30th Annual Symposium on Combinatorial Pattern Matching

CPM 2019, June 18–20, 2019, Pisa, Italy

Edited by

Nadia Pisanti

Solon P. Pissis

Editors

Nadia Pisanti
University of Pisa, Italy
pisanti@di.unipi.it

Solon P. Pissis
CWI Amsterdam, the Netherlands
solon.pissis@cwi.nl

ACM Classification 2012
Mathematics of computing → Discrete mathematics; Applied computing → Computational biology; Mathematics of computing → Information theory; Information systems → Information retrieval; Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-103-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-103-0.

Publication date
June, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.CPM.2019.0

ISBN 978-3-95977-103-0 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

To all algorithmic stringologists in the world

Contents

Invited Talks

Regular Papers

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The Annual Symposium on Combinatorial Pattern Matching (CPM) is the international research forum in the areas of combinatorial pattern matching, string algorithms and related applications. The studied objects include strings as well as trees, regular expressions, graphs, and point sets, and the goal is to design efficient algorithms and data structures based on their properties, in order to design efficient algorithmic solutions for the addressed computational problems. The problems this conference deals with include those in bioinformatics and computational biology, coding and data compression, combinatorics on words, data mining, information retrieval, natural language processing, pattern matching and discovery, string algorithms, string processing in databases, symbolic computation, and text searching and indexing. This volume contains the papers presented at the 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019) held on June 18-20, 2019 in Pisa, Italy. The conference programme includes 29 contributed papers and three invited talks by Paweł Gawrychowski (University of Wrocław, Poland), Antonio Restivo (University of Palermo, Italy), and Michal Ziv-Ukelson (Ben Gurion University of the Negev, Israel). Contributions of the invited lectures are also included in this volume. For the first time, this edition of CPM will also include the "Highlights of CPM" special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this first edition we have invited Diptarka Chakraborty (Weizmann Institute of Science, Israel) to present his FOCS 2018 (best) paper "Approximating edit distance within constant factor in truly sub-quadratic time", and Nicola Prezza (University of Pisa, Italy) to present his STOC 2018 paper "At the roots of dictionary compression: string attractors". The contributed papers were selected out of 50 submissions, corresponding to an acceptance ratio of about 58%. Each submission received at least three reviews. We thank the members of the Programme Committee and all the additional external reviewers that are listed below for their hard and invaluable and collaborative work that resulted in an excellent scientific programme.

The Annual Symposium on Combinatorial Pattern Matching started in 1990, and has since then taken place every year. Previous CPM meetings were held in Paris, London (UK), Tucson, Padova, Asilomar, Helsinki, Laguna Beach, Aarhus, Piscataway, Warwick, Montreal, Jerusalem, Fukuoka, Morelia, Istanbul, Jeju Island, Barcelona, London (Ontario, Canada), Pisa, Lille, New York, Palermo, Helsinki, Bad Herrenalb, Moscow, Ischia, Tel Aviv, Warsaw, and Qingdao. From 1992 to the 2015 meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, the CPM proceedings appear in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM 2016), 78 (CPM 2017), and 105 (CPM 2018). The whole submission and review process was carried out with the help of the EasyChair conference system. We thank the CPM Steering Committee for supporting Pisa as the site for CPM 2019 and for their advice. The conference is generously hosted by the Computer Science Department of the University of Pisa. We thank Alessio Conte, Roberto Grossi, Veronica Guerrini, Giulio Pibiri, Nicola Prezza, Giovanna Rosone, Rossano Venturini (University of Pisa), and Andrea Marino (University of Florence) for their involvement and help in local arrangements. Finally, we would like to thank the MSCA RISE 2015 BIRDS Project “Bioinformatics and Information Retrieval Data Structures Analysis & Design” (GA 690941) and the UniPi PRA2017-44 Project for providing generous financial support to the conference.

Nadia Pisanti & Solon P. Pissis

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee

Nadia Pisanti (Co-Chair)
University of Pisa, Italy

Solon P. Pissis (Co-Chair)
CWI Amsterdam, the Netherlands

Golnaz Badkobeh
Goldsmiths University of London, UK

Carl Barton
European Bioinformatics Institute, UK

Djamal Belazzougui
DTISI-CERIST, Algeria

Christina Boucher
University of Florida, USA

Gianluca Della Vedova
University of Milano-Bicocca, Italy

Gabriele Fici
University of Palermo Palermo, Italy

Szymon Grabowski
Lodz University of Technology, Poland

Inge Li Gørtz
Technical University of Denmark, Denmark

Wing-Kai Hon
National Tsing Hua University, Taiwan

Tomohiro I
Kyushu Institute of Technology, Japan

Tomasz Kociumaka
University of Warsaw, Poland

Christian Komusiewicz
Philipps-Universität Marburg, Germany

Tsvi Kopelowitz
Bar Ilan University, Israel

Florin Manea
Universität Kiel, Germany

Veli Mäkinen
University of Helsinki, Finland

Robert Mercas
Loughborough University, UK

Gonzalo Navarro
University of Chile, Chile

Yakov Nekrich
University of Waterloo, Canada

Cyril Nicaud
Université Paris-Est, France

Alberto Policriti
Università degli Studi di Udine, Italy

Simon J. Puglisi
University of Helsinki, Finland

Jakub Radoszewski
University of Warsaw, Poland

Giovanna Rosone
University of Pisa, Italy

Eva Rotenberg
Technical University of Denmark, Denmark

Marie-France Sagot
INRIA and Univ. Claude Bernard, France

Jamie Simpson
Curtin University, Australia

Tatiana Starikovskaya
École Normale Supérieure, France

Jens Stoye
Universität Bielefeld, Germany

Wing-Kin Sung
National University of Singapore, Singapore

Sharma V. Thankachan
University of Central Florida, USA

Oren Weimann
University of Haifa, Israel

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Anders Aamand
Hideo Bannai
Frederique Bassino
Olivier Bodini
Tiziana Calamoneri
Bastien Cazaux
Panagiotis Charalampopoulos
Anders Roy Christiansen
Julien Clément
Anne Condon
Graham Cormode
Maxime Crochemore
Alessandro De Luca
Sebastian Deorowicz
Daniel Doerr
Guillaume Ducoffe
Bartlomiej Dudek
Pawel Gawrychowski
Simon Gog
Shay Golan
Niels Grüttemeier
Danny Hermelin
Matthew Hoban
Stepan Holub
Oded Lachish
Dominik Kempa
Christian Konrad
Dominik Köppl
Dmitry Kosolobov
Tomasz Marek Kowalski
Gregory Kucherov
Alan Kuhnle
Thierry Lecroq
Avivit Levy
Sabrina Mantaci
Daniel Martin
Yuto Nakashima
Pascal Ochem
Murray Patterson
Yann Ponty
Marco Previtali
Nicola Prezza
Eric Rivals
Raffaella Rizzi
Marinella Sciortino
Arseny Shur
Blerina Sinaimeri
Frank Sommer
Teresa Anna Steiner
Juliusz Straszyński
Alexander Tiskin
Rossano Venturini
Tomasz Waleń
Roland Wittler
Robert Zimmer
Michal Ziv-Ukelson

