having  $n_1$  and  $n_2$  nodes respectively. The weight of a node  $u$  in either of the two trees is given by  $W(u)$  and  $W(u) > W(\text{parent}(u))$ , where  $\text{parent}(u)$  is the parent node of  $u$ . For simplicity, node  $u$  means the node with pre-order rank  $u$ . Each tree has exactly  $m \leq \min\{n_1, n_2\}$  leaves. Leaves in both trees are labeled and the labeling of the leaves in  $\mathcal{T}_2$  is a permutation of the labeling of the leaves in  $\mathcal{T}_1$ . Two nodes, one each from  $\mathcal{T}_1$  and  $\mathcal{T}_2$ , are *induced* if the leaves in the respective subtrees have at least one common label. For any two nodes  $u$  and  $v$  in a tree, the node  $v$  is an ancestor of  $u$  iff  $v$  is on the path from  $u$  to the root of the tree. Moreover,  $v$  is a proper ancestor  $u$  iff  $u \neq v$ . We revisit the following problem, which has several interesting applications in string matching.

► **Problem 1** (Heaviest Induced Ancestor Problem). Given a node  $u_1 \in \mathcal{T}_1$  and a node  $u_2 \in \mathcal{T}_2$ , find  $\text{HIA}(u_1, u_2)$ , which is defined as the pair of *induced* nodes  $(u_1^*, u_2^*)$  with the highest combined weight  $W(u_1^*) + W(u_2^*)$ , such that  $u_1^*$  (resp.,  $u_2^*$ ) is an ancestor of  $u_1$  (resp.,  $u_2$ ).

Here and henceforth,  $\epsilon$  is an arbitrarily small positive constant and  $n = n_1 + n_2$  is the total number of nodes in the two trees. The model of computation is the standard Word RAM with word size  $\Omega(\log n)$  bits. Gagie et al. [8] presented the following several results for the Heaviest Induced Ancestor problem.

- an  $O(n \log^2 n)$ -word index with  $O(\log n \log \log n)$  query time
- an  $O(n \log n)$ -word index with  $O(\log^2 n)$  query time
- an  $O(n)$ -word index with  $O(\log^{3+\epsilon} n)$  query time.

Our contribution is summarized in the following Theorem.

► **Theorem 2.** *A heaviest induced ancestor query over two trees of  $n$  nodes in total can be answered*

- *in  $O(\log n \log \log n)$  time using an  $O(n \log n)$ -word data structure, or*
- *in  $O\left(\frac{\log^2 n}{\log \log n}\right)$  time using an  $O(n)$ -word data structure.*

### 1.1 Applications to String Matching

One motivation to study the heaviest induced ancestor problem is to design an LZ77 [20] compressed text index that can answer longest common substring  $\text{LCS}(S, P)$  queries efficiently. Formal definition is below.

► **Problem 3** (Longest Common Substring in LZ77 Compressed Strings [8]). *Given a string  $S$  of length  $N$ , whose LZ77 parsing contains  $n$  phrases, build a data structure that can efficiently report  $\text{LCS}(S, P)$ , i.e., the longest common substring of  $S$  and  $P$ , where  $P$  is a query string of length  $|P|$ .*

If one were to forego the compression requirement, the problem can be easily solved by maintaining a suffix tree [17] of  $S$  in  $O(N)$  words yielding  $O(|P|)$  query time. On the other hand, we can also answer  $\text{LCS}(S, P)$  queries using compressed/succinct data structures, such as the FM Index or Compressed Suffix Array [6, 9, 13], with a slight penalty in query time. However, for strings having a repetitive structure, LZ77-based compression techniques offer better space-efficiency than that obtained using FM-Index or Compressed Suffix Array.

Gagie et al. [8] showed that Problem 3 can be solved using an  $O(n \log N + n \log^2 n)$ -word index with very high probability in  $O(|P| \log n \log \log n)$  query time. Alternatively, they also



presented an  $O(n \log N)$ -word index with  $O(|P| \log^2 n)$  query time. Using Theorem 2 and the techniques in [8], we present an improved result for Problem 3 (see Theorem 4). We omit the details as they are immediate from the discussions in [8].

► **Theorem 4.** *Given a string  $S$  of length  $N$ , we can build an  $O(n \log N)$ -word index that reports  $\text{LCS}(S, P)$  in  $O(|P| \log n \log \log n)$  time with very high probability, where  $n$  is the number of phrases in an LZ77 parsing of  $S$  and  $|P|$  is the length of the input query string  $P$ .*

Another problem that we study is the recently introduced *longest common substring after one substitution* problem [1], defined as follows.

► **Problem 5 (Longest Common Substring after One Substitution [1]).** *Given two strings  $X$  and  $Y$  of total length  $n$  over an alphabet set  $\Sigma$ , build a data structure that can efficiently report  $\text{LCS}_{(i, \alpha)}(X, Y)$ , the length of the longest common substring of  $X_{\text{new}}$  and  $Y$ , where  $X_{\text{new}}$  is  $X$  after replacing its  $i$ th character by  $\alpha \in \Sigma$ .*

An  $O(n|\Sigma|)$  space and  $O(1)$  time solution is straightforward, but not efficient when  $|\Sigma|$  is large. The solution by Amir et al. [1] takes  $O(n \log^3 n)$  space and  $O(\log^3 n)$  query time. Theorem 2 combined with other techniques implies an improved result to this problem, as summarized in the following theorem.

► **Theorem 6.** *Given two strings  $X$  and  $Y$  of total length  $n$ , we can build indexes with the following space-time trade-offs for reporting  $\text{LCS}_{(i, \alpha)}(X, Y)$*

1. *an  $O(n \log n)$  space data structure with  $O(\log n \log \log n)$  query time*
2. *an  $O(n)$  space data structure with  $O\left(\frac{\log^2 n}{\log \log n}\right)$  query time.*

Straightforward modifications to our approach leads to an index that can also support the case of single letter insertions or deletions in  $X$ , i.e., insert the character  $\alpha$  after position  $i$  and delete the character at position  $i$ .

## 1.2 Map

In Section 2, we revisit some of the well-known data structures that have been used to arrive at our results. Section 3 presents an overview of our techniques, as an intermediate step into the final data structures. The final data structures for Theorem 2 are presented in Section 4. Section 5.1 is left to sketch our solution to Problem 5.

## 2 Preliminaries and Terminologies

### 2.1 Predecessor/Successor Queries

Let  $\mathcal{S}$  be a subset of  $\mathcal{U} = \{0, 1, 2, 3, \dots, U - 1\}$  of size  $n$ . A predecessor search query  $p$  on  $\mathcal{S}$  asks to return  $p$  if  $p \in \mathcal{S}$ , else return  $\max\{q < p \mid q \in \mathcal{S}\}$ . Similarly, a successor query  $p$  on  $\mathcal{S}$  asks to return  $p$  if  $p \in \mathcal{S}$ , else return  $\min\{q > p \mid q \in \mathcal{S}\}$ . By preprocessing  $\mathcal{S}$  into a  $y$ -fast trie of size  $O(n)$  words, we can answer such queries in  $O(\log \log U)$  time [18].

### 2.2 Fully-Functional Succinct Tree

Let  $\mathcal{T}$  be a tree having  $n$  nodes, such that nodes are numbered from 1 to  $n$  in the ascending order of their pre-order rank. Also, let  $\ell_i$  denotes the  $i$ th leftmost leaf. Then by maintaining an index of size  $2n + o(n)$  bits, we can answer the following queries on  $\mathcal{T}$  in constant time [14]:

- $\text{parent}_{\mathcal{T}}(u)$  = parent of node  $u$ .
- $\text{size}_{\mathcal{T}}(u)$  = number of leaves in the subtree of  $u$ .
- $\text{nodeDepth}_{\mathcal{T}}(u)$  = number of nodes on the path from  $u$  to the root of  $\mathcal{T}$ .
- $\text{levelAncestor}_{\mathcal{T}}(u, D)$  = ancestor  $w$  of  $u$  such that  $\text{nodeDepth}(w) = D$ .
- $\text{lMost}_{\mathcal{T}}(u) = i$ , where  $\ell_i$  is the leftmost leaf in the subtree of  $u$ .
- $\text{rMost}_{\mathcal{T}}(u) = j$ , where  $\ell_j$  is the rightmost leaf in the subtree of  $u$ .
- $\text{lca}_{\mathcal{T}}(u, v)$  = lowest common ancestor (LCA) of two nodes  $u$  and  $v$ .

We omit the subscript “ $\mathcal{T}$ ” if the context is clear.

### 2.3 Range Maximum Query (RMQ) and Path Maximum Query (PMQ)

Let  $A[1, n]$  be an array of  $n$  elements. A range maximum query  $\text{RMQ}_A(a, b)$  asks to return  $k \in [a, b]$ , such that  $A[k] = \max\{A[i] \mid i \in [a, b]\}$ . Path maximum query (PMQ) (or bottleneck edge query [5]) is a generalization of RMQ from arrays to trees. Let  $\mathcal{T}$  be a tree having  $n$  nodes, such that each node  $u$  is associated with a score. A path maximum query  $\text{PMQ}_{\mathcal{T}}(a, b)$  returns the node  $k$  in  $\mathcal{T}$ , where  $k$  is a node with highest score among all nodes on the path from node  $a$  to node  $b$ . Cartesian tree based solutions exist for both problems. The space and query time are  $2n + o(n)$  bits and  $O(1)$ , respectively [5, 7].

### 2.4 Orthogonal Range Queries in 2-Dimension

Let  $\mathcal{P}$  be a set of  $n$  points in an  $[1, n] \times [1, n]$  grid. Then,

- An orthogonal range counting query  $(a, b, c, d)$  on  $\mathcal{P}$  returns the cardinality of  $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \in [c, d]\}$
- An orthogonal range emptiness query  $(a, b, c, d)$  on  $\mathcal{P}$  returns “EMPTY” if the cardinality of the set  $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \in [c, d]\}$  is zero. Otherwise, it returns “NOT-EMPTY”.
- An orthogonal range predecessor query  $(a, b, c)$  on  $\mathcal{P}$  returns the point in  $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \leq c\}$  with the highest  $y$ -coordinate value, if one exists.
- An orthogonal range successor query  $(a, b, c)$  on  $\mathcal{P}$  returns the point in  $\{(x, y) \in \mathcal{P} \mid x \in [a, b], y \geq c\}$  with the lowest  $y$ -coordinate value, if one exists.
- An orthogonal range selection query  $(a, b, k)$  on  $\mathcal{P}$  returns the point in  $\{(x, y) \in \mathcal{P} \mid x \in [a, b]\}$  with the  $k$ th lowest  $y$ -coordinate value.

By maintaining an  $O(n)$  word structure, we can answer orthogonal range counting queries in  $O(\log / \log \log n)$  time [11], orthogonal range emptiness queries in  $O(\log^\epsilon n)$  time [3], orthogonal range predecessor/successor queries in  $O(\log^\epsilon n)$  time [12] and orthogonal range selection queries in  $O(\log n / \log \log n)$  time [2, 4]. Alternatively, by maintaining an  $O(n \log \log n)$  space structure, we can answer orthogonal range emptiness and orthogonal range predecessor/successor queries in  $O(\log \log n)$  time [3, 19].

### 2.5 Heavy Path and Heavy Path Decomposition

We now define the heavy path decomposition [10, 15] of a tree  $\mathcal{T}$  having  $n$  nodes. First, the nodes in  $\mathcal{T}$  are categorized into light and heavy. The root node is *light* and exactly one child of every internal node is heavy. Specifically, the child having the largest number of nodes in its subtree (ties are broken arbitrarily). The first heavy path of  $\mathcal{T}$  is the path starting at  $\mathcal{T}$ 's root, and traversing through every heavy node to a leaf. Each off-path subtree of the first heavy path is further decomposed recursively. Clearly, a tree with  $m$  leaves has  $m$  heavy paths. Let  $u$  be a node on a heavy path  $H$ , then  $\text{hp\_root}(u)$  is the highest node on  $H$  and  $\text{hp\_leaf}(u)$  is the lowest node on  $H$ . Note that  $\text{hp\_root}(\cdot)$  is always light.

► **Fact 7.** For a tree having  $n$  nodes, the path from the root to any leaf traverses at most  $\lceil \log n \rceil$  light nodes. Consequently, the sum of the subtree sizes of all light nodes (i.e., the starting node of a heavy path) put together is at most  $n \lceil \log n \rceil$ .

### 3 Our Framework

We assume that both trees  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are compacted, i.e., any internal node has at least two children. This ensures that the number of internal nodes is strictly less than the number of leaves ( $m$ ). Thus,  $n \leq 4m - 2$ . We remark that this assumption can be easily removed without affecting the query time. We maintain the tree topology of  $\mathcal{T}_1$  and  $\mathcal{T}_2$  succinctly in  $O(n)$  bits with constant time navigational support (refer to Section 2.2). Define two arrays,  $\text{Label}_k[1, m]$  for  $k = 1$  and  $2$ , such that  $\text{Label}_k[j]$  is the label associated with the  $j$ th leaf node in  $\mathcal{T}_k$ . The following is a set of  $m$  two-dimensional points based on tree labels.

$$\mathcal{P} = \{(i, j) \mid i, j \in [1, m] \text{ and } \text{Label}_1[i] = \text{Label}_2[j]\}$$

We pre-process  $\mathcal{P}$  into a data structure, so as to support various range queries described in Section 2.4. For range counting and selection, we maintain data structures with  $O(n)$  space and  $O(\log n / \log \log n)$  time. For range successor/predecessor and emptiness queries, we have two options: and  $O(n \log \log n)$  space structure with  $O(\log \log n)$  time, and an  $O(n)$  space structure with  $O(\log^\epsilon n)$  time. We employ the first result in our  $O(n \log n)$  space solution and the second result in our  $O(n)$  space solution.

#### 3.1 Basic Queries

► **Lemma 8 (Induced-Check).** *Given two nodes  $x, y$ , where  $x \in \mathcal{T}_1$  and  $y \in \mathcal{T}_2$ , we can check if they are induced or not*

- *in  $O(\log \log n)$  time using an  $O(n \log \log n)$  space structure, or*
- *in  $O(\log^\epsilon n)$  time using an  $O(n)$  space structure.*

**Proof.** The task can be reduced to a range emptiness query, because  $x$  and  $y$  are induced iff the set  $\{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(x), \text{rMost}(x)] \times [\text{lMost}(y), \text{rMost}(y)]\}$  is not empty. ◀

► **Definition 9 (Partner).** The partner of a node  $x \in \mathcal{T}_1$  w.r.t a node  $y \in \mathcal{T}_2$ , denoted by  $\text{partner}(x/y)$  is the lowest ancestor  $y'$  of  $y$ , such that  $x$  and  $y'$  are induced. Likewise,  $\text{partner}(y/x)$  is the lowest ancestor  $x'$  of  $x$ , such that  $x'$  and  $y$  are induced.

► **Lemma 10 (Find Partner).** *Given two nodes  $x, y$ , where  $x \in \mathcal{T}_1$  and  $y \in \mathcal{T}_2$ , we can find  $\text{partner}(x/y)$  as well as  $\text{partner}(y/x)$*

- *in  $O(\log \log n)$  time using an  $O(n \log \log n)$  space structure, or*
- *in  $O(\log^\epsilon n)$  time using an  $O(n)$  space structure.*

**Proof.** To find  $\text{partner}(x/y)$ , first check if  $x$  and  $y$  are induced. If yes, then  $\text{partner}(x/y) = y$ . Otherwise, find the last leaf node  $\ell_a \in \mathcal{T}_2$  before  $y$  in pre-order, such that  $x$  and  $\ell_a$  are induced ( $\ell_a$  denotes  $a$ -th leftmost leaf). Also, find the first leaf node  $\ell_b \in \mathcal{T}_2$  after  $y$  in pre-order, such that  $x$  and  $\ell_b$  are induced. Both tasks can be reduced to orthogonal range predecessor/successor queries.

$$(\cdot, a) = \arg \max_j \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(x), \text{rMost}(x)] \times [1, \text{lMost}(y)]\}$$

$$(\cdot, b) = \arg \min_j \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(x), \text{rMost}(x)] \times [\text{rMost}(y), m]\}$$

Clearly, an ancestor of  $y$  and  $x$  are induced iff either  $\ell_a$  or  $\ell_b$  is in its subtree. Therefore, we report the lowest node among  $u_a = \text{lca}(\ell_a, y)$  and  $u_b = \text{lca}(\ell_b, y)$  as  $\text{partner}(x/y)$ . The computation of  $\text{partner}(y/x)$  is analogous. ◀

### 3.2 Overview

For any two nodes  $u$  and  $v$  in the same tree  $\mathcal{T}$ , define  $\text{Path}(u, v, \mathcal{T})$  as the set of nodes on the path from  $u$  to  $v$ . Let  $\text{root}_1$  be the root of  $\mathcal{T}_1$  and  $\text{root}_2$  be the root of  $\mathcal{T}_2$ . Throughout this paper,  $(u_1, u_2)$  denotes the input and  $\text{HIA}(u_1, u_2) = (u_1^*, u_2^*)$  denotes the output. Clearly,  $u_2^* = \text{partner}(u_1^*/u_2)$  and  $u_1^* = \text{partner}(u_2^*/u_1)$ . Therefore,

$$(u_1^*, u_2^*) = \arg \max_{(x,y)} \{W(x) + W(y) \mid y \in \text{Path}(\text{root}_2, u_2, \mathcal{T}_2) \text{ and } x = \text{partner}(y/u_1)\}$$

To evaluate the above equation efficiently, we explore the heavy path decomposition of  $\mathcal{T}_2$ .

► **Definition 11 (Special Nodes).** For each light node in  $w \in \mathcal{T}_2$ , we identify a set  $\text{Special}(w)$  of nodes in  $\mathcal{T}_1$  (which we call special nodes) as follows: a leaf node  $\ell_i \in \mathcal{T}_1$  is *special* iff  $\ell_i$  and  $w$  are induced. An internal node in  $\mathcal{T}_1$  is special iff it is the lowest common ancestor of two special leaves. Additionally, for each node  $x \in \text{Special}(w)$ , define its score w.r.t.  $w$  as the sum of weights of  $x$  and the node  $\text{partner}(x/\text{hp\_leaf}(w)) \in \mathcal{T}_2$ . Formally,

$$\text{score}_w(x) = W(x) + W(\text{partner}(x/\text{hp\_leaf}(w)))$$

Moreover,  $|\text{Special}(w)| \leq 2\text{size}(w) - 1$  and  $\sum_{w \text{ is a light node}} |\text{Special}(w)| = O(n \log n)$ .

To answer an HIA query  $(u_1, u_2)$ , we first identify some nodes in  $\mathcal{T}_1$  and  $\mathcal{T}_2$  as follows. Nodes  $w_1 = \text{root}_2, w_2, \dots, w_k$  are the *light* nodes in  $\text{Path}(\text{root}_2, u_2, \mathcal{T}_2)$  (in the ascending order of their pre-order ranks). Nodes  $t_1, t_2, \dots, t_k$  are also in  $\text{Path}(\text{root}_2, u_2, \mathcal{T}_2)$ , such that  $t_k = u_2$  and  $t_h = \text{parent}(w_{h+1})$  for  $h < k$ . Therefore,  $\text{Path}(\text{root}_2, u_2, \mathcal{T}_2) = \cup_{h=1}^k \text{Path}(w_h, t_h, \mathcal{T}_2)$ . Next,  $\alpha_1, \alpha_2, \dots, \alpha_k$  and  $\beta_1, \beta_2, \dots, \beta_k$  are nodes in  $\text{Path}(\text{root}_1, u_1, \mathcal{T}_1)$ , such that for  $h = 1, 2, \dots, k$ ,  $\alpha_h = \text{partner}(t_h/u_1)$  and  $\beta_h = \text{partner}(w_h/u_1)$ . Clearly, there exists an  $f \in [1, k]$  such that  $u_2^* \in \text{Path}(w_f, t_f, \mathcal{T}_2)$ . See Figure 1 for an illustration. We now present several lemmas, which forms the basis of our solution.

► **Lemma 12.** *The node  $u_1^* \in \text{Path}(\alpha_f, \beta_f, \mathcal{T}_1)$ .*

**Proof.** We prove this via proof by contradiction arguments.

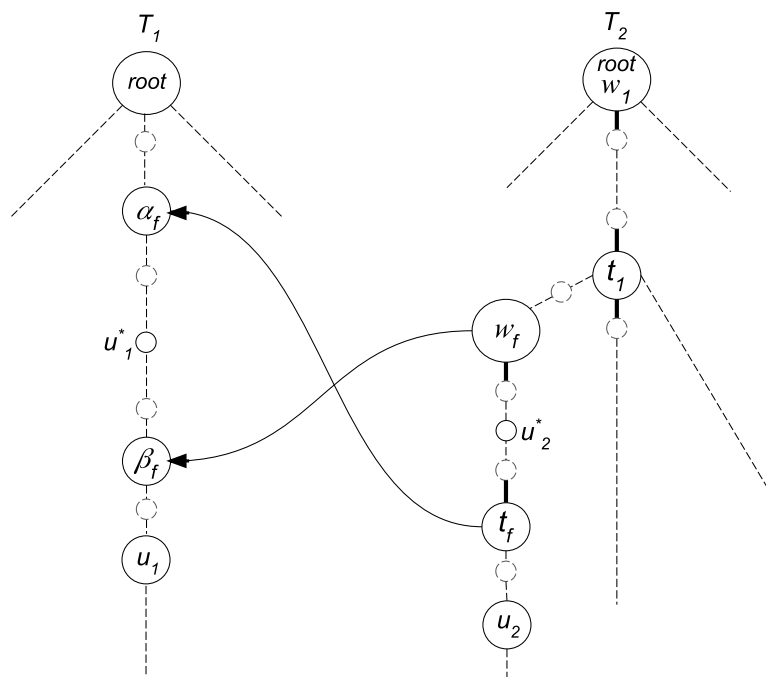
- Suppose  $u_1^*$  is a proper ancestor of  $\alpha_f$ . Then,  $\alpha_f$  and  $t_f$  are induced and  $W(\alpha_f) + W(t_f) > W(u_1^*) + W(u_2^*)$ , a contradiction. Therefore,  $u_1^*$  is in the subtree of  $\alpha_f$ .
- Suppose  $u_1^*$  is in the proper subtree of  $\beta_f$ . Then,  $u_1^*$  and  $w_f$  are also induced. Therefore,  $\text{partner}(w_f/u_1)$  is  $u_1^*$  or a node in the subtree of  $u_1^*$ . This implies,  $\beta_f = \text{partner}(w_f/u_1)$  is in the proper subtree of  $\beta_f$ , a contradiction. Therefore,  $u_1^*$  is an ancestor of  $\beta_f$ .

This completes the proof. ◀

► **Lemma 13.** *The node  $u_1^* \in \text{Special}(w_f) \cup \{\beta_f\}$ .*

**Proof.** Let  $z$  (if exists) be the first node in  $\text{Special}(w_f)$  on the path from  $u_1^*$  to  $\beta_f$ . Then,

- if  $z$  exists, then  $u_1^* \notin \text{Special}(w_f)$  gives a contradiction as follows. The intersection of the following two sets is empty: (i) set of labels of the leaves in the subtree of  $u_1^*$ , but not in the subtree of  $z$  and (ii) set of labels associated with the leaves in the subtree of  $w_f$ . This implies,  $z$  and  $u_2^*$  are induced (because  $u_1^*$  and  $u_2^*$  are induced) and  $W(z) + W(u_2^*) > W(u_1^*) + W(u_2^*)$ , a contradiction.



■ **Figure 1** We refer to Section 3.2 for the description of this figure.

- otherwise, if  $z$  does not exist, then it is possible that  $u_1^* \notin \text{Special}(w)$ . However, in this case,  $u_1^* = \beta_f$  (proof follows from similar arguments as above).

In summary,  $u_1^* \in \text{Special}(w_f) \cup \{\beta_f\}$ . ◀

► **Lemma 14.** For any  $x \in \text{Path}(\alpha_f, \beta_f, \mathcal{T}_1) \setminus \{\alpha_f\}$ ,  $\text{partner}(x/u_2) = \text{partner}(x/\text{hp\_leaf}(w_f))$ .

**Proof.** We claim that for any  $x \in \text{Path}(\alpha_f, \beta_f, \mathcal{T}_1) \setminus \{\alpha_f\}$ ,  $\text{partner}(x/u_2)$  is a proper ancestor of  $t_f$ . The proof follows from contradiction as follows. Suppose, there exists an  $x \in \text{Path}(\alpha_f, \beta_f, \mathcal{T}_1) \setminus \{\alpha_f\}$ , such that  $\text{partner}(x/u_2)$  is in the subtree of  $t_f$ . Then,  $x$  and  $t_f$  are induced. This means,  $\alpha_f = \text{partner}(t_f/u_1)$  is a node in the subtree of  $x$ , a contradiction.

Since,  $\text{partner}(x/u_2)$  is a proper ancestor of  $t_f$ ,  $\text{partner}(x/u_2) = \text{partner}(x/r)$  for any node  $r$  in the subtree of  $t_f$ . Therefore, by choosing  $r = \text{hp\_leaf}(w_f)$ , we obtain Lemma 14. ◀

► **Corollary 15.** For any  $x \in (\text{Path}(\alpha_f, \beta_f, \mathcal{T}_1) \setminus \{\alpha_f\})$ ,

$$W(x) + W(\text{partner}(x/u_2)) = W(x) + W(\text{partner}(x/\text{hp\_leaf}(w_f))) = \text{score}_{w_f}(x)$$

► **Lemma 16.** The node  $u_1^* \in \{\alpha_f, \beta_f, \gamma_f\}$ , where

$$\gamma_f = \arg \max_x \{\text{score}_{w_f}(x) \mid x \in \text{Special}(w_f) \cap (\text{Path}(\alpha_f, \beta_f, \mathcal{T}_1) \setminus \{\alpha_f, \beta_f\})\}$$

**Proof.** Follows from Lemma 12, Lemma 13, Lemma 14 and Corollary 15. ◀

► **Lemma 17.** Let  $\mathcal{C} = \cup_{h=1}^k \{\alpha_h, \beta_h, \gamma_h\}$ , where

$$\gamma_h = \arg \max_x \{\text{score}_{w_h}(x) \mid x \in \text{Special}(w_h) \cap (\text{Path}(\alpha_h, \beta_h, \mathcal{T}_1) \setminus \{\alpha_h, \beta_h\})\}$$

Then,

$$(u_1^*, u_2^*) = \arg \max_{(x,y)} \{W(x) + W(y) \mid x \in \mathcal{C} \text{ and } y = \text{partner}(x/u_2)\}$$

**Proof.** Since  $f$  is unknown, we invoke Lemma 14 for  $f = 1, 2, 3, \dots, k \leq \log n$ . ◀

Next, we show how to transform the result in Lemma 17 into an efficient data structure.

## 4 Our Data Structures

We start by defining a crucial component of our solution.

► **Definition 18** (Induced Subtree). The induced subtree of  $\mathcal{T}_1(w)$  of  $\mathcal{T}_1$  w.r.t. a light node  $w \in \mathcal{T}_2$  is a tree having exactly  $|\text{Special}(w)|$  number of nodes, such that

- for each node  $x \in \mathcal{T}_1(w)$ , there exists a node  $\text{Map}_w(x) \in \text{Special}(w)$  and
- for each  $x' \in \text{Special}(w)$ , there exists a node  $\text{invMap}_w(x') \in \mathcal{T}_1(w)$ , such that

$$\text{lca}_{\mathcal{T}_1}(\text{Map}_w(x), \text{Map}_w(y)) = \text{Map}_w(\text{lca}_{\mathcal{T}_1(w)}(x, y))$$

Note that a node  $x$  is a leaf in  $\mathcal{T}_1(w)$  iff  $\text{Map}_w(x)$  is a leaf in  $\mathcal{T}_1(w)$ . In the following lemmas, we present two space-time trade-offs on induced subtrees.

► **Lemma 19.** *By maintaining an  $O(n \log n)$  space structure, we can compute  $\text{Map}_w(\cdot)$  and  $\text{invMap}_w(\cdot)$  for any light node  $w \in \mathcal{T}_2$  in time  $O(1)$  and  $O(\log \log n)$ , respectively.*

**Proof.** Let  $L_w[1, |\text{Special}(w)|]$  be an array, such that  $L_w[x] = \text{Map}_w(x)$ . For each  $w$ , maintain  $L_w$  and a y-fast trie [18] over it. The total space is  $O(n \log n)$ . Now, any  $\text{Map}_w(\cdot)$  query can be answered in constant time. Also, for any  $x' \in \text{Special}(w)$ ,  $\text{invMap}_w(x')$  is the number of elements in  $L_w$  that are  $\leq x'$ . Therefore, an  $\text{invMap}_w(\cdot)$  can be reduced to a predecessor search and answered in  $O(\log \log n)$  time. ◀

► **Lemma 20.** *By maintaining an  $O(n)$  space structure, we can compute  $\text{Map}_w(\cdot)$  and  $\text{invMap}_w(\cdot)$  for any light node  $w \in \mathcal{T}_2$  in time  $O(\log n / \log \log n)$ .*

**Proof.** Let node  $p$  be the  $r$ th leaf in  $\mathcal{T}_1(w)$  and  $q = \text{Map}_w(p)$  be the  $s$ th leaf in  $\mathcal{T}_1$ . Then,  $s$  is the  $x$ -coordinate of the  $r$ th point in  $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, m] \times [\text{lMost}(w), \text{rMost}(w)]\}$  in the ascending order of  $x$ -coordinates. Also,  $r$  is the number of points in  $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, s] \times [\text{lMost}(w), \text{rMost}(w)]\}$ . Therefore, given  $p$ , we can compute  $r$ , then  $s$  and  $q$  in  $O(\log n / \log \log n)$  time via a range selection query on  $\mathcal{P}$ . Similarly, given  $q$ , we can compute  $s$  and then  $r$  and  $p$  in  $O(\log n / \log \log n)$  time via a range counting query on  $\mathcal{P}$ .

Now, if  $p$  is an internal node in  $\mathcal{T}_1(w)$ , then  $\text{Map}_w(p)$  is  $\text{lca}_{\mathcal{T}_1}(\text{Map}_w(\ell_L), \text{Map}_w(\ell_R))$ , where  $\ell_L$  and  $\ell_R$  are the first and last leaves in the subtree of  $p$ . Similarly, if  $q$  is an internal node in  $\mathcal{T}_1$ , then  $\text{invMap}_w(q) = \text{lca}_{\mathcal{T}_1(w)}(\text{invMap}_w(\ell_A), \text{invMap}_w(\ell_B))$  as follows:

$$(A, \cdot) = \arg \min_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(q), \text{rMost}(q)] \times [\text{lMost}(w), \text{rMost}(w)]\}$$

$$(B, \cdot) = \arg \max_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(q), \text{rMost}(q)] \times [\text{lMost}(w), \text{rMost}(w)]\}$$

Here,  $A$  and  $B$  can be computed via range successor/predecessor queries in  $O(\log^\epsilon n)$  time. Therefore, the total time is  $\log^\epsilon n + \log n / \log \log n = O(\log n / \log \log n)$  time. ◀

► **Lemma 21.** *Given an input  $(a, b, w)$ , where  $w$  is a light node in  $\mathcal{T}_2$  and,  $a$  and  $b$  are nodes in  $\mathcal{T}_1(w)$ , we can report the node with the highest  $\text{score}_w(\text{Map}_w(\cdot))$  over all nodes on the path from  $a$  to  $b$  in  $\mathcal{T}_1(w)$  in  $O(1)$  time using an  $O(n)$  space structure.*

**Proof.** For each  $\mathcal{T}_1(w)$ , we maintain the Cartesian tree for answering path maximum query (refer to Section 2.3). Space for a particular  $w$  is  $|\text{Special}(w)|(2 + o(1))$  bits and space over all light nodes  $w$  in  $\mathcal{T}_2$  is  $O(n \log n)$  bits (from Fact 7), equivalently  $O(n)$  words. For an input  $(a, b, w)$ , the answer is  $\text{PMQ}_{\mathcal{T}_1(w)}(a, b)$ .  $\blacktriangleleft$

#### 4.1 Our $O(n \log n)$ space data structure

We maintain  $\mathcal{T}_1$  and  $\mathcal{T}_2$  explicitly, so that the weight of any node in either of the trees can be accessed in constant time. Moreover, we maintain fully-functional succinct representation of their topologies (refer to Section 2.2) for supporting various operations in  $O(1)$  time. Additionally, we maintain the structures for answering Induced-Check and Find-Partner queries in  $O(\log \log n)$  time, data structures for range predecessor/successor queries on  $\mathcal{P}$  in  $O(\log \log n)$  time (refer to Section 2.4) and the structures described in Lemma 19 and Lemma 21. Thus, the total space is  $O(n \log n)$  words.

We now present the algorithm for computing the output  $(u_1^*, u_2^*)$  for a given input  $(u_1, u_2)$ . Following are the key steps.