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Authors of Selected Papers

Mai Alzamel
Ali Akhavi
Hayam Alamro
Amir Amihood
Golnaz Badkobeh
Hideo Bannai
Djamal Belazzougui
Giulia Bernardini
Paola Bonizzoni
Laurent Bulteau
Bastien Cazaux
Julien Clément
Maxime Crochemore
Fabio Cunial
Konrad K. Dabrowski
Dimitri Darthenay
Gianluca Della Vedova
Diego Diaz-Dominguez
Guillaume Fertin
Mitsuru Funakoshi
Travis Gagie
Michał Gańczorz
Pawel Gawrychowski
Raffaele Giancarlo
Jiong Guo
Diptarama Hendrian
Costas S. Iliopoulos
Shunsuke Inenaga
Haitao Jiang
Matthew Johnson
Juha Kärkkäinen
Niko Kiirala
Tomasz Kociumaka
Eitan Kondratovsky
Dominik Köppl
Dmitry Kosolobov
Karim Labib
Loïck Lhote
Gad M. Landau
Giovanni Manzini
Oleg Merkurev
Yuto Nakashima
Gonzalo Navarro
Takaaki Nishimoto
Julian Pape-Lange
Kunsoo Park
Sung Gwan Park
Murray Patterson
Daniel Paulusma
Marcin Piątkowski
Nicola Prezza
Simon J. Puglisi
Leena Salmela
Arseny Shur
Nikita Sivukhin
Tatiana Starikovskaya
Juliusz Straszyński
Jakub Radoszewski
Antonio Restivo
Eric Rivals
Giovanna Rosone
Wojciech Rytter
Marinella Sciortino
Jan Studený
Ryo Sugahara
Yasuo Tabei
Takuya Takagi
Masayuki Takeda
Alexandru I. Tomescu
Yuki Urabe
Przemysław Uznański
Brigitte Vallee
Stephane Vialette
Tomasz Waleń
Daniel Wolleb-Graf
Binhai Zhu
Daming Zhu
Michal Ziv-Ukelson
Wiktor Zuba

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

How to Exploit Periodicity

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

Abstract

Periodicity is a fundamental combinatorial property of strings. We say that p is a period of a string $s[1..n]$ when $s[i] = s[i+p]$ for every i such that both $s[i]$ and $s[i+p]$ are defined. While this notion is interesting on its own, it can be often used as a tool for designing efficient algorithms. At a high level, such algorithms often operate differently depending on whether a given string does or does not have a small period, where small usually means smaller than half of its length (or, say, quarter). In other words, we design an algorithm that is efficient if the given string is repetitive, and another algorithm that is efficient if the given string is non-repetitive, in every case carefully exploiting either the periodicity or the fact that input looks sufficiently "random", and then choose the appropriate algorithm depending on the input. Of course, in some cases, one needs to proceed in a more complex manner, for example by classifying the whole string look at its substrings and process each of them differently depending on its structure.

I will survey results, mostly connected to different version of pattern matching, that are based on this paradigm. This will include the recent generalization of periodicity that can be applied in approximate pattern matching, and some examples of how the notion of periodicity can be applied to design a better data structure.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases periodicity, pattern matching, Hamming distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.1

Category Invited Talk

Some Variations on Lyndon Words

Francesco Dolce
IRIF, Université Paris Diderot, France
dolce@irif.fr

Antonio Restivo
Dipartimento di Matematica e Informatica, Università degli Studi di Palermo, Italy
antonio.restivo@unipa.it

Christophe Reutenauer
LaCIM, Université du Québec à Montréal, Canada
reutenauer.christophe@uqam.ca

Abstract

In this paper we compare two finite words u and v by the lexicographical order of the infinite words u^ω and v^ω. Informally, we say that we compare u and v by the infinite order. We show several properties of Lyndon words expressed using this infinite order. The innovative aspect of this approach is that it allows to take into account also non trivial conditions on the prefixes of a word, instead that only on the suffixes. In particular, we derive a result of Ufnarovskij [V. Ufnarovskij, *Combinatorial and asymptotic methods in algebra*, 1995] that characterizes a Lyndon word as a word which is greater, with respect to the infinite order, than all its prefixes. Motivated by this result, we introduce the prefix standard permutation of a Lyndon word and the corresponding (left) Cartesian tree. We prove that the left Cartesian tree is equal to the left Lyndon tree, defined by the left standard factorization of Viennot [G. Viennot, *Algèbres de Lie libres et monoïdes libres*, 1978]. This result is dual with respect to a theorem of Hohlweg and Reutenauer [C. Hohlweg and C. Reutenauer, *Lyndon words, permutations and trees*, 2003].

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Lyndon words, Infinite words, Left Lyndon trees, Left Cartesian trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.2

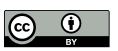
Category Invited Talk

1 Introduction

Let A be a totally ordered alphabet. A word w is called a *Lyndon word* if for each nontrivial factorization $w = uv$, one has $w < v$ (here $<$ is the lexicographical order). Lyndon words were introduced in [19].

A well-known theorem of Lyndon states that every finite word w can be decomposed in a unique way as a nonincreasing product $w = \ell_1\ell_2\cdots\ell_n$ of Lyndon words. This theorem, which is a combinatorial counterpart of the famous theorem of Poincaré-Birkhoff-Witt, provides an example of a factorization of the free monoid (see [18]). It has also many algorithmic applications and it may be computed in an efficient way. Indeed, Duval proposed in [11] a linear-time algorithms to compute it, while Apostolico and Crochemore proposed in [1] a $O(\lg n)$-time parallel algorithm.

The (right) *Lyndon tree* of a Lyndon word w corresponds recursively to the following *right standard factorization* of w, when no reduced to a single letter: w can be written as $w = uv$ where v is the longest proper nonempty suffix of w which is a Lyndon word. The word u is then also a Lyndon word. Remark that one can also define a *left standard factorization* of a Lyndon word, and then a left Lyndon tree (cf. [24] and [5]).



On the other hand, one can associate to a Lyndon word w the *Cartesian tree* corresponding to its suffix standard permutation (also commonly known as the inverse suffix array of w). Hohlweg and Reutenauer have proved that the (right) Lyndon tree of a Lyndon word is equal to its Cartesian tree (see [15]). This connection is useful for the computation of runs in a word (see, e.g. [9, 8, 2, 3]).

In this paper we consider a new approach that uses infinite words: the relation between two finite words u and v is determined by the lexicographical order of the infinite words u^ω and v^ω (where $u^\omega = uuu\cdots$). Informally, we say that we *compare u and v using the infinite order*.

Note that one can have $u < v$ but $u^\omega > v^\omega$. For instance, if $a < b$, one has $ab < aba$ but $(aba)^\omega < (ab)^\omega$.

This new relation between words has been used in some important results in combinatorics on words, as, for instance in the bijection of Gessel and Reutenauer (cf. [13]), which is at the basis of some extentions of the Burrows-Wheeler transform (see [20] and [16]).

We show that several properties of Lyndon words can be expressed by using the infinite order. We prove (Corollary16) that a word w is a Lyndon word if and only if $w^\omega < v^\omega$ for each proper suffix v of w, i.e., w is smaller, with respect to the infinite order, than all its proper suffixes. Moreover, we show that in the classical factorization theorem by Lyndon (every word can be factorized in a unique way as a non-increasing product of Lyndon words) the product is non-increasing also with respect to the infinite order. We also deduce (Theorems 21 and 23) new characterizations of the first and of the last factor of the factorization in Lyndon words.

The innovative aspect of this new approach in the study of Lyndon words is that it takes into account also conditions on the prefixes of a word, instead that only on the suffixes. In particular, we derive (Corollary17) a result of Ufnarovskij (cf. [23]) that characterizes a Lyndon word as a word which is greater, with respect to the infinite order, than all its proper prefixes.

In the last section, motivated by the Ufnarovskij's Theorem, we show that the ordering of the prefixes of a word according to the infinite order is non-trivial, and this leads to the notion *prefix array* of a word. We then introduce the *prefix standard permutation* of a word (the inverse of the prefix array) and the corresponding left Cartesian tree. We prove, as a result which is dual with respect to that of Hohlweg and Reutenauer, that the left Cartesian tree of a Lyndon words is equal to its left Lyndon tree (Theorem 36).

Some of the results of Sections 3 and 4 can be extended by considering a *generalized order* relation, i.e., an order in which the comparison between two words depends on the length of their common prefix. A word w is called a *generalized Lyndon word* if $w^\omega < v^\omega$ (with respect the generalized order) for each proper suffix v of w. Generalized Lyndon words have been introduced in [22] and their theory has been further developed in [10]. A very special case of generalized order is given by the *alternating order*, in which the comparison between two words depends on the *parity* of the length of their common prefix. The generalized Lyndon words with respect to the alternating order are called *Galois words*. A bijection, similar to that of Gessel and Reutenauer, for the alternating order, has been proved in [12]: it leads to the definition of the *Alternate Burrows-Wheeler Transform* (ABWT), that has been also studied in [14].

2 Definitions and notations

For undefined notation we refer to [17] and [18]. We denote by A a finite alphabet, by A^* the free monoid and by A^+ the free semigroup. Elements of A^* are called *words* and the identity element, denoted by 1 is called the *empty word*. We say that a word u is a *factor* of

the word w if $w = xuy$ for some words x, y; u is a *prefix* (resp. *suffix*) if $x = 1$ (resp. $y = 1$); it is *nontrivial* if $u \neq 1$ and *proper* if $u \neq w$. We say that $w = ps$ is a *nontrivial factorization* of w whenever p, s are both nonempty. The length of a word $w = a_1 a_2 \cdots a_n$, where $a_i \in A$ for all i, is equal to n and it is denoted by $|w|$.

A *period* of a word $a_1 a_2 \cdots a_n$ is a natural integer p such that $a_i = a_{i+p}$ for any i such that $i, i + p \in \{1, \ldots, n\}$; it is called *a nontrivial period* if $0 < p < n$. A word having a nontrivial period is called *periodic*.

We say that v *is a fractional power of* u if $u = u_1 u_2$ and $v = u^k u_1$ for some nonnegative integer k. In this case, one writes $v = u^r$, where $r = k + |u_1|/|u|$ is a positive rational. Note that for $k = 0$ (or $r < 1$) this means that v is a prefix of u. Fractional powers are also known as *sesquipowers* (see, e.g., [21]).

We say that the v is *a strict fractional power of* u if v is a fractional power of u and, with the notations above, $k \geq 1$ or, equivalently, that $r \geq 1$. In this case u is a prefix of v.

▶ **Example 1.** Let $u = abcdef$. Then $u^{2/3} = abcd$ and $u^{5/3} = abcdefabcd$. The last one is, in particular, a strict fractional power of u.

We denote by A^ω the set of sequences over A, also called *infinite words*; such a sequence $(a_n)_{n\geq 1}$ is also written $a_1 a_2 a_3 \cdots$. If w is a (finite) word of length $n \geq 1$, w^ω denotes the infinite word having w as a prefix and of period n.

We denote by $A^\infty = A^* \cup A^\omega$ the set of finite and infinite words.

A *border* of a word w of length n is a word which is simultaneously a nontrivial proper prefix and suffix of w. A word is called *unbordered* if it has no border. It is well-known that a word has a border if and only if it is periodic.

3 Infinite order on finite words

Given an order $<$ on the alphabet A, we can define the *lexicographical order* $<_{lex}$ (or simply $<$ when it is clear from the context) on A^∞ in the following way : $u <_{lex} v$ if either u is a proper prefix of v (in which case u must be in A^*) or we may write $u = pau'$, $v = pbv'$ for some words $p \in A^*$, $u', v' \in A^\infty$ and some letters $a, b \in A$ such that $a < b$.

▶ **Definition 2.** *Let s, t be two distinct elements of A^ω such that we have a factorization $s = u_1 \cdots u_k s_0$ with $u_1, \cdots, u_k$ finite nonempty words and s_0 is an infinite word. We say that* the comparison between s and t takes place within u_k *if $u_1 \cdots u_{k-1}$ is a prefix of t, but $u_1 \cdots u_k$ is not. If moreover $u_1, u_2, \ldots, u_k$ are letters we say that the comparison takes place* at position k.

Note that, when the comparison takes place within u_k, one may write $t = u_1 \cdots u_{k-1} u'_k t'$, for some $t' \in A^\omega$ and $u'_k \neq u_k$ such that $|u'_k| = |u_k|$.

Suppose that u, v are finite nonempty words. The following fact is well-known: one has $u^\omega = v^\omega$ if and only if u, v are power of a common word, and this is true if and only if u and v commute (see, for instance, [18, Corollary 6.2.5]).

In the sequel we compare two finite words u and v by comparing the infinite words u^ω and v^ω, with respect to the lexicographical order. Informally, we say that we *compare u and v with respect to the infinite order*.

Note that, given two nonempty finite words u, v, it is not true that $u^\omega < v^\omega \Leftrightarrow u < v$, as shown in the following example.

▶ **Example 3.** Let us consider the two words $u = b$, $v = ba$. Since u is a prefix of v we have $u < v$. Nevertheless $u^\omega = bbb \cdots > bababa \cdots = v^\omega$.

In the next section we will show that this equivalence holds for Lyndon words (Theorem 20).

Even though we use the term "infinite order", when $u^\omega \neq v^\omega$ the comparison between u^ω and v^ω takes place at a position bounded by a function of the lengths of u and v, as stated by the following lemma, which is a reformulation of the well known Fine and Wilf theorem (see, for instance, [17]).

▶ **Lemma 4.** *Let u, v be nonempty words such that $u^\omega \neq v^\omega$. Then the comparison between u^ω and v^ω takes place at a position $k \leq |u| + |v| - \gcd(|u|, |v|)$.*

The following example shows that the bound given in the previous lemma is tight.