1. Find  $w_h$  and  $t_h$  for  $h = 1, 2, \dots, k \leq \log n$ .
2. Find  $\alpha_h$  and  $\beta_h$  for  $h = 1, 2, \dots, k \leq \log n$ .
3. Let  $\alpha'_h$  be the first and  $\beta'_h$  be the last special node (w.r.t.  $w_h$ ) on the path from  $\alpha_h$  (excluding  $\alpha_h$ ) to  $\beta_h$  (excluding  $\beta_h$ ). Also, let

$$\gamma_h = \text{Map}_{w_h} \left( \text{PMQ}_{\mathcal{T}_1(w_h)} \left( \text{invMap}_{w_h}(\alpha'_h), \text{invMap}_{w_h}(\beta'_h) \right) \right)$$

Compute  $\gamma_h$  for  $h = 1, 2, \dots, k \leq \log n$ .

4. Obtain  $\mathcal{C} = \cup_{h=1}^k \{\alpha_h, \beta_h, \gamma_h\}$  and report

$$(u_1^*, u_2^*) = \arg \max_{(x,y)} \{W(x) + W(y) \mid x \in \mathcal{C} \text{ and } y = \text{partner}(x/u_2)\}$$

The correctness follows immediately from Lemma 17. We now bound the time complexity. Step 1 takes  $O(k)$  time and step 2 takes  $O(k)$  number of Find-Partner queries with  $O(\log \log n)$  time per query. The procedure for computing  $\alpha'_h$  and  $\beta'_h$  is the following.

- Find the child  $\alpha''_h$  of  $\alpha_h$  on the path from  $\alpha_h$  to  $\beta_h$ . Then  $\alpha'_h = \text{lca}_{\mathcal{T}_1}(\ell_{a_h}, \ell_{b_h})$ , where  $\ell_{a_h}$  (resp.  $\ell_{b_h}$ ) is the first (resp. last) special leaf in the subtree of  $\alpha''_h$  (w.r.t.  $w_h$ ). To compute  $a_h$  and  $b_h$ , we rely on range predecessor/successor queries on  $\mathcal{P}$ :

$$(a_h, \cdot) = \arg \min_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(\alpha''_h), \text{rMost}(\alpha''_h)] \times [\text{lMost}(w_h), \text{rMost}(w_h)]\}$$

$$(b_h, \cdot) = \arg \max_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{lMost}(\alpha''_h), \text{rMost}(\alpha''_h)] \times [\text{lMost}(w_h), \text{rMost}(w_h)]\}$$

- Find the rightmost special (w.r.t.  $w_h$ ) leaf  $\ell_{d_h}$  before  $\beta_h$  and the leftmost special (w.r.t.  $w_h$ ) leaf  $\ell_{g_h}$  after the last leaf in the subtree of  $\beta_h$ . For this, we rely on range predecessor/successor queries on  $\mathcal{P}$ :

$$(d_h, \cdot) = \arg \max_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [1, \text{lMost}(\alpha''_h) - 1] \times [\text{lMost}(w_h), \text{rMost}(w_h)]\}$$

$$(g_h, \cdot) = \arg \min_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\text{rMost}(\alpha''_h) + 1, m] \times [\text{lMost}(w_h), \text{rMost}(w_h)]\}$$

Then,  $\beta'_h = \text{lca}_{\mathcal{T}_1}(\ell_{d_h}, \ell_{g_h})$  if  $\beta_h$  and  $w_h$  are not induced (i.e., there does not exist a special node (w.r.t.  $w_h$ ) under  $\beta_h$ ). Otherwise,  $\beta'_h$  is the lowest node among  $\text{lca}_{\mathcal{T}_1}(\ell_{d_h}, \beta_h)$  and  $\text{lca}_{\mathcal{T}_1}(\beta_h, \ell_{g_h})$ .



The time for a range predecessor/successor query on  $\mathcal{P}$  is  $O(\log \log n)$ . Therefore, computation of  $\alpha'_h$  and  $\beta'_h$  takes  $O(\log \log n)$  time, and an additional  $O(\log \log n)$  for evaluating  $\gamma_h$ . Therefore, the total time for step 3 is  $O(k \log \log n)$ . Finally, step 4 also takes  $O(k \log \log n)$  time. By putting every thing together, the total time complexity is  $k \log \log n = O(\log n \log \log n)$ .

## 4.2 Our Linear Space Data Structure

We obtain our linear space data structure by replacing all super-linear space components in the previous solution by their space efficient counter parts. Specifically, we use linear space structures for Induced-Check, Find-Partner, and range predecessor/successor with query time  $O(\log^\epsilon n)$ . Also, we use the structure in Lemma 20 instead of the structure in Lemma 19. Thus, the total space is  $O(n)$  words.

The query algorithm remains the same. The time complexity is:  $O(k)$  for step 1,  $O(k \log^\epsilon n)$  for step 2,  $O(k \log n / \log \log n)$  for step 3 and  $O(k \log^\epsilon n)$  for step 4. Thus, total time is  $O(\log^2 n / \log \log n)$ .

## 5 Applications

### 5.1 Longest Common Substring after One Substitution

Let  $X$  and  $Y$  be two strings of total length  $n$  over an alphabet set  $\Sigma$ . Define,  $\text{LCS}(X, Y)$  as the length of the longest common substring of  $X$  and  $Y$  and  $\text{LCS}_{(i, \alpha)}(X, Y)$  as the length of the longest common substring of  $X_{new}$  and  $Y$ , where  $X_{new}$  is  $X$  after replacing its  $i$ th character by  $\alpha \in \Sigma$ . Our task is to build a data structure for  $X$  and  $Y$ , so that  $\text{LCS}_{(i, \alpha)}(X, Y)$  for any input  $(i, \alpha)$  can be reported efficiently.

#### 5.1.1 The Data Structure

Let  $\text{LCS}_{(i, \alpha)}$  be  $X[l, r]$ . As observed by Amir et al. [1], two possible scenarios are:  $i \notin [l, r]$  and  $i \in [l, r]$ . We handle each of these scenarios separately, i.e., we find the new longest common substring (with the character at position  $i$  replaced by  $\alpha$ ) with position  $i$  (a) not covered and (b) covered, and choose the longest. To obtain (a), simply store an array  $A[1, |X|]$ , where

$$A[i] = \max\{\text{LCS}(Y, X[1 \dots (i-1)]), \text{LCS}(Y, X[(i+1) \dots |X|])\}$$

For case (b), we maintain the following structures.

1. A generalized suffix tree [17] of  $X$  and  $Y$  (GST), which is a compact trie over all suffixes of  $X$  and  $Y$ , after appending each string from  $X$  (resp.,  $Y$ ) with a unique symbol  $\$1$  (resp.,  $\$2$ ).
2. A compact trie of reverse of all prefixes of  $X$  and  $Y$  (GPT), after appending each string from  $X$  (resp.,  $Y$ ) with a unique symbol  $\$1$  (resp.,  $\$2$ ).
3. For each character  $\alpha \in \Sigma$ ,
  - a compact trie  $\mathcal{T}_\alpha$  of all strings in  $\{Y[(i+1) \dots] \mid Y[i] = \alpha\}$  after appending each suffix with  $\$2$ . We label  $Y[(i+1) \dots]$  with  $i$ .
  - Another compact trie  $\mathcal{T}'_\alpha$  of all strings in  $\{\overleftarrow{Y[1 \dots (i-1)]} \mid Y[i] = \alpha\}$ . Here  $\overleftarrow{Y[1 \dots (i-1)]}$  is the reverse of  $Y[1 \dots (i-1)]$  and we label it with  $i$ .



- The data structure for HIA queries on  $(\mathcal{T}_\alpha, \mathcal{T}'_\alpha)$ . Here the weight of a node is its string-depth. Therefore, we can easily generalize our solution to the HIA problem to the case where the input  $(u_1, u_2)$  is such that  $u_1$  and  $u_2$  are not necessarily nodes, but locations on edges.

The total space is proportional to the size of an HIA structure over an input of size  $n$ .

### 5.1.2 Processing a query $(i, \alpha)$

Get the LCS not covering  $i$  in constant time from the array  $A$ . For LCS covering  $i$ , do the following steps.

- Let  $\ell_p$  be the leaf in GST corresponding to the suffix  $X[(i+1)\dots]$ . Find the lowest ancestor  $u$  of  $\ell_p$  with at least one leaf corresponding to a suffix of  $Y$  (say  $Y[a\dots]$ ) in its subtree.
- Let  $\ell_q$  be the leaf in GPT corresponding to the reverse of the prefix  $X[1\dots(i-1)]$ . Find the lowest ancestor  $v$  of  $\ell_q$  with at least one leaf corresponding to a reverse of a prefix of  $Y$  (say  $Y[\dots b]$ ) in its subtree.
- Issue an HIA query  $\text{HIA}(x, y)$  on  $(\mathcal{T}_\alpha, \mathcal{T}'_\alpha)$ , where
  1.  $x$  is the location in  $\mathcal{T}_\alpha$  on the path of the leaf corresponding to  $Y[a\dots]$  at a distance of string-depth of  $u$  from the root.
  2.  $y$  is the location in  $\mathcal{T}'_\alpha$  on the path of the leaf corresponding to  $\overleftarrow{Y[\dots b]}$  at a distance of string-depth of  $v$  from the root.

Let  $(x^*, y^*)$  be the output. Then,  $\text{LCS}_{(i,\alpha)}(X, Y)$  covering position  $i$  is  $W(x^*) + W(y^*)$ .

Therefore, final  $\text{LCS}_{(i,\alpha)}(X, Y)$  is  $\max\{A[i], W(x^*) + W(y^*)\}$ . Steps 1 and 2 can be performed in  $O(\log n)$  time binary searches. Therefore, total time is dominated by the time for an HIA query. The correctness can be easily verified.

## 5.2 All-Pairs Longest Common Substring Problem

Let  $\mathcal{S} = \{S_1, S_2, S_3, \dots, S_n\}$  be a collection of  $n$  strings and let  $L$  be the length of the longest string in  $\mathcal{S}$ . We consider the problem of finding  $\text{LCS}(S_i, S_j)$  for all  $(i, j)$  pairs. This problem can be easily solved in  $O(n^2L)$  time. However, it is also possible to obtain a conditional lower bound of  $\tilde{\Omega}(n^2L)$  via a reduction from the boolean matrix multiplication [16]. To this end, we remark that the following run-time is possible with the aid of HIA framework.

$$\tilde{O}\left(nL + \sum_i \sum_{j < i} \frac{L}{\text{LCS}(S_i, S_j)}\right)$$

We defer details to the full version of this paper.

---

### References

- 1 Amihoud 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, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2017. doi:10.1007/978-3-319-67428-5\_2.

- 2 Gerth Stølting Brodal and Allan Grønlund Jørgensen. Data structures for range median queries. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 822–831. Springer, 2009. doi:10.1007/978-3-642-10631-6\_83.
- 3 Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.
- 4 Timothy M. Chan and Bryan T. Wilkinson. Adaptive and approximate orthogonal range counting. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 241–251. SIAM, 2013. doi:10.1137/1.9781611973105.18.
- 5 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/s00453-012-9683-x.
- 6 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.
- 7 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.
- 8 Travis Gagie, Pawel Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013, Waterloo, Ontario, Canada, August 8-10, 2013*. Carleton University, Ottawa, Canada, 2013. URL: [http://cccg.ca/proceedings/2013/papers/paper\\_29.pdf](http://cccg.ca/proceedings/2013/papers/paper_29.pdf).
- 9 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.
- 10 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 11 Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568. Springer, 2004. doi:10.1007/978-3-540-30551-4\_49.
- 12 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In Fedor V. Fomin and Petteri Kaski, editors, *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings*, volume 7357 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2012. doi:10.1007/978-3-642-31155-0\_24.
- 13 Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 225–232. ACM/SIAM, 2002. URL: <http://dl.acm.org/citation.cfm?id=545381.545410>.
- 14 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149. SIAM, 2010. doi:10.1137/1.9781611973075.13.

- 15 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 114–122. ACM, 1981. doi:10.1145/800076.802464.
- 16 Sharma V. Thankachan, Sriram P. Chockalingam, and Srinivas Aluru. An efficient algorithm for finding all pairs k-mismatch maximal common substrings. In *Bioinformatics Research and Applications - 12th International Symposium, ISBRA 2016, Minsk, Belarus, June 5-8, 2016, Proceedings*, pages 3–14, 2016.
- 17 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.
- 18 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.
- 19 Gelin Zhou. Two-dimensional range successor in optimal time and almost linear space. *Inf. Process. Lett.*, 116(2):171–174, 2016. doi:10.1016/j.ipl.2015.09.002.
- 20 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.




# Superstrings with multiplicities

**Bastien Cazaux**

Department of Computer Science, University of Helsinki, Helsinki, Finland;  
L.I.R.M.M., Université Montpellier, Montpellier, France & Institute of Computational Biology,  
Montpellier, France  
bastien.cazaux@cs.helsinki.fi

**Eric Rivals**

L.I.R.M.M., Université Montpellier, Montpellier, France & Institute of Computational Biology,  
Montpellier, France  
rivals@lirmm.fr  
 <https://orcid.org/0000-0003-3791-3973>

---

## Abstract

A superstring of a set of words  $P = \{s_1, \dots, s_p\}$  is a string that contains each word of  $P$  as substring. Given  $P$ , the well known Shortest Linear Superstring problem (SLS), asks for a shortest superstring of  $P$ . In a variant of SLS, called Multi-SLS, each word  $s_i$  comes with an integer  $m(i)$ , its multiplicity, that sets a constraint on its number of occurrences, and the goal is to find a shortest superstring that contains at least  $m(i)$  occurrences of  $s_i$ . Multi-SLS generalizes SLS and is obviously as hard to solve, but it has been studied only in special cases (with words of length 2 or with a fixed number of words). The approximability of Multi-SLS in the general case remains open. Here, we study the approximability of Multi-SLS and that of the companion problem Multi-SCCS, which asks for a shortest cyclic cover instead of shortest superstring. First, we investigate the approximation of a greedy algorithm for maximizing the compression offered by a superstring or by a cyclic cover: the approximation ratio is  $1/2$  for Multi-SLS and 1 for Multi-SCCS. Then, we exhibit a linear time approximation algorithm, Concat-Greedy, and show it achieves a ratio of 4 regarding the superstring length. This demonstrates that for both measures Multi-SLS belongs to the class of APX problems.

**2012 ACM Subject Classification** Mathematics of computing → Discrete mathematics, Theory of computation → Approximation algorithms analysis

**Keywords and phrases** greedy algorithm, approximation, overlap, cyclic cover, APX, subset system

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.21

**Funding** This work is supported by the Institut de Biologie Computationnelle <http://www.ibc-montpellier.fr> (ANR-11-BINF-0002) and Défi MASTODONS C3G <http://www.lirmm.fr/~rivals/C3G/> from CNRS.

**Acknowledgements** We thank for the reviewers for their comments.

## 1 Introduction

Given a set of  $p$  words  $P := \{s_1, s_2, \dots, s_p\}$  over a finite alphabet  $\Sigma$ , a superstring of  $P$  is a string containing each  $s_i$  for  $1 \leq i \leq p$  as a substring. The **Shortest Linear Superstring (SLS)** problem is an optimization problem that asks for a superstring of  $P$  of minimal length. It is also known as the Shortet Common Superstring problem, which does not convey the fact that the output superstring is a linear rather than cyclic word. SLS has been studied in depth for its applications in data compression, where a superstring is an alternative representation



© Bastien Cazaux and Eric Rivals;  
licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 21; pp. 21:1–21:16

Leibniz International Proceedings in Informatics



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

of  $P$ , and in bioinformatics [11]. SLS is known to be hard to solve (**NP-hard** provided the input words are of length at least three) and to approximate (**MAX-SNP-hard**), and these difficulties remain even if one considers instances over a binary alphabet [10, 3, 17]. In bioinformatics, SLS models the initial step of genome assembly in a shotgun sequencing approach [1], whose input is a large and redundant set of "reads". This first step consists in merging overlapping words to obtain partial substrings of the target genome. These output strings are called *contigs*. In practice, one never obtains a single superstring covering the genome, but a large set of contigs. A major difficulty that is inherent to biology comes from the presence of repeated regions in genomes. When assembled, the distinct copies of a repeat tend to collapse into a single occurrence, and the corresponding contig then exhibits a higher density of merged words [1]. By comparing the local density of a contig to the expected density, one can estimate the underlying number of copies for a repeat. The assembly process can then be rerun using these *multiplicities*, that is for each word, the number of times it must appear in the superstring. To take into account the issue of repeated regions in SLS, Crochemore et al. have proposed a variant of SLS called **Multi-SLS**<sup>1</sup>: the input consists in  $P$  with a function  $m$  giving the multiplicity of each word of  $P$ , and the output *multi superstring* must contain at least  $m(s_i)$  occurrences of  $s_i$ , for any  $1 \leq i \leq p$  [8]. They present two polynomial time algorithms to solve two special cases of **Multi-SLS**: First, the case where the number of input words is constant, and second the case where each input word has length 2. The latter generalizes SLS for words of length 2, which can also be solved in polynomial time [10].

**Contributions** To our knowledge, the approximability of **Multi-SLS** in the general case (*i.e.*, with an unbounded number of words of length  $\geq 2$ ) is wide open. As for SLS, two measures can be considered: the superstring length or its compression – the superstring length minus the sum of the lengths of all required occurrences of words of  $P$ . In general, for an optimization problem  $\mathcal{P}$ , we denote by  $\mathcal{P}_{\text{comp}}$  the related problem that maximizes the compression measure.

► **Example 1.** Consider the instance  $(P, m)$  with  $P := \{aab, abaa, baba\}$  with multiplicities  $m(aab) = 2$ ,  $m(abaa) = 1$ , and  $m(baba) = 2$ . Then  $w := aabaabababa$  is a multi superstring of  $(P, m)$ , it has length 11 and achieves of compression of 9 symbols. Similarly, the string  $y := aabaababaababababa$ , which results from concatenating the required words, also is a multi superstring of  $(P, m)$  of length 18 and thus yields a compression of 0.

In Section 3, we study the greedy algorithm for **Multi-SLS**<sub>comp</sub> and show it has a compression ratio of 1/2 in Theorem 9. In Section 5, we propose for **Multi-SLS** the first polynomial time approximation algorithm, called **Concat-Greedy**, and prove in Theorem 15 that it admits an approximation ratio of 4 for the superstring length measure. Hence, we demonstrate:

► **Theorem 2.** *Both **Multi-SLS** and **Multi-SLS**<sub>comp</sub> belong to the class **APX**.*

In fact, the ratio of 4 follows from a stronger bound on the length of a solution of **Concat-Greedy** (see Proposition 14 p. 11). Note that the same ratio of 4 was proven for the classical SLS problem by [3] in 1994, using the **Concat-Cycles** algorithm. To achieve this bound, **Concat-Greedy** must solve a related problem called **Multi-SCCS**, where the

---

<sup>1</sup> According to the notation of [8], this variant was termed **MULTI-SCS**.

solution is a set of cyclic strings that collectively contain all the required occurrences of words of  $P$ . Such a set is called a *cyclic cover of strings*, or *cyclic cover* for short. First, we show in Section 3 that a greedy algorithm solves exactly **Multi-SCCS**, and then exhibit in Section 4 a graph based algorithm for it and bound its time complexity, which yields:

► **Theorem 3.** *The **Multi-Greedy** algorithm (Algo. 2) solves the **Multi-SCCS** problem in a time that is linear in the size of its output.*

## 2 Preliminaries

Here, we introduce basic notions on strings, permutations, superstrings and formally define the two problems **Multi-SLS** and **Multi-SCCS**. Then, we derive a logical, but important fact: all multi superstrings (resp. multi cyclic cover) we need to consider are induced by permutations. For any finite set  $U$ ,  $|U|$  denotes its cardinality.

**About strings.** Let  $u, v$  be two linear strings. We denote by  $|u|$  the length of  $u$ , and by  $uv$  their concatenation. Given a linear string  $u$ , we obtain the *circular string*  $\langle u \rangle$  by linking the last letter of the linear string  $u$  to its first letter. The length of the circular string  $\langle u \rangle$  is the length of the linear string  $u$ . Given a set of linear or circular strings  $P$ , we call the *norm* of  $P$ , denoted by  $\|P\|$ , *i.e.* the sum of the lengths of the strings of  $P$ .

Let  $x := x_1 \dots x_n$  and  $y := y_1 \dots y_m$  be two linear strings (where for any  $1 \leq j \leq m$ ,  $y_j$  is the  $j^{\text{th}}$  letter of  $y$ ). We denote by  $\text{Occ}(y, x)$  the set of the occurrences of  $y$  in  $x$ , *i.e.*, the set of positions  $i$  between 1 and  $n - m + 1$  such that  $x_i \dots x_{i+m-1} = y_1 \dots y_m$ . Whenever  $\text{Occ}(y, x)$  is not empty,  $y$  is said to be a *substring* of  $x$ . We extend the notion of substring to circular strings by extending the set of occurrences: we denote by  $\text{Occ}(y, \langle x \rangle)$  the set  $\text{Occ}(y, x^\infty) \cap \{1, \dots, |x|\}$  (where  $x^\infty = xx \dots$ ). A *prefix*  $y$  (respectively a *suffix*) of a linear string  $x$  is a substring beginning (respectively ending)  $x$ , *i.e.*,  $1 \in \text{Occ}(y, x)$  (resp.  $|x| - |y| + 1 \in \text{Occ}(y, x)$ ). Furthermore, we say that  $y$  is a *proper* substring of  $x$  if  $|y| < |x|$  (Definitions of a proper prefix/suffix are similar). Let  $M$  be a set of linear or circular strings; we denote by  $\text{Occ}(x, M)$  the set of all occurrences of  $x$  in all strings of  $M$ .

**Problem definitions.** Throughout the article, let  $P := \{s_1, \dots, s_p\}$  be a set of linear strings  $P$  and a function  $m$  from  $P$  to  $\mathbb{N}^*$  giving the multiplicity of each string. We assume that  $P$  is factor-free, *i.e.*,  $s_i$  is not a substring of  $s_j$  for any  $i, j$  in  $\{1, \dots, p\}$ . The pair  $(P, m)$  is the input of the problems **Multi-SLS** and **Multi-SCCS**. A *superstring* of  $P$  is a word  $w$  such that for any  $1 \leq i \leq p$ ,  $|\text{Occ}(s_i, w)| \geq 1$ .

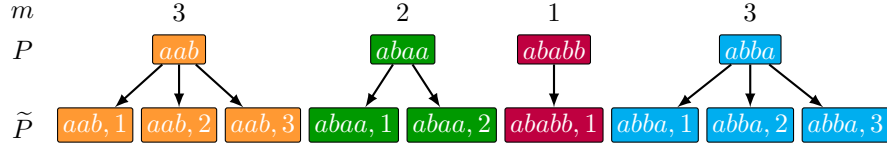
Let us define formally the two minimization problems **Multi-SLS** and **Multi-SCCS**: both seek to minimize their output length. Note that Definition 4 is equivalent to that **MULTI-SCS(k)** from [8].

► **Definition 4** (Multi Shortest Linear Superstring (**Multi-SLS**)). Let  $P := \{s_1, \dots, s_p\}$  be a set of strings and  $m$  a function from  $P$  to  $\mathbb{N}^*$ . It seeks a linear string  $w$  of minimal length and such that for all  $s_i \in P$ ,  $|\text{Occ}(s_i, w)| \geq m(s_i)$ .

► **Definition 5** (Multi Shortest Cyclic Cover of Strings (**Multi-SCCS**)). Let  $P := \{s_1, \dots, s_p\}$  be a set of strings and  $m$  a function from  $P$  to  $\mathbb{N}^*$ . It seeks a set  $C$  of circular strings of minimal norm and such that for all  $s_i \in P$ ,  $|\text{Occ}(s_i, C)| \geq m(s_i)$ .

In any solution of **Multi-SLS** or of **Multi-SCCS**, each string  $s$  of  $P$  must occur at least  $m(s)$  times. Let us define  $\tilde{P}$  to be the set containing  $m(s)$  copies of each word  $s$  of  $P$ ; to

## 21:4 Superstrings with multiplicities



■ **Figure 1** Example of  $\tilde{P}$  for the instance  $(P, m)$  of Example 6. Due to space constraints, for a pair of  $\tilde{P}$  we may write  $\boxed{aab, 2}$  or  $\boxed{2}$ .

distinguish its copies we denote any element of  $\tilde{P}$  by a pair  $(s, i)$  for  $1 \leq i \leq m(s)$  – see Example 6 and Figure 1. Formally, *i.e.*

$$\tilde{P} = \bigcup_{s \in P} \left( \bigcup_{i=1}^{m(s)} \{(s, i)\} \right).$$

For an element  $(s, i)$  of  $\tilde{P}$ , we denote by  $\text{word}((s, i))$  the word  $s$  of  $P$ , *i.e.*,  $\text{word}((s, i)) = s$ .

Note that for some instances – when words of  $P$  do not overlap each other – an optimal solution for **Multi-SLS** is the concatenation of all strings in  $\tilde{P}$ , and has length  $\|\tilde{P}\| := \sum_{i=1}^p m(s_i) |s_i|$ . This observation remains valid for **Multi-SCCS**. Any algorithm solving **Multi-SLS** or **Multi-SCCS** has its complexity bounded by the length of its output, *i.e.*, by  $\|\tilde{P}\|$ , which we consider to be linear in the input size. In [8], the authors seek to find a compressed representation of the output; we dwell on this question on page 10.

► **Example 6** (see Figure 1). This same instance  $(P, m)$  is used as running example throughout the paper. Let  $P = \{aab, abaa, ababb, abba\}$  be a set of strings and  $m$  be the function from  $P$  to  $\mathbb{N}^*$  such that  $m(aab) = 3$ ,  $m(abaa) = 2$ ,  $m(ababb) = 1$  and  $m(abba) = 3$ . We have that

$$\tilde{P} = \{(aab, 1), (aab, 2), (aab, 3), (abaa, 1), (abaa, 2), (ababb, 1), (abba, 1), (abba, 2), (abba, 3)\}.$$

**About permutations.** Given a permutation  $\sigma$  of a set  $E$ , a *successor*  $y$  of an element  $x$  of  $E$  by  $\sigma$ , is an element of  $E$  such that  $y = \sigma^k(x)$  where  $\sigma^1(x) = \sigma(x)$  and  $\sigma^k(x) = \sigma^{k-1}(\sigma(x))$ . We denote by  $\text{Part}(E, \sigma)$  the partition  $\{E_1, \dots, E_p\}$  of  $E$  where each element of  $E$  and its successors are in the same subset  $E_i$ . A permutation is said *circular* if all the elements of  $E$  are successors of any element of  $E$ , *i.e.*  $\text{Part}(E, \sigma) = \{E\}$ . Moreover, we denote by  $\text{Decomp}(E, \sigma)$  the decomposition into circular permutations of the permutation  $\sigma$ , *i.e.*, the set of pairs  $(E_i, \sigma_i)$  where  $E_i \in \text{Part}(E, \sigma)$  and where  $\sigma_i$  is the restriction of  $\sigma$  to the elements of  $E_i$ .

**About linear and circular superstrings.** Given two linear strings  $u$  and  $v$ , an *overlap* from  $u$  over  $v$  is a linear string that is a proper suffix of  $u$  and a proper prefix of  $v$ . We denote by  $\text{ov}(u, v)$  the longest overlap from  $u$  to  $v$  (also termed maximal overlap). Overlaps are not symmetrical. The *prefix from  $u$  to  $v$* , denoted by  $\text{pr}(u, v)$  is the string satisfying  $u = \text{pr}(u, v)\text{ov}(u, v)$ . The *merge* from  $u$  to  $v$  is the linear string  $\text{pr}(u, v)v$  if  $u \neq v$ , and the circular string  $\langle \text{pr}(u, u) \rangle$  otherwise. Given a set of strings  $P$ , we denote by  $\text{Ov}(P)$  the set of all the maximal overlaps between any two strings of  $P$ .

Let  $P = \{s_1, \dots, s_p\}$  be a set of linear strings. We denote by  $\text{Linear}(s_1, \dots, s_p)$  (resp. by  $\text{Circular}(s_1, \dots, s_p)$ ) the linear (resp. circular) string defined by the merge of  $s_1, \dots, s_p$  in this order:

$$\text{Linear}(s_1, \dots, s_p) := \text{pr}(s_1, s_2)\text{pr}(s_2, s_3) \dots \text{pr}(s_{p-1}, s_p)s_p$$



and

$$\text{Circular}(s_1, \dots, s_p) := \langle \text{pr}(s_1, s_2) \text{pr}(s_2, s_3) \dots \text{pr}(s_{p-1}, s_p) \text{pr}(s_p, s_1) \rangle.$$

► **Remark.** The starting point of the merge does not impact  $\text{Circular}()$ . Formally, for all  $j \in \{1, \dots, p\}$ ,  $\text{Circular}(s_1, \dots, s_p) = \text{Circular}(s_j, \dots, s_p, s_1, \dots, s_{j-1})$ .

**About multi superstrings and multi cyclic covers induced by a permutation.** The number of possible superstrings or cyclic covers of  $\tilde{P}$  is infinite, which makes the search space for **Multi-SLS** / **Multi-SCCS** unpractical. Hence, a crucial issue is whether we can restrict this search space. For this sake, we introduce the notion of multi superstring/cyclic cover induced by a permutation.

Let  $\tau$  be a permutation of  $\tilde{P}$ . If  $\tau$  is a circular permutation (meaning that all its elements are successors of each other), we can define the *multi superstring induced by  $\tau$*  and by an element  $\tilde{s}$  of  $\tilde{P}$  as follows:

$$\text{Lin}(\tilde{P}, \tau, \tilde{s}) = \text{Linear}(\text{next\_word}(\tilde{s}, 1), \dots, \text{next\_word}(\tilde{s}, |\tilde{P}|))$$

where  $\text{next\_word}(\tilde{s}, k) = \text{word}(\tau^k(\tilde{s}))$ . Here, the term  $\text{Linear}()$  of this equation is the merge of the words of  $\tilde{P}$  in the order given by  $\tau$  and ending with the chosen element  $\tilde{s}$  (indeed,  $\text{next\_word}(\tilde{s}, |\tilde{P}|) = \text{word}(\tilde{s})$ ).

In general,  $\tau$  is not circular. It can be decomposed in several circular permutations (see Fig. 2a); we denote its decomposition by  $\text{Decomp}(\tilde{P}, \tau)$ . We define,  $\text{CC}(\tilde{P}, \tau)$ , the *multi cyclic cover of strings induced by  $\tau$*  as follows:

$$\text{CC}(\tilde{P}, \tau) = \bigcup_{(\tilde{P}_i, \sigma_i) \in \text{Decomp}(\tilde{P}, \tau)} \{\text{Circular}(\text{next\_word}(\tilde{s}, 1), \dots, \text{next\_word}(\tilde{s}, |\tilde{P}_i|))\}$$

where  $\tilde{s}$  is any element of  $\tilde{P}_i$  and  $\text{next\_word}(\tilde{s}, k) = \text{word}(\sigma_i^k(\tilde{s}))$ .  $\text{CC}(\tilde{P}, \tau)$  is a set of cyclic strings, each obtained by merging the words in the order given by a sub-permutation  $\sigma_i$ .

► **Example 7.** Let  $\sigma_1$  and  $\sigma_2$  be the permutations of  $\tilde{P}$  of Figure 2a and Figure 2b. Consider the pair  $(\text{abba}, 3)$  in  $\tilde{P}$  (node 3 in figures 2a and b); its direct successor with  $\sigma_1$  is itself, *i.e.*,  $\sigma_1((\text{abba}, 3)) = (\text{abba}, 3)$ , and with  $\sigma_2$ , it is the node 1 in Figure 2b, *i.e.*,  $\sigma_2((\text{abba}, 3)) = (\text{ababb}, 1)$ .

Some thoughts lead to the observation that any optimal multi superstring or multi cyclic cover is necessarily induced by a permutation on  $\tilde{P}$ . This yields this proposition, which indeed restricts the search spaces of both problems. Due to space constraints, the proofs of some results (marked with a  $\star$ ) are not included here; some proofs are given in the appendix.