▶ **Example 5.** Let $u = abaab$ and $v = abaababa$. Then,

$$u^\omega = abaababaabab\ldots \quad \text{and} \quad v^\omega = abaababaabaa\ldots.$$

One has that $v^\omega < u^\omega$ and that the comparison takes place at position $12 = |u| + |v| - 1$.

▶ **Lemma 6.** *Let u, v be nonempty words such that $u^\omega \neq v^\omega$. Then the comparison between u^ω and v^ω takes place within the first factor v of v^ω if and only if v is not a fractional power of u.*

Proof. The comparison between the two infinite words takes place within the first v if and only if the two prefixes of length $|v|$ of u^ω and v^ω are different. The conclusion follows from the fact that v is a fractional power of u if and only if v is a prefix of u^ω. ◀

▶ **Lemma 7.** *Let $s, t \in A^\omega$ be as in Definition 2 (and the sentence following it). Then $s < t$ (resp. $s > t$) implies $u_1 \cdots u'_k s' < u_1 \cdots u_k t'$ (resp. $u_1 \cdots u'_k s' > u_1 \cdots u_k t'$) for any infinite words s', t'.*

▶ **Lemma 8.** *Let u, v be nonempty finite words such that $u^\omega < v^\omega$ and let x, y be two finite words. Then*

- **(i)** *if neither u or v is a prefix of the other, then $(ux)^\omega < (vy)^\omega$;*
- **(ii)** *if v is not a fractional power of u, then $(u^{k+1}x)^\omega < (vy)^\omega$, where k is the largest integer such that u^k is a prefix of v. In particular $u^\omega < (vy)^\omega$.*

Proof. In case (i), the comparison between the two infinite words takes place within the prefix of length $\min(|u|, |v|)$. Hence we conclude using Lemma 7.

Suppose now that the hypothesis of (ii) holds. Then we can write $u = u'au_1$ and $v = u^k u' b v_1$, with $u' \in A^*$, $a, b \in A$ such that $a \neq b$, and $u_1, v_1 \in A^*$. Let $m = |u^k u'|$. Since $u^\omega = u^k u' a u_1 u^\omega$ and since $u^\omega < v^\omega$, we have that $a <_{m+1} b$. The two infinite words u^ω and $(u^{k+1}x)^\omega$ share the same prefix of length $m+1$, and the same do v^ω and $(vy)^\omega$. Thus the comparison between between $(u^{k+1}x)^\omega$ and $(vy)^\omega$ takes place at position $m+1$. Since $a <_{m+1} b$, we can conclude. ◀

We use several times the following observation: the opposite order $\tilde{<}$ of an order $<$ is also a lexicographical order.

▶ **Example 9.** Let $<$ be the usual lexicographical order on $\{a, b\}$, that is such that $a < b$. Then $\tilde{<}$ is defined by $b \tilde{<} a$.

▶ **Theorem 10.** *Let u, v be finite nonempty words such that $u^\omega \neq v^\omega$, and s, t be infinite words in $\{u, v\}^\omega$. Then*

$$u^\omega < v^\omega \iff us < vt.$$

Proof. Without loss of generality, we may assume that u, v are primitive.

Suppose that $u^\omega < v^\omega$. Let us first consider the case when v is not a fractional power of u. Then we can write $u = u_1au_2$, $v = u^k u_1 b v_2$ with letters $a \neq b$ and $k \geq 0$. Then $u^\omega = u^k u_1 a \cdots$ and $v^\omega = u^k u_1 b \cdots$. Since $u^\omega < v^\omega$, we must have $a < b$. Note that by hypothesis, one has either $us = u^\omega$ or $us = u^l v \cdots$, with $l \geq 1$. In both cases us begins by u^{k+1}. Thus $us = u^k u_1 a \cdots$. Moreover vt begins by v and hence $vt = u^k u_1 b \cdots$. Therefore $us < vt$.

Suppose now that v is a strict fractional power of u. We can write $v = u^k u_1$, $u = u_1 u_2$, for some $k \geq 1$, and u_1, u_2 nonempty finite words (since $u^\omega \neq v^\omega$). In particular, we have $u_1u_2 \neq u_2u_1$ since u is primitive. Then $u^\omega = u^k u_1 u_2 u_1 \cdots$ and $v^\omega = u^k u_1 u_1 u_2 \cdots$ (since $k \geq 1$). From the inequality $u^\omega < v^\omega$, we deduce that $u_2u_1 < u_1u_2$. We claim that $us = u^k u_1 u_2 u_1 \cdots$: indeed, either $s = u^l v \cdots$, $1 \leq l \leq \infty$, so that u^{k+2} is a prefix of us, implying the claim; or $s = v \cdots$, so that us begins by $uv = uu^k u_1 = u^k u_1 u_2 u_1$ and the claim is true, too. Moreover $vt = u^k u_1 u_1 u_2 \cdots$. Therefore $us < vt$.

It remains the case where v is a proper prefix of u. Then u is either not a fractional power of v, or u is a strict fractional power of v. In this case, we have $v^\omega \mathrel{\widetilde{<}} u^\omega$. Hence the previous arguments imply that $vt \mathrel{\widetilde{<}} us$ and therefore $us < vt$.

Suppose now that $u^\omega < v^\omega$ does not hold. Since $u^\omega \neq v^\omega$, we have $u^\omega \mathrel{\widetilde{<}} v^\omega$. From the previous arguments it follows that $us \mathrel{\widetilde{<}} vt$, hence $vt < us$ and therefore $us < vt$ does not hold. ◀

Using the previous theorem we can prove the following result (part of the it is stated, in a more general context, in [22], see also [10]).

▶ **Corollary 11.** *The following conditions are equivalent for nonempty words $u, v \in A^*$:*

(1) $u^\omega < v^\omega$;
(2) $(uv)^\omega < v^\omega$;
(3) $u^\omega < (vu)^\omega$;
(4) $(uv)^\omega < (vu)^\omega$;
(5) $u^\omega < (uv)^\omega$;
(6) $(vu)^\omega < v^\omega$.

Proof. It follows from Theorem 10 that Condition (1) is equivalent to each of the Conditions (2), (3) and (4).

Condition (5) is equivalent to condition (3): indeed $u^\omega < (vu)^\omega \Leftrightarrow uu^\omega < u(vu)^\omega \Leftrightarrow u^\omega < (uv)^\omega$. Similarly, condition (6) is equivalent to condition (2): indeed, $(uv)^\omega < v^\omega \Leftrightarrow v(uv)^\omega < vv^\omega \Leftrightarrow (vu)^\omega < v^\omega$. ◀

Note that previous corollary implies a result proved by Bergman in [4, Lemma 5.1] (see also [23, p.34 and pp.101–102]).

▶ **Corollary 12.** *Let u, v be two finite words such that $u^\omega < v^\omega$. Then*

$$u^\omega < (uv)^\omega < (vu)^\omega < v^\omega. \tag{1}$$

Another immediate consequence of Corollary 11 is that one can define the "infinite order" without any explicit use of infinite words, as stated by the following corollary.