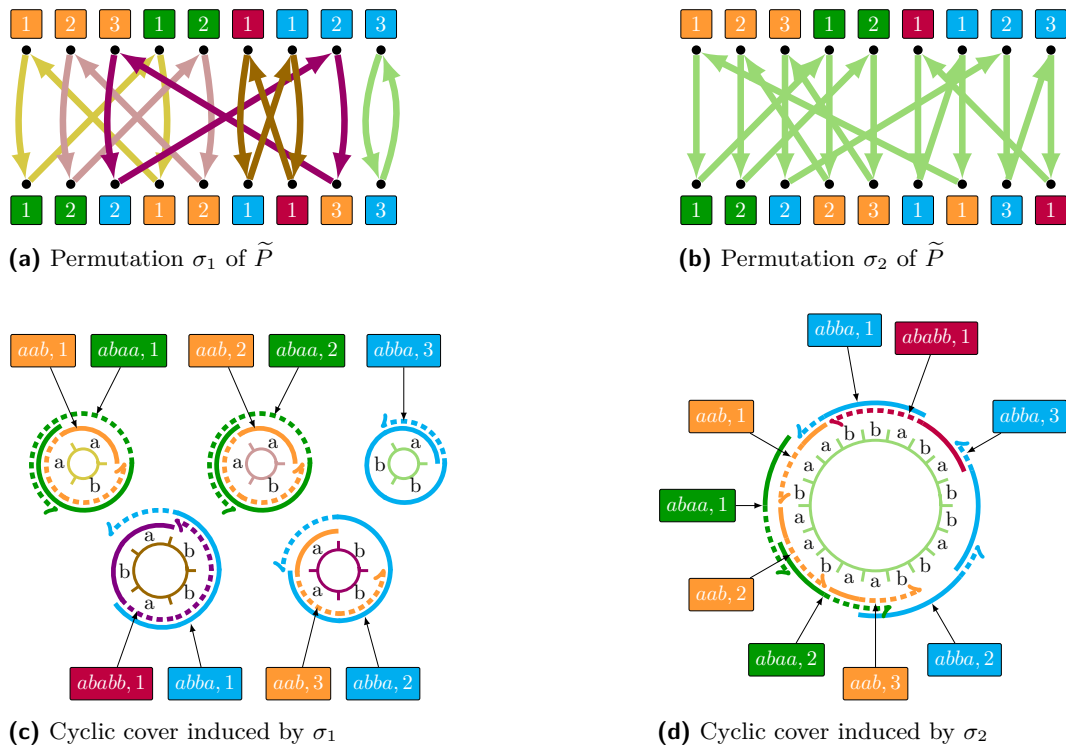
► **Proposition 8 ( $\star$ ).** *Let  $(P, m)$  be an instance of **Multi-SLS** and of **Multi-SCCS**. Let  $w_{\text{opt}}$  be an optimal solution of **Multi-SLS** and let  $C_{\text{opt}}$  be an optimal solution of **Multi-SCCS**. Then, there exist*

1. a permutation  $\tau$  of  $\tilde{P}$  such that  $C_{\text{opt}} = \text{CC}(\tilde{P}, \tau)$ .
2. a circular permutation  $\varphi$  of  $\tilde{P}$  and an element  $\tilde{s}$  of  $\tilde{P}$  such that  $w_{\text{opt}} = \text{Lin}(\tilde{P}, \varphi, \tilde{s})$ .

### 3 Approximation

Here, let us define the greedy algorithms for **Multi-SLS** and **Multi-SCCS** problems and exhibit their approximation ratios for the measure of compression.

21:6 Superstrings with multiplicities



■ **Figure 2** Running example: two possible permutations of  $\tilde{P}$  (Fig. a & b), and the cyclic covers induced by these permutations (Fig. c and d). Permutation  $\sigma_1$  in (a) is decomposed in 5 circular permutations (five colors in a) and induces 5 cyclic strings (c), while permutation  $\sigma_2$  cannot be decomposed and induces a single cyclic string (d). In (c, d) input words are drawn as arrows around the cyclic strings, and the dashed part represents the overlap with the successor.

**Greedy algorithms.** By Proposition 8, we have that each optimal solution of Multi-SCCS can be induced by a permutation on  $\tilde{P}$ . We can generalize the greedy algorithm for SCCS [5] to Multi-SCCS.

The basic principle of the greedy algorithm for SLS or SCCS is 1/ to merge a pair of strings at each step until all merge possibilities have been exhausted, and 2/ to consider pairs of strings to be merged in order of decreasing overlap length, and 3/ to break ties randomly. It is greedy because it chooses merge operations that yield the best compression first, and never backtracks on these choices. In fact, the greedy algorithm determines a total ordering on the merge operations (it is the greedy algorithm of a precise subset system – see [6] for details). In stringology, the greedy algorithm is usually presented as in Algorithm 1: the initial set of words (set  $Q$  in Algorithm 1) is iteratively modified at each iteration of the main loop: a pair of strings is chosen, those strings removed from the set, and the string resulting from the merge is (re-)inserted in the set. The two formulations of the algorithm are equivalent [6], basically because the new string offers the same overlaps with remaining words as the strings that were merged.

Of course the algorithm differs between the linear and cyclic cases. For SLS or Multi-SLS, the loop merges pairs of words until getting a single linear string, which is the final result. For SCCS or Multi-SCCS, the result is a set of cyclic strings, which is iteratively built (solution set  $S$ ). A merge of two linear string results in a linear string, but the merge of a single string that self-overlaps yields a cyclic string. A cyclic string has no overlap and cannot be merged.

---

**Algorithm 1:** The greedy algorithm for Multi-SCCS.
 

---

```

1 Input: a pair  $(P, m)$ ; Output:  $S$ : a cyclic cover of strings covering  $\tilde{P}$ ;
2  $S := \emptyset$ ; // the solution set in construction
3  $Q := \tilde{P}$ ;
4 newIndex :=  $|Q|$ ;
5 while  $|Q| > 0$  do
6    $(u, i)$  and  $(v, j)$  two elements of  $Q$  such that  $u$  and  $v$  have the longest overlap;
   //  $u$  can be equal to  $v$  and  $i$  equal to  $j$ 
7    $w$  is the merge of  $u$  and  $v$ ;
8    $Q := Q \setminus \{(u, i), (v, j)\}$ ;
9   if  $u = v$  and  $i = j$  (i.e.,  $w$  is a cyclic string) then  $S := S \cup \{w\}$ ;
10  else  $Q := Q \cup \{(w, \text{newIndex}++)\}$ ;
11 return  $S$ 

```

---

Hence, each cyclic string is directly inserted into the solution set (set  $S$ , line 9), while a linear string is re-inserted in the set of strings remaining to be merged (set  $Q$ , line 10). This explains why the loop condition is  $|Q| > 0$  (line 5).

► **Remark.** Algorithm 1 is equivalent to iteratively merging the two elements  $u$  and  $v$  of  $\tilde{P}$  having the longest overlap, provided that  $u$  is not merged on its right<sup>2</sup> more than  $m(u)$  times and  $v$  is not merged on its left more than  $m(v)$  times. The word that results from the merge is inserted back into  $Q$  when it is linear, and inserted in the solution set  $S$  if it is cyclic. As elements of  $Q$  are pairs, we number each inserted word with a variable newIndex that is incremented on line 9.

We can also generalize the greedy algorithm for Multi-SCCS to Multi-SLS. To do so, we just need to change in Algorithm 1, the while condition " $|Q| > 0$ " by " $|Q| > 1$ " and, on line 6 "**and  $i$  equal to  $j$** " by "**but  $i$  cannot be equal to  $j$** ".

**Measure of compression.** For the both problems Multi-SLS and Multi-SCCS, we want to minimize the length of the multi superstring or the norm of the multi cyclic cover of strings. If instead, we want to maximize the compression, that is the difference between the norm of  $\tilde{P}$  and the output size, we call the corresponding problems Multi-SLS<sub>comp</sub> and Multi-SCCS<sub>comp</sub>.

As the size of the input is constant, all optimal solutions of Multi-SCCS are also optimal solutions of Multi-SCCS<sub>comp</sub>, and vice versa. The set of optimal solutions of Multi-SLS is also equal to the set of optimal solutions of Multi-SLS<sub>comp</sub>. By Proposition 8, as we can restrict to solutions induced by a permutation of  $\tilde{P}$ , the compression can be seen as the sum of the lengths of the overlaps between two successive strings in the permutation. Indeed, for a permutation  $\tau$  of  $\tilde{P}$ ,

$$\|\tilde{P}\| - |\text{CC}(\tilde{P}, \tau)| = \sum_{(\tilde{P}_i, \sigma_i) \in \text{Decomp}(\tilde{P}, \tau)} \left( \sum_{j=1}^{|\tilde{P}_i|} |\text{ov}(\text{next\_word}(\tilde{s}, j), \text{next\_word}(\tilde{s}, j+1))| \right)$$

where  $\tilde{s} \in \tilde{P}_i$  and  $\text{next\_word}(\tilde{s}, k) = \text{word}(\sigma_i^k(\tilde{s}))$ . Similarly, we get that Multi-SLS<sub>comp</sub>

---

<sup>2</sup> Merged on its right (resp. left) means using an overlap of its suffix (resp. prefix).

maximizes the sum of the lengths of the successive overlaps in a multi superstring induced by a permutation.

**Approximation for compression.** We can see the greedy algorithm for SLS (and  $\text{SLS}_{\text{comp}}$ ) as the greedy algorithm for finding a maximum weighted Hamiltonian path (*Maximum Asymmetric Travelling Salesman Problem – Max-ATSP*) in the overlap graph [15]. The overlap graph is a complete digraph labelled on the arcs, where each input word is a node, and where the length of the maximal overlap between two words is a weight on the corresponding arc [3]. Theorem 9 generalizes the half compression of greedy algorithm for SLS from [15] to **Multi-SLS** (full proof in the Appendix).

► **Theorem 9.** *The greedy algorithm for  $\text{Multi-SLS}_{\text{comp}}$  has a  $\frac{1}{2}$  approximation ratio.*

**Proof.** (See details in Appendix.) In [6], we show that one can prove the approximation ratio of the greedy algorithm for  $\text{SLS}_{\text{comp}}$  by combining the Monge inequality [14] with subset systems that simulate the greedy algorithm for **Max-ATSP** in graphs [13]. By building the overlap graph for  $\tilde{P}$  (see Figure 4a), we can use the same subset system on the maximal overlaps of  $\tilde{P}$  and obtain the same approximation ratio for the greedy algorithm of  $\text{Multi-SLS}_{\text{comp}}$  as for that of  $\text{SLS}_{\text{comp}}$ . ◀

With the same arguments, we can show that the approximation ratio of the greedy algorithm for  $\text{Multi-SCCS}_{\text{comp}}$  equals that of the greedy algorithm for  $\text{SCCS}_{\text{comp}}$ , which is 1 [6]. This yields Theorem 10.

► **Theorem 10.** *For both problems  $\text{Multi-SCCS}_{\text{comp}}$  and  $\text{Multi-SCCS}$ , the greedy algorithm (Algorithm 1) yields an optimal solution.*

By Proposition 8 and by the fact that greedy solutions for **Multi-SCCS** are optimal, we can represent each greedy solution by a permutation of  $\tilde{P}$ . For any instance  $(P, m)$ , let  $\text{GreedyPerm}(\tilde{P})$  denote the set of permutations of  $\tilde{P}$  corresponding to greedy solutions for **Multi-SCCS**.

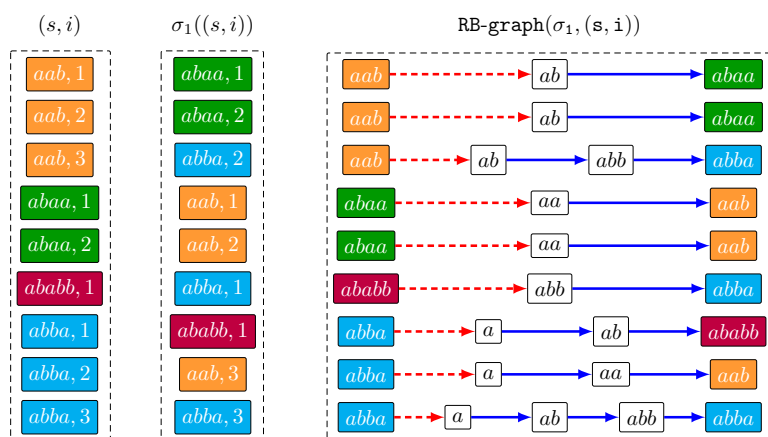
## 4 Linear construction of Multi-SCCS

In this section, we show how to compute a greedy solution for **Multi-SCCS** in linear time in the norm of the set of strings of the input and in the norm of an optimal solution of **Multi-SCCS**. To achieve this, we adapt the superstring graph [5] in order to model greedy solutions for **Multi-SCCS**. Now, assume that one stores  $m(w)$ , the multiplicity of a string  $w$ , in constant space ( $O(1)$  bits); hence the input,  $(P, m)$ , has size  $O(\|P\|)$ .

**Red-Blue graphs.** To begin with, we define the Red-Blue graphs, which are intermediate digraphs needed to define the multi superstring graph – see Figure 3 (or Figure 6 in appendix). Let  $\tau$  be a permutation for  $\tilde{P}$  and  $\tilde{s}$  an element of  $\tilde{P}$ . We define,  $\text{RB-Graph}(\tau, \tilde{s}) := (V, R, B)$ , the Red-Blue graph of  $\tilde{s}$  for the permutation  $\tau$  as

$$\begin{aligned} V &= \{\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s}))\} \cup \{y \in \text{Ov}(P) : |y| \geq |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))|, \text{ and} \\ &\quad (y \text{ suffix of } \text{word}(\tilde{s}) \text{ or } y \text{ prefix of } \text{word}(\tau(\tilde{s})))\}, \\ R &= \{(u, v) \in V \times V \mid v \text{ is the longest proper suffix of } u \text{ in } V\}, \\ B &= \{(u, v) \in V \times V \mid u \text{ is the longest proper prefix of } v \text{ in } V\}. \end{aligned}$$

By the properties of prefixes/suffixes, Red-Blue graphs are path graphs, which we illustrate in Figure 3 (running example and permutation  $\sigma_1$  from Fig. 2a). Note that a Red-Blue graph of  $\tilde{s}$  depends on  $\text{Ov}(P)$ : it may contain a suffix/prefix that is an overlap of another pair of



■ **Figure 3** Running example: set of all the Red-Blue graphs of  $(s, i) \in \tilde{P}$  for the permutation  $\sigma_1$  (see Figure 2a). A dashed arc (in red) links a string to its longest proper suffix, while a plain arc (in blue) links a longest proper prefix of a string to this string.

words  $(\in \{\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})) \mid \tilde{s} \in \tilde{P}\})$ . In Figure 3, it happens on the graph for the pair  $aab$  to  $abba$  since  $abb$  is not their maximal overlap.

Let  $u$  and  $v$  be in  $P \cup \text{Ov}(P)$ . By the definition of Red-Blue graphs, the arc linking  $u$  to  $v$  occurs only once in a given Red-Blue graph, *i.e.*,  $|\{(u, v)\} \cap (R \cup B)| \in \{0, 1\}$  (see Lemma 16 in Appendix). We define  $\text{NbOcc}(\tau, (u, v))$  as the number of occurrences of the arc  $(u, v)$  in all Red-Blue graphs for all  $\tilde{s}$  in  $\tilde{P}$ . Thus, we get:

$$\text{NbOcc}(\tau, (u, v)) := \sum_{\substack{\tilde{s} \in \tilde{P} \\ (V, R, B) = \text{RB-Graph}(\tau, \tilde{s})}} |\{(u, v)\} \cap (R \cup B)|.$$

Furthermore, we define  $\text{PrefixArc}(P)$  (resp.  $\text{SuffixArc}(P)$ ), as the set of arcs  $(u, v)$  (resp.  $(v, u)$ ) of  $(P \cup \text{Ov}(P))^2$  such that  $u$  is the longest prefix (resp. suffix) of  $v$  in  $P \cup \text{Ov}(P)$ .

For a permutation  $\tau$  of  $\tilde{P}$  that corresponds to a greedy solution for **Multi-SCCS**, we can count the  $\text{NbOcc}(\tau, (u, v))$  for all  $(u, v) \in \text{PrefixArc}(P) \cup \text{SuffixArc}(P)$ . For the sake of simplicity, we extend the function  $m$  to elements of  $\text{Ov}(P)$  and set:  $m(w) = 0$  for any  $w$  in  $\text{Ov}(P)$ .

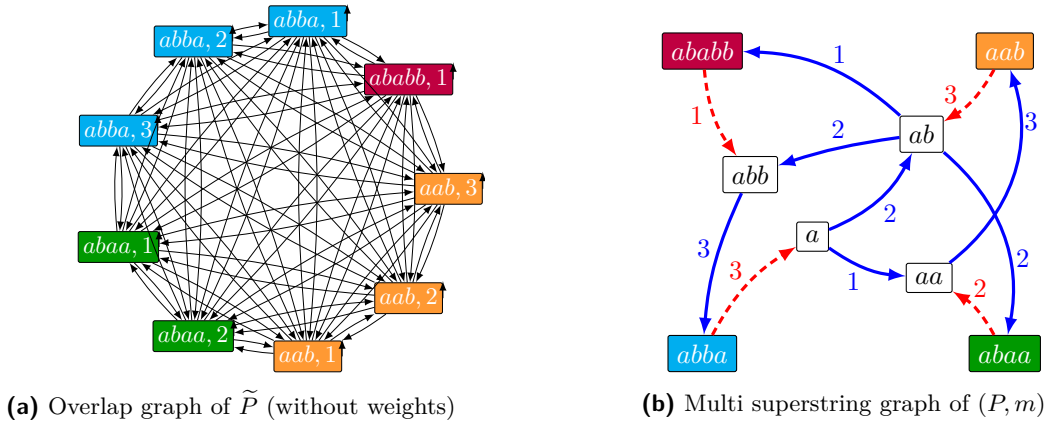
► **Proposition 11.** *Let be  $\tau \in \text{GreedyPerm}(\tilde{P})$  and  $(u, v) \in \text{PrefixArc}(P) \cup \text{SuffixArc}(P)$ . We have that*

$$\text{NbOcc}(\tau, (u, v)) = \begin{cases} \text{Max}(m(v), -a(v)) & \text{if } |u| \leq |v| \\ \text{Max}(m(u), a(u)) & \text{if } |u| > |v| \end{cases}$$

$$\text{where } a(w) = \sum_{(w', w) \in \text{SuffixArc}(P)} \text{NbOcc}(\tau, (w', w)) - \sum_{(w, w') \in \text{PrefixArc}(P)} \text{NbOcc}(\tau, (w, w')).$$

**Multi superstring graph.** Let  $\tau$  be a permutation of  $\tilde{P}$ . We define  $G_p(\tau) := (V, R, B, l)$  as the graph labelled on its arcs, which results from the merge of all Red-Blue graphs for all elements of  $\tilde{P}$  and for permutation  $\tau$ . Formally:

$$\begin{aligned} V &= \text{Ov}(P) \setminus U \\ R &= \{(u, v) \in \text{SuffixArc}(P) \mid \text{NbOcc}(\tau, (u, v)) \neq 0\} \\ B &= \{(u, v) \in \text{PrefixArc}(P) \mid \text{NbOcc}(\tau, (u, v)) \neq 0\} \\ l &: (u, v) \mapsto \text{NbOcc}(\tau, (u, v)) \end{aligned}$$



■ **Figure 4** Running example: overlap graph of  $\tilde{P}$  and multi superstring graph of  $(P, m)$ .

where  $U = \{v \in \text{Ov}(P) \mid v \text{ is not an extremity of an arc of } R \cup B\}$ .

By Proposition 11, we have that for a permutation  $\tau$  of  $\text{GreedyPerm}(\tilde{P})$  and  $(u, v) \in \text{PrefixArc}(P) \cup \text{SuffixArc}(P)$ , the number of occurrences of the arc  $(u, v)$ , *i.e.*  $\text{NbOcc}(\tau, (u, v))$ , is independent of the permutation  $\tau$ . From this observation and arguments from [6], we deduce Proposition 12.

► **Proposition 12** ( $\star$ ). *Let  $\tau_1, \tau_2$  be two permutations of  $\text{GreedyPerm}(\tilde{P})$ . Then,  $G_p(\tau_1) = G_p(\tau_2)$ .*

By Proposition 12, all permutations inducing a greedy solution for an instance of **Multi-SCCS** yield the same graph, which we call the *multi superstring graph* and denote by  $\text{SG}(P, m)$  (see Figure 4b). Using data structures like the (generalised) suffix tree to determine  $\text{Ov}(P)$  [16], and with Proposition 11, we can build the multi superstring graph of  $(P, m)$  recursively and we obtain the following proposition.

► **Proposition 13** ( $\star$ ). *The multi superstring graph can be built in linear time and space in  $\|P\|$ .*

**Linear construction.** By Proposition 11, we know that for  $\text{SG}(P, m) = (V, R, B, l)$  the multi superstring graph of  $(P, m)$  the following equality holds:

$$\forall v \in V, \quad \sum_{(v,u) \in R} l((v,u)) - \sum_{(u,v) \in B} l((u,v)) = \sum_{(u,v) \in R} l((u,v)) - \sum_{(v,u) \in B} l((v,u)).$$

Hence, it follows that the multi superstring graph, in which the label of an arc is seen as a multi-arc, is Eulerian on each of its connected components. In Figure 4b, the arc from *abba* to *a* labelled by 3 means the Eulerian cycle must traverse this arc exactly thrice. Conversely, we can show that every set of cycles covering the multi superstring graph corresponds to a greedy solution for **Multi-SCCS**. As finding an Eulerian cycle cover of  $\text{SG}(P, m)$  takes a time linear in  $\|P\|$ , we deduce Theorem 3 (p. 3).

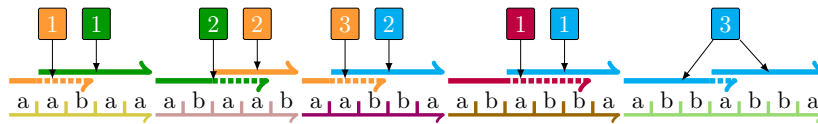
**Compressed output representation.** At the beginning of this section, we have assumed that for each word of  $P$ , we can store the multiplicity in constant space. To improve the complexity, in this paragraph we assume that we can store the multiplicity of each string in  $O(\|P\|)$  bits. In [8], the authors present a compact representation of a solution for **Multi-SLS**

---

**Algorithm 2:** The **Multi-Greedy** algorithm for **Multi-SCCS**

---

- 1 **Input:** a pair  $(P, m)$ . **Output:**  $W$  a greedy solution for **Multi-SCCS**;
  - 2 build  $\text{SG}(P, m)$  the multi superstring graph of  $(P, m)$ ;
  - 3 compute an Eulerian multi-cycle  $c = (c_1, \dots, c_n)$  of  $G_P$ ;
  - 4 **for**  $j \in [1, n]$  **do**
  - 5     traverse  $c_j$ : list the words of  $P$  whose node is in  $c_j$  and insert the cyclic string of the concatenation of the corresponding prefixes in  $W$ ;
  - 6 **return**  $W$
- 



■ **Figure 5** Running example: linearization  $\text{LinCC}(\tilde{P}, \sigma_1, W)$  of a cyclic cover of strings induced by permutation  $\sigma_1$  (see Figure 2a) for  $W := (\text{abaa}, 1, \text{aab}, 2, \text{abba}, 2, \text{abba}, 1, \text{abba}, 3)$ .

with strings of length 2. They show that this compact representation has a size in  $O(\|P\|^2)$  and can be computed in  $O(\|P\|^2)$  time.

We can apply their technique to the multi superstring graph defined for **Multi-SCCS**. First, build the multi superstring graph of  $(P, m)$ , and then using the algorithm *EulerianCycle* from [8] on  $\text{SG}(P, m)$ , compute a compact representation of a multi cyclic cover of size  $O(\|P\|^2)$  in  $O(\|P\|^2)$  time. Now, as any connected component of  $\text{SG}(P, m)$  can be represented just by a permutation and its first element, one gets a compact representation of size  $O(\|P\| \times |P|)$ , therefore improving on [8].

## 5 Approximation algorithm for Multi-SLS

Now, we propose an approximation algorithm for **Multi-SLS** and derive its approximation ratio with respect to the multi superstring length. By Theorem 9, we know that the greedy algorithm for **Multi-SLS<sub>comp</sub>** has an approximation ratio of 1/2, and thus it belongs to **APX**. Here, we extend the **Concat-Cycles** algorithm from [3] and we show that this new algorithm, called **Concat-Greedy**, has an approximation ratio of 4 for **Multi-SLS**. The idea is to build an Eulerian multi-cycle of the multi superstring graph of  $(P, m)$ , to break each cycle and merge its words to create linear strings, and to concatenate all these linear strings in an arbitrary order. Figure 5 displays an example of linearization.

To define formally the linearization of a cyclic cover of strings induced by permutation  $\tau$  of  $\tilde{P}$ , we denote  $\text{LinCC}(\tilde{P}, \tau, (w_1, \dots, w_p))$  the following linearization

$$\text{LinCC}(\tilde{P}, \tau, (w_1, \dots, w_p)) = \text{Lin}(\tilde{P}_1, \sigma_1, w_1) \dots \text{Lin}(\tilde{P}_p, \sigma_p, w_p)$$

where  $\text{Decomp}(\tilde{P}, \tau) = \{(\tilde{P}_1, \sigma_1), \dots, (\tilde{P}_p, \sigma_p)\}$  and  $(w_1, \dots, w_p) \in \tilde{P}_1 \times \dots \times \tilde{P}_p$ .

Now, let us define the algorithm **Concat-Greedy** by Algorithm 3.

Adapting the proof by Blum *et al.* of the approximation ratio of **Concat-Cycles** from [3], one gets the following bound on the length of a multi superstring computed by **Concat-Greedy**.



---

**Algorithm 3:** The algorithm **Concat-Greedy** for Multi-SLS
 

---

- 1 **Input:** a pair  $(P, m)$ . **Output:** a linear solution for Multi-SLS;
  - 2 build  $\text{SG}(P, m)$  the multi superstring graph of  $(P, m)$ ;
  - 3 compute an Eulerian multi-cycle of  $G_P$  and take  $\tau$  the permutation in  $\text{GreedyPerm}(\tilde{P})$  corresponding to this multi-cycle;
  - 4 take a tuple  $W$  of  $E_1 \times \dots \times E_p$  where  $\text{Part}(\tilde{P}, \tau) = \{E_1, \dots, E_p\}$ ;
  - 5 **return**  $\text{LinCC}(\tilde{P}, \tau, W)$
- 

► **Proposition 14** (\*). Let  $w_{CG}$  be a solution of Algorithm 3,  $w_{OPT(\text{Multi-SLS})}$  be an optimal solution of **Multi-SLS**, and  $w_{OPT(\text{SLS})}$  be an optimal solution of **SLS**. We have:

$$|w_{CG}| \leq |w_{OPT(\text{Multi-SLS})}| + 3 \times |w_{OPT(\text{SLS})}|.$$

As an optimal solution of **Multi-SLS** is longer than or equal to an optimal solution of **SLS**, one gets the following approximation ratio for **Concat-Greedy**, which is not tight.

► **Theorem 15.** The approximation ratio of Algorithm **Concat-Greedy** for **Multi-SLS** is 4.

► **Remark.** As we have made for **Multi-SCCS**, we can compute a compact representation of **Multi-SLS** of size  $O(|P| \times |P|)$  in time  $O(|P|^2)$ . Indeed, we linearize the compact representation of **Multi-SCCS** using **Concat-Greedy** to get a compact representation for **Multi-SLS**.

## 6 Conclusion

Here, we provide the first study of **Multi-SLS** in the general case, that is without restriction on the number of words, nor on the word length. **Multi-SLS** can be approximated for both the superstring length measure and for the compression measure. Finally, both **Multi-SLS** and **Multi-SLS<sub>comp</sub>** admit a constant approximation ratio, and thus belong to the class of **APX** problems. Proposition 14 shows that the difference in length between a multi-superstring returned by **Concat-Greedy** and an optimal multi-superstring is bounded by a term proportional to the length of an optimal superstring for **SLS**, on which the multiplicities have no impact. In practice, **Concat-Greedy** may produce solutions way below this bound. A future line of research is to implement this algorithm and evaluate its ratio experimentally, an approach of great interest for superstring problems. Indeed, for the classical **SLS** problem, a simple greedy like algorithm seems to yield superstrings very close to the optimum, achieving a ratio that is orders of magnitude smaller than the theoretical bound [4]. Indeed, experimental tests allow to compare approximation algorithms and may help pinpointing hard instances. Of course, the theoretical ratio of the greedy algorithm, and the best possible approximation ratio remain open questions for **Multi-SLS**.

Our main result regarding **Multi-SCCS** is its solvability in linear time. The **Multi-Greedy** algorithm paves the way to the design of new approximation algorithms for **Multi-SLS**, as was done for the classical **SLS** problem. Let us stress that even if our algorithm builds the multi superstring graph for  $(P, m)$ , the multiplicities do not impact the numbers of nodes or of arcs, but only the weights on the arcs. As shown in Figure 4b, it is crucial that these numbers are independent of the multiplicities. Another issue is to understand what influences the number of cycles in a solution of **Multi-SCCS**; minimizing it may improve the output of **Concat-Greedy**, which "loses" some symbols each time it breaks a cycle.



Regarding future work, numerous variants of SLS (with reversals, with DNA strings [12, 9]) or restrictions of SLS (*e.g.* to strings of the same length [7]) can also be investigated with multiplicities. The question of updating a shortest superstring when the instance changes is challenging [2]. Here, a change of multiplicity can be considered as an alteration of the instance.

---

## References

- 1 Eric L. Anson and Eugene W. Myers. Algorithms for whole genome shotgun sequencing. In Sorin Istrail, Pavel A. Pevzner, and Michael S. Waterman, editors, *Proceedings of the Third Annual International Conference on Research in Computational Molecular Biology, RECOMB 1999, Lyon, France, April 11-14, 1999*, pages 1–9. ACM, 1999. doi:10.1145/299432.299442.
- 2 Davide Bilò, Hans-Joachim Böckenhauer, Dennis Komm, Richard Královic, Tobias Mömke, Sebastian Seibert, and Anna Zych. Reoptimization of the shortest common superstring problem. *Algorithmica*, 61(2):227–251, 2011. doi:10.1007/s00453-010-9419-8.
- 3 Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.
- 4 Bastien Cazaux, Samuel Juhel, and Eric Rivals. Practical lower and upper bounds for the shortest linear superstring. In Gianlorenzo D’Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA) 2018, June 27–29, 2018, L’Aquila, Italy*, volume 103 of *LIPICs*, page in press, 2018. doi:10.4230/LIPICs.SEA.2018.18.
- 5 Bastien Cazaux and Eric Rivals. A linear time algorithm for Shortest Cyclic Cover of Strings. *J. Discrete Algorithms*, 37:56–67, 2016.
- 6 Bastien Cazaux and Eric Rivals. The power of greedy algorithms for approximating Max-ATSP, Cyclic Cover, and superstrings. *Discrete Applied Mathematics*, 212:48–60, 2016.
- 7 Bastien Cazaux and Eric Rivals. Relationship between superstring and compression measures: New insights on the greedy conjecture. *Discrete Applied Mathematics*, 2017. doi:10.1016/j.dam.2017.04.017.
- 8 Maxime Crochemore, Marek Cygan, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Algorithms for three versions of the shortest common superstring problem. In Amihoud Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 2010. doi:10.1007/978-3-642-13509-5\_27.
- 9 Gabriele Fici, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. On the greedy algorithm for the Shortest Common Superstring problem with reversals. *Inf. Proc. Letters*, 116(3):245–251, 2016.
- 10 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20:50–58, 1980.
- 11 Theodoros P. Gevezes and Leonidas S. Pitsoulis. *Optimization in Science and Engineering: In Honor of the 60th Birthday of Panos M. Pardalos*, chapter The Shortest Superstring Problem, pages 189–227. Springer New York, New York, NY, 2014. doi:10.1007/978-1-4939-0808-0\_10.
- 12 Tao Jiang, Ming Li, and Ding-Zhu Du. A note on shortest superstrings with flipping. *Information Processing Letters*, 44(4):195–199, 1992.
- 13 Julián Mestre. Greedy in Approximation Algorithms. In *Proceedings of 14th Annual European Symposium on Algorithms (ESA)*, volume 4168 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2006.

- 14 Gaspard Monge. Mémoire sur la théorie des déblais et des remblais. In *Mémoires de l'Académie Royale des Sciences*, pages 666–704, 1781.
- 15 Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57:131–145, 1988. doi:10.1016/0304-3975(88)90167-3.
- 16 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
- 17 Virginia Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In *30th Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 793–800. Springer, 2005.

## A Details on the proofs for Theorems 9 and 10

This section summarizes the main lines of the proofs for Theorems 9 and 10 – formal proofs are left for a full version of this article. The proof of Theorem 9 (resp. Theorem 10) follows that of Theorem 3 (resp. Theorem 4) in [6]. We refer the reader to [13] for details on subset systems and the notion of extendibility.

Both proofs rely on a subset system to analyze the greedy algorithm for solving **Max-ATSP** in general graphs, and on the proof of its approximation ratio on **Overlap Graphs**. The goal of **Max-ATSP** is to find a maximum weighted Hamiltonian path in a digraph  $G = (V, A)$ . The subset system enforces three conditions on the arcs incorporated in a greedy solution:

1. any two arcs must start from distinct nodes
2. any two arcs must end in distinct nodes (*i.e.*, the symmetrical of the first condition)
3. there exist no cycle of length smaller than the cardinality of  $V$ .

These conditions ensure that the greedy algorithm indeed builds a Hamiltonian path. Thanks to its 3-extendibility and to Theorem 1 from [13], one deduce that the greedy algorithm yields a  $1/3$  approximation ratio for **Max-ATSP**, and similarly a  $1/2$  ratio for the **Maximum Weighted Cycle Cover** problem. However, these are the ratios for general graphs. In the case of overlap graphs, which satisfy the Monge condition [14], the proof of Theorem 3 in [6] shows by analyzing finely the greedy approximation, that the greedy algorithm yields a  $1/2$  approximation ratio for **Max-ATSP**. Since, it is known that an approximation ratio for **Max-ATSP** translates directly to an approximation ratio for **Maximum Compression** [11], which is the version of **Shortest Linear Superstring** that seeks to maximize compression measure, one gets a  $1/2$  approximation ratio for **SLS**. By applying this result on the overlap graph of  $\tilde{P}$ , one derives the  $1/2$  ratio for **Multi-SLS<sub>comp</sub>**. A similar proof ends up with an approximation ratio of 1 for the **Maximum Weighted Cycle Cover** problem on overlap graph. This yields the same ratio for **Multi-SCCS<sub>comp</sub>**, thereby showing that the greedy algorithm solves this problem exactly.

## B Proof of Lemma 16 and Proposition 11

► **Lemma 16.** *Let  $\tau$  be a permutation of  $\tilde{P}$  and  $\tilde{s} \in \tilde{P}$ . Consider  $\text{RB-Graph}(\tau, \tilde{s}) := (V, R, B)$  be the Red-Blue graph of  $\tilde{s}$ , and let  $u$  and  $v$  be two strings of  $V$ . Then, the arc  $(u, v)$  occurs only once in the Red-Blue graph, in other words*

$$|\{(u, v)\} \cap (R \cup B)| = 1.$$

**Proof of Lemma 16.** We face two alternatives: any arc belongs either to  $B$  or to  $R$ . By the definition of  $R$ , if  $(u, v)$  belongs to  $R$ , then  $v$  is the longest proper suffix of  $u$  in  $V$ . Thus, the length of  $u$  is strictly larger than that of  $v$ . By the definition of  $B$ , if  $(u, v) \in B$ , then  $u$

is the longest proper prefix of  $v$  in  $V$ . Thus,  $|u| < |v|$ . Hence, any arc of  $R \cup B$  is either in  $R$  or in  $B$ , *i.e.*,  $R \cap B = \emptyset$ . By the unicity of the longest proper prefix/suffix,  $(u, v)$  cannot appear more than once in  $R$  nor in  $B$ , which concludes the proof.  $\blacktriangleleft$

**Proof of Proposition 11.** By definition,

$$\text{NbOcc}(\tau, (u, v)) := \sum_{\substack{\tilde{s} \in \tilde{P} \\ (V, R, B) = \text{RB-Graph}(\tau, \tilde{s})}} |\{(u, v)\} \cap (R \cup B)|.$$

By Lemma 16,  $\text{NbOcc}(\tau, (u, v))$  is the number of times the arc  $(u, v)$  occurs in all Red-Blue graphs of all the elements of  $\tilde{P}$ .

To simplify the proof, we consider four alternative cases.

**The case where  $u$  is an element of  $P$ .** As  $P$  is factor-free,  $(u, v)$  is an arc of a Red-Blue graph  $(V, R, B)$ , and  $(u, v)$  is an element of  $R$  (since  $|u| > |v|$ ). Moreover,  $a(u) = 0$  because the set  $\{(w', w) \in \text{SuffixArc}(P)\} \cup \{(w, w') \in \text{PrefixArc}(P)\}$  is empty. And thus,

$$\begin{aligned} \text{NbOcc}(\tau, (u, v)) &= |\{u \mid \exists k \in \mathbb{N}, (u, k) \in \tilde{P}\}| \\ &= m(u) \\ &= \mathbf{Max}(m(u), a(u)). \end{aligned}$$

**The case where  $v$  is an element of  $P$ .** As  $P$  is factor-free, we get that  $(u, v)$  is an element of  $B$  since  $|u| < |v|$ , and that  $a(v) = 0$ . Hence,

$$\begin{aligned} \text{NbOcc}(\tau, (u, v)) &= |\{v \mid \exists k \in \mathbb{N}, (v, k) \in \tilde{P}\}| \\ &= m(v) \\ &= \mathbf{Max}(m(v), -a(v)). \end{aligned}$$

**The case where  $u \notin P$ ,  $v \notin P$  and  $|u| < |v|$ .** As  $|u| < |v|$ , the arc  $(u, v)$  is an element of  $B$ . As  $u \notin P$  and  $v \notin P$ ,  $m(u) = m(v) = 0$ .

$$\begin{aligned} \text{NbOcc}(\tau, (u, v)) &= |\{\tilde{s} \in \tilde{P} \mid (u, v) \text{ is an arc of } \text{RB-Graph}(\tau, \tilde{s})\}| \\ &= |\{\tilde{s} \in \tilde{P} \mid \text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s}))) \text{ is a prefix of } u\}| \\ &= |\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| \leq |u|\}|. \end{aligned}$$

As  $\tau$  is a permutation of  $\text{GreedyPerm}(\tilde{P})$ , and assuming that the set

$$\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| \leq |u|\}$$

is not empty (otherwise, we would have  $\text{NbOcc}(\tau, (u, v)) = 0$ ), we deduce that

$$\begin{aligned} \text{NbOcc}(\tau, (u, v)) &= |\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s}))\}| \\ &\quad - |\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| = |v|\}| \\ &\quad - |\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| > |v|\}| \\ &= \sum_{(v, w) \in \text{PrefixArc}(P)} (|\{\tilde{s} \in \tilde{P} \mid u \text{ and } w \text{ are prefixes of } \text{word}(\tau(\tilde{s}))\}| \\ &\quad - |\{\tilde{s} \in \tilde{P} \mid u \text{ and } w \text{ are prefixes of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| \geq |w|\}|) \\ &= -|\{\tilde{s} \in \tilde{P} \mid u \text{ is a proper prefix of } \text{word}(\tau(\tilde{s})), |\text{ov}(\text{word}(\tilde{s}), \text{word}(\tau(\tilde{s})))| = |v|\}| \\ &\quad + \sum_{(v, w) \in \text{PrefixArc}(P)} \text{NbOcc}(\tau, (v, w)) - \sum_{(w', v) \in \text{SuffixArc}(P)} \text{NbOcc}(\tau, (w', v)). \end{aligned}$$

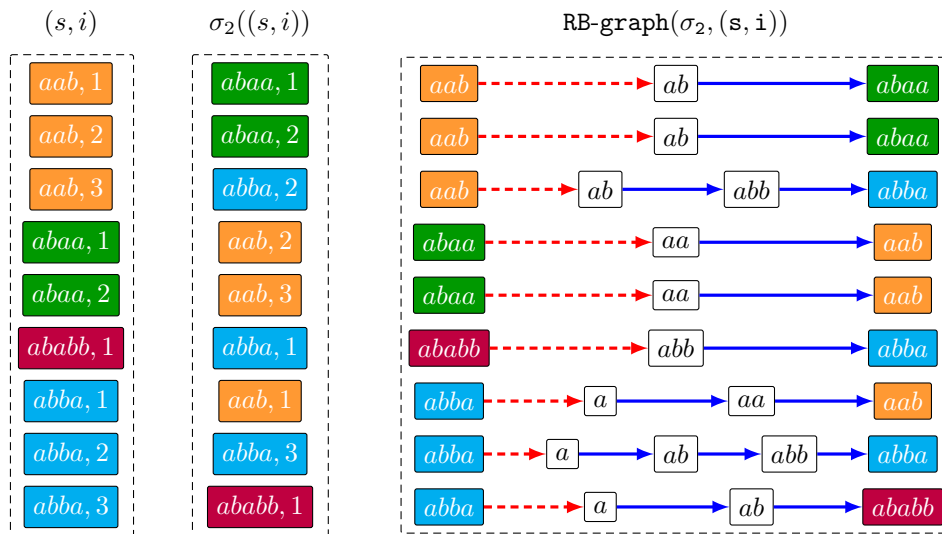
Hence,

$$\text{NbOcc}(\tau, (u, v)) = \mathbf{Max}(m(v), -a(v)).$$

**The case where  $u \notin P$ ,  $v \notin P$  and  $|u| > |v|$**  is similar to the previous case, where  $u \notin P$ ,  $v \notin P$  and  $|u| < |v|$ .

All cases have been considered and this concludes the proof.  $\blacktriangleleft$

**C** Example: set of all Red-Blue graphs



**Figure 6** Running example: set of all the Red-Blue graphs of  $(s, i) \in \tilde{P}$  for the permutation  $\sigma_2$  (see Figure 2b).

# Linear-time algorithms for the subpath kernel

Kilho Shin<sup>1</sup>

Graduate School of Applied Informatics, University of Hyogo  
Minatojima-Minamimachi, Chuo, Kobe, Japan  
yshn@ai.u-hyogo.ac.jp

Taichi Ishikawa

Graduate School of Applied Informatics, University of Hyogo  
Minatojima-Minamimachi, Chuo, Kobe, Japan  
t.i.tkgw@gmail.com

---

## Abstract

---

The subpath kernel is a useful positive definite kernel, which takes arbitrary rooted trees as input, no matter whether they are ordered or unordered. We first show that the subpath kernel can exhibit excellent classification performance in combination with SVM through an intensive experiment. Secondly, we develop a theory of irreducible trees, and then, using it as a rigid mathematical basis, reconstruct a bottom-up linear-time algorithm for the subtree kernel, which is a correction of an algorithm well-known in the literature. Thirdly, we show a novel top-down algorithm, with which we can realize a linear-time parallel-computing algorithm to compute the subpath kernel.

**2012 ACM Subject Classification** Theory of computation → Kernel methods

**Keywords and phrases** tree, kernel, suffix tree

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.22

## 1 Introduction

Recently, designing efficient kernel functions for tree-type data has become more important in various fields including bioinformatics, natural language processing (NLP) and so forth. First of all, we have many applications where data are represented in the form of trees. For instance, glycans are attracting wide attention of researchers as the third life molecule that follows DNA and proteins, and their chemical structures are trees in contrast that DNA and proteins are sequences. Also, results of syntactical analysis of natural languages and documents created according to markup languages such as HTML/XML are all represented as trees. In this paper, a tree always means a rooted tree.

To capture features of tree-type data, kernel functions are known useful. The basic nature of kernel functions is a measure to evaluate similarity of data. Furthermore, when used with various methods of multivariate analysis such as PCA and SVM, kernels are significantly useful for the purposes of classification, clustering, regression and so forth.

Kernel functions applicable to tree-type data have been intensively studied in the literature. In fact, since Haussler first introduced a generic class of positive definite kernels for semi-structured data, named the *convolution kernel* [6], a variety of tree kernels have been proposed: for example, Collins and Duffy designed the first tree kernel for the study of parse trees of natural languages [3]; Kashima and Koyanagi relaxed application-specific constraints of the

---

<sup>1</sup> This work was supported by JSPS KAKENHI Grant Number JP17H007623 and JP16K12491.



*parse tree kernel* by Collins and Duffy and introduced the *elastic tree kernel* [8]. The idea that underlies these kernels is to count shared sub-structures.

In parallel, Shin and Kuboyama [14] showed a method to derive kernel functions from various tree edit distances such as Tai distance [16] and the constrained distance [18]. In fact, these counting-up-based and distance-based tree kernels can be discussed within the common generalized framework of the *mapping kernel* [14]. In [15], a wide variety of tree kernels designed within the mapping kernel framework are investigated from the accuracy performance point of view.

This paper focuses on the *subpath kernel*, which extends and generalizes the *spectrum kernel* [11] and the all-sequences kernel for strings, and the spectrum kernel for trees [10]. We see that the subpath kernel outperforms the benchmark tree kernels in prediction performance, and its superiority is statistically significant. Furthermore, we present linear-time fast algorithms to compute it with mathematical proof for their correctness.

## 2 The Subpath Kernel (SPK) for Trees

The *subpath kernel* takes two rooted labeled trees  $T_1$  and  $T_2$  as an input and returns a real value.

The idea of the subpath kernel dates back to the *spectrum kernel* for strings that Leslie et al. proposed [11]. Leslie's spectrum kernel counts up all the pairs of congruent substrings of a fixed length such that one substring appears in the first input string, while the other does in the second.

Kuboyama et al. [10] have extended Leslie's idea to trees and introduced the spectrum tree kernel, which counts up congruent *subpaths* instead of substrings.

- Starting from an arbitrary vertex  $v$ , a subpath of length  $q$  is the sequence of vertices  $\pi = (v, p(v), p^2(v), \dots, p^{q-1}(v))$ , where  $p(w)$  denotes the parent of a vertex  $w$ .
- From  $\pi$ , we obtain a string  $\ell(\pi) = \ell(v)\ell(p(v)) \dots \ell(p^{q-1}(v)) \in \Sigma^q$ , where  $\Sigma$  is an alphabet of labels and  $\ell(w)$  is the label of a vertex  $w$ .
- For a tree  $T$  and  $s \in \Sigma^q$ , we let  $c(s; T)$  denote the number of subpaths  $\pi$  with  $\ell(\pi) = s$ .
- Finally, the  $q$ -spectrum kernel  $K_q$  is define by

$$K_q(T_1, T_2) = \sum_{s \in \Sigma^q} c(s; T_1) \cdot c(s; T_2).$$

► **Definition 1.** With a *decay factor*  $\lambda \in (0, 1)$  and spectrum tree kernels  $K_q$ , the subpath kernel is defined by  $\text{SPK}(T_1, T_2) = \sum_{q \in \mathbb{N}} \lambda^q K_q(T_1, T_2)$ .

The subpath kernel is positive definite and the yields an inner product function in the reproducing kernel Hilbert space [1].

## 3 High accuracy performance as a similarity measure.

We first see that the subpath kernel has prediction accuracy superior to major tree kernels known in the literature through an intensive experiment.

### 3.1 Datasets

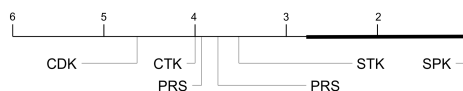
In the experiment, we use ten datasets, which cover three different areas of applications: bioinformatics (three), natural language processing (six) and web access analysis (one). Three (COLON, CYSTIC and LEUKEMIA) are retrieved from the KEGG/GLYCAN database ([5]) and contain glycan structures annotated relating to colon cancer, cystic fibrosis and leukemia

■ **Table 1** Datasets: Number of examples, averaged sizes and averaged heights of trees.

DATASET	AIMED	BIOINFER	COLON	CYSTIC	HPRD50	IEPA	LEUKEMIA	LLL	SYNTACTIC	WEB
EXAMPLES	100	70	134	160	100	100	442	100	225	500
SIZE	94.4	116.4	8.4	8.3	84.4	105.2	13.5	106.4	19.7	12.0
HEIGHT	13.5	14.1	5.6	5.0	12.7	13.6	7.4	14.3	6.5	4.3

■ **Table 2** Accuracy scores, averaged ranks, and  $p$ -values in Hommel test

Kernel	AIMED	BIOINF.	COLON	CYSTIC	HPRD50	IEPA	LEUK.	LLL	SYN.	WEB	Av. Rnk.	p-Val.
SPK	<b>0.75</b>	<b>0.84</b>	<b>0.91</b>	<b>0.79</b>	<b>0.70</b>	<b>0.71</b>	<b>0.90</b>	<b>0.65</b>	<b>0.87</b>	<b>0.82</b>	1.1	–
PRS	<b>0.75</b>	0.81	0.77	0.60	0.64	0.60	0.89	0.63	0.65	0.77	3.75	0.0031
ELS	0.74	0.81	0.82	0.67	0.60	0.64	0.88	0.60	0.68	0.77	3.95	0.0020
CDK	<b>0.75</b>	0.81	0.83	0.66	0.57	0.59	0.87	0.59	0.68	0.77	4.65	0.0001
CTK	0.73	0.78	0.88	0.72	0.60	0.60	0.88	0.59	0.76	0.78	4.0	0.0016
STK	0.72	0.79	0.90	0.73	0.61	0.59	0.88	0.60	0.82	0.78	3.55	0.0034



■ **Figure 1** Hommel test:  $p < 0.01$ .

cells. One (SYNTACTIC) is the dataset PropBank provided in [12]. This dataset includes parse trees labeled with two syntactic role classes for modeling the syntactic/semantic relation between a predicate and the semantic roles of its arguments in a sentence. Five (AIMED, BIOINFER, HPRD50 IEPA and LLL) are the corpora that include parse trees obtained by analyzing documents regarding protein-protein interaction (PPI) extraction ([13]). PPI is an intensively studied problem of the BioNLP field. The remaining one (WEB), used in [17], consists of trees representing web-page accesses by users, and the annotation is based on whether the user is from a .edu site or not. Table 1 describes the basic features of these datasets.

### 3.2 Kernels to compare

The benchmark kernels to compare with are the parse tree kernel (PRS) [3], the elastic tree kernel (ELS) [8], the sparse path kernel (STK) and the contiguous kernel (CRS). The STK and CTK are two kernels that performed the best in an intensive experiment in [15]. Each kernel includes two adjustable parameters  $\alpha$  and  $\beta$  with  $1 \geq \alpha \geq \beta > 0$ .

### 3.3 Experimental results

Table 2 shows the results of the experiments. We run ten-fold cross validation with a libSVM classifier [2] and measure accuracy scores by the accuracy index, determined by  $ACC = \frac{TP+TN}{TP+TN+FP+FN}$ . The accuracy values in Table 2 are the best values obtained through grid search changing the parameters. The subpath kernel includes one adjustable parameter to tune, a decay factor  $\lambda$ , while the others include two,  $\alpha$  and  $\beta$ .

Remarkably, for all of the datasets tested, the subtree kernel is ranked top. Also, Table 2 specifies the  $p$ -values obtained when we perform the Hommel multiple comparison test as recommended by [4]. With a significance level 0.01, we can conclude that the exhibited superiority of the subpath kernel is statistically significant (Figure 1).

## 4 Linear-time algorithms for the subpath kernel

The subpath kernel “is known” to be one of a few tree kernels that have linear-time complexity in the size of the input trees. In fact, [9] presented a linear-time algorithm but at the same time reported not so good accuracy performance. For example, the accuracy scores the subpath kernel with LEUKEMIA was the lowest of the five tree kernels tested. This could not help raising a question with us, and we have found the reason for this. The algorithm proposed in [9] was wrong.

In this section, we reconstruct the algorithm based on the mathematically rigid ground, a theory of irreducible trees (Section 4.1) and further, introduce a novel algorithm for the subpath kernel, which realizes parallel computation of the subpath kernel in combination with the corrected algorithm.

### 4.1 A theory of irreducible trees

An irreducible tree is rooted, ordered and labeled. A rooted tree  $T$  is a partially ordered set (poset) with respect to an *generation order*:  $v < w$  means that a vertex  $v$  is an ancestor of another vertex  $w$ , and hence, the root  $r_T$  of  $T$  is the unique minimum vertex. Furthermore, we let  $p(v)$  denote the parent of a vertex  $v$ , and  $p^k(v)$  does the ancestor of  $v$  for  $k > 0$  such that there are exactly  $k - 1$  intermediate vertices between  $v$  and  $p^k(v)$ . If a vertex is not the parent of any other vertices, we call it a *leaf*. From the generation order, the *nearest common ancestor* of a pair vertices  $(v, w)$  can be naturally introduced.

► **Definition 2.** For any  $\{v, w\} \subseteq T$ ,  $v \smile w = \max_{\leq} \{u \in T \mid u \leq v, u \leq w\}$  is the *nearest common ancestor* of  $v$  and  $w$ .

To define an *ordered* tree  $T$ , it is common to introduce a sibling order, but we deploy the following definition, since we are only interested in a numbering of the leaves of  $T$ .

► **Definition 3.** When the entire leaves of a rooted tree  $T$  is numbered as  $(l_1, \dots, l_n)$ ,  $T$  is said to be *ordered*, if, and only if,  $l_i \smile l_k = l_i \smile l_j \smile l_k$  holds for any  $1 \leq i < j < k \leq n$ .

► **Proposition 4.** For a vertex  $v$  of an ordered tree,  $\{i \mid l_i \geq v\} = [a, b]$  holds for some  $a$  and  $b$  in  $\{1, \dots, n\}$ . We say that  $[a, b]$  is the *span* of  $v$ .

**Proof.** We let  $a = \min\{i \mid l_i \geq v\}$  and  $b = \max\{i \mid l_i \geq v\}$ . For any  $i \in (a, b)$ ,  $l_i \smile l_a \geq l_a \smile l_b \geq v$  holds. In particular, we have  $l_i \geq v$ . ◀

Finally, we define an *irreducible* tree in Definition 5.

► **Definition 5.** A rooted and ordered tree is *irreducible*, iff no vertex has only one child.

For study of irreducible trees,  $\alpha_i$  defined below plays a crucial role.

► **Definition 6.** For  $i \in \{1, 2, \dots, n - 1\}$ ,  $\alpha_i$  denotes  $l_i \smile l_{i+1}$ .

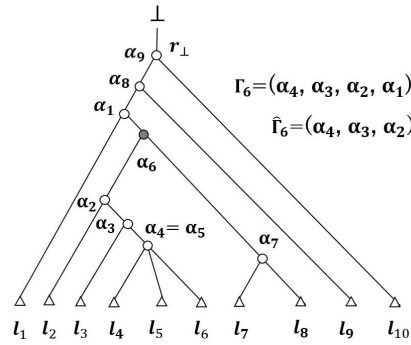
The rightmost (leftmost) leaf of a vertex  $v$  is  $l_b$  ( $l_a$ ), when  $v$  spans  $[a, b]$ . The rightmost leaf of  $v$  can be characterized by  $\alpha_i$  as follows.

► **Proposition 7.** We assume  $v < l_i$ .  $l_i$  is the rightmost leaf of  $v$ , if, and only if,  $v > \alpha_i$ .

**Proof.** If  $l_i$  is the rightmost leaf,  $\alpha_i \not\geq v$  holds, since  $l_{i+1} \geq v$  holds, otherwise; If  $\alpha_i < v$ ,  $l_i \smile l_k \leq \alpha_i < v$ , and therefore,  $l_k \not\geq v$  holds for any  $k > i$ . ◀

Any non-leaf vertex  $v$  has at least one  $i$  such that  $v = \alpha_i$ . We have





■ **Figure 2** An irreducible tree.

► **Proposition 8.** For a non-leaf vertex  $v$ , we let  $w$  be the leftmost child of  $v$  and  $l_i$  be the rightmost leaf of  $w$ . Then,  $i = \min\{j \mid \alpha_j = v\}$  holds.

**Proof.** By Proposition 7,  $\alpha_i < w$  holds. On the other hand, since  $l_i$  is not the rightmost leaf of  $v$ ,  $\alpha_i \geq v$  holds.  $\alpha_i = v$  immediately follows. ◀

► **Definition 9.** For an intermediate vertex  $v$ ,  $\gamma(v)$  denotes  $\min\{i \mid \alpha_i = v\}$ .

For example, in Figure 2,  $\alpha_4$  and  $\alpha_5$  are identical, and  $\gamma(\alpha_5) = 4$  holds. Corollary 10 will play a central role when we introduce a top-down algorithm for the subpath kernel in Section 4.5. For the convenience of explanation, without loss of generality, we add an imaginary root  $\perp$  on top of  $r_T$  and let  $\alpha_n = \perp$ .

► **Corollary 10.** If a non-leaf vertex  $v$  that spans  $[a, b]$  has children  $w_1, \dots, w_t$ , their rightmost leaves are  $l_{i_1}, \dots, l_{i_t}$  with  $\{i_1, \dots, i_t\} = \{j \mid j \in [a, b], \alpha_j \leq v\}$ .

**Proof.** We assume  $i_1 < \dots < i_t$ .  $i_t = b$  follows from Proposition 7.  $l_{i_1}$  is the rightmost leaf of  $w_1$  by Proposition 8. To verify that  $l_{i_i}$  is the rightmost leaf of  $w_i$  for  $1 < i < t$ , we have only to eliminate  $w_1, \dots, w_{i-1}$  and their subordinates and then to apply Proposition 8. ◀

Theorem 14 and 16 stated below will be a theoretical basis to justify the correctness of the bottom-up traversal algorithm introduced in [7] and to correct errors of the algorithm to compute the subpath kernel proposed in [9]. We start with defining  $\Gamma_i$  and  $\widehat{\Gamma}_i$ .

► **Definition 11.**  $\Gamma_i$  and  $\widehat{\Gamma}_i$  are the subsequences of the subpath  $(p^1(l_i), p^2(l_i), \dots, p^{l_i}(l_i) = r_T)$  consisting of the vertices  $p^j(l_i)$  such that  $\gamma(p^j(l_i)) < i$  and  $p^j(l_i) > \alpha_i$ , respectively.

► **Example 12.** In Figure 2,  $\Gamma_i$  and  $\widehat{\Gamma}_i$  for  $i = 1, \dots, 10$  are determined as follows.

$i$	$\Gamma_i$	$\widehat{\Gamma}_i$	$i$	$\Gamma_i$	$\widehat{\Gamma}_i$
1	()	()	6	$(\alpha_4, \alpha_3, \alpha_2, \alpha_1)$	$(\alpha_4, \alpha_3, \alpha_2)$
2	$(\alpha_1)$	()	7	$(\alpha_6, \alpha_1)$	()
3	$(\alpha_2, \alpha_1)$	()	8	$(\alpha_7, \alpha_6, \alpha_1)$	$(\alpha_7, \alpha_6, \alpha_1)$
4	$(\alpha_3, \alpha_2, \alpha_1)$	()	9	$(\alpha_8)$	$(\alpha_8)$
5	$(\alpha_4, \alpha_3, \alpha_2, \alpha_1)$	()	10	$(\alpha_9)$	$(\alpha_9)$

► **Proposition 13.** Any  $v \in \widehat{\Gamma}_i$  has  $\gamma(v) < i$ . Hence,  $\widehat{\Gamma}_i \subseteq \Gamma_i$  holds.

**Proof.**  $l_i \smile l_k \leq \alpha_i < v$  holds for  $k > i$ , and hence,  $l_k \not\leq v$  holds. ◀

► **Theorem 14.** The sequence  $\prod_{i=1}^n [(l_i) \cdot \widehat{\Gamma}_i]$  yields the bottom-up traversal of the vertices of  $T$ . Given two sequences  $s$  and  $t$ ,  $s \cdot t$  denotes their concatenation.

**Proof.** Since every vertex  $v$  has a unique leftmost leaf, it appears in the sequence exactly once. On the other hand, for a vertex  $w$  with  $w > v$ , the span of  $w$  is a subset of the span of  $v$ , and hence,  $w$  appears before  $v$  in the sequence. ◀

► **Example 15.** In Figure 2,  $(l_i) \cdot \widehat{\Gamma}_i$  for  $i = 1, \dots, 10$  is determined as follows.

$i$	$\widehat{\Gamma}_i$	$(l_i) \cdot \widehat{\Gamma}_i$	$i$	$\widehat{\Gamma}_i$	$(l_i) \cdot \widehat{\Gamma}_i$
1	()	$(l_1)$	6	$(\alpha_4, \alpha_3, \alpha_2)$	$(l_6, \alpha_4, \alpha_3, \alpha_2)$
2	()	$(l_2)$	7	()	$(l_7)$
3	()	$(l_3)$	8	$(\alpha_7, \alpha_6, \alpha_1)$	$(l_8, \alpha_7, \alpha_6, \alpha_1)$
4	()	$(l_4)$	9	$(\alpha_8)$	$(l_9, \alpha_8)$
5	()	$(l_5)$	10	$(\alpha_9)$	$(l_{10}, \alpha_9)$

In fact, their concatenation  $(l_1, l_2, l_3, l_4, l_5, l_6, \alpha_4, \alpha_3, \alpha_2, l_7, l_8, \alpha_7, \alpha_6, \alpha_1, l_9, \alpha_8, l_{10}, \alpha_9)$  gives the bottom-up traversal of the vertices of the tree.

► **Theorem 16.** For  $i = 1, \dots, n - 1$ , the following hold.

1. If  $\alpha_i \in \Gamma_i$ ,  $\Gamma_{i+1} = \Gamma_i \setminus \widehat{\Gamma}_i$ .
2. If  $\alpha_i \notin \Gamma_i$ ,  $\Gamma_{i+1} = (\alpha_i) \cdot (\Gamma_i \setminus \widehat{\Gamma}_i)$ .

**Proof.** If  $j$  with  $j < i$  meets  $\alpha_j < l_{i+1}$ ,  $\alpha_j \leq \alpha_i$  holds. In fact, since  $\alpha_j \geq l_j \smile l_{i+1}$ , we have  $\alpha_j = l_j \smile l_{i+1} \leq \alpha_i$ , and hence,  $\alpha_j \in \Gamma_i \setminus \widehat{\Gamma}_i$ . If  $\alpha_i \in \Gamma_i \setminus \widehat{\Gamma}_i$ ,  $\Gamma_{i+1} = \Gamma_i \setminus \widehat{\Gamma}_i$  holds. Otherwise, we prepend  $\alpha_i$  to  $\Gamma_i \setminus \widehat{\Gamma}_i$  to obtain  $\Gamma_{i+1}$ . ◀

► **Example 17.** In Figure 2,  $\alpha_i \in \Gamma_i$  holds only for  $i = 5$ . In fact,  $\Gamma_6 = (\alpha_4, \alpha_3, \alpha_2, \alpha_1)$  is identical to  $\Gamma_5 \setminus \widehat{\Gamma}_5 = (\alpha_4, \alpha_3, \alpha_2, \alpha_1) \setminus ()$ . For the other  $i$ ,  $\Gamma_{i+1} = (\alpha_i) \cdot (\Gamma_i \setminus \widehat{\Gamma}_i)$  holds. For example,  $\widehat{\Gamma}_5 = (\alpha_4, \alpha_3, \alpha_2)$  and  $(\alpha_6) \cdot (\Gamma_5 \setminus \widehat{\Gamma}_5) = (\alpha_6, \alpha_1) = \Gamma_7$  hold.

► **Definition 18.**  $h : T \rightarrow \mathbb{N}$  is a *height function*, if  $h(v) > h(w)$  holds for any  $(v, w) \in T^2$  with  $v > w$ , and if  $h(r_T) = 0$ .

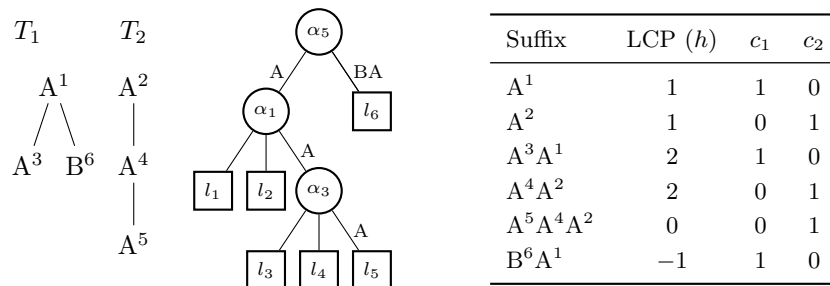
A height function can be defined for an arbitrary rooted tree, which is not necessarily irreducible.

► **Example 19.** For a rooted tree  $T$  and a vertex  $v$  in  $T$ , we let  $h_v$  denote the number of ancestors of  $v$ : that is,  $h_v = |\{w \in T \mid w < v\}|$ . Evidently,  $h_v$  is a height function.

## 4.2 Suffix arrays and suffix trees

The well known suffix tree is an example of irreducible trees.

Consider two rooted labeled trees  $T_1$  and  $T_2$ , which are not necessarily ordered. For each vertex  $v \in T_i$ , its entire path is the sequence of vertices  $(v, p(v), p^2(v), \dots, p^{h_v}(v) = r_T)$ , and the *suffix* of  $v$  is the string “ $L(v)L(p(v)) \dots L(p^{h_v}(v))$ ”, where  $L(v)$  denotes the label of a vertex  $v$ . To determine the suffix array for  $T_1$  and  $T_2$ , we collect all the suffices across all the vertices of  $T_1$  and  $T_2$ , and then sort them in the lexicographical order as strings. The suffix array includes  $n = |T_1| + |T_2|$  entries. In Figure 3, the first column of the right table describes the suffix array for  $T_1$  and  $T_2$  depicted by the same figure.



■ **Figure 3** A suffix array (right) and the associated suffix tree (middle).

The *suffix tree*  $ST$  for  $T_1$  and  $T_2$  is derived from the suffix array. The leaf vertices  $l_1, l_2, \dots, l_{|T_1|+|T_2|}$  of the suffix tree uniquely correspond to the entries of the suffix array in the order in which they appear in the array: the leaf  $l_i$  represents the suffix  $s_i$ , which is the entry of the suffix array at position  $i$ . Because there is a one-to-one correspondence between the entries of the suffix array and the vertices of  $T_1$  and  $T_2$ , each leaf of the suffix tree also uniquely represents a vertex in  $T_1$  or  $T_2$ . Furthermore, each edge of  $ST$  is labeled with a string of vertex labels so that the following conditions are met:

1. The concatenation of the edge labels of the path from the root  $r_{ST}$  to  $l_i$  is identical to  $s_i$ .
2. The labels of two downward edges from the same vertex of the suffix tree have no common prefix.

Combined with the condition that the suffix tree is irreducible, these conditions uniquely determine the suffix tree  $ST$ .

The center tree displayed in Figure 3 describes the suffix tree derived from  $T_1$  and  $T_2$  depicted in the same figure. Note that an edge label is omitted, if it is an empty string. For example,  $l_5$  corresponds to the fifth entry of the suffix array, and therefore, represents the vertex  $A^5$  in  $T_2$ . In fact, the downward concatenation of the labels for the entire path of  $l_5$  is identical to  $s_5 = AAA$ .