▶ **Corollary 13.** *Let u, v be two finite words. Then $u^\omega < v^\omega$ if and only if $uv < vu$.*

4 Lyndon words

In this section we show some fundamental properties of Lyndon words by using the infinite order, instead of the classical lexicographical order. Moreover, the infinite order allows to introduce an innovative point of view about Lyndon words, by taking into account conditions on the prefixes of a word (instead that only on the suffixes). In particular, we derive a result by Ufnarovskij (Corollary 17) that characterizes a Lyndon word as a word which is greater (with respect to the infinite order) than its proper prefixes.

Let us start with the classical definition of Lyndon words in terms of the usual lexicographical order (see, for instance, [11]).

▶ **Definition 14.** *A word w is a Lyndon word if any one of the following three equivalent conditions holds:*

(i) *for any nontrivial factorization $w = uv$, $u < v$,*
(ii) *for any nontrivial factorization $w = uv$, $uv < v$,*
(iii) *for any nontrivial factorization $w = uv$, $uv < vu$.*

The following theorem provides a characterization of Lyndon words by using the infinite order.

▶ **Theorem 15.** *A word w is a Lyndon word if and only if, for any nontrivial factorization $w = uv$, any one of the six equivalent conditions of Corollary 11 holds.*

Proof. From Corollary 13 it follows that condition (iii) of Definition 14 is equivalent to condition (1) of Corollary 11. ◀

In next corollary (part of whose has already been proved, in a more general context, in [22, Proposition 2.1], see also [10]) we highlight conditions (1) and (2) of Corollary 11. Even though such conditions show some formal similarity with the conditions (i) and (ii) of Definition 14, they are essentially different.

▶ **Corollary 16.** *A word w is a Lyndon word if and only if one of the following condition is satisfied for any nontrivial factorization $w = uv$:*

1. $u^\omega < v^\omega$.
2. $w^\omega < v^\omega$.

The following result, due to Ufnarovskij (see also [23, Theorem 2, p.35]) follows from Theorem 15 and condition (5) of Corollary 11. It provides a characterization of a Lyndon word that takes into account its prefixes, instead than its suffixes.

▶ **Corollary 17** (Ufnarovskij)**.** *A word w is a Lyndon word if and only if for any nontrivial factorization $w = ps$, one has $p^\omega < w^\omega$.*

▶ **Example 18.** The word $w = aabab$ is a Lyndon word. We have $a^\omega = (aa)^\omega < (aaba)^\omega < (aab)^\omega < w^\omega$.

The following result is classical (see, for instance [17]).

▶ **Theorem 19.** *Each word in A^* can be factorized in a unique way as a nonincreasing product of Lyndon words.*

The term "nonincreasing" in the previous theorem is referred to the classical lexicographical order. The following theorem (cf. [6, Theorem 8]) shows that the factorization in Lyndon words is non-increasing also with respect to the infinite order.

▶ **Theorem 20.** *Let u, v be two Lyndon words. Then $u < v$ if and only if $u^\omega < v^\omega$,*

Proof. Suppose that $u < v$. If u is not a prefix of v, then obviously $u^\omega < v^\omega$.

If u is a prefix of v, then $v = uy$, for some nonempty word y. From $u < v$ and $v < y$ (since v is a Lyndon word) we get $u^2 < uy = v$, and thus $u^\omega < v^\omega$.

To prove the converse implication, let us suppose that $u^\omega < v^\omega$. By contradiction, suppose $v < u$. By the first part of the proof we would have $v^\omega < u^\omega$, which gives us a contradiction. ◀

In the following theorem we characterize the last element of the factorization in Lyndon words by using the infinite order.

▶ **Theorem 21.** *Let $w = \ell_1\ell_2\cdots\ell_n$, with ℓ_i Lyndon words such that $\ell_1^\omega \geq \ell_2^\omega \geq \ldots \geq \ell_n^\omega$. Then ℓ_n is the shortest among all nontrivial suffixes s of w such that s^ω is minimum.*

Proof. Let z be the shortest among all nontrivial suffixes of w such that s^ω is minimum. If $w = z$ then w is a Lyndon word and $\ell_1 = \ell_n = z$.

Otherwise, we can write $w = uz$. Consider the factorization of u in Lyndon words: $u = \ell'_1\ell'_2\cdots\ell'_k$, with $\ell'^\omega_1 \geq \ell'^\omega_2 \geq \cdots \geq \ell'^\omega_k$. By hypothesis, $(\ell'_k z)^\omega > z^\omega$. Then, using Corollary 11, one has $\ell'^\omega_k > z^\omega$. It follows that $\ell'_1\ell'_2\cdots\ell'_k z$ is a non-increasing factorization of w in Lyndon words. Since the factorization is unique, we have that $z = \ell_n$. ◀

▶ **Example 22.** Let $w = ababaab$. Its non-increasing factorization into Lyndon words is $w = (ab)(ab)(aab)$. One can check that $(aab)^\omega < (abaab)^\omega < w^\omega < (ab)^\omega < (baab)^\omega < (babaab)^\omega < b^\omega$.

In the previous results we gave a characterization of the last element of the factorization of a word in Lyndon words. Now, we focus on the first factor. This result is motivated by point 1 of Corollary 16: the fact that a word w is not a Lyndon word implies the existence of a prefix u such that $u^\omega \geq v^\omega$, where v is the corresponding suffix. If one chooses the shortest prefix satisfying this property, this turns out to be the first factor in the Lyndon factorization. In the same vein, it is motivated by Ufnarovskijj's Theorem (Corollary 17 above).

▶ **Theorem 23.** *Let $w = \ell_1\ell_2\cdots\ell_n$ be the nonincreasing factorization into Lyndon words of a finite nonempty word w.*

1. *The word ℓ_1 is the shortest nontrivial prefix p of w such that, when writing $w = ps$, one has either $s = 1$ or $p^\omega \geq s^\omega$.*
2. *The word ℓ_1 is the shortest nontrivial prefix p of w such that $p^\omega \geq w^\omega$.*

In order to prove Theorem 23 we need a preliminary result which refines Corollary 11 in the case of the usual lexicographical order.

Note that, for any infinite words s, t such that $s < t$, with $<$ the classical order, and for any finite word w, one has $ws < wt$.

▶ **Corollary 24.** *If $\ell_1, \ell_2, \ldots, \ell_n$, with $n \geq 2$, are Lyndon words such that $\ell_1^\omega \geq \ell_2^\omega \geq \cdots \geq \ell_n^\omega$, then $\ell_1^\omega \geq (\ell_2\cdots\ell_n)^\omega$.*

Proof. The case $n = 2$, it is trivial. Let consider the case $n \geq 3$. By induction hypothesis we have $\ell_2^\omega \geq (\ell_3\cdots\ell_n)^\omega$. From Corollary 11 it follows that $\ell_2^\omega \geq (\ell_2\cdots\ell_n)^\omega$. Hence, $\ell_1^\omega \geq (\ell_2\cdots\ell_n)^\omega$. ◀

It order to prove Theorem 23 let us recall the well-known fact that all Lyndon words are unbordered (see, for instance, [7]).