An LCP value  $h(i)$  for an entry at the position  $i$  in a suffix array gives the length of the longest common prefix between  $s_i$  and  $s_{i+1}$ . For example, in Figure 3, we have  $s_2 = A$  and  $s_3 = AA$ , and therefore, the LCP value  $h(2)$  turns out to be 1. For the last entry of the suffix array, we define its LCP value to be  $-1$  for convenience of computation. In the corresponding suffix tree,  $h(i)$  determines a height of the intermediate vertex  $\alpha_i$ .

Finally, we introduce two arrays  $c_1$  and  $c_2$  in addition to the LCP array  $h$ .  $c_1(i)$  and  $c_2(i)$  for the entry at position  $i$  of a suffix array describes to which the suffix  $s_i$  belongs,  $T_1$  or  $T_2$ :  $c_1(i) = 1$ , if  $s_i$  is a subpath of  $T_1$ , and  $c_2(i) = 1$ , if  $s_i$  is a subpath of  $T_2$ .

To compute the subpath kernel, we have only to input these three arrays  $h$ ,  $c_1$  and  $c_2$  into algorithms.

In [9], an algorithm to generate suffix arrays and suffix trees whose time complexity is linear to the size of trees is proposed.

### 4.3 Reconstruction of the bottom-up traversal algorithm of [7]

We first reconstruct a linear-time bottom-up traversal algorithm based on the theory shown in Section 4.1, which is equivalent to the one introduced in [7]. Algorithm 1 shows the algorithm Theorem 14 and 16 clearly explain the algorithm and at the same time give a mathematical justification for its correctness.

---

**Algorithm 1** A bottom-up traversal algorithm of an irreducible tree.
 

---

**Require:**  $(h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n$   $\triangleright h(\alpha_i)$ : the height of  $\alpha_i$   
**Ensure:** A sequence  $(v_1, \dots, v_{|T|})$  of vertices of  $T$  in the bottom-up traversal order.

```

1: Clear a stack  $\Gamma$   $\triangleright \text{pop}_\Gamma, \text{push}_\Gamma(\cdot), \text{top}_\Gamma$  are operations on  $\Gamma$ 
2: for  $i = 1, 2, \dots, n$  do
3:   Write  $l_i$ 
4:   while  $\Gamma \neq \emptyset \wedge h(\text{top}_\Gamma) > h(\alpha_i)$  do  $\triangleright \text{top}_\Gamma > \alpha_i \Leftrightarrow h(\text{top}_\Gamma) > h(\alpha_i)$ 
5:     Write  $\text{top}_\Gamma$ 
6:     Do  $\text{pop}_\Gamma$ 
7:   end while
8:   if  $\Gamma = \emptyset \vee h(\text{top}_\Gamma) \neq h(\alpha_i)$  then  $\triangleright \alpha_i \in \Gamma \Leftrightarrow \alpha_i = \text{top}_\Gamma$ 
9:     Do  $\text{push}_\Gamma(\alpha_i)$ 
10:  end if
11: end for
    
```

---

1. The first-in-last-out stack  $\Gamma$  holds  $\Gamma_i$  for each  $i$  of the **for** loop. If  $\Gamma_i = (v_1, \dots, v_k)$  with  $v_1 > \dots > v_k$ ,  $v_1$  is stored at the top, and  $v_k$  is stored at the bottom of  $\Gamma$ .
2. Note that, if  $\alpha_i$  and  $\alpha_j$  are comparable with respect to the generation order, we have  $\alpha_i < \alpha_j \Leftrightarrow h(\alpha_i) < h(\alpha_j)$ . Therefore, the exit condition of the **while** loop is for  $h(\text{top}_\Gamma) \leq h(\alpha_i)$  to hold.
3. The **while** loop outputs the elements of  $\widehat{\Gamma}_i$  in the decreasing direction of the generation order. Hence, Theorem 14 asserts that the algorithm outputs the vertices of  $T$  in the bottom-up traverse order.
4. The **while** loop also eliminates  $\widehat{\Gamma}_i$  from  $\Gamma_i$  in the stack  $\Gamma$ . This is done by performing  $\text{pop}_\Gamma$ . By Theorem 16, this updates  $\Gamma_i$  to  $\Gamma_{i+1}$ , if  $\alpha_i \in \Gamma_i$ .
5. If  $\alpha_i \notin \Gamma_i$ , by Theorem 16,  $\alpha_i$  is to be prepended to  $\Gamma_i \setminus \widehat{\Gamma}_i$  to obtain  $\Gamma_{i+1}$ . In fact, this is done by performing  $\text{push}_\Gamma(\alpha_i)$ .
6. To know whether  $\alpha_i \in \Gamma_i$ , we have only to examine whether  $\text{top}_\Gamma = \alpha_i$ , equivalently, whether  $h(\text{top}_\Gamma) = h(\alpha_i)$ .

#### 4.4 A linear-time bottom-up algorithm for the subpath kernel

In [9], the key formula to compute  $\text{SPK}(T_1, T_2)$  is given as

$$\text{SPK}(T_1, T_2) = \sum_{v \in ST} (w(h(v)) - w(h(p(v)))) \cdot c_1(v) \cdot c_2(v). \quad (1)$$

The function  $w$  is determined by  $w(h) = \sum_{i=1}^h \lambda^i$  and  $c_i(v)$  is the number of leaves below  $v$  that belong to  $T_i$ . Algorithm 2 computes  $\text{SPK}(T_1, T_2)$  by Eq. (1) and is also a correction to the algorithm exhibited in [9].

The steps commented with “ $\triangleright$  For bottom-up traversal” are to perform bottom-up traversal of vertices of the suffix tree  $ST$  derived from  $T_1$  and  $T_2$ , the following are added to Algorithm 1.

- The stack  $\Gamma$  stores a triplet  $(\alpha_i, c_1, c_2)$  (Step 13) instead of  $\alpha_i$ . The second and third components store intermediate values to compute  $c_1(v)$  and  $c_2(v)$ .
- The value  $(w(h(v)) - w(h(p(v)))) \cdot c_1(v) \cdot c_2(v)$  computed for each vertex  $v$  is accumulated in the variable *kernel* (Step 9).
- When  $\alpha_i \in \Gamma_i$  (Step 14), the triplet  $(v, c'_1, c'_2)$  is updated so that leaves found during eliminating  $\widehat{\Gamma}_i$  from  $\Gamma_i$  are counted (Step 16).

---

**Algorithm 2** A bottom-up algorithm for SPK (correction to [9]).

---

**Require:**  $(h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n$ ;  $(c_1(l_1), \dots, c_1(l_n)) \in \mathbb{Z}_2^n$ ;  $(c_2(l_1), \dots, c_2(l_n)) \in \mathbb{Z}_2^n \triangleright h(\alpha_i)$ : the height of  $\alpha_i$ ;  $c_i(l_j)$ : belonging of  $l_j$  to  $T_i$

**Ensure:**  $\text{SPK}(T_1, T_2)$

```

1: procedure  $\text{SPK}_{\text{BU}}(h(\alpha_1), \dots, h(\alpha_n); c_1(l_1), \dots, c_1(l_n); c_2(l_1), \dots, c_2(l_n))$ 
2:   Clear a stack  $\Gamma$  ▷ For bottom-up traversal
3:   Let  $kernel = 0$ 
4:   for  $i = 1, 2, \dots, n$  do ▷ For bottom-up traversal
5:     Let  $c_1 = c_1(l_i)$  and  $c_2 = c_2(l_i)$ 
6:     while  $\Gamma \neq \emptyset \wedge h(\text{top}_\Gamma[0]) > h(\alpha_i)$  do ▷ For bottom-up traversal
7:       Let  $(v, c'_1, c'_2) = \text{top}_\Gamma$ 
8:       Do  $\text{pop}_\Gamma$  ▷ For bottom-up traversal
9:       Let  $c_1 = c_1 + c'_1$  and  $c_2 = c_2 + c'_2$  ▷  $c_i = c_i(v)$ 
10:      if  $h(v) \neq 0$  then
11:        Let  $kernel = kernel + (w(h(v)) - w(h(p(v)))) \cdot c_1 \cdot c_2$ 
12:      end if ▷ Eq. (1)
13:    end while ▷ For bottom-up traversal
14:    if  $\Gamma = \emptyset \vee h(\text{top}_\Gamma[0]) \neq h(\alpha_i)$  then ▷ For bottom-up traversal
15:      Do  $\text{push}_\Gamma(\alpha_i, c_1, c_2)$  ▷ For bottom-up traversal
16:    else
17:      Let  $(v, c'_1, c'_2) = \text{top}_\Gamma$ 
18:      Let  $\text{top}_\Gamma = (v, c'_1 + c_1, c'_2 + c_2)$ 
19:    end if ▷ For bottom-up traversal
20:  end for ▷ For bottom-up traversal
21: end procedure

```

---

**Algorithm 3** Computation of  $p(v)$ .

---

**Require:**  $(h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n$ ;  $\Gamma = \Gamma_i \setminus \{v_1, \dots, v_j\}$ ;  $v = v_j$  ▷  $\{v_1, \dots, v_j\} \subseteq \widehat{\Gamma}_i$

**Ensure:**  $p(v)$

```

1: if  $\Gamma = \emptyset \vee h(\text{top}_\Gamma[0]) < h(\alpha_i)$  then ▷  $\text{top}_\Gamma[0] < \alpha_i \Leftrightarrow h(\text{top}_\Gamma[0]) < h(\alpha_i)$ 
2:   return  $\alpha_i$ 
3: else
4:   return  $\text{top}_\Gamma[0]$ 
5: end if

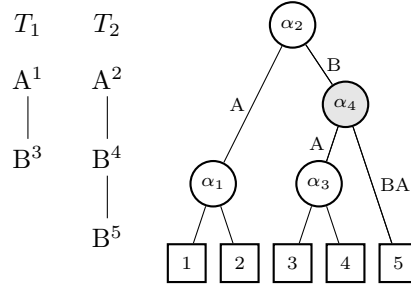
```

---

We should be careful when computing  $p(v)$  in Step 10. Proposition 13 asserts that, when  $(v, c'_1, c'_2)$  is the last element eliminated from  $\Gamma$ ,  $p(v) = \text{top}_\Gamma[0]$  holds, if  $\text{top}_\Gamma[0] > \alpha_i$ , and  $p(v) = \alpha_i$  holds, otherwise.

The most important error of the algorithm of [9] was that it wrongly assumed  $p(v) = \text{top}_\Gamma[0]$  unconditionally. For example, for two trees  $T_1$  and  $T_2$  and the suffix tree derived from them depicted by Figure 4, the subpath kernel value for  $T_1$  and  $T_2$  turns out to be  $\lambda^2 + 3\lambda$ , because the subpaths of  $T_1$  are  $\{A^1, B^3, B^3A^1\}$ , while the subpaths of  $T_2$  are  $\{A^2, B^4, B^5, B^4A^2, B^5B^4, B^5B^4A^2\}$ .

In Algorithm 1, the bottom-up traversal visits  $\alpha_3$ , when  $i = 4$ . Since  $p(\alpha_3)$  is  $\alpha_4$ , the value  $(w(h(\alpha_3)) - w(h(\alpha_4))) \cdot 1 \cdot 1 = \lambda^2 + \lambda - \lambda = \lambda^2$  is added to the variable  $kernel$  at Step 10. On the other hand, since  $\Gamma_4 = (\alpha_3, \alpha_2)$  holds, the algorithm of [9] adds  $(w(h(\alpha_3)) - w(h(\alpha_2))) \cdot 1 \cdot 1 = \lambda^2 + \lambda$ , instead. By this, the kernel value that the algorithm of [9] computes becomes  $\lambda^2 + 4\lambda$ .



■ **Figure 4** A counter example.

---

**Algorithm 4** Decomposition into child trees.

---

**Require:**  $a; h(v); \{h(\alpha_1), \dots, h(\alpha_n)\} \subset \mathbb{N}^n$   $\triangleright l_a$ : the leftmost leaf of  $v$   
**Ensure:**  $((i_1, h_1), \dots, (i_t, h_t))$   
 1:  $\triangleright (w_1, \dots, w_t)$ : the children of  $v$ ;  $l_{i_j}$  is the rightmost leaf of  $w_j$ ;  $h_j = h(w_j)$   
 2: Let  $i = a$   
 3: **while** true **do**  
 4:      $min_h = h(\alpha_i)$   
 5:     **while**  $h(\alpha_i) > h(v)$  **do**  $\triangleright \alpha_i > v \Leftrightarrow h(\alpha_i) > h(v)$   
 6:          $min_h = \min\{h(\alpha_i), min_h\}$   
 7:         Let  $i = i + 1$   
 8:     **end while**  
 9:     Write  $(i, min_h)$   
 10:     **if**  $h(\alpha_i) < h(v)$  **then**  $\triangleright \alpha_i < v \Leftrightarrow h(\alpha_i) < h(v)$   
 11:         **return**  
 12:     **end if**  
 13:     Let  $i = i + 1$   
 14: **end while**

---

#### 4.5 A Top-Down Algorithm for the subpath kernel

We introduce a novel algorithm that computes the subpath kernel leveraging recursive function calls. Algorithm 4 below is the key component of the algorithm, which decomposes a tree into a sequence of child trees. Corollary 10 guarantees the correctness of Algorithm 4.

Algorithm 5 defines the function  $\text{SPK}_{\text{TD}}$  that computes the subpath kernel. For convenience of explanation, we simply assume that we call the function  $\text{SPK}_{\text{TD}}$  specifying an interval of leaves as an input to obtain three values: the number of leaves that belong to  $T_1$  in the interval; the number of leaves that belong to  $T_2$  in the interval; and the kernel value computed for the interval. To be specific,  $\text{SPK}_{\text{TD}}(I)$  is formulated by

$$\text{SPK}_{\text{TD}}(I) = \left( |STL_1 \cap I|, |STL_2 \cap I|, \sum_{i \in STL_1 \cap I} \sum_{j \in STL_2 \cap I} w(h(l_i \sim l_j)) \right), \quad (2)$$

where  $STL_i = \{j \mid l_j \in T_i\}$  for  $i = 1, 2$  and  $I = [a, b]$  for  $1 \leq a \leq b \leq n$ . Evidently,  $\text{SPK}_{\text{TD}}([1, n]) = \text{SPK}(T_1, T_2)$  holds.

The function first performs Algorithm 4 to decompose the input interval of leaves, spanned by an intermediate vertex  $v$  in  $ST$ , into more than one intervals, each of which is spanned by a child of  $v$  (Step 5). Then, the function recursively applies itself to each interval obtained (Step 10).

The time complexity of computing  $\text{SPK}_{\text{TD}}(I)$  can be estimated to be  $O(|I| \cdot \text{dp}(v))$ , where the depth function  $\text{dp}(v)$  gives the longest length of downward paths in the suffix tree

**Algorithm 5** A top-down algorithm for SPK

---

**Require:**  $a; b; h(v); (h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n; (c_1(l_1), \dots, c_1(l_n)) \in \mathbb{Z}_2^n; (c_2(l_1), \dots, c_2(l_n)) \in \mathbb{Z}_2^n$   $\triangleright$   
 $v$ : a vertex that spans  $(l_a, \dots, l_b)$  in  $ST$

**Ensure:**  $c'_1; c'_2; kernel' \triangleright c'_i$ : the number of leaves of  $T_i$  in  $[a, b]$ ;  $kernel'$ : the kernel value for  $[a, b]$

```

1: procedure SPKTD( $a; b; h(v); h(\alpha_a), \dots, h(\alpha_b); c_1(l_a), \dots, c_1(l_b); c_2(l_a), \dots, c_2(l_b)$ )
2:   if  $a = b$  then
3:     return  $(c_1(l_a), c_2(l_b), 0.0)$ 
4:   end if
5:   Compute  $((i_1, h_1), \dots, (i_t, h_t))$  by Algorithm 4
6:            $\triangleright (w_1, \dots, w_t)$ : the children of  $v$ ;  $l_{i_j}$ : the leftmost leaf of  $w_j$ ;  $h_j = h(w_j)$ 
7:   Let  $i_0 = a - 1$ 
8:   Let  $c_1, c_2, kernel = 0, 0, 0.0$ 
9:   for  $j = 1, \dots, t$  do
10:    Let  $(c'_1, c'_2, kernel') = \text{SPK}_{\text{TD}}(i_{j-1} + 1; i_j; h_j; h(\alpha_{i_{j-1}+1}), \dots, h(\alpha_{i_j});$ 
11:           $c_1(l_{i_{j-1}+1}), \dots, c_1(l_{i_j}); c_2(l_{i_{j-1}+1}), \dots, c_2(l_{i_j}))$ 
12:    Let  $kernel = kernel + w(h(v)) \cdot (c_1 \cdot c'_2 + c_2 \cdot c'_1) + kernel'$ 
13:           $\triangleright w(h) = \lambda + \lambda^2 + \dots + \lambda^h$ , where  $\lambda$  is a decay factor
14:    Let  $c_1, c_2 = c_1 + c'_1, c_2 + c'_2$ 
15:  end for
16:  return  $(c_1, c_2, kernel)$ 
17: end procedure

```

---

that start at the vertex  $v$ . This can be proven by mathematical induction as follows. Since Algorithm 4 scans all the leaves in  $I$  exactly one time for each, its time complexity is  $O(|I|)$ . As a result of running Algorithm 4,  $I$  is partitioned to intervals  $I_1, \dots, I_t$ . Since a suffix tree is irreducible,  $t > 1$  holds. By the hypothesis of mathematical induction, we suppose that the time complexity to execute Algorithm 5 for  $I_i$  is  $O(|I_i| \cdot \text{dp}(w_i))$ , where  $w_i$  is a child of  $v$  in the suffix tree and spans  $I_i$ . Hence, the time complexity to execute Algorithm 5 for  $I$  is bounded above by

$$O(|I|) + \sum_{i=1}^t O(|I_i| \cdot \text{dp}(w_i)) \leq O(|I|) + \sum_{i=1}^t O(|I_i| \cdot (\text{dp}(v) - 1)) = O(|I| \cdot \text{dp}(v))$$

In particular, the time complexity of Algorithm 5 for two trees  $T_1$  and  $T_2$  is bounded above by  $O((|T_1| + |T_2|) \cdot \max\{\text{dp}(T_1), \text{dp}(T_2)\})$ , where  $\text{dp}(T_i)$  is the depth of the root of  $T_i$  in  $T_i$ .

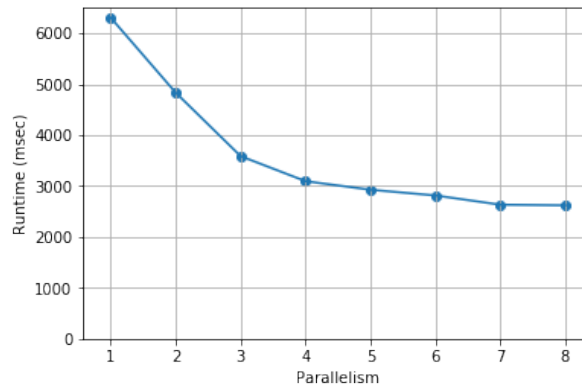
Although this top-down algorithm is not linear with respect to the size of trees, it leads us to a hybrid parallel-computing linear-time algorithm as shown in the next section.

## 4.6 A hybrid parallel-computing linear-time algorithm

The top-down algorithm (Algorithm 5) enables us to compute the subpath kernel within the parallel computing framework. The idea is:

1. Apply the decomposition algorithm of Algorithm 4 until the entire tree is decomposed into an appropriate number of subtrees;
2. Use the bottom-up subpath kernel algorithm of Algorithm 2 to compute the kernel values for the subtrees obtained in Step 1;
3. Call the SPK<sub>TD</sub> function of Algorithm 5 recursively until reaching the subtrees precomputed in Step 2.

Since the time complexity of Step 1 and Step 3 is linear to  $|T_1| + |T_2|$ , since the parallelism is a constant number. On the other hand, the time complexity of Step 2 is evidently linear, and hence, the total time complexity of the hybrid algorithm is linear.



■ **Figure 5** Runtime to compute 20 kernel values.

We conducted an experiment to compare the run-time of

For the experiment, we used a Mac Book Pro with 2.9GHz Quad Core Intel Core™ i7 CPU and ran the program written in Scala on macOS High Sierra 10.13.4. For parallel computation, we used the `ParArray` collection class.

The dataset used in the experiment consists of 20 pairs of randomly generated synthetic trees, each of which consists of  $\frac{10^7-1}{9} = 1,111,111$  vertices and uniformly has degree 10 and height 7. The size of the alphabet of vertex labels is 100.

Figure 5 shows the run-time sores in milliseconds to compute the 20 kernel values, when we change the parallelism from 1 to 8. Since the CPU includes four cores, the runtime rapidly decreases until the parallelism reaches three. For the parallelism greater than three, although the gradient of the curve becomes gentler, the runtime steadily decreases.

## 5 Conclusion

We have shown superiority of the subpath kernel to other benchmark tree kernels in classification performance. The superiority has proven to be statistically significant through Hommel multiple comparison test with the significance level 0.01. In addition, we presented a linear-time bottom-up algorithm for the subpath kernel as well as a top-down algorithm. We have given mathematical proofs for the correctness of these algorithms based on a theory that we have developed. By combining the bottom-up and top-down algorithms, we can build hybrid linear-time parallel-computing algorithm, which has proven to improve the run-time performance through experiments. Considering all the above, we conclude that the subpath kernel should be the best kernel for analyzing tree data. As future studies, we will investigate their performance for other purposes of data analysis such as classification and regression.

---

## References

- 1 Christensen Berg, C. and R. J. P. R., Ressel. Harmonic analysis on semigroups. theory of positive definite and related functions. *Springer*, 1984.
- 2 C. C. Chang and C. J. Lin. Libsvm: a library for support vector machines, 2001. URL: <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>.
- 3 M. Collins and N. Duffy. Convolution kernels for natural language. *Neural Information Processing Systems*, 2001.



- 4 J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Theory*, 7:1–30, 2006.
- 5 K. Hashimoto, S. Goto, S. Kawano, K. F. Aoki-Kinoshita, and N. Ueda. KEGG as a glycome informatics resource. *Glycobiology*, 16:63R–70R, 2006.
- 6 D. Haussler. Convolution kernels on discrete structures. *UCSC-CRL 99-10*, 1999.
- 7 T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *the 12th Annual Symposium on Combinatorial Pattern Matching. pp.*, 2001.
- 8 H. Kashima and T. Koyanagi. Kernels for semi-structured data. in: the 9th international conference on machine learning. *ICML*, 2002.
- 9 D. Kimura and H. Kashima. Fast computation of subpath kernel for trees. *ICML*, 2012.
- 10 T. Kuboyama, K. Hirata, H. Kashima, K.F. Aoki-Kinoshita, and H. Yasuda. A spectrum tree kernel. *JSAI*, 2007.
- 11 C. S. Leslie, E. Eskin, and W. Stafford Noble. The spectrum kernel: A string kernel for SVM protein classification. *Pacific Symposium on Biocomputing*, 2002.
- 12 Alessandro Moschitti. Example data for TREE KERNELS IN SVM-LIGHT. URL: <http://disi.unitn.it/moschitti/Tree-Kernel.htm>.
- 13 S. Pyysalo, A. Airola, J. Heimonen, J. Bjorne, F. Ginter, and T. Salakoski. Comparative analysis of five protein-protein interaction corpora. *BMC Bioinformatics*, 9(S-3), 2008.
- 14 K. Shin and T. Kuboyama. A generalization of Haussler’s convolution kernel - mapping kernel. *ICML*, 2008.
- 15 K. Shin and T. Kuboyama. A comprehensive study of tree kernels. in: Jsai-isai post-workshop proceedings. *Lecture Notes in Artificial Intelligence*, 2014.
- 16 K. C. Tai. The tree-to-tree correction problem. *journal of the ACM*, 1979.
- 17 M. J. Zaki and C. C. Aggarwal. XRules: An effective algorithm for structural classification of XML data. *Machine Learning*, 62:137–170, 2006.
- 18 K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 1995.



# Linear-Time Algorithm for Long LCF with $k$ Mismatches

## Panagiotis Charalampopoulos

Department of Informatics, King's College London, London, UK  
panagiotis.charalampopoulos@kcl.ac.uk  
 <https://orcid.org/0000-0002-6024-1557>

## Maxime Crochemore

Department of Informatics, King's College London, London, UK  
maxime.crochemore@kcl.ac.uk  
 <https://orcid.org/0000-0003-1087-1419>

## Costas S. Iliopoulos

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


## Tomasz Kociumaka

Institute of Informatics, University of Warsaw, Warsaw, Poland  
kociumaka@mimuw.edu.pl  
 <https://orcid.org/0000-0002-2477-1702>

## Solon P. Pissis

Department of Informatics, King's College London, London, UK  
solon.pissis@kcl.ac.uk  
 <https://orcid.org/0000-0002-1445-1932>


## Jakub Radoszewski

Institute of Informatics, University of Warsaw, Warsaw, Poland  
jrad@mimuw.edu.pl  
 <https://orcid.org/0000-0002-0067-6401>

## Wojciech Rytter

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

## Tomasz Waleń

Institute of Informatics, University of Warsaw, Warsaw, Poland  
walen@mimuw.edu.pl  
 <https://orcid.org/0000-0002-7369-3309>

---

### Abstract

In the Longest Common Factor with  $k$  Mismatches ( $\text{LCF}_k$ ) problem, we are given two strings  $X$  and  $Y$  of total length  $n$ , and we are asked to find a pair of maximal-length factors, one of  $X$  and the other of  $Y$ , such that their Hamming distance is at most  $k$ . Thankachan et al. [27] show that this problem can be solved in  $\mathcal{O}(n \log^k n)$  time and  $\mathcal{O}(n)$  space for constant  $k$ . We consider the  $\text{LCF}_k(\ell)$  problem in which we assume that the sought factors have length at least  $\ell$ . We use difference covers to reduce the  $\text{LCF}_k(\ell)$  problem with  $\ell = \Omega(\log^{2k+2} n)$  to a task involving  $m = \mathcal{O}(n / \log^{k+1} n)$  *synchronized* factors. The latter can be solved in  $\mathcal{O}(m \log^{k+1} m)$  time, which results in a linear-time algorithm for  $\text{LCF}_k(\ell)$  with  $\ell = \Omega(\log^{2k+2} n)$ . In general, our solution to the  $\text{LCF}_k(\ell)$  problem for arbitrary  $\ell$  takes  $\mathcal{O}(n + n \log^{k+1} n / \sqrt{\ell})$  time.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Pattern matching



© Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń; licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 23; pp. 23:1–23:16



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

**Keywords and phrases** longest common factor, longest common substring, Hamming distance, heavy-light decomposition, difference cover

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.23

**Related Version** A full version of the paper is available at <http://arxiv.org/abs/1802.06369>.

**Funding** Jakub Radoszewski is supported by the “Algorithms for text processing with errors and uncertainties” project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund. Wojciech Rytter and Tomasz Waleń are supported by the Polish National Science Center, grant no. 2014/13/B/ST6/00770.

## 1 Introduction

The longest common factor (LCF) problem is a classical and well-studied problem in theoretical computer science. It consists in finding a maximal-length factor of a string  $X$  occurring in another string  $Y$ . When  $X$  and  $Y$  are over a linearly-sortable alphabet, the LCF problem can be solved in the optimal  $\mathcal{O}(n)$  time and space [17, 15], where  $n$  is the total length of  $X$  and  $Y$ . Considerable efforts have thus been made on improving the *additional* working space; namely, the space required for computations, not taking into account the space providing read-only access to  $X$  and  $Y$ . We refer the interested reader to [25, 21].

In many bioinformatics applications and elsewhere, it is relevant to consider potential alterations within the pair of input strings (e.g. DNA sequences). It is thus natural to define the LCF problem under a distance metric model. The problem then consists in finding a pair of maximal-length factors of  $X$  and  $Y$  whose distance is at most  $k$ . In fact, this problem has received much attention recently, in particular due to its applications in alignment-free sequence comparison [29, 22].

Under the Hamming distance model, the problem is known as the LONGEST COMMON FACTOR WITH AT MOST  $k$  MISMATCHES (LCF $_k$ ) problem. The restricted case of  $k = 1$  was first considered in [4], where an  $\mathcal{O}(n^2)$ -time and  $\mathcal{O}(n)$ -space solution was given. It was later improved by Flouri et al. [12], who built heavily on a technique by Crochemore et al. [11] to obtain  $\mathcal{O}(n \log n)$  time and  $\mathcal{O}(n)$  space.

For a general value of  $k$ , the problem can be solved in  $\mathcal{O}(n^2)$  time and space by a dynamic programming algorithm, but more efficient solutions have been devised. Leimeister and Morgenstern [22] first suggested a greedy heuristic algorithm. Flouri et al. [12] proposed an  $\mathcal{O}(n^2)$ -time algorithm that uses  $\mathcal{O}(1)$  additional space. Grabowski [13] presented two algorithms with running times  $\mathcal{O}(n((k+1)(\ell_0+1))^k)$  and  $\mathcal{O}(n^2k/\ell_k)$ , where  $\ell_0$  and  $\ell_k$  are, respectively, the length of an LCF of  $X$  and  $Y$  and the length of an LCF of  $X$  and  $Y$  with at most  $k$  mismatches. Thankachan et al. [27] proposed an  $\mathcal{O}(n \log^k n)$ -time and  $\mathcal{O}(n)$ -space algorithm (for any constant  $k$ ).

Abboud et al. [1] employed the polynomial method to obtain a  $k^{1.5}n^2/2^{\Omega(\sqrt{\frac{\log n}{k}})}$ -time randomized algorithm. Kociumaka et al. [20] showed that a strongly subquadratic-time algorithm for the LCF $_k$  problem, for binary strings and  $k = \Omega(\log n)$ , refutes the Strong Exponential Time Hypothesis [19, 18]. Thus, subquadratic-time solutions for approximate variants of the problem have been developed [20, 24]. The average-case complexity of this problem has also been considered [28, 2, 3].

## 1.1 Our Contribution

We consider the following variant of the LONGEST COMMON FACTOR WITH AT MOST  $k$  MISMATCHES problem in which the result is constrained to have at least a given length. Let  $\text{LCF}_k(X, Y)$  denote the length of the longest common factor of  $X$  and  $Y$  with at most  $k$  mismatches.

LCF OF LENGTH AT LEAST  $\ell$  WITH AT MOST  $k$  MISMATCHES ( $\text{LCF}_k(X, Y, \ell)$ )

**Input:** Two strings  $X$  and  $Y$  of total length  $n$  and integers  $k \geq 0$  and  $\ell \geq 1$

**Output:**  $\text{LCF}_k(X, Y)$  if it is at least  $\ell$ , and “NONE” otherwise.

We focus on a special case of this problem with  $\ell = \Omega(\log^{2k+2} n)$ . Apart from its theoretical interest, solutions to the  $\text{LCF}_k(X, Y, \ell)$  problem may prove to be useful from a practical standpoint. The  $\text{LCF}_k$  length has been used as a measure of sequence similarity [29, 22]. It is thus assumed that similar sequences share relatively long factors with  $k$  mismatches.

We show an  $\mathcal{O}(n)$ -time algorithm for the  $\text{LCF}_k(X, Y, \ell)$  problem with  $\ell = \Omega(\log^{2k+2} n)$ . Moreover, we prove that the  $\text{LCF}_k(X, Y, \ell)$  problem can be solved in  $\mathcal{O}(n + n \log^{k+1} n / \sqrt{\ell})$  time for arbitrary  $\ell$  and constant  $k$ . In the final section we discuss the complexity for  $k = \mathcal{O}(\log n)$ . This unveils that the  $\mathcal{O}(\cdot)$  notation hides a multiplicative factor that is actually subconstant in  $k$ .

For simplicity, we only describe how to compute the length  $\text{LCF}_k(X, Y)$ . It is straightforward to amend our solution so that it extracts the corresponding factors of  $X$  and  $Y$ .

**Toolbox.** We use the following algorithmic tools:

- Difference covers (see, e.g., [23, 8]) let us reduce the  $\text{LCF}_k(X, Y, \ell)$  problem to searching for longest common prefixes and suffixes with at most  $k$  mismatches ( $\text{LCP}_k, \text{LCS}_k$ ) at positions belonging to sets  $A$  in  $X$  and  $B$  in  $Y$  such that  $|A|, |B| = \mathcal{O}(n/\sqrt{\ell})$ .
- We use a technique of recursive heavy-path decompositions by Cole et al. [9], already applied in the context of the  $\text{LCF}_k$  problem by Thankachan et al. [27], to reduce computing  $\text{LCP}_k, \text{LCS}_k$  to computing  $\text{LCP}, \text{LCS}$  in sets of modified prefixes and suffixes starting at positions in  $A$  and  $B$ . Modifications consist in at most  $k$  changes and increase the size of the problem by a factor of  $\mathcal{O}(\log^k n)$ . We adjust the original technique of Cole et al. [9] so that all modified strings are stored in one compacted trie.
- Finally we apply to the compacted trie a solution to a problem on colored trees that is the cornerstone of the previous  $\mathcal{O}(n \log n)$ -time solution for the  $\text{LCF}_1$  problem by Flouri et al. [12] (and originates from efficient merging of AVL trees [7]).