Proof of Theorem 23. Let us prove the first assertion. When $n = 1$, then $w = \ell_1$ is a Lyndon word and the result is true by point 1 of Corollary 16.

Suppose now that $n \geq 2$. Then, by Corollary 24, we have $\ell_1^\omega \geq (\ell_2 \cdots \ell_n)^\omega$. Let p be a nontrivial prefix of w shorter then ℓ_1. Thus, we have a nontrivial factorization $\ell_1 = pq$ for some $q \neq 1$. By Corollary 16, we know that $p^\omega < q^\omega$. Since ℓ_1 is unbordered, q cannot be a fractional power of p. Thus, by point (ii) of Lemma 8, one has $p^\omega < (q\ell_2 \cdots \ell_n)^\omega$, which prove the first part of the theorem.

The second assertion just follows from the first one. Indeed, using Corollary 11, we have that if $s \neq 1$, then $p^\omega \geq s^\omega$ is equivalent to $p^\omega \geq (ps)^\omega$. ◀

▶ **Example 25.** Let $w = ababaab$. As seen in Example 22, its nonincreasing factorization into Lyndon words is $w = (ab)(ab)(aab)$. One can check that $(ab)^\omega > w^\omega > (abaab)^\omega$ while $a^\omega < w^\omega < (babaab)^\omega$.

5 Left Lyndon tree and prefix standardization

In [15], the authors associate with each Lyndon word w an increasing tree, based on the suffixes of w; they show that the completion of this tree, with leaves appropriately labeled by the letters of w, is equal to the tree obtained by iterating the right standard factorization of w (equivalently the Lie bracketing associated with w).

In this section we give a similar construction and result, based on the prefixes of w instead that on the suffixes. This construction is motivated by Ufnarovskij's Theorem (Corollary 17).

5.1 Left Lyndon tree

In [24] (cf. also [5]) Viennot introduced the notion of left standard factorization of a Lyndon word. Let us consider a Lyndon word w having length at least 2. The *left standard factorization* of w is the factorization $w = uv$, where u is the longest nonempty proper prefix of w which is a Lyndon word.

The following is a well-known result (see [24, p. 14]).

▶ **Proposition 26.** *Let $w = uv$ be a Lyndon word with its left standard factorization. Then both u and v are Lyndon words and $u < v$. Moreover, either v is a letter, or given its left standard factorization $v = v_1v_2$ one has $v_1 \leq u$.*

▶ **Corollary 27.** *Let u, v, v_1 and v_2 as in Proposition 26. Then v_1 is a prefix of u.*

Proof. By Proposition 26 we have $v_1 \leq u < v_1v_2$. By a classical property of lexicographical order, this implies that $u = v_1m$, for a certain word m such that $m < v_2$. Thus v_1 is a prefix of u. ◀

▶ **Example 28.** Let us consider the Lyndon word $w = aabaacab$ on the alphabet $\{a, b, c\}$ with $a < b < c$. Its left standard factorization is $w = (aabaac)(ab)$. The suffix $v = ab$ is a Lyndon word with left standard factorization $(a)(b)$ and one has $a \leq aabaac$.

The *free magma* $M(A)$ over A is the set of *complete trees* over A defined recursively as follows:

- each letter is a tree;
- if $\mathfrak{t}_1, \mathfrak{t}_2$ are trees, then $(\mathfrak{t}_1, \mathfrak{t}_2)$ is a tree.

We will use the classical notions of *root*, *internal node* and *leaf* for a tree.

There is a canonical surjective mapping φ from $M(A)$ onto A^+ defined as follows: given a tree $\mathfrak{t}$, its image $\varphi(\mathfrak{t})$, called its *foliage*, is defined recursively by:

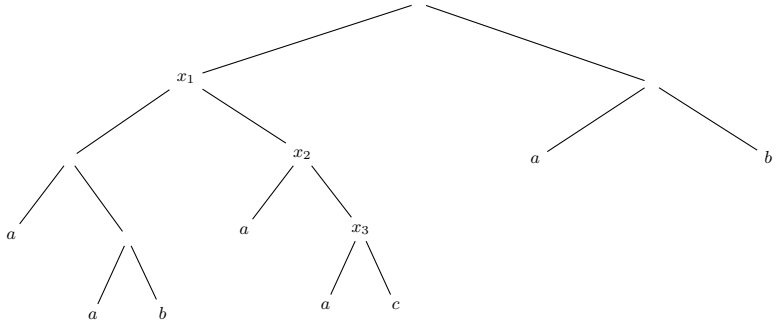


Figure 1 The left Lyndon tree of $w = aabaacab$.

- $\varphi(a) = a$ for any a in A;
- $\varphi((\mathfrak{t}_1, \mathfrak{t}_2)) = \varphi(\mathfrak{t}_1)\varphi(\mathfrak{t}_2)$ for any two trees $\mathfrak{t}_1, \mathfrak{t}_2$.

In other words, $\varphi(\mathfrak{t})$ is obtained by vertically projecting the leaves of $\mathfrak{t}$ onto a horizontal line.

▶ **Example 29.** The foliage of the tree in Figure 1 is the word $aabaacab$.

To each Lyndon word in A^+ we can associate a complete tree $\mathcal{L}(w)$ in $M(A)$, called the *left Lyndon tree* of w, defined recursively as follows:

- $\mathcal{L}(a) = a$ for each letter $a \in A$;
- $\mathcal{L}(w) = (\mathcal{L}(u), \mathcal{L}(v))$ for each Lyndon word w of length at least 2 with left standard factorization $w = uv$.

It is clear by the definition that $\varphi(\mathcal{L}(w)) = w$.

▶ **Example 30.** The left Lyndon tree associated to the Lyndon word $w = aabaacab$ is shown in Figure 1 (disregarding the labels of the internal nodes).

5.2 Prefix standardization

Consider an alphabet A and a total order $<$ on A. We define an order $\prec$ on the free monoid as follows: $u \prec v$ if either

- $u^\omega < v^\omega$, or
- $u^\omega = v^\omega$ (which means that u, v are power of the same word) and if u is longer than v.

▶ **Example 31.** Let us consider the above order on $\{a, b\}^*$ induced by $a < b$. One has $aa \prec a \prec ab \prec ba \prec b$.

Let w be a word of length $n + 1$ on a totally ordered alphabet. Let us consider the sequence $p_1, p_2, \ldots, p_n, p_{n+1} = w$ of its nonempty prefixes, in increasing length.

Motivated by Ufnarovskij's Theorem (Corollary 17), we call *prefix standard permutation* of w the unique permutation $\sigma \in \mathfrak{S}_{n+1}$ such that

$$p_{\sigma^{-1}(1)} \prec p_{\sigma^{-1}(2)} \prec \cdots \prec p_{\sigma^{-1}(n)} \prec p_{\sigma^{-1}(n+1)}.$$

In other words, we number each letter in w by $1, 2, \ldots, n, n + 1$ as follows: 1 for the letter where the $\prec$-smallest prefix ends, 2 for the second smallest one in the order $\prec$, and so on. Then σ is the represented by the word w translated in this new alphabet. We denote such a permutation by $\mathrm{pstd}(w)$.