In total we arrive at  $\mathcal{O}(n + n \log^{k+1} n / \sqrt{\ell})$  time complexity for the  $\text{LCF}_k(X, Y, \ell)$  problem.

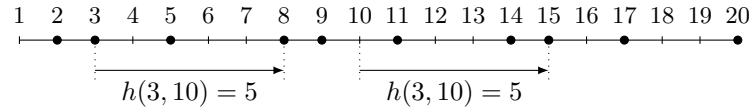
## 2 Preliminaries

Henceforth we denote the input strings by  $X$  and  $Y$  and their common length by  $n$ .

The  $i$ -th letter of a string  $U$ , for  $1 \leq i \leq |U|$ , is denoted by  $U[i]$ . By  $[i..j]$  we denote the integer interval  $\{i, \dots, j\}$  and by  $U[i..j]$  we denote the string  $U[i] \dots U[j]$  that we call a factor of  $U$ . For simplicity, we denote  $U[..i] = U[1..i]$  and  $U[i..] = U[i..|U|]$ . By  $U^R$  we denote the mirror image of  $U$ .

For a pair of strings  $U$  and  $V$  such that  $|U| = |V|$ , we define their Hamming distance as  $d_H(U, V) = |\{1 \leq i \leq |U| : U[i] \neq V[i]\}|$ . For two strings  $U, V$  and a non-negative integer  $d$ , we define

$$\text{LCP}_d(U, V) = \max\{p \leq |U|, |V| : d_H(U[1..p], V[1..p]) \leq d\}.$$



■ **Figure 1** An example of a 6-cover  $\mathbf{S}_{20}(6) = \{2, 3, 5, 8, 9, 11, 14, 15, 19, 20\}$ , with the elements marked as black circles. For example, we may have  $h(3, 10) = 5$  since  $3 + 5, 10 + 5 \in \mathbf{S}_{20}(6)$ .

Let  $T$  be the trie of a collection of strings  $\mathcal{F}$ . The *compacted trie* of  $\mathcal{F}$ ,  $\mathcal{T}(\mathcal{F})$ , contains the root, the branching nodes, and the terminal nodes of  $T$ . Each edge of the compacted trie may represent several edges of  $\mathcal{T}$  and is labeled by a factor of one of the strings  $F_i$ , stored in  $\mathcal{O}(1)$  space. The edges outgoing from a node are labeled by the first letter of the respective strings. The size of a compacted trie is  $\mathcal{O}(|\mathcal{F}|)$ . The best-known example of a compacted trie is the suffix tree of a string; see [10].

## 2.1 Difference covers

We say that a set  $\mathbf{S}(d) \subseteq \mathbb{Z}_+$  is a  $d$ -cover if there is a constant-time computable function  $h$  such that for  $i, j \in \mathbb{Z}_+$  we have  $0 \leq h(i, j) < d$  and  $i + h(i, j), j + h(i, j) \in \mathbf{S}(d)$  (see Figure 1). The following fact synthesizes a well-known construction implicitly used in [8], for example.

► **Fact 1** ([23, 8]). *For each  $d \in \mathbb{Z}_+$  there is a  $d$ -cover  $\mathbf{S}(d)$  such that  $\mathbf{S}_n(d) := \mathbf{S}(d) \cap [1..n]$  is of size  $\mathcal{O}(\frac{n}{\sqrt{d}})$  and can be constructed in  $\mathcal{O}(\frac{n}{\sqrt{d}})$  time.*

## 2.2 Colored Trees Problem

As a component of our solution we use the following problem for colored trees:

### COLORED TREES PROBLEM

**Input:** Two trees  $T_1$  and  $T_2$  containing blue and red leaves such that each internal node is branching (except for, possibly, the root). Each leaf has a number between 1 and  $m$ . Each tree has at most one red leaf and at most one blue leaf with a given number. The nodes of  $T_1$  and  $T_2$  are weighted such that children are at least as heavy as their parent.  
**Output:** A node  $v_1$  of  $T_1$  and a node  $v_2$  of  $T_2$  with maximum total weight such that  $v_1$  and  $v_2$  have at least one blue leaf of the same number and at least one red leaf of the same number in their subtrees.

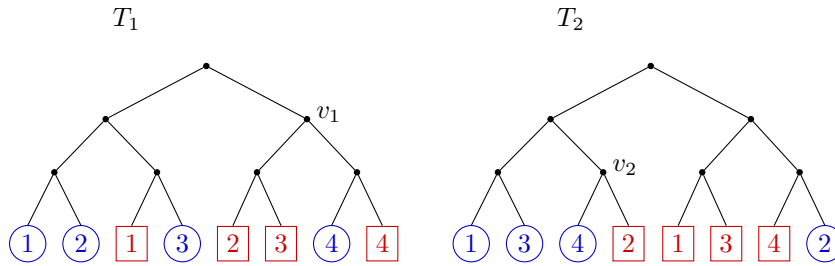
This abstract problem lies at the heart of the algorithm of Flouri et al. [12] for the LONGEST COMMON FACTOR WITH 1 MISMATCH problem. They solve it in  $\mathcal{O}(m \log m)$  time applying a solution inspired by an algorithm of Crochemore et al. [11] finding the longest repeat with a block of  $k$  don't cares, which, in turn, is based on the fact that two AVL trees can be merged efficiently [7].

► **Fact 2** ([11, 12]). *COLORED TREES PROBLEM can be solved in  $\mathcal{O}(m \log m)$  time.*

In our solution we actually use the following problem related to families of strings represented on a compacted trie. It reduces to the COLORED TREES PROBLEM.

### TWO STRING FAMILIES LCP PROBLEM

**Input:** A compacted trie  $\mathcal{T}(\mathcal{F})$  of a family of strings  $\mathcal{F}$  and two sets  $\mathcal{P}, \mathcal{Q} \subseteq \mathcal{F}^2$   
**Output:** The value  $\max\text{PairLCP}(\mathcal{P}, \mathcal{Q})$ , defined as  
 $\max\text{PairLCP}(\mathcal{P}, \mathcal{Q}) = \max\{\text{LCP}(P_1, Q_1) + \text{LCP}(P_2, Q_2) : (P_1, P_2) \in \mathcal{P} \text{ and } (Q_1, Q_2) \in \mathcal{Q}\}$



■ **Figure 2** Example instance for COLORED TREES PROBLEM. Assuming that each node has weight equal to the distance from the root, the optimal solution is a pair of nodes  $(v_1, v_2)$  as shown in the figure. Both  $v_1$  and  $v_2$  have as a descendant a blue leaf with number 4 and a red leaf with number 2.

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

**Proof.** First, we create two copies  $\mathcal{T}_1$  and  $\mathcal{T}_2$  of the tree  $\mathcal{T}(\mathcal{F})$ , removing the edge labels but preserving the node weights  $w(v)$  equal to the sum of lengths of edges on the path to the root.

Next, for each  $(P_1, P_2) \in \mathcal{P}$  we attach a blue leaf to the terminal node of  $\mathcal{T}_1$  representing  $P_1$  and to the terminal of  $\mathcal{T}_2$  representing  $P_2$ . We label these two blue leaves with a unique label, denoted here  $L_{\mathcal{P}}(P_1, P_2)$ . Similarly, for each  $(Q_1, Q_2) \in \mathcal{Q}$ , we attach red leaves to the terminal node of  $\mathcal{T}_1$  representing  $Q_1$  and the terminal node of  $\mathcal{T}_2$  representing  $Q_2$ . We label these two red leaves with a unique label  $L_{\mathcal{Q}}(Q_1, Q_2)$ . Finally, in both  $\mathcal{T}_1$  and  $\mathcal{T}_2$  we remove all nodes which do not contain any colored leaf in their subtrees and dissolve all nodes with exactly one child (except for the roots). This way, each tree  $\mathcal{T}_i$  contains  $\mathcal{O}(|\mathcal{P}| + |\mathcal{Q}|)$  nodes, including  $|\mathcal{P}| + |\mathcal{Q}|$  leaves, each with a distinct label.

Observe that for  $(P_1, P_2) \in \mathcal{P}$ ,  $(Q_1, Q_2) \in \mathcal{Q}$ , and  $j \in \{1, 2\}$ , the value  $\text{LCP}(P_j, Q_j)$  is the weight of the lowest common ancestor (LCA) in  $\mathcal{T}_j$  of the two leaves with labels  $L_{\mathcal{P}}(P_1, P_2)$  and  $L_{\mathcal{Q}}(Q_1, Q_2)$ . Consequently, our task can be formulated as follows: Find a pair of internal nodes  $v_1 \in \mathcal{T}_1$  and  $v_2 \in \mathcal{T}_2$  of maximal total weight  $w(v_1) + w(v_2)$  so that the subtrees rooted at  $v_1$  and  $v_2$  contain blue leaves with the same label and red leaves with the same label. This is exactly the COLORED TREES PROBLEM that can be solved in  $\mathcal{O}(m \log m)$  time, where  $m = |\mathcal{T}_1| + |\mathcal{T}_2| = \mathcal{O}(|\mathcal{P}| + |\mathcal{Q}|)$  (Fact 2). ◀

### 3 Reduction of $\text{LCF}_k(\ell)$ problem to multiple synchronized $\text{LCP}_k$ 's

Let  $U$  be a string of length  $n$ . We denote:

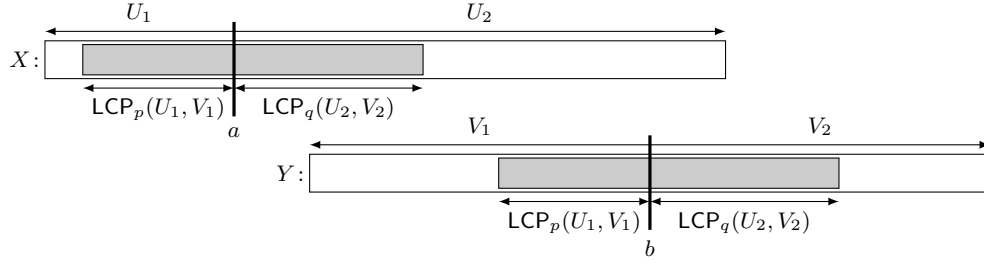
$$\text{Pairs}_{\ell}(U) = \{((U[. . i - 1])^R, U[i . .]) : i \in \mathbf{S}_n(\ell)\}.$$

Observe that  $|\text{Pairs}_{\ell}(U)| = |\mathbf{S}_n(\ell)| = \mathcal{O}(n/\sqrt{\ell})$ .

► **Lemma 4.** *If  $\text{LCF}_k(X, Y) \geq \ell$ , then*

$$\text{LCF}_k(X, Y) = \max_{p+q=k} \{\text{LCP}_p(U_1, V_1) + \text{LCP}_q(U_2, V_2) : (U_1, U_2) \in \text{Pairs}_{\ell}(X), (V_1, V_2) \in \text{Pairs}_{\ell}(Y)\}.$$

**Proof.** First, assume that  $(U_1, U_2) \in \text{Pairs}_{\ell}(X)$ ,  $(V_1, V_2) \in \text{Pairs}_{\ell}(Y)$ , and  $k = p + q$ . Let  $\tilde{U}_1$  and  $\tilde{V}_1$  be prefixes of  $U_1$  and  $V_1$  (respectively) of length  $\text{LCP}_p(U_1, V_1)$ , and let  $\tilde{U}_2$  and  $\tilde{V}_2$  be



■ **Figure 3** If  $\text{LCF}_k(X, Y) \geq \ell$ , then there exist  $(U_1, U_2) \in \text{Pairs}_\ell(X)$  and  $(V_1, V_2) \in \text{Pairs}_\ell(Y)$  such that  $\text{LCF}_k(X, Y) = \text{LCP}_p(U_1, U_2) + \text{LCP}_q(V_1, V_2)$  for some  $p + q = k$ .

prefixes of  $U_2$  and  $V_2$  (respectively) of length  $\text{LCP}_q(U_2, V_2)$ . Observe that  $\tilde{U}_1^R \tilde{U}_2$  is a factor of  $X = U_1^R U_2$  and  $\tilde{V}_1^R \tilde{V}_2$  is a factor of  $Y = V_1^R V_2$ . Moreover,

$$d_H(\tilde{U}_1^R \tilde{U}_2, \tilde{V}_1^R \tilde{V}_2) = d_H(\tilde{U}_1, \tilde{U}_2) + d_H(\tilde{V}_1, \tilde{V}_2) \leq p + q = k.$$

Consequently,

$$\text{LCF}_k(X, Y) \geq |\tilde{U}_1^R \tilde{U}_2| = |\tilde{U}_1| + |\tilde{U}_2| = \text{LCP}_p(U_1, V_1) + \text{LCP}_q(U_2, V_2).$$

This concludes the proof of the lower bound on  $\text{LCF}_k(X, Y)$ .

Next, let  $X[i..i']$  and  $Y[j..j']$  be an optimal pair of factors; see Figure 3. They satisfy

$$|X[i..i']| = |Y[j..j']| = \text{LCF}_k(X, Y) \geq \ell \quad \text{and} \quad d_H(X[i..i'], Y[j..j']) \leq k.$$

Denote  $a = i + h(i, j)$  and  $b = j + h(i, j)$ , where  $h$  is the shift function associated with the  $\ell$ -cover  $\mathbf{S}(\ell)$ . Note that  $a \in [i..i'] \cap \mathbf{S}(\ell)$  and  $b \in [j..j'] \cap \mathbf{S}(\ell)$ . Consequently,  $(U_1, U_2) := ((X[. . a - 1])^R, X[a..]) \in \text{Pairs}_\ell(X)$  and  $(V_1, V_2) := ((Y[. . b - 1])^R, Y[b..]) \in \text{Pairs}_\ell(Y)$ . Moreover,

$$k \geq d_H(X[i..i'], Y[j..j']) = d_H(X[i..a-1], Y[j..b-1]) + d_H(X[a..i'], Y[b..j']).$$

Therefore, for  $p = d_H(X[i..a-1], Y[j..b-1])$  and  $q = k - p$ , we have

$$\text{LCP}_p(U_1, V_1) + \text{LCP}_q(V_2, U_2) \geq |X[i..a-1]| + |X[a..i']| = |X[i..i']| = \text{LCF}_k(X, Y).$$

This concludes the proof. ◀

#### 4 The case of $k = 0$ and of $k = 1$ and $\sigma = 2$

In this section, as a warm-up, we show how the TWO STRING FAMILIES LCP PROBLEM can be used to solve two special cases of the  $\text{LCF}_k(X, Y, \ell)$  problem. Then in Section 6 we explain how it can be used to solve the problem in full generality.

In order to solve the  $\text{LCF}_k(X, Y, \ell)$  problem for  $k = 0$ , we observe that, by Lemma 4, if  $\text{LCF}_0(X, Y) \geq \ell$ , then  $\text{LCF}_0(X, Y) = \max\text{PairLCP}(\text{Pairs}_\ell(X), \text{Pairs}_\ell(Y))$ . Thus, we simply build the joint suffix tree  $\mathcal{T}$  of  $X, Y, X^R$ , and  $Y^R$ , and we solve the appropriate instance of TWO STRING FAMILIES LCP PROBLEM.

The preprocessing time is clearly  $\mathcal{O}(n)$ , while solving the TWO STRING FAMILIES LCP PROBLEM takes  $\mathcal{O}(n + n \log n / \sqrt{\ell})$  time, which is  $\mathcal{O}(n)$  provided that  $\ell = \Omega(\log^2 n)$ .



For  $k \geq 1$ , we would ideally like to extend the family  $\text{Pairs}_\ell(S)$  to  $\text{Pairs}_\ell^{(k)}(S)$  replacing the suffixes and reversed prefixes of  $S$  with their approximate copies so that

$$\text{LCP}_k(X, Y) = \max \text{PairLCP}(\text{Pairs}_\ell^{(k)}(X), \text{Pairs}_\ell^{(k)}(Y)).$$

A very naive solution would be to extend the alphabet  $\Sigma$  to  $\Sigma_\$$  adding a symbol  $\$ \notin \Sigma$ , and for each  $(S_1, S_2) \in \text{Pairs}_\ell(S)$  to replace an arbitrary subset of  $k$  symbols with  $\$$ 's. However, this results in  $\binom{n}{k}$  copies of each  $(S_1, S_2) \in \text{Pairs}_\ell(S)$ , which is by far too much.

Our approach is therefore based on the technique of Cole et al. [9], which has already been used in the context of the LONGEST COMMON FACTOR WITH AT MOST  $k$  MISMATCHES problem by Thankachan et al. [27]. It allows us to reduce the number of approximate copies of each  $(S_1, S_2) \in \text{Pairs}_\ell(S)$  to  $\mathcal{O}(\log^k n)$ . However, the sets  $\text{Pairs}_\ell^{(k)}(X)$  and  $\text{Pairs}_\ell^{(k)}(Y)$  cannot be constructed independently, and we actually have to build several pairs of such sets rather just one.

Below, we explain the main points for  $k = 1$  and  $\sigma = 2$ .

Let  $\mathcal{F}$  be a family consisting of the suffixes of  $X$ ,  $X^R$ ,  $Y$ , and  $Y^R$ , appearing in  $\text{Pairs}_\ell(X)$  or  $\text{Pairs}_\ell(Y)$ . We apply the heavy-light decomposition on the compacted trie  $\mathcal{T}(\mathcal{F})$ ; this technique can be summarized as follows:

► **Fact 5** (Tarjan [26]). *If  $T$  is a rooted tree, then in linear time we can mark some edges in  $T$  as light so that:*

- *each node has at most one outgoing edge which is not light,*
- *each root-to-leaf path contains  $\mathcal{O}(\log |T|)$  light edges.*

Next, for each string  $F \in \mathcal{F}$ , we construct a set  $N(F)$  consisting of  $F$  and any string which can be obtained from  $F$  by flipping the first symbol on one of the light edges on the path representing  $F$  in  $\mathcal{T}(\mathcal{F})$ . By Fact 5, we have  $|N(F)| = \mathcal{O}(\log |\mathcal{F}|) = \mathcal{O}(\log n)$ .

Let us introduce two subsets of  $N(F)$ :  $N_0(F) = \{F\}$  and  $N_1(F) = N(F)$ . These sets have been constructed so that they enjoy the following crucial property:

► **Lemma 6.** *If  $F, G \in \mathcal{F}$ , then*

$$\text{LCP}_1(F, G) = \max_{d_1+d_2=1} \{\text{LCP}(F', G') : F' \in N_{d_1}(F), G' \in N_{d_2}(G)\}.$$

**Proof.** First, let us bound  $\text{LCP}_1(F, G)$  from below. Let  $p = \text{LCP}(F', G')$  be the maximum on the right-hand side, We have

$$\begin{aligned} d_H(F[..\ p], G[..\ p]) &= d_H(F[..\ p], F'[..\ p]) + d_H(G'[..\ p], G[..\ p]) \leq \\ &\leq d_H(F, F') + d_H(G', G) \leq d_1 + d_2 = 1. \end{aligned}$$

Consequently,  $\text{LCP}_1(F, G) \geq p$  as claimed.

To bound  $\text{LCP}_1(F, G)$  from above, let us consider terminal nodes  $v_F$  and  $v_G$  in  $\mathcal{T}(\mathcal{F})$  representing  $F$  and  $G$ , respectively, and their LCA  $v$ . If  $v = v_F$  or  $v = v_G$ , then  $\text{LCP}_1(F, G) = \text{LCP}(F, G)$  and the claimed bound holds due to  $F \in N_0(F)$  and  $G \in N_1(G)$  (and vice versa). Otherwise, the edge from  $v$  towards  $v_F$  or the edge from  $v$  towards  $v_G$  has to be light (according to Fact 5). If the former edge is light, then  $N_1(F)$  contains a string  $F'$  obtained from  $F$  by flipping the first letter on that edge. Such a string  $F'$  satisfies  $\text{LCP}_1(F, G) = \text{LCP}(F', G)$ , so the claimed bound holds due to  $G \in N_0(G)$ . Symmetrically, if the edge towards  $v_G$  is light, then  $\text{LCP}_1(F, G) = \text{LCP}(F, G')$  for some  $G' \in N_1(G)$ . ◀

For  $S \in \{X, Y\}$  and  $d \in \{0, 1\}$ , let us define

$$\text{Pairs}_\ell^{(d)}(S) = \bigcup_{\substack{(U_1, U_2) \in \text{Pairs}_\ell(S) \\ d_1 + d_2 = 1}} \{(U'_1, U'_2) : U'_1 \in N_{d_1}(U_1), U'_2 \in N_{d_2}(U_2)\}.$$

Observe that  $\text{Pairs}_\ell^{(0)}(S) = \text{Pairs}_\ell(S)$ , whereas the set  $\text{Pairs}_\ell^{(1)}(S)$  satisfies  $|\text{Pairs}_\ell^{(1)}(S)| = \mathcal{O}(|\text{Pairs}_\ell(S)| \log |\mathcal{F}|) = \mathcal{O}(n \log n / \sqrt{\ell})$ . Lemmas 4 and 6 yield the following

► **Corollary 7.** *If  $\text{LCF}_1(X, Y) \geq \ell$ , then*

$$\text{LCF}_1(X, Y) = \max_{k_1 + k_2 = 1} \max \text{PairLCP}(\text{Pairs}_\ell^{(k_1)}(X), \text{Pairs}_\ell^{(k_2)}(Y)).$$

**Proof.** By Lemma 4, we have  $\text{LCF}_1(X, Y) = \text{LCP}_p(U_1, V_1) + \text{LCP}_q(U_2, V_2)$  for some  $(U_1, U_2) \in \text{Pairs}_\ell(X)$ ,  $(V_1, V_2) \in \text{Pairs}_\ell(Y)$ , and  $p + q = 1$ . Lemma 6 yields that  $\text{LCP}_p(U_1, V_1) = \text{LCP}(U'_1, V'_1)$  for some  $U'_1 \in N_{p_1}(U_1)$  and  $V'_1 \in N_{p_2}(V_1)$  such that  $p = p_1 + p_2$ . Similarly,  $\text{LCP}_q(U_2, V_2) = \text{LCP}(U'_2, V'_2)$  for some  $U'_2 \in N_{q_1}(U_2)$  and  $V'_2 \in N_{q_2}(V_2)$ . Observe that  $(U'_1, U'_2) \in \text{Pairs}_\ell^{(p_1 + q_1)}(X)$  and  $(V'_1, V'_2) \in \text{Pairs}_\ell^{(p_2 + q_2)}(Y)$ , so

$$\text{LCF}_1(X, Y) \leq \max \text{PairLCP}(\text{Pairs}_\ell^{(k_1)}(X), \text{Pairs}_\ell^{(k_2)}(Y))$$

for  $k_i = p_i + q_i$  (which satisfy  $k_1 + k_2 = p + q = 1$ , as claimed).

Next, suppose that  $(U'_1, U'_2) \in \text{Pairs}_\ell^{(k_1)}(X)$  and  $(V'_1, V'_2) \in \text{Pairs}_\ell^{(k_2)}(Y)$ . We shall prove that  $\text{LCF}_1(X, Y) \geq \text{LCP}(U'_1, V'_1) + \text{LCP}(U'_2, V'_2)$ . Note that  $U'_1 \in N_{p_1}$  and  $U'_2 \in N_{q_1}(U_2)$  for some  $p_1 + q_1 = k_1$  and  $(U_1, U_2) \in \text{Pairs}_\ell(X)$ ; symmetrically,  $V'_1 \in N_{p_2}$  and  $V'_2 \in N_{q_2}(V_2)$  for some  $p_2 + q_2 = k_2$  and  $(V_1, V_2) \in \text{Pairs}_\ell(Y)$ . By Lemma 6,  $\text{LCP}(U'_1, V'_1) \leq \text{LCP}_{p_1 + p_2}(U_1, V_1)$  and  $\text{LCP}(U'_2, V'_2) \leq \text{LCP}_{q_1 + q_2}(U_2, V_2)$ . Hence, the claimed bound holds due to Lemma 4:

$$\text{LCF}_1(X, Y) \geq \text{LCP}_{p_1 + p_2}(U_1, V_1) + \text{LCP}_{q_1 + q_2}(U_2, V_2) \geq \text{LCP}(U'_1, V'_1) + \text{LCP}(U'_2, V'_2).$$

This concludes the proof. ◀

Consequently, it suffices to solve two instances of TWO STRING FAMILIES LCP PROBLEM, with  $(\mathcal{P}, \mathcal{Q})$  equal to  $(\text{Pairs}_\ell^{(0)}(X), \text{Pairs}_\ell^{(1)}(Y))$  and  $(\text{Pairs}_\ell^{(1)}(X), \text{Pairs}_\ell^{(0)}(Y))$ , respectively.

► **Proposition 8.** *The  $\text{LCF}_k(X, Y, \ell)$  problem for  $k = 1$  and binary alphabet can be solved in  $\mathcal{O}(n + n \log^2 n / \sqrt{\ell})$  time. If  $\ell = \Omega(\log^4 n)$ , this running time is  $\mathcal{O}(n)$ .*

**Proof.** First, we build the sets  $\text{Pairs}_\ell(X)$  and  $\text{Pairs}_\ell(Y)$ . Next, we construct the joint suffix tree of strings  $X, Y, X^R, Y^R$  (along with a component for constant-time LCA queries [5]) and we extract the compacted trie  $\mathcal{T}(\mathcal{F})$  of the family  $\mathcal{F}$ . Then, we process light edges on  $\mathcal{T}(\mathcal{F})$  (determined by Fact 5) to construct the sets  $N(F)$  as defined above for each  $F \in \mathcal{F}$ . We initialize each set  $N(F)$  with  $F$ ; then, for every light edge  $e$ , we traverse the subtree below  $e$  and for each terminal node (representing  $F \in \mathcal{F}$ ), we insert to  $N(F)$  a string  $F'$  obtained from  $F$  by flipping the first letter represented by  $e$ . Technically, in  $N(F)$  we just store the set of positions for which  $F$  should be flipped to obtain  $F'$ .

To compute the compacted trie  $\mathcal{T}(\mathcal{F}')$  of a family  $\mathcal{F}' = \bigcup_{F \in \mathcal{F}} N(F)$ , we sort the strings in  $F' \in \mathcal{F}'$  using a comparison-based algorithm. Next, we extend the representation of  $N(F)$  so that each  $F' \in N(F)$  stores a pointer to the corresponding terminal node in  $\mathcal{T}(\mathcal{F}')$ . This way, we can generate sets  $\text{Pairs}_\ell^d(S)$  for  $d \in \{0, 1\}$  and  $S \in \{X, Y\}$  with strings represented as pointers to terminal nodes of  $\mathcal{T}(\mathcal{F}')$ . Finally, we solve two instances of TWO STRING FAMILIES LCP PROBLEM according to Corollary 7.

We conclude with the running-time analysis. In the preprocessing, we spend  $\mathcal{O}(n)$  time to construct the joint suffix tree. Then, applying Fact 5 to build the sets  $N(F)$  for  $F \in \mathcal{F}$  takes  $\mathcal{O}(|\mathcal{F}| \log |\mathcal{F}|) = \mathcal{O}(n \log n / \sqrt{\ell})$  time. We spend further  $\mathcal{O}(|\mathcal{F}'| \log |\mathcal{F}'|) = \mathcal{O}(n \log^2 n / \sqrt{\ell})$  time to construct  $\mathcal{T}(\mathcal{F}')$ . Since  $|\text{Pairs}_\ell^{(d)}(S)| = \mathcal{O}(|\mathcal{F}| \log |\mathcal{F}|)$  for  $d \in \{0, 1\}$  and  $S \in \{X, Y\}$ , the time to solve both instances of the TWO STRING FAMILIES LCP PROBLEM is also  $\mathcal{O}(n \log^2 n / \sqrt{\ell})$  (see Lemma 3). Hence, the overall time complexity is  $\mathcal{O}(n + n \log^2 n / \sqrt{\ell})$ . ◀

## 5 Arbitrary $k$ and $\sigma$

In this section, we describe the core concepts of our solution for arbitrary number of mismatches  $k$  and alphabet size  $\sigma$ . They depend heavily on the ideas behind the  $\mathcal{O}(n \log^k n)$ -time solution to the LCF $_k$  problem [27], which originate in approximate indexing [9].

► **Definition 9.** Consider strings  $U, V \in \Sigma^*$  and an integer  $d \geq 0$ . We say that strings  $U', V' \in \Sigma_\$^*$  form a  $(U, V)_d$ -pair if

- $|U'| = |U|$  and  $|V'| = |V|$ ;
- if  $i > \text{LCP}_d(U, V)$  or  $U[i] = V[i]$ , then  $U'[i] = U[i]$  and  $V'[i] = V[i]$ ;
- otherwise,  $U'[i] = V'[i] \in \{U[i], V[i], \$\}$ .

For a string  $S \in \Sigma_\$^*$  let us define  $\#_\$(S) = |\{1 \leq i \leq |S| : S[i] = \$\}|$ . The following observation specifies key properties of  $(U, V)_d$  pairs.

► **Observation 10.** Consider strings  $U, V \in \Sigma^*$  and an integer  $d \geq 0$ . If strings  $U', V' \in \Sigma_\$^*$  form a  $(U, V)_d$ -pair, then the following conditions hold:

- (a)  $\text{LCP}(U', V') = \text{LCP}_d(U, V)$ ,
- (b)  $\#_\$(U') = \#_\$(V')$ ,
- (c)  $d = d_H(U, U') - \frac{1}{2}\#_\$(U') + d_H(V, V') - \frac{1}{2}\#_\$(V')$ .

► **Definition 11.** Consider a finite family of strings  $\mathcal{F} \subseteq \Sigma^*$ . We say that sets  $N(F) \subseteq \Sigma_\$^*$  for  $F \in \mathcal{F}$  form a  $k$ -complete family if for every  $U, V \in \mathcal{F}$  and  $0 \leq d \leq k$ , there exists a  $(U, V)_d$ -pair  $U', V'$  with  $U' \in N(U)$  and  $V' \in N(V)$ .

► **Remark.** A simple (yet inefficient) way to construct a  $k$ -complete family is to include in  $N(F)$  all strings which can be obtained from  $F$  by replacing up to  $k$  letters with  $\$$ 's. An example of a more efficient family is shown in Table 1.

The following lemma states a property of  $k$ -complete families that we will use in the algorithm. For  $F \in \mathcal{F}$  and  $0 \leq d \leq k$ , let us define  $N_d(F) = \{F' \in N(F) : d_H(F, F') \leq d\}$ . Moreover, for a half integer<sup>1</sup>  $d'$ ,  $0 \leq d' \leq d$ , let

$$N_{d,d'}(F) = \{F' \in N_d(F) : d_H(F, F') - \frac{1}{2}\#_\$(F') \leq d'\},$$

► **Lemma 12.** Let  $N(F)$  for  $F \in \mathcal{F}$  be a  $k$ -complete family. If  $F_1, F_2 \in \mathcal{F}$  and  $0 \leq d \leq k$ , then

$$\text{LCP}_d(F_1, F_2) = \max_{\substack{d_1+d_2=d \\ F'_i \in N_{d,d_i}(F_i)}} \text{LCP}(F'_1, F'_2) = \max_{\substack{d_1+d_2 < d+1 \\ F'_i \in N_{k,d_i}(F_i)}} \text{LCP}(F'_1, F'_2).$$

<sup>1</sup> Here, a half integer is a number of the form  $\frac{a}{2}$ , where  $a$  is an integer.

23:10 Linear-Time Algorithm for Long LCF with  $k$  Mismatches

■ **Table 1** A sample 1-complete family for  $\mathcal{F} = \{\text{abacb}, \text{bacb}, \text{acb}, \text{cb}, \text{b}\}$  (the suffixes of **abacb**) is  $N(\text{b}) = \{\text{a}, \text{b}, \$\}$ ,  $N(\text{cb}) = \{\text{ab}, \text{cb}, \$\text{b}\}$ ,  $N(\text{acb}) = \{\text{abb}, \text{acb}\}$ ,  $N(\text{bacb}) = \{\text{aacb}, \text{bacb}, \$\text{acb}\}$ , and  $N(\text{abacb}) = \{\text{abacb}\}$ . The  $(U, V)_1$ -pairs for all  $U, V \in \mathcal{F}$  are illustrated in the table above. Observe that  $\text{LCP}_1(U, V) = \text{LCP}(U', V')$  for the corresponding  $(U, V)_1$ -pair  $(U', V')$ . Also, note that  $\text{LCP}_1(\text{acb}, \text{cb}) = 1$  even though  $\text{abb} \in N(\text{acb})$ ,  $\text{ab} \in N(\text{cb})$ , and  $\text{LCP}(\text{abb}, \text{ab}) = 2$ .