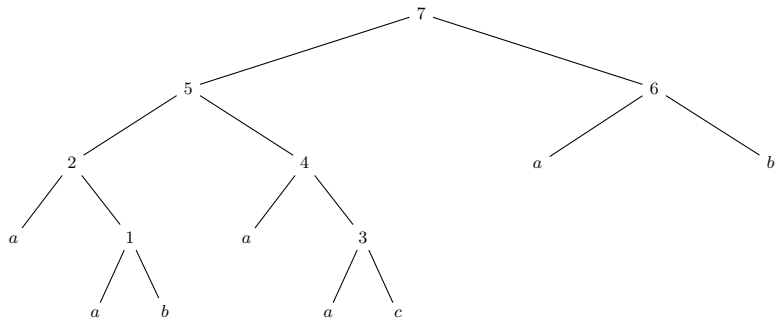


Figure 2 The tree $\mathcal{C}(aabaacab)$.

▶ **Example 32.** The prefix standard permutation of the word $w = aabaacab$ is the permutation $\mathrm{pstd}(w) = 21543768$. Indeed on can check that $aa \prec a \prec aabaa \prec aaba \prec aab \prec aabaaca \prec aabaac \prec w$.

▶ Remark 33. In [15] the authors define the "suffix standard permutation" of a word. In literature, this is also commonly known as "inverse suffix array". Therefore, our prefix standard permutation could also be called "inverse prefix array".

An *injective word* is a word without repetition of letters. Each permutation in $\mathfrak{S}_n$ can be seen as a complete injective word, that is an injective word in the ordered alphabet $\{1 < 2 < \ldots < n\}$, having all n letters as factors.

With each injective word α on a totally ordered alphabet, and in particular with each permutation $\alpha \in \mathfrak{S}_n$, we can associate bijectively a binary noncomplete labeled *decreasing tree* (i.e., such that each node is smaller than its father) defined recursively as follows:

- the root is n, the maximum letter in α;
- its left and right subtrees (which may be empty) are the trees associated to u and v respectively, where u, v are injective words in $\{1, 2, \ldots, n\}$ such that $\alpha = unv$.

The inverse bijection is obtained by vertically projecting the labels on the horizontal line.

▶ **Example 34.** The decreasing tree associated to $\alpha = 2154376$ is the tree shown in Figure 2 restricted to its internal nodes.

Let us now associate a tree with a Lyndon word, called the *left Cartesian tree* of w, and denoted $\mathcal{C}(w)$, as follows (see also [9] and the references there). To each letter $a \in A$ we define $\mathcal{C}(a) = a$. Otherwise, let $\sigma = \mathrm{pstd}(w) \in \mathfrak{S}_{n+1}$, with $|w| = n + 1$, and let $\alpha \in \mathfrak{S}_n$ be the permutation obtained by removing the last digit in σ (which is $n + 1$, because of Corollary 17). Using the construction before we obtain a noncomplete binary decreasing tree $\mathfrak{t}^*$ having n nodes. We define $\mathcal{C}(w)$ as the complete binary tree having $\mathfrak{t}^*$ as tree of internal nodes and the letters of w as leaves in order from left to right, i.e., such that $\varphi(\mathfrak{t}) = w$.

▶ **Example 35.** The tree $\mathcal{C}(w)$ for $w = aabaacab$ is shown in Figure 2.

For our purpose, we give an alternative construction of $\mathcal{C}(w)$. Consider the sequence of proper prefixes $(p_1, p_2, \ldots, p_n)$ of w, viewed as a word of length n on the alphabet A^*, totally ordered by $\prec$. Since it is an injective word, we can associate with it a decreasing tree as before; call it $t^*(w)$. Also as before, we can complete $t^*(w)$ to a tree $t(w)$, having the letters of w as leaves in ordrer from left to right, i.e., such that $\varphi(t(w)) = w$. The completed tree $t(w)$, disregarding the labels of the internal nodes, coincides with $\mathcal{C}(w)$. Indeed, the word α coincides with the previous injective word, up to the unique increasing order isomorphism from $\{1, \ldots, n\}$ into the set of proper prefixes of w, sending i to the i-th prefix for the order $\prec$.

5.3 Equivalence of the trees

From the constructions seen before it is clear that $\varphi(\mathcal{L}(w)) = \varphi(\mathcal{C}(w))$ for every Lyndon word w. We actually have a stronger result.

▶ **Theorem 36.** *Let w be a Lyndon word. The trees $\mathcal{L}(w)$ and $\mathcal{C}(w)$ are equal.*

To prove the theorem we need some intermediate result. Let x be an internal node of some planar binary complete tree $\mathfrak{t} = (\mathfrak{t}_1, \mathfrak{t}_2)$. We call *left subtrees sequence* with respect to the tree $\mathfrak{t}$ and the node x, denoted $\mathrm{lss}(\mathfrak{t}, x)$, the sequence of subtrees of $\mathfrak{t}$ hanging at the left of the path from the root to x, that is the sequence of subtrees of $\mathfrak{t}$ recursively defined as follows:

- if x is the root, then $\mathrm{lss}(\mathfrak{t}, x) = (\mathfrak{t}_1)$;
- if x is in $\mathfrak{t}_1$, then $\mathrm{lss}(\mathfrak{t}, x) = \mathrm{lss}(\mathfrak{t}_1, x)$;
- if x is in $\mathfrak{t}_2$, then $\mathrm{lss}(\mathfrak{t}, x) = (\mathfrak{t}_1, \mathrm{lss}(\mathfrak{t}_2, x))$.

▶ **Lemma 37.** *Let w be a Lyndon word and let x be an internal node of $\mathcal{L}(w)$. Let $lss(\mathfrak{t}, x) = (\mathfrak{t}_1, \ldots, \mathfrak{t}_n)$ and let ℓ_i be the foliage of the tree $\mathfrak{t}_i$ for each i. Then all ℓ_i are Lyndon words. Moreover, ℓ_{i+1} is a prefix of ℓ_i for each $1 \leq i \leq n-1$.*

Proof. The fact that each ℓ_i is a Lyndon word follows from the more general fact that the foliage of each subtree of $\mathcal{L}(w)$ is a Lyndon word, as follows recursively from the construction of $\mathcal{L}(w)$.

We prove the other assertion by induction on the size of the tree. If x is the root, then $n = 1$ and there is nothing to prove.

Let us suppose that x is not the root of the tree and let us write $\mathcal{L}(w) = (\mathfrak{t}', \mathfrak{t}'')$.

If x is an internal node of $\mathfrak{t}'$, then $\mathrm{lss}(\mathfrak{t}, x) = \mathrm{lss}(\mathfrak{t}', x)$, and we can conclude by induction.