	<b>b</b>	<b>cb</b>	<b>acb</b>	<b>bacb</b>	<b>abacb</b>
<b>abacb</b>	<u>a</u> <u>abacb</u>	<u>ab</u> <u>abacb</u>	<u>abb</u> <u>abacb</u>	<u>aacb</u> <u>abacb</u>	<u>abacb</u> <u>abacb</u>
<b>bacb</b>	<u>b</u> <u>bacb</u>	<u>\$b</u> <u>\$acb</u>	<u>acb</u> <u>aacb</u>	<u>bacb</u> <u>bacb</u>	<u>abacb</u> <u>aacb</u>
<b>acb</b>	<u>a</u> <u>acb</u>	<u>ab</u> <u>acb</u>	<u>acb</u> <u>acb</u>	<u>aacb</u> <u>acb</u>	<u>abacb</u> <u>abb</u>
<b>cb</b>	<u>\$</u> <u>\$b</u>	<u>cb</u> <u>cb</u>	<u>acb</u> <u>ab</u>	<u>\$acb</u> <u>\$b</u>	<u>abacb</u> <u>ab</u>
<b>b</b>	<u>b</u> <u>b</u>	<u>\$b</u> <u>\$</u>	<u>acb</u> <u>a</u>	<u>bacb</u> <u>b</u>	<u>abacb</u> <u>a</u>

**Proof.** We shall prove that

$$\max_{\substack{d_1+d_2=d \\ F'_i \in N_{d,d_i}(F_i)}} \text{LCP}(F'_1, F'_2) \geq \text{LCP}_d(F_1, F_2) \geq \max_{\substack{d_1+d_2 < d+1 \\ F'_i \in N_{k,d_i}(F_i)}} \text{LCP}(F'_1, F'_2).$$

This is sufficient due to the fact that  $N_{d,d'}(F)$  is monotone with respect to both  $d$  and  $d'$ .

For the first inequality, observe that (by definition of a  $k$ -complete family) the sets  $N(F_1)$  and  $N(F_2)$  contain an  $(F_1, F_2)_d$ -pair  $(F'_1, F'_2)$ . By Observation 10, we have

$$d \geq d_H(F_1[\cdot \cdot |P|], F_2[\cdot \cdot |P|]) = d_H(F_1, F'_1) - \frac{1}{2}\#\$_(F'_1) + d_H(F_2, F'_2) - \frac{1}{2}\#\$_(F'_2).$$

Consequently,  $F'_i \in N_{d,d_i}(F_i)$  for  $d_i = d_H(F_i, F'_i) - \frac{1}{2}\#\$_(F'_i)$ . If  $d > d_1 + d_2$ , we may increase  $d_1$  or  $d_2$ .

For the second inequality, suppose that  $F'_i \in N_{k,d_i}(F_i)$  for  $d_1 + d_2 < d + 1$ . Let  $P$  be the longest common prefix of  $F'_1$  and  $F'_2$ . Then

$$\begin{aligned} d_H(F_1[\cdot \cdot |P|], F_2[\cdot \cdot |P|]) &\leq d_H(F_1[\cdot \cdot |P|], P) + d_H(F_2[\cdot \cdot |P|], P) - \#\$_(P) \leq \\ &\leq d_H(F_1, F'_1) - \#\$_(F'_1) + d_H(F_2, F'_2) - \#\$_(F'_2) + \#\$_(P) \leq d_1 + d_2. \end{aligned}$$

Consequently,  $d_H(F_1[\cdot \cdot |P|], F_2[\cdot \cdot |P|]) \leq d_1 + d_2 < d + 1$ , i.e.,  $d_H(F_1[\cdot \cdot |P|], F_2[\cdot \cdot |P|]) \leq d$ , as claimed. ◀

In the algorithms, we represent a  $k$ -complete family using the compacted trie  $\mathcal{T}(\mathcal{F})$  of the union  $\mathcal{F}' = \bigcup_{F \in \mathcal{F}} N(F)$ . Its terminal nodes  $F'$  are marked with a subset of strings  $F \in \mathcal{F}$  for which  $F' \in N(F)$ ; for convenience we also store  $\#\$_(F')$  and  $d_H(F, F')$ . Each edge is labeled by a factor of  $F \in \mathcal{F}$ , perhaps prepended by  $\$$ .

Our construction of a  $k$ -complete family is based on the results of [9, 27], but below we provide a self-contained proof.

► **Proposition 13** (see also [9, 27]). *Let  $\mathcal{F} \subseteq \Sigma^*$  be a finite family of strings and let  $k \geq 0$  be an integer. There exists a  $k$ -complete family  $N$  such that  $|N_d(F)| \leq 2^d \binom{\log |\mathcal{F}| + d}{d}$  for each  $F \in \mathcal{F}$  and  $0 \leq d \leq k$ . Moreover, the compacted trie  $\mathcal{T}(\mathcal{F}')$  can be constructed in  $\mathcal{O}(2^k |\mathcal{F}| \binom{\log |\mathcal{F}| + k + 1}{k+1})$  time provided constant-time LCP queries for suffixes of the strings  $F \in \mathcal{F}$ .*

---

**Algorithm 1:** A recursive procedure inserting strings with prefix  $P$  to sets  $N(F)$ .

---

**Function**  $\text{Generate}(P, \mathcal{F}_P)$  **is**

$h :=$  a most frequent element of  $\{S[1] : (S, F, b) \in \mathcal{F}_P \text{ and } S \neq \varepsilon\}$ ;

**foreach**  $(S, F, b) \in \mathcal{F}_P$  **do**  $// b = k - d_H(F, PS) \geq 0$

**if**  $S = \varepsilon$  **then**  $N(F) := N(F) \cup \{P\}$ ;

**else**

$c := S[1]$ ;

$\mathcal{F}_{Pc} := \mathcal{F}_{Pc} \cup \{(S[2..], F, b)\}$ ;

**if**  $c \neq h$  **and**  $b > 0$  **then**

$\mathcal{F}_{Ph} := \mathcal{F}_{Ph} \cup \{(S[2..], F, b - 1)\}$ ;

$\mathcal{F}_{P\$} := \mathcal{F}_{P\$} \cup \{(S[2..], F, b - 1)\}$ ;

**foreach**  $c \in \Sigma \cup \{\$\}$  **such that**  $\mathcal{F}_{Pc} \neq \emptyset$  **do**

$\text{Generate}(Pc, \mathcal{F}_{Pc})$ ;

---

**Proof.** We apply a recursive procedure that builds the subtree rooted at the node representing  $P$ . The input  $\mathcal{F}_P$  consists of tuples  $(S, F, b)$  such that  $F \in \mathcal{F}$ ,  $S$  is a suffix of  $F$  of length  $|S| = |F| - |P|$ , and  $b = k - d_H(F, PS) \geq 0$ . Intuitively, the parameter  $b$  can be seen as a “budget” of remaining symbol changes in the string that prevents exceeding the number  $k$  of mismatches. In the first call we have  $P = \varepsilon$  and  $\mathcal{F}_P = \{(F, F, k) : F \in \mathcal{F}\}$ .

In the pseudocode below we state this procedure in an abstract way; afterwards we explain how to implement it efficiently. The 1-complete family from Table 1 is a subset of the family constructed by that procedure.

Correctness of Algorithm 1 is relatively easy to derive. Due to space constraints, the proof of the following claim can be found in the full version.

► **Claim 14.** *For every  $S, T \in \mathcal{F}$  and  $0 \leq d \leq k$ , there exists an  $(S, T)_d$ -pair  $(S', T')$  with  $S' \in N(S)$  and  $T' \in N(T)$ .*

We also refer to the full version for a complete proof of the following bound on  $N_d(F)$ .

► **Claim 15.** *For each  $F \in \mathcal{F}$ , we have  $|N_d(F)| \leq 2^d \binom{\log |\mathcal{F}| + d}{d}$ .*

The idea is to define  $N_{d,P}(F) = \{F' \in N_d(F) : P \text{ is a prefix of } F'\}$  for each  $P \in \Sigma_\$^*$  and to prove the following bound by induction on decreasing  $|P|$ :

$$|N_{d,P}(F)| \leq \begin{cases} 2^b \binom{\log |\mathcal{F}_P| + b}{b} & \text{if } (S, F, b + k - d) \in \mathcal{F}_P \text{ and } b \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

Below, we show that the  $k$ -complete family  $N$  represented as a trie  $T_N$  can be constructed in  $\mathcal{O}(|\mathcal{F}| 2^k \binom{\log |\mathcal{F}| + k + 1}{k + 1})$  time provided constant-time LCP queries for suffixes of strings  $F \in \mathcal{F}$ .

To a tuple  $(S, F, b) \in \mathcal{F}_P$  we assign a number of tokens:

$$\text{Tokens}_P(S, F, b) = C(2^{b+1} - 1) \binom{\log |\mathcal{F}_P| + b + 1}{b + 1}$$

where  $C$  is a sufficiently large constant. We shall inductively prove that  $\text{Generate}(P, \mathcal{F}_P)$  can be implemented in time  $\sum_{(S, F, b) \in \mathcal{F}_P} \text{Tokens}_P(S, F, b)$ .

In the implementation of the procedure we use finger search trees [14], which maintain subsets of a linearly-ordered universe supporting constant-time queries. Among many applications (see [6] for a survey), they support the following two operations [16, 6]:

- insert an element into a set  $A$ , which takes  $\mathcal{O}(\log |A|)$  time,
- for a given key  $t$ , split the set  $A$  into  $A_{\leq t} = \{a \in A : a \leq t\}$  and  $A_{>t} = \{a \in A : a > t\}$ , which takes  $\mathcal{O}(\log \min(|A_{\leq t}|, |A_{>t}|))$  time.

We are now ready to specify how the arguments to the procedure **Generate**  $(P, \mathcal{F}_P)$  are given. The string  $P$  is represented by the corresponding node of the constructed trie  $T_N$ ; we also explicitly store  $|P|$  and  $\#_{\S}(P)$ . The set  $\mathcal{F}_P$  is stored in a finger search tree with tuples  $(S, F, b)$  ordered by  $S$ . However,  $S$  is not stored itself as it is uniquely specified as a suffix of  $F$  of length  $|F| - |P|$ . Thus each element in the tree is stored in  $\mathcal{O}(1)$  space.

First, we process tuples  $(S, F, b)$  with  $S = \varepsilon$ , conveniently located at the beginning of  $\mathcal{F}_P$ . We remove these tuples from  $\mathcal{F}_P$  and store  $F$  at the current node of  $T_N$ . This simulates inserting  $P$  to  $N(F)$ ; we also store auxiliary values  $d_H(P, F) = k - b$  and  $\#_{\S}(P)$ .

Next, we compute the length of longest common prefix  $P'$  of non-empty strings  $S$  with  $(S, F, b) \in \mathcal{F}_P$ . For this, we make an LCP query for the smallest and the largest of these suffixes. If the longest common prefix  $P'$  is non-empty, we observe that  $\mathcal{F}_{PP'} = \mathcal{F}_P$  (with the stored representation unchanged) and Algorithm 1 does not explore any other branch. Hence, we immediately call **Generate**  $(PP', \mathcal{F}_{PP'})$  which corresponds to creating a complete compacted edge of the resulting trie. This step takes  $\mathcal{O}(1)$  time, but it guarantees that **Generate**  $(PP', \mathcal{F}_{PP'})$  outputs or branches. Hence, this time gets amortized.

If  $P' = \varepsilon$ , we partition  $\mathcal{F}_P$  into at most  $\sigma$  finger search trees  $\mathcal{F}_{P,c}$  each storing tuples sharing the letter  $S[1] = c$ , and we identify the heavy letter  $h$  by choosing the largest  $\mathcal{F}_{P,c}$ . For this, we iteratively split out the tree with the smallest unprocessed  $S[1]$ , which takes time proportional to  $\sum_{c \neq h} \log |\mathcal{F}_{P,c}|$ .

The sets  $\mathcal{F}_{P,c}$  for  $c \neq h$  already represented by  $\mathcal{F}_{P,c}$  (note that the order does not change, and the tuples need not be altered since the “budget”  $b$  remains the same and  $S$  is stored implicitly). Similarly, we can build  $\mathcal{F}_{P,h}$  by inserting new tuples into  $\mathcal{F}_{P,h}$ .

Thus, we define  $\mathcal{L}_P := \{(S, F, b) \in \mathcal{F}_P : S \neq \varepsilon \text{ and } S[1] \neq h\}$  and insert to  $\mathcal{F}_{P,h}$  and  $\mathcal{F}_{P,\S}$  tuples  $(S[2..], F, b - 1)$  for  $(S, F, b) \in \mathcal{L}_P$  with  $b > 0$ , which takes  $\mathcal{O}(\log |\mathcal{F}_P|)$  time per element.

In total, the processing time is  $\mathcal{O}(1)$  for each element of  $\mathcal{L}_P$  with  $b = 0$ , and  $\mathcal{O}(\log |\mathcal{F}_P|)$  when  $b > 0$ . Additionally, we may spend  $\mathcal{O}(1)$  time for a tuple with  $S = \varepsilon$ . Let us check that the difference in the number of tokens is sufficient to cover the running time of these operations.

The tuples with  $S = \varepsilon$  do not appear in future computations. Hence, we spend all their tokens on the computations related to them. It is indeed sufficient:

$$\text{Tokens}_P(\varepsilon, F, b) = C(2^{b+1} - 1) \binom{\log |\mathcal{F}_P| + b + 1}{b+1} \geq C \binom{\log |\mathcal{F}_P| + 1}{1} = C(\log |\mathcal{F}_P| + 1) \geq C.$$

We don't spend any time on tuples with  $S[1] = h$ , and number of tokens for such a tuple does not increase:

$$\begin{aligned} & \text{Tokens}_P(S, F, b) - \text{Tokens}_{P,h}(S, F, b) = \\ & C(2^{b+1} - 1) \binom{\log |\mathcal{F}_P| + b + 1}{b+1} - C(2^{b+1} - 1) \binom{\log |\mathcal{F}_{P,h}| + b + 1}{b+1} \geq 0. \end{aligned}$$

Finally, for a tuple with  $S[1] \neq h$  (i.e., in  $\mathcal{L}_P$ ) the difference in the number of tokens is

$$\begin{aligned} & \text{Tokens}_P(S, F, b) - \text{Tokens}_{P,c}(S', F, b) - \text{Tokens}_{P,h}(S', F, b - 1) - \text{Tokens}_{P,\S}(S', F, b - 1) = \\ & = C(2^{b+1} - 1) \binom{\log |\mathcal{F}_P| + b + 1}{b+1} - C(2^{b+1} - 1) \binom{\log |\mathcal{F}_{P,c}| + b + 1}{b+1} - C(2^b - 1) \binom{\log |\mathcal{F}_{P,h}| + b}{b} \\ & \quad - C(2^b - 1) \binom{\log |\mathcal{F}_{P,\S}| + b}{b} \geq C \binom{\log |\mathcal{F}_P| + b}{b} \end{aligned}$$

where  $c = S[1]$  and  $S' = S[2..]$ . It is sufficient since we spend constant time for  $b = 0$  and  $\mathcal{O}(\log |\mathcal{F}_P|)$  time for  $b \geq 1$ .

The claimed bound on the overall running time follows.  $\blacktriangleleft$

## 6 Main Result

Let  $\mathcal{F}$  be a family of suffixes and reverse prefixes of  $X$  and  $Y$  occurring in  $\text{Pairs}_\ell(X)$  or  $\text{Pairs}_\ell(Y)$ , and let us fix a  $k$ -complete family  $N(F) : F \in \mathcal{F}$ . For a half integer  $k'$ ,  $0 \leq k' \leq k$ , and a string  $S \in \{X, Y\}$  let us define

$$\text{Pairs}_\ell^{(k,k')}(S) = \bigcup_{(U_1, U_2) \in \text{Pairs}_\ell(S)} \{(U'_1, U'_2) : U'_i \in N_{d_i, d'_i}(U_i), k = d_1 + d_2, k' = d'_1 + d'_2\}.$$

Intuitively, we extend  $(U_1, U_2) \in \text{Pairs}_\ell(S)$ , arbitrarily splitting the budgets  $k$  and  $k'$  between  $U_1$  and  $U_2$ . To bound the size of  $\text{Pairs}_\ell^{(k,k')}(S)$ , we observe that for  $d_1 + d_2 = k$  and  $k = \mathcal{O}(\log n)$

$$|N_{d_1}(U_1)| \cdot |N_{d_2}(U_2)| \leq 2^k \binom{\log |\mathcal{F}| + d_1}{d_1} \binom{\log |\mathcal{F}| + d_2}{d_2} = \frac{2^{\mathcal{O}(k)} \log^k |\mathcal{F}|}{k^k}.$$

Hence,  $|\text{Pairs}_\ell^{(k,k')}(S)| = \frac{2^{\mathcal{O}(k)} |\mathcal{F}| \log^k |\mathcal{F}|}{k^k \sqrt{\ell}}$ . Combining Lemmas 4 and 12, we obtain the following.

► **Corollary 16.** *If  $\text{LCF}_k(X, Y) \geq \ell$ , then*

$$\text{LCF}_k(X, Y) = \max_{k_1 + k_2 = k} \max \text{PairLCP}(\text{Pairs}_\ell^{(k, k_1)}(X), \text{Pairs}_\ell^{(k, k_2)}(Y)).$$

**Proof.** By Lemma 4, there exist  $(U_1, U_2) \in \text{Pairs}_\ell(X)$ ,  $(V_1, V_2) \in \text{Pairs}_\ell(Y)$ , and  $p + q = k$  such that  $\text{LCF}_k(X, Y) = \text{LCP}_p(U_1, V_1) + \text{LCP}_q(U_2, V_2)$ . Lemma 12 further yields the existence of half integers  $p'_1 + p'_2 \leq p$  and  $q'_1 + q'_2 \leq q$  such that  $\text{LCF}_k(X, Y) = \text{LCP}(U'_1, V'_1) + \text{LCP}(U'_2, V'_2)$  for some  $U'_1 \in N_{p, p'_1}(U_1)$ ,  $V'_1 \in N_{p, p'_1}(V_1)$ ,  $U'_2 \in N_{q, q'_1}(U_2)$ , and  $V'_2 \in N_{q, q'_1}(V_2)$ .

We set  $k'_1 = p'_1 + q'_1$  and  $k'_2 = k - k'_1 \geq p'_2 + q'_2$  so that  $(U'_1, U'_2) \in \text{Pairs}_\ell^{(k, k'_1)}(X)$  and  $(V'_1, V'_2) \in \text{Pairs}_\ell^{(k, k'_2)}(Y) \subseteq \text{Pairs}_\ell^{(k, k'_2)}(Y)$ . Consequently,

$$\text{LCF}_k(X, Y) \leq \max \text{PairLCP}(\text{Pairs}_\ell^{(k, k'_1)}(X), \text{Pairs}_\ell^{(k, k'_2)}(Y)),$$

which concludes the proof of the upper bound on  $\text{LCF}_k(X, Y)$ .

For the lower bound, we shall prove that  $\text{LCF}_k(X, Y) \geq \text{LCP}(U'_1, V'_1) + \text{LCP}(U'_2, V'_2)$  for all  $(U'_1, U'_2) \in \text{Pairs}_\ell^{(k, k'_1)}(X)$  and  $(V'_1, V'_2) \in \text{Pairs}_\ell^{(k, k'_2)}(Y)$  such that  $k'_1 + k'_2 \leq k$ . By definition of  $\text{Pairs}_\ell^{(k, k'_1)}$ , there exist  $(U_1, U_2) \in \text{Pairs}_\ell(X)$  such that  $U'_1 \in N_{k, p'_1}(U_1)$  and  $U'_2 \in N_{k, q'_1}(U_2)$  for half integers  $p'_1 + q'_1 \leq k'_1$ . Similarly, there exist  $(V_1, V_2) \in \text{Pairs}_\ell(Y)$  such that  $V'_1 \in N_{k, p'_2}(V_1)$  and  $V'_2 \in N_{k, q'_2}(V_2)$  for half integers  $p'_2 + q'_2 \leq k'_2$ . We set  $p = \lfloor p'_1 + p'_2 \rfloor$  and  $q = \lfloor q'_1 + q'_2 \rfloor$ , and observe that  $\text{LCP}_p(U_1, V_1) \geq \text{LCP}(U'_1, V'_1)$  as well as  $\text{LCP}_q(U_2, V_2) \geq \text{LCP}(U'_2, V'_2)$  due to Lemma 12. Now, Lemma 4 yields  $\text{LCF}_k(X, Y) \geq \text{LCP}_p(U_1, V_1) + \text{LCP}_q(U_2, V_2) \geq \text{LCP}(U'_1, V'_1) + \text{LCP}(U'_2, V'_2)$ , as desired.  $\blacktriangleleft$

► **Theorem 17.** *For  $k = \mathcal{O}(\log n)$ , the  $\text{LCF}_k(X, Y, \ell)$  problem can be solved in time  $\mathcal{O}(n + \frac{2^{\mathcal{O}(k)} n \log^{k+1} n}{k^k \sqrt{\ell}})$ . For  $k = \mathcal{O}(1)$ , this running time becomes  $\mathcal{O}(n + \frac{n \log^{k+1} n}{\sqrt{\ell}})$ .*

**Proof.** First, we build the joint suffix tree of  $X$ ,  $X^R$ ,  $Y$ , and  $Y^R$ , as well as the family  $\mathcal{F}$ . A component for the LCA queries on the suffix tree lets us compare any suffixes of  $F \in \mathcal{F}$  in constant time [5]. This allows us to build the  $k$ -complete family  $N(F) : F \in \mathcal{F}$ , represented



as a compacted trie of  $\mathcal{F}' := \bigcup\{N(F) : F \in \mathcal{F}\}$  using Proposition 13. Next, we construct the sets  $\text{Pairs}_\ell^{(k,k')}(X) \subseteq (\mathcal{F}')^2$  and  $\text{Pairs}_\ell^{(k,k')}(Y) \subseteq (\mathcal{F}')^2$  for  $k' = 0, \frac{1}{2}, \dots, k - \frac{1}{2}, k$ , and solve the  $2k + 1$  instances of TWO STRING FAMILIES LCP PROBLEM, as specified in Corollary 16.

We conclude with running-time analysis. Preprocessing takes  $\mathcal{O}(n)$  time, and the procedure of Proposition 13 runs in  $\mathcal{O}(2^k |\mathcal{F}'|^{\frac{\log |\mathcal{F}'| + k + 1}{k+1}}) = \frac{2^{\mathcal{O}(k)} n \log^{k+1} n}{k^k \sqrt{\ell}}$  time. We have  $\text{Pairs}_\ell^{k,k'}(X) = \frac{2^{\mathcal{O}(k)} n \log^k n}{k^k \sqrt{\ell}}$ , so solving all instances of TWO STRING FAMILIES LCP PROBLEM also takes  $\frac{2^{\mathcal{O}(k)} n \log^{k+1} n}{k^k \sqrt{\ell}}$  time (Lemma 3). The overall running time is therefore as claimed.  $\blacktriangleleft$

In particular, for  $k = \mathcal{O}(\log n)$ , there exists  $\ell_0 = \frac{2^{\mathcal{O}(k)} \log^{2k+2} n}{k^{2k}}$  such that the  $\text{LCF}_k(X, Y, \ell)$  problem can be solved in  $\mathcal{O}(n)$  time for  $\ell \geq \ell_0$ . For  $k = \mathcal{O}(1)$ , we have  $\ell_0 = \mathcal{O}(\log^{2k+2} n)$ , while for  $k = o(\log n)$ , we have  $\ell_0 = n^{o(1)}$ . We arrive at the main result.

► **Corollary 18.** *The  $\text{LCF}_k(X, Y, \ell)$  problem with  $\ell = \Omega(\log^{2k+2} n)$  can be solved in  $\mathcal{O}(n)$  time.*

---

## References

- 1 Amir Abboud, Richard Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.
- 2 Hayam Alamro, Lorraine A. K. Ayad, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with  $k$ -mismatches and applications. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiri Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February 2, 2018, Proceedings*, volume 10706 of *Lecture Notes in Computer Science*, pages 636–649. Springer, 2018. doi:10.1007/978-3-319-73117-9\_45.
- 3 Lorraine A. K. Ayad, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with  $k$ -errors and applications. *CoRR*, abs/1801.04425, 2018. arXiv:1801.04425.
- 4 Maxim A. Babenko and Tatiana A. Starikovskaya. Computing the longest common substring with one mismatch. *Probl. Inf. Transm.*, 47(1):28–33, 2011. doi:10.1134/S0032946011010030.
- 5 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839\_9.
- 6 Gerth Stølting Brodal. Finger search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.ch11.
- 7 Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979. doi:10.1145/322123.322127.
- 8 Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. 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 55–69. Springer, 2003. doi:10.1007/3-540-44888-8\_5.



- 9 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.
- 10 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.
- 11 Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, and Marie-France Sagot. Longest repeats with a block of  $k$  don't cares. *Theor. Comput. Sci.*, 362(1-3):248–254, 2006. doi:10.1016/j.tcs.2006.06.029.
- 12 Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen. Longest common substrings with  $k$  mismatches. *Inf. Process. Lett.*, 115(6-8):643–647, 2015. doi:10.1016/j.ipl.2015.03.006.
- 13 Szymon Grabowski. A note on the longest common substring with  $k$ -mismatches problem. *Inf. Process. Lett.*, 115(6-8):640–642, 2015. doi:10.1016/j.ipl.2015.03.003.
- 14 Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 49–60. ACM, 1977. doi:10.1145/800105.803395.
- 15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- 16 Kurt Hoffman, Kurt Mehlhorn, Pierre Rosenstiehl, and Robert Endre Tarjan. Sorting jordan sequences in linear time using level-linked search trees. *Information and Control*, 68(1-3):170–184, 1986. doi:10.1016/S0019-9958(86)80033-X.
- 17 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6\_19.
- 18 Russell Impagliazzo and Ramamohan Paturi. On the complexity of  $k$ -sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 19 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.
- 20 Tomasz Kociumaka, Jakub Radoszewski, and Tatiana A. Starikovskaya. Longest common substring with approximately  $k$  mismatches. *CoRR*, abs/1712.08573, 2017. arXiv:1712.08573.
- 21 Tomasz Kociumaka, Tatiana A. Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2014. doi:10.1007/978-3-662-44777-2\_50.
- 22 Chris-André 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.
- 23 Mamoru Maekawa. A square root  $N$  algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.
- 24 Tatiana A. Starikovskaya. Longest common substring with approximately  $k$  mismatches. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPICs*,

- pages 21:1–21:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.21.
- 25 Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 223–234. Springer, 2013. doi:10.1007/978-3-642-38905-4\_22.
- 26 Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979. doi:10.1145/322154.322161.
- 27 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.
- 28 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.
- 29 Igor Ulitsky, David Burstein, Tamir Tuller, and Benny Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006. doi:10.1089/cmb.2006.13.336.

# Lyndon Factorization of Grammar Compressed Texts Revisited

**Isamu Furuya**


Graduate School of IST, Hokkaido University, Japan  
furuya@ist.hokudai.ac.jp

**Yuto Nakashima**

Department of Informatics, Kyushu University, Japan  
yuto.nakashima@inf.kyushu-u.ac.jp

**Tomohiro I**

Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan  
tomohiro@ai.kyutech.ac.jp


 <https://orcid.org/0000-0001-9106-6192>

**Shunsuke Inenaga**

Department of Informatics, Kyushu University, Japan  
inenaga@inf.kyushu-u.ac.jp

**Hideo Bannai**

Department of Informatics, Kyushu University, Japan  
RIKEN Center for Advanced Intelligence Project, Japan  
bannai@inf.kyushu-u.ac.jp

 <https://orcid.org/0000-0002-6856-5185>

**Masayuki Takeda**

Department of Informatics, Kyushu University, Japan  
takeda@inf.kyushu-u.ac.jp

---

## Abstract

We revisit the problem of computing the Lyndon factorization of a string  $w$  of length  $N$  which is given as a straight line program (SLP) of size  $n$ . For this problem, we show a new algorithm which runs in  $O(P(n, N) + Q(n, N)n \log \log N)$  time and  $O(n \log N + S(n, N))$  space where  $P(n, N)$ ,  $S(n, N)$ ,  $Q(n, N)$  are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. Our algorithm improves the algorithm proposed by I et al. (TCS '17), and can be more efficient than the  $O(N)$ -time solution by Duval (J. Algorithms '83) when  $w$  is highly compressible.

**2012 ACM Subject Classification** Mathematics of computing → Combinatorial algorithms

**Keywords and phrases** Lyndon word, Lyndon factorization, Straight line program

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2018.24

**Funding** This work was supported by JSPS KAKENHI Grant Numbers JP17H06923 (YN), JP16K16009 (TI), JP17H01697 (SI), JP16H02783 (HB), and JP25240003 (MT).

## 1 Introduction

A string  $w$  is said to be a Lyndon word if  $w$  is lexicographically smaller than any of its proper suffixes. For instance, **abb** is a Lyndon word, but **bba** and **aba** are not. The Lyndon factorization of a string  $w$  is the sequence of strings  $\ell_1^{p_1}, \dots, \ell_m^{p_m}$  such that  $w = \ell_1^{p_1} \dots \ell_m^{p_m}$ ,  $\ell_i$



© Isamu Furuya, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda;

licensed under Creative Commons License CC-BY

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).

Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 24; pp. 24:1–24:10



Leibniz International Proceedings in Informatics

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

is a Lyndon word,  $p_i \geq 1$  ( $1 \leq i \leq m$ ), and  $\ell_i \succ \ell_{i+1}$  ( $1 \leq i < m$ ) [4]. Lyndon factorizations are used, for example, in a bijective variant of Burrows-Wheeler transform [13, 7] and an algorithm to check digital convexity [2].

Let  $LF_w$  denote the Lyndon factorization of a string  $w$ . Given a string  $w$  of length  $N$ ,  $LF_w$  can be computed on-line in  $O(N)$  time [6]. When the length  $N$  of the string  $w$  is huge, even the  $O(N)$ -time solution may not be efficient enough. I et al. [10] showed an efficient Lyndon factorization algorithm when the string  $w$  is given as a *straight line program* (SLP), which is a compressed representation of the string based on a context free grammar that derives only  $w$ . The algorithm runs in  $O(n^2 + P(n, N) + Q(n, N)n \log n)$  time and  $O(n^2 + S(n, N))$  space where  $P(n, N)$ ,  $S(n, N)$ ,  $Q(n, N)$  are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. This algorithm can be more efficient than the  $O(N)$ -time solution when  $w$  is highly compressible.

In this paper, we revisit the Lyndon factorization problem on SLPs and give a more efficient solution. Given an SLP  $\mathcal{S}$  of size  $n$  representing a string  $w$  of length  $N$ , our new algorithm runs in  $O(P(n, N) + Q(n, N)n \log \log N)$  time and  $O(n \log N + S(n, N))$  space. If we use the LCE data structure of [8], we can compute the Lyndon factorization in  $O(n \log N \log \log N)$  time and  $O(n \log N)$  space. This improves the previous algorithm since  $\log N \leq n$  holds.

We note that the previous algorithm [10] computes the Lyndon factorization in a bottom-up manner, which requires us to store the Lyndon factorization for *every* variable of a given SLP. This implies that we must use  $\Omega(n^2)$  space (and thus time) in total because the size of each Lyndon factorization can be  $\Omega(n)$ . We show that the Lyndon factorization of  $w$  can be computed without computing the Lyndon factorization of each variable.

## 2 Preliminaries

### 2.1 Strings and model of computation

Let  $\Sigma$  be an ordered finite *alphabet*. An element of  $\Sigma^*$  is called a *string*. The length of a string  $w$  is denoted by  $|w|$ . The empty string  $\varepsilon$  is a string of length 0. Let  $\Sigma^+$  be the set of non-empty strings, i.e.,  $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ . For a string  $w = xyz$ ,  $x$ ,  $y$  and  $z$  are called a *prefix*, *substring*, and *suffix* of  $w$ , respectively. A prefix  $x$  of  $w$  is called a *proper prefix* of  $w$  if  $x \neq w$ . The  $i$ -th character of a string  $w$  is denoted by  $w[i]$ , where  $1 \leq i \leq |w|$ . For a string  $w$  and two integers  $1 \leq i \leq j \leq |w|$ , let  $w[i..j]$  denote the substring of  $w$  that begins at position  $i$  and ends at position  $j$ . For convenience, let  $w[i..j] = \varepsilon$  when  $i > j$ . For any string  $w$  let  $w^1 = w$ , and for any integer  $k \geq 2$  let  $w^k = ww^{k-1}$ , i.e.,  $w^k$  is a  $k$ -time repetition of  $w$ .

If character  $a$  is lexicographically smaller than another character  $b$ , then we write  $a \prec b$ . For any strings  $x, y$ , let  $lcp(x, y)$  be the length of the longest common prefix of  $x$  and  $y$ . We write  $x \prec y$  iff either  $x[lcp(x, y) + 1] \prec y[lcp(x, y) + 1]$  or  $x$  is a proper prefix of  $y$ .

Our model of computation is the word RAM. We assume the computer word size is at least  $\lceil \log_2 |w| \rceil$ , and hence, standard operations on values representing lengths and positions of string  $w$  can be manipulated in  $O(1)$  time. Space complexities will be determined by the number of computer words (not bits).

### 2.2 Lyndon words and Lyndon factorization of strings

Two strings  $x$  and  $y$  are *conjugates*, if  $x = uv$  and  $y = vu$  for some strings  $u$  and  $v$ . A string  $w$  is said to be a *Lyndon word*, if  $w$  is lexicographically strictly smaller than all of its conjugates. Namely,  $w$  is a Lyndon word, if for any factorization  $w = uv$ , it holds that

$uv \prec vu$ . An equivalent definition of Lyndon words is: a string  $w$  is a *Lyndon word*, if  $w \prec v$  for any non-empty proper suffix  $v$  of  $w$ .

The *Lyndon factorization* of a string  $w$ , denoted  $LF_w$ , is the factorization  $\ell_1^{p_1}, \dots, \ell_m^{p_m}$  of  $w$ , such that each  $\ell_i \in \Sigma^+$  is a Lyndon word,  $p_i \geq 1$ , and  $\ell_i \succ \ell_{i+1}$  for all  $1 \leq i < m$ . The size of  $LF_w$  is  $m$  and denoted by  $|LF_w|$ .  $LF_w$  can be represented by the sequence  $(|\ell_1|, p_1), \dots, (|\ell_m|, p_m)$  of integer pairs, where each pair  $(|\ell_i|, p_i)$  represents the  $i$ -th Lyndon factor  $\ell_i^{p_i}$  of  $w$ . Note that this representation requires  $O(m)$  space.

In the literature, the Lyndon factorization is sometimes defined to be a sequence of lexicographically non-increasing Lyndon words, namely, each Lyndon factor  $\ell^p$  is decomposed into a sequence of  $p$   $\ell$ 's. In this paper, each Lyndon word  $\ell$  in the Lyndon factor  $\ell^p$  is called a *decomposed Lyndon factor*.

For any string  $w$ , let  $LF_w = \ell_1^{p_1}, \dots, \ell_m^{p_m}$ . Let  $lfb_w(i)$  denote the position where the  $i$ -th Lyndon factor begins in  $w$ , i.e.,  $lfb_w(1) = 1$  and  $lfb_w(i) = lfb_w(i-1) + |\ell_{i-1}^{p_{i-1}}|$  for any  $2 \leq i \leq m$ . For any  $1 \leq i \leq m$ , let  $lfs_w(i) = \ell_i^{p_i} \ell_{i+1}^{p_{i+1}} \dots \ell_m^{p_m}$  and  $lfp_w(i) = \ell_1^{p_1} \ell_2^{p_2} \dots \ell_i^{p_i}$ . For convenience, let  $lfs_w(m+1) = lfp_w(0) = \varepsilon$ .

► **Example 1.** Let  $w = \text{abcabcabababcbabababcbababab}$ . Then,

- $LF_w = (\text{abc})^2, \text{abababcb}, \text{abababc}, (\text{ab})^2, \text{a}$ ;
- the decomposed Lyndon factorization of  $w$  is  $\text{abc}, \text{abc}, \text{abababcb}, \text{abababc}, \text{ab}, \text{ab}, \text{a}$ .

Moreover,  $lfb_w(2) = 7$ ,  $lfs_w(3) = \text{abababcbababab}$ , and  $lfp_w(2) = \text{abcabcabababcb}$ .

The following is a useful lemma concerning Lyndon factorizations.

► **Lemma 2** (Lemma 4 of [11]). *Let  $LF_w = \ell_1^{p_1}, \dots, \ell_m^{p_m}$  and  $1 \leq i, j \leq m$ . Assume that  $\ell_i^{p_i} \dots \ell_j^{p_j}$  has an occurrence to the left in  $w$ . Then,*

1. *the leftmost occurrence of  $\ell_i^{p_i} \dots \ell_j^{p_j}$  is a prefix of  $\ell_k$  for some  $k < i$ ;*
2.  *$\ell_i^{p_i} \dots \ell_j^{p_j}$  is a prefix of every  $\ell_h$  with  $k \leq h < i$ .*

### 2.3 Straight line programs (SLPs)

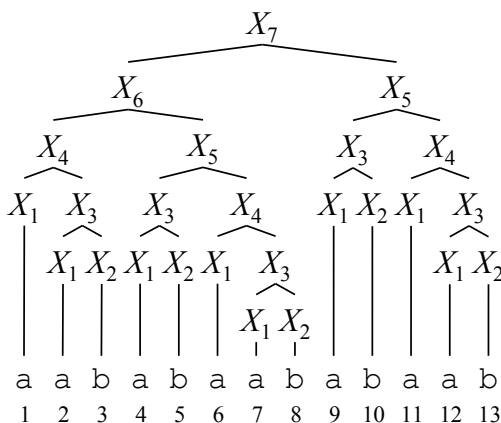
A *straight line program (SLP)* is a set of productions  $\mathcal{S} = \{X_i \rightarrow \text{expr}_i\}_{i=1}^n$ , where each  $X_i$  is a variable and each  $\text{expr}_i$  is an expression of the form  $\text{expr}_i = a$  ( $a \in \Sigma$ ), or  $\text{expr}_i = X_l X_r$  ( $i > l, r$ ). Let  $\text{val}(X_i)$  denote the string derived by  $X_i$ . Also let  $\text{val}(a) = a$  for  $a \in \Sigma$ . We will sometimes associate  $\text{val}(X_i)$  with  $X_i$  and denote  $|\text{val}(X_i)|$  as  $|X_i|$ . An SLP  $\mathcal{S}$  *represents* the string  $w = \text{val}(X_n)$ . The *size* of the program  $\mathcal{S}$  is the number  $n$  of productions in  $\mathcal{S}$ . If  $N$  is the length of the string represented by SLP  $\mathcal{S}$ , then  $N$  can be as large as  $2^{n-1}$ .

The derivation tree  $T_{\mathcal{S}}$  of SLP  $\mathcal{S}$  is a labeled ordered tree obtained by recursively applying the productions of variable  $X_i$ , starting from  $X_n$ , i.e., the root node has label  $X_n$ , and for each internal node labeled  $X_i$ , if  $X_i \rightarrow X_l X_r$ , then its left child is labeled  $X_l$  and its right child is labeled  $X_r$ , if  $X_i \rightarrow a$ , then its single child is labeled  $a$ .

The height of SLP  $\mathcal{S}$  is the height of  $T_{\mathcal{S}}$ . We associate to each leaf of  $T_{\mathcal{S}}$  the corresponding position in string  $w = \text{val}(X_n)$ . An example of the derivation tree of an SLP is shown in Figure 1.

It is known that the size of the Lyndon factorization of  $w$  is a lower bound of the size of smallest SLP which derives  $w$ .

► **Lemma 3** (Lemma 17 of [10]). *For any string  $w$ , let  $m = |LF_w|$ , and  $n$  be the size of an SLP which derives  $w$ . Then  $m \leq n$  holds.*



■ **Figure 1** The derivation tree of SLP  $\mathcal{S} = \{X_1 \rightarrow a, X_2 \rightarrow b, X_3 \rightarrow X_1X_2, X_4 \rightarrow X_1X_3, X_5 \rightarrow X_3X_4, X_6 \rightarrow X_4X_5, X_7 \rightarrow X_6X_5\}$ , representing string  $w = \text{val}(X_7) = \text{aabaabaabaab}$ .

### 2.4 Longest common extension problem on SLPs

The longest common extension (LCE) problem on SLPs is to preprocess an SLP so that we can efficiently answer LCE queries that ask to compute  $\text{lcp}(\text{val}(X_i)[k_1..|X_i|], \text{val}(X_i)[k_2..|X_i|])$  for any variable  $X_i$  and  $1 \leq k_1, k_2 \leq |X_i|$ . Currently, the best known deterministic solution to this problem is the following.

► **Lemma 4** (Theorem 2 of [8]). *Given an SLP of size  $n$  representing a string of length  $N$ , we can preprocess in  $O(n \log(N/n))$  time and  $O(n + t \log(N/t))$  space to support LCE queries in  $O(\log N)$  time where  $t$  is the size of non-overlapping LZ77 factorization.*

In order to describe the complexity of our algorithm independent from the choice of LCE data structures, the preprocessing time, space and query time of the chosen LCE data structure are denoted by  $P(n, N)$ ,  $S(n, N)$  and  $Q(n, N)$ , respectively.

## 3 Lyndon factorization algorithm for SLP

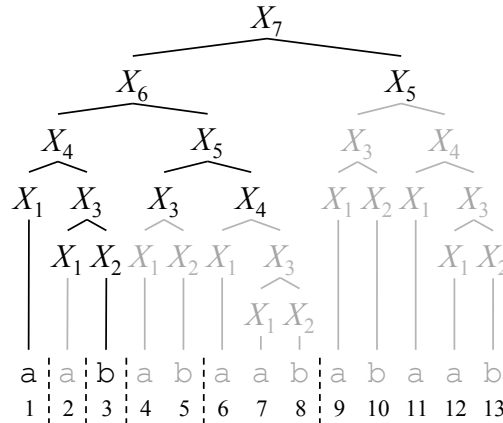
In this paper, we propose a new Lyndon factorization algorithm for an SLP compressed text. More formally, we are given an SLP  $\mathcal{S}$  which derives a string  $w$ , and we compute  $LF_w$ .

Firstly, we explain the idea of our algorithm. We compute the Lyndon factorization of  $w$  from left to right based on *G-factorization* defined as follows. The G-factorization of a string  $w$  that is derived by an SLP  $\mathcal{S}$  is defined by the *Partial Parse Tree* of  $\mathcal{S}$ .

► **Definition 5** (Partial Parse Tree [14]). The partial parse tree of an SLP  $\mathcal{S}$  is a subtree of the derivation tree of  $\mathcal{S}$  such that each variable occurs exactly once as a label of an internal node and the occurrence is the leftmost possible.

► **Definition 6** (G-factorization [14]). The G-factorization of a string  $w$  that is derived by an SLP  $\mathcal{S}$  is  $GF_{(w, \mathcal{S})} = \text{val}(\text{leaf}_1), \dots, \text{val}(\text{leaf}_g)$ , where  $\text{leaf}_1, \text{leaf}_2, \dots, \text{leaf}_g$  is the sequence of leaf labels of the partial parse tree of an SLP  $\mathcal{S}$ .

Figure 2 shows the partial parse tree and the G-factorization of SLP  $\mathcal{S}$  which was shown in Figure 1. Since the number of internal nodes of the partial parse tree of  $\mathcal{S}$  is exactly the same as the number of variables of  $\mathcal{S}$ , it is clear that  $|GF_{(w, \mathcal{S})}|$  as well as the size of the partial parse tree is  $O(n)$ , where  $n$  is the size of SLP  $\mathcal{S}$ .



■ **Figure 2** The partial parse tree of SLP  $\mathcal{S}$  which was shown in Figure 1. The G-factorization of this string is shown by dash lines on the string.

Let  $GF_{(w,S)} = val(leaf_1), \dots, val(leaf_g)$  and  $w_j = val(leaf_1) \cdots val(leaf_j)$  for any  $1 \leq j \leq g$ . Our algorithm consists of the following two parts.

1. Compute the set of significant suffixes of  $X_i$  for all  $1 \leq i \leq n$ .
2. Compute the Lyndon factorization and the significant suffixes of  $w_{j+1}$ .

Here, *significant suffixes* of a string are suffixes of the string, as defined in [9, 10] (also used in [12]), closely related to Lyndon factorizations, and will be explained in detail in Section 3.1.

The second part of our algorithm consists of  $g$  steps: In the  $(j+1)$ -th step of our algorithm, we compute  $LF_{w_{j+1}}$  by using information computed for  $w_j$  and  $leaf_i$ . More precisely, we compute  $LF_{w_{j+1}}$  by using:

- the Lyndon factorization of  $w_j$ ,
- the significant suffixes of  $w_j$ , and
- the significant suffixes of  $leaf_i$ .

The rest of this section is organized as follows. In Section 3.1, we explain what significant suffixes are. We also show properties on significant suffixes which are used in our algorithm. In Section 3.2 and 3.3, we describe respectively, the first part and the second part of our algorithm.

### 3.1 Significant suffix

Assume that  $LF_u = u_1^{p_1}, \dots, u_m^{p_m}$ .  $lfs_u(i)$  is said to be a significant suffix of  $u$  if  $lfs_u(i+1)$  is a prefix of  $lfs_u(i)$  for any  $1 \leq i \leq m$ . It is clear that  $lfs_u(m)$  (i.e., the last Lyndon factor) is always a significant suffix of  $u$ .

► **Lemma 7.** Assume that  $lfs_u(i+1)$  is a prefix of  $lfs_u(i)$  for some  $1 \leq i \leq m$ . Then  $lfs_u(j+1)$  is a prefix of  $lfs_u(j)$  for any  $i < j \leq m$ .

**Proof.** Assume that  $lfs_u(i+1)$  is a prefix of  $lfs_u(i)$  for some  $1 \leq i \leq m$ . Let  $i < j \leq m$ . By the definition,  $lfs_u(j)$  and  $lfs_u(j+1)$  are suffixes of  $lfs_u(i+1)$ . Thus,  $lfs_u(j)$  and  $lfs_u(j+1)$  are proper substrings of  $lfs_u(i)$ . By Lemma 2,  $lfs_u(j)$  and  $lfs_u(j+1)$  have to be proper prefixes of  $lfs_u(i)$ . Since  $|lfs_u(j)| > |lfs_u(j+1)|$ , then  $lfs_u(j+1)$  is a prefix of  $lfs_u(j)$ . ◀

Let  $\lambda_u$  be the minimum integer such that  $lfs_u(i+1)$  is a prefix of  $u_i$  for any  $\lambda_u \leq i \leq m$ . We define the set of significant suffixes  $\Lambda_u$  of  $u$  as  $\Lambda_u = \{lfs_u(i) \mid \lambda_u \leq i \leq m\}$ .



► **Example 8.** Let  $w = \text{abcabcabababcbabababcbababa}$  (same as Example 1). Then,  $\lambda_w = 3$  and  $\Lambda_w = \{\text{abababcbababa}, \text{ababa}, \mathbf{a}\}$  since  $lfs_w(3)$  is not a prefix of  $lfs_w(2)$ , but  $lfs_w(4)$  is a prefix of  $lfs_w(3)$ .

It is clear from the definition of Lyndon factorization and  $\lambda_u$  that for any  $\lambda_u \leq i \leq m$ ,  $u_i = lfs_u(i+1)y_i$  for some non-empty string  $y_i$ . We will represent  $\Lambda_u$  by the sequence  $(lfb_u(\lambda_u), p_{\lambda_u}), \dots, (lfb_u(m), p_m)$  of integer pairs. Note that this representation requires  $O(\log |u|)$  space by the following lemma.

► **Lemma 9** (Lemma 12 of [10]). *For any string  $u$ ,  $|\Lambda_u| = O(\log |u|)$ .*

### 3.2 Computing significant suffixes

For any strings  $u, v$ , let  $LF_u = u_1^{p_1}, \dots, u_m^{p_m} = U_1, \dots, U_m$  where  $U_i = u_i^{p_i}$  and  $LF_v = v_1^{q_1}, \dots, v_{m'}^{q_{m'}} = V_1, \dots, V_{m'}$  where  $V_i = v_i^{q_i}$ . Our idea of computing significant suffixes is based on the following lemma used in [10].

► **Lemma 10** ([1, 5]).  *$LF_{uv} = U_1, \dots, U_c, z^k, V_{c'}, \dots, V_{m'}$  for some  $0 \leq c \leq m$ ,  $1 \leq c' \leq m' + 1$  and  $LF_{lfs_u(c+1)lfp_v(c'-1)} = z^k$ .*

This lemma says that  $LF_{uv}$  can be obtained from  $LF_u$  and  $LF_v$  by computing the medial Lyndon factor  $z^k$  since the other Lyndon factors remain unchanged in  $uv$ .

Let  $X_i = X_\ell X_r$  ( $1 \leq \ell, r < i \leq n$ ). Assume that we have computed  $\Lambda_{X_\ell}$  and  $\Lambda_{X_r}$ . Then we compute  $\Lambda_{X_i}$  from this information. The following lemmas are useful for our algorithm.

► **Lemma 11** (Lemma 16 of [10]).  $\lambda_u \leq c + 1$ .

► **Lemma 12.**  $lfb_{uv}(\lambda_{uv}) \in \{lfb_u(i) \mid \lambda_u \leq i \leq c + 1\} \cup \{|u| + lfb_v(\max\{c', \lambda_v\})\}$ .

**Proof.** By Lemma 10,  $V_j$  is a Lyndon factor of  $uv$  for any  $c' \leq j \leq |LF_v|$ . Hence,  $lfs_v(k)$  is a significant suffix of  $uv$  for any  $\max\{c', \lambda_v\} \leq k \leq m$ . Let  $1 \leq j < \lambda_u$ . By Lemma 11,  $U_j$  is a Lyndon factor of  $uv$ . By the definition of significant suffix and Lemma 7,  $lfs_u(j+1)$  is not a prefix of  $lfs_u(j)$ . From this fact, it is easy to see that  $lfs_u(j+1)v$  is not a prefix of  $lfs_u(j)v$ , i.e.,  $lfs_{uv}(j+1)$  is not a prefix of  $lfs_{uv}(j)$ . Thus,  $lfb_{uv}(\lambda_{uv}) \geq lfb_u(\lambda_u)$ . Therefore, this lemma holds. ◀

From Lemma 10 and the definition of Lyndon factorization, there exists exact one  $z$  which begins in  $u$  and ends in  $v$  (if  $u_m \prec v_1$ ). We refer to this  $z$  as *crossing factor*. By the next lemma, we can determine the lexicographic order between  $u_m$  and  $v_1$  using a single LCE query  $lcp(u_m v, v)$ . We remark that we do not have to know  $|v_1|$  as well as the Lyndon factorization of  $v$ .

► **Lemma 13.** *Let  $\alpha = lcp(u_m v, v)$ . Then,*

1.  $u_m \succ v_1$  if  $\alpha < |u_m|$  and  $u_m v \succ v$ ;
2.  $u_m = v_1$  if  $\alpha \geq |u_m|$  and  $u_m v \succ v$ ;
3.  $u_m \prec v_1$  if  $u_m v \prec v$ .

**Proof.**

1. If  $\alpha < |u_m|$  and  $u_m v \succ v$ , then the prefix of  $v$  of length  $\alpha$  is not a prefix of any Lyndon words. This implies that the longest prefix of  $v$  which is a Lyndon word is shorter than  $\alpha$ . Thus,  $u_m \succ v_1$  holds.
2. If  $\alpha \geq |u_m|$  and  $u_m v \succ v$ , then the prefix of  $v$  of length  $\alpha + 1$  can be represented as  $u_m^i u_m' c$  such that  $i \geq 1$ ,  $u_m'$  is a prefix of  $u_m$ , and  $u_m[|u_m'| + 1] \succ c \in \Sigma$ . This implies that  $u_m$  is the longest prefix of  $v$  which is a Lyndon word. Thus,  $u_m = v_1$  holds.



3. If  $\alpha \geq |u_m|$  and  $u_m v \prec v$ , then the prefix of  $v$  of length  $\alpha + 1$  can be represented as  $u_m^i u_m' c$  such that  $i \geq 1$ ,  $u_m'$  is a prefix of  $u_m$ , and  $u_m[|u_m'| + 1] \prec c \in \Sigma$ . This implies that the prefix of  $v$  of length  $\alpha + 1$  is a Lyndon word which has  $u_m$  as a prefix. Thus,  $u_m \prec v_1$ . If  $\alpha < |u_m|$  and  $u_m v \prec v$ , then the prefix of  $v$  of length  $\alpha + 1$  is a Lyndon word which is lexicographically larger than  $u_m$ . Thus,  $u_m \prec v_1$ . ◀

Due to the following lemma, we can compute the crossing factor efficiently.

► **Lemma 14** (Lemma 16 of [10]). *Assume that  $u_m \prec v_1$ . Let  $i, j$  be the beginning position and the ending position of the crossing factor  $z$ , respectively. Then*

- $i = \text{lf}_u(i')$  for some  $\lambda_u \leq i' \leq m$ ,
- $\text{ls}_u(1)v \succ \dots \succ \text{ls}_u(i')v \prec \dots \prec \text{ls}_u(m+1)v$ ,
- $j = \text{lf}_v(j')$  such that  $\text{ls}_v(j'-1) \succ \text{ls}_u(i')v \succ \text{ls}_v(j')$ .

The following lemma is the main result of this section.

► **Lemma 15.** *We can compute all significant suffixes for each variable of an SLP  $\mathcal{S}$  by  $O(n \log \log N)$  lexicographical string comparisons.*

**Proof.** Let  $i \leq n$ . Assume that we have computed all significant suffixes of variable  $X_j$  for any  $j < i$ . We show how to compute all significant suffixes of variable  $X_i = X_\ell X_r$  ( $\ell, r < i$ ). Let  $LF_{X_\ell} = U_1, \dots, U_m$  and  $LF_{X_r} = V_1, \dots, V_{m'}$ . Firstly, we compute the lexicographic order between  $u_m$  and  $v_1$  by Lemma 13. This can be done by one LCE query.

Suppose that  $u_m \succ v_1$ . Then  $LF_{X_i} = U_1, \dots, U_m, V_1, \dots, V_{m'}$  by the definition of Lyndon factorization. Since we have all significant suffixes of  $X_\ell$  and  $X_r$ , we can compute all significant suffixes of  $X_i$  by  $O(\log \log |\text{val}(X_\ell)| + \log \log |\text{val}(X_r)|)$  lexicographical string comparisons from Lemmas 7 and 12. It is clear that the last decomposed Lyndon factor of  $X_i$  is the same as  $X_r$ .

Suppose that  $u_m = v_1$ . Then  $LF_{X_i} = U_1, \dots, U_{m-1}, U_m V_1, V_2, \dots, V_{m'}$  by the definition of Lyndon factorization. We can compute all significant suffixes of  $X_i$  in a similar way.

Suppose that  $u_m \prec v_1$ . We can compute the beginning position of the crossing factor  $z$  by  $O(\log |\Lambda_u|)$  lexicographic string comparisons from Lemma 14. Let  $b = \text{lf}_u(j)$  be this position. Next, we check whether  $b$  is also the beginning position of  $z^k$  or not. We can do this with one LCE query as follows. If  $j = \lambda_u$ , then  $b$  is the beginning position of  $z^k$  (since  $u_{j-1}$  does not have  $\text{ls}_u(j)$  as a prefix). If the length of the longest common prefix between  $u_{j-1}$  and  $\text{ls}_u(j)v$  is  $|u_{j-1}|$ , then  $u_{j-1} = z$  and  $\text{lf}_u(j-1)$  is the beginning position of  $z^k$ . Suppose that  $u_{j-1} = z$ . Then we can compute  $z^k$  by a constant number of lexicographic string comparisons. Thus we can also compute all significant suffixes by Lemma 12 from significant suffixes of  $X_\ell$  and  $X_r$ . Suppose that  $u_{j-1} \neq z$ . Firstly, we check whether the ending position of  $z$  which begins in  $u$  and ends in  $v$  is larger than  $|u| + \text{lf}_v(\lambda_v)$  or not. We can do this by  $O(\log |\Lambda_v|)$  lexicographic string comparisons from Lemma 14. If so, by Lemmas 7 and 12, we can compute all significant suffixes of  $X_i$  by additional  $O(\log |\Lambda_u|)$  lexicographic string comparisons. Otherwise,  $\Lambda_{uv} \subseteq \Lambda_v$  holds.

Therefore, we can compute all significant suffixes of  $X_i$  by  $O(\log \log N)$  lexicographical string comparisons. ◀

### 3.3 Computing Lyndon factorization

Let  $GF_{(w, \mathcal{S})} = \text{val}(\text{leaf}_1), \dots, \text{val}(\text{leaf}_g)$  and  $w_j = \text{val}(\text{leaf}_1) \cdots \text{val}(\text{leaf}_j)$  for any  $1 \leq j \leq g$ . We consider computing the Lyndon factorization and the significant suffixes of  $w_{j+1}$  assuming that we have computed the Lyndon factorization and the significant suffixes of  $w_j$ , and

also computed the significant suffixes for each variable of  $\mathcal{S}$  by Section 3.2. Notice that for  $w_{j+1} = w_j \text{val}(\text{leaf}_{j+1})$ ,  $\text{val}(\text{leaf}_{j+1})$  has already occurred in  $w_j$  at least once if  $\text{leaf}_{j+1}$  is a variable. The following lemma is very useful for our algorithm.

► **Lemma 16** (Lemma 21 of [10]). *Let  $w$  be non-empty string such that  $w = xvyvz$  with  $v \in \Sigma^+$  and  $x, y, z \in \Sigma^*$ . If  $|xvy| < \text{lfb}_w(k) \leq |xvyv|$  for some  $k$ , then  $\text{lfb}_w(k) \in \{|xvy| + \text{lfb}_v(j) \mid \lambda_v \leq j \leq m'\}$ , where  $m' = |\text{LF}_v|$ .*

This lemma implies that the ending position of  $z^k$  is restricted by significant suffixes of  $v$ . Below, we change this lemma for the ending position of the crossing factor rather than the ending position of  $z^k$ .

► **Lemma 17.** *Let  $i - 1$  be the ending position of the crossing factor. Assume that  $v$  is a substring of  $u$ . Then  $i \in \{|u| + \text{lfb}_v(j) \mid \lambda_v \leq j \leq m'\}$ , where  $m' = |\text{LF}_v|$ .*

**Proof.** Assume for a contradiction that  $i < |u| + \text{lfb}_v(\lambda_v)$ . From Lemma 16,  $i = \text{lfb}_v(\lambda_v - 1)$  and  $v_{\lambda_v - 1} = z$  holds. Moreover,  $V_{\lambda_v}, \dots, V_{m'}$  are also Lyndon factors of  $uv$ . Since  $v$  is a substring of  $u$ ,  $V_{\lambda_v} \cdots V_{m'}$  has a left occurrence. By Lemma 2,  $v_{\lambda_v - 1} = z$  has  $V_{\lambda_v} \cdots V_{m'}$  as a prefix. This contradicts that  $\text{lfs}_v(\lambda_v - 1)$  is not a significant suffix of  $v$ . ◀

Thus, we can compute the ending position of the crossing factor by significant suffixes of an added string  $v$  (i.e., we do not need the whole Lyndon factorization of  $v$ ).

► **Lemma 18.** *Given an SLP  $\mathcal{S}$  of size  $n$  representing string  $w$  of length  $N$ , we can compute  $\text{LF}_w$  in  $O(n \log \log N)$  lexicographic string comparisons.*

**Proof.** Let  $GF_{(w, \mathcal{S})} = \text{val}(\text{leaf}_1), \dots, \text{val}(\text{leaf}_g)$  and  $w_j = \text{val}(\text{leaf}_1) \cdots \text{val}(\text{leaf}_j)$  for any  $1 \leq j \leq g$ .

Firstly we compute significant suffixes for all variables in  $\mathcal{S}$ . By Lemma 15, it can be done in  $O(n \log \log N)$  lexicographical string comparisons.

Next we consider computing  $\text{LF}_{w_{j+1}}$  and significant suffixes of  $w_{j+1}$  assuming that we have computed  $\text{LF}_{w_j} = U_1, \dots, U_m$ . Suppose now that  $\text{leaf}_{j+1}$  is a variable (otherwise  $\text{leaf}_{j+1} \in \Sigma$ ). Let  $\text{leaf}_{j+1} = X_i$  and  $\text{LF}_{X_i} = V_1, \dots, V_{m'}$  (remark that we do not actually have  $\text{LF}_{X_i}$ ). According to the lexicographic order between  $U_m$  and  $V_1$ , which can be checked using a single LCE query by Lemma 13, we proceed as follows:

- $U_m \succ V_1$ . This implies that  $\text{LF}_{w_{j+1}} = U_1, \dots, U_m, V_1, \dots, V_{m'}$ . By Lemma 16,  $\lambda_{X_i} = 1$  holds, which means that the significant suffixes of  $X_i$  hold the whole information of  $\text{LF}_{X_i}$ . Thus we can get  $\text{LF}_{w_{j+1}}$  without string comparisons. Finally, we can compute significant suffixes of  $w_{j+1}$  by  $O(\log |\Lambda_{w_j}|)$  string comparisons (same as Lemma 15).
- $U_m = V_1$ . This implies that  $\text{LF}_{X_i} = U_1, \dots, U_{m-1}, U_m V_1, V_2, \dots, V_{m'}$ . We can compute  $\text{LF}_{w_{j+1}}$  without string comparisons in a similar way to the previous case. We can also compute  $\Lambda_{w_{j+1}}$  by  $O(\log |\Lambda_{w_j}|)$  string comparisons in a similar way to the previous case.
- $U_m \prec V_1$ . Firstly, we compute the crossing factor  $z$  of  $w_j X_i$  by  $O(\log |\Lambda_{w_j}| + \log |\Lambda_{X_i}|)$  string comparisons from Lemmas 14 and 17. Next we compute  $z^k$  by checking consecutive Lyndon factors, which can be done using two LCE queries. Then we can obtain  $\text{LF}_{w_{j+1}} = U_1, \dots, U_c, z^k, V_{c'}, \dots, V_{m'}$  because  $V_{c'}, \dots, V_{m'}$  are part of the significant suffixes of  $X_i$  due to Lemma 17. Finally, we can compute significant suffixes of  $w_{j+1}$  by  $O(\log |\Lambda_{w_j}|)$  string comparisons (same as Lemma 15).

In either case, we can compute  $\text{LF}_{w_{j+1}}$  and the significant suffixes of  $w_{j+1}$  by  $O(\log |\Lambda_{w_j}| + \log |\Lambda_{X_i}|) = O(\log \log N)$  string comparisons from Lemma 9.

Now suppose that  $leaf_{j+1} \in \Sigma$ . Since  $leaf_{j+1} = c$  is a new character that does not appear in  $w_j$ , the situation does not directly match with the condition of Lemma 17. Still it is easy to see that  $LF_c = c$ , and thus, we can compute  $LF_{w_{j+1}}$  and the significant suffixes of  $w_{j+1}$  in a similar way to the case that  $leaf_{j+1}$  is a variable.

Note that we do not have to “rebuild” the whole Lyndon factorization of  $w_{j+1}$  (which would take  $O(n)$  time) because each Lyndon factor of  $LF_{w_{j+1}}$  whose beginning position is in  $[lfb_{w_j}(\lambda_{w_j}), lfb_{w_{j+1}}(\lambda_{w_{j+1}}))$  remains as a Lyndon factor while appending strings to it, and thus, it is a Lyndon factor of  $LF_w$  to output. Hence, we can compute  $LF_w$  while treating only the last  $O(\log N)$  Lyndon factors that are corresponding to the significant suffixes of the current  $w_{j+1}$ 's.

Therefore, we can compute  $LF_w = LF_{w_g}$  by  $O(n \log \log N)$  string comparisons. ◀

Here, we analyze the space requirement. We need  $O(n \log N)$  space for all significant suffixes. In each step, the size of the Lyndon factorization is less than  $n$  by Lemma 3. Thus we need  $O(n)$  space for storing the Lyndon factorization. Finally, we get the following result by using an LCE data structure for string comparisons.

► **Theorem 19.** *Given an SLP of size  $n$  representing string  $w$  of length  $N$ , we can compute  $LF_w$  in  $O(P(n, N) + Q(n, N)n \log \log N)$  time and  $O(n \log N + S(n, N))$  space.*

When we use an LCE data structure of Lemma 4, we can get the following (since the size of LZ factorization is a lower bound on the smallest grammar [3]).

► **Corollary 20.** *Given an SLP of size  $n$  representing string  $w$  of length  $N$ , we can compute  $LF_w$  in  $O(n \log N \log \log N)$  time and  $O(n \log N)$  space.*

## 4 Conclusion

We revisited the problem of computing the Lyndon factorization on SLPs. Given an SLP  $G$  of size  $n$  representing a string  $w$  of length  $N$ , our new algorithm runs in  $O(P(n, N) + Q(n, N)n \log \log N)$  time and  $O(n \log N + S(n, N))$  space where  $P(n, N)$ ,  $S(n, N)$ ,  $Q(n, N)$  are respectively the pre-processing time, space, and query time of a data structure for longest common extensions (LCE) on SLPs. If we use the LCE data structure of [8], we can compute the Lyndon factorization in  $O(n \log N \log \log N)$  time and  $O(n \log N)$  space.

The paper [10] also proposed an algorithm to compute in  $O(s \log s)$  time and space the Lyndon factorization of a string that is compressed by LZ78 in  $s$  size. Future work would include improving this result and/or deriving new algorithms working on other compression schemes.

---

## References

- 1 Alberto Apostolico and Maxime Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995.
- 2 Srećko Brlek, Jacques-Olivier Lachaud, Xavier Provençal, and Christophe Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42(10):2239–2246, 2009.
- 3 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
- 4 K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

- 5 Jacqueline W. Daykin, Costas S. Iliopoulos, and William F. Smyth. Parallel RAM algorithms for factorizing words. *Theor. Comput. Sci.*, 127(1):53–67, 1994.
- 6 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
- 7 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.
- 8 Tomohiro I. Longest common extensions with recompression. In *Proc. CPM 2017*, pages 18:1–18:15, 2017.
- 9 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Lyndon factorization of grammar compressed text. In *Proc. CPM 2013*, pages 153–164, 2013.
- 10 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016.
- 11 Juha Kärkkäinen, Dominik Kempa, Yuto Nakashima, Simon J. Puglisi, and Arseny M. Shur. On the size of Lempel-Ziv and Lyndon factorizations. In *Proc. STACS 2017*, pages 45:1–45:13, 2017.
- 12 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *Proc. CPM 2016*, pages 28:1–28:12, 2016.
- 13 Manfred Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. PSC 2009*, pages 65–79, 2009.
- 14 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.