Suppose now that x is an internal node of $\mathfrak{t}''$, and let $\mathfrak{t}'' = (\mathfrak{s}_1, \mathfrak{s}_2)$. Thus $\mathrm{lss}(\mathfrak{t}, x) = (\mathfrak{t}_1, \mathfrak{t}_2, \ldots, \mathfrak{t}_n)$, with $\mathfrak{t}_1 = \mathfrak{t}'$ and $(\mathfrak{t}_2, \ldots \mathfrak{t}_n) = \mathrm{lss}(\mathfrak{t}'', x)$ (note that $n \geq 2$). By induction it is enough to show that ℓ_2 is a prefix of ℓ_1. Let v_1, v_2 be respectively the foliages of $\mathfrak{s}_1, \mathfrak{s}_2$. Then by the property of the left standard factorization and by the construction of the tree $\mathcal{L}(w)$, we have that v_1 is a prefix of ℓ_1. Now either x is in $\mathfrak{s}_2$ and $\mathfrak{t}_2 = \mathfrak{s}_1$ and $\ell_2 = v_1$ or x is in $\mathfrak{s}_1$ and ℓ_2 is a proper prefix of v_1 In both cases ℓ_2 is a prefix of ℓ_1. ◀

▶ **Example 38.** Let us consider the tree $\mathfrak{t}$ in Figure 1 and its internal node x_3. The left subtrees sequence $\mathrm{lss}(\mathfrak{t}, x_3)$ is equal to $(\mathfrak{t}_1, \mathfrak{t}_2, \mathfrak{t}_3)$, where $\mathfrak{t}_i$ is the subtree of $\mathfrak{t}$ having root x_i. The foliages of the three subtrees are respectively the words $\ell_1 = aab$, $\ell_2 = a$ and $\ell_3 = a$. Each of them is a prefix of the previous one.

▶ **Lemma 39.** *Let $\ell_1, \ldots, \ell_n$, be Lyndon words such that ℓ_{i+1} is a prefix of ℓ_i for each $i = 1, \ldots, n-1$. Let $\ell_n = \ell'_n \ell''_n$ be the left standard factorization of ℓ_n. Then*

(i) $(\ell_1 \cdots \ell_{n-1} \ell'_n)^\omega < (\ell_1 \cdots \ell_n)^\omega$;

(ii) *moreover, if $n \geq 2$, one has $(\ell_1 \cdots \ell_n)^\omega \leq (\ell_1 \cdots \ell_{n-1})^\omega$.*

Proof. If $n = 1$, we have $\ell_1'^\omega < \ell_1''^\omega$ by Corollary 16 and $\ell_1'^\omega < (\ell'_1 \ell''_1)^\omega$ by Corollary 11. Hence $\ell_1'^\omega < \ell_1^\omega$.

Suppose now that $n \geq 2$. Since both $(\ell_1 \cdots \ell_{n-1} \ell'_n)^\omega$ and $(\ell_1 \cdots \ell_n)^\omega$ start with the word $\ell_1 \cdots \ell_{n-1} \ell'_n$, we have

$$(\ell_1 \cdots \ell_{n-1} \ell'_n)^\omega < (\ell_1 \cdots \ell_n)^\omega \iff (\ell_1 \cdots \ell_{n-1} \ell'_n)^\omega < \ell''_n (\ell_1 \cdots \ell_n)^\omega.$$

Since ℓ_n is a prefix of ℓ_1, the last inequality is of the form $\ell_n s_0 < \ell''_n t_0$, with $s_0, t_0 \in A^\omega$, and this is true since $\ell_n < \ell''_n$ by Corollary 16 and because ℓ_n is not a prefix of ℓ''_n.

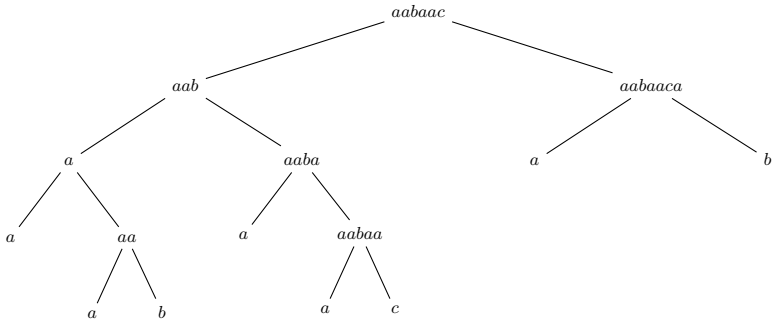


Figure 3 Variant of the left Lyndon tree of $w = aabaacab$ labeling the internal nodes with their left foliage.

Let us now prove point (ii). If $\ell_1 = \ell_2 = \cdots = \ell_n$, we trivially have $(\ell_1 \cdots \ell_n)^\omega = (\ell_1 \cdots \ell_{n-1})^\omega$. Thus, let us suppose that $\ell_1 \neq \ell_n$. Since both terms of the inequality start with $\ell_1 \cdots \ell_{n-1}$, we have

$$(\ell_1 \cdots \ell_n)^\omega \ \leq \ (\ell_1 \cdots \ell_{n-1})^\omega \quad \Longleftrightarrow \quad \ell_n(\ell_1 \cdots \ell_n)^\omega \ \leq \ (\ell_1 \cdots \ell_{n-1})^\omega.$$

Since ℓ_n is a prefix of ℓ_1, and $\ell_n \neq \ell_1$ we can write $\ell_1 = \ell_n v$ for a nonempty finite word v. Thus, the last inequality is of the form $\ell_n \ell_1 s_0 \leq \ell_n v t_0$ with $s_0, t_0 \in A^\omega$, and this is equivalent to $\ell_1 s_0 \leq v t_0$, which is true because of Corollary 16. ◀

Before proving the main result let us introduce the following notation. Let $\mathfrak{t} = (\mathfrak{t}_1, \mathfrak{t}_2)$ be a complete binary labeled tree. To each internal node x of $\mathfrak{t}$ we associate the word $g_\mathfrak{t}(x)$, called the *left foliage* of x in $\mathfrak{t}$, as follows:

- if x is the root of $\mathfrak{t}$, then $g_\mathfrak{t}(x) = \varphi(\mathfrak{t}_1)$;
- if x is in $\mathfrak{t}_1$, then $g_\mathfrak{t}(x) = g_{\mathfrak{t}_1}(x)$;
- if x is in $\mathfrak{t}_2$, then $g_\mathfrak{t}(x) = \varphi(\mathfrak{t}_1) g_{\mathfrak{t}_2}(x)$.

For further use, we note that the length of $g_\mathfrak{t}(x)$ is equal to the number of leaves located at the left of x in $\mathfrak{t}$.

Also, we use the following result.

▶ **Lemma 40.** *Let $\mathfrak{t}$ be a complete binary labeled tree and x an internal note of $\mathfrak{t}$. Then $g_\mathfrak{t}(x) = \ell_1 \cdots \ell_n$, where $\ell_1, \ldots, \ell_n$ are the foliages of the trees hanging at the left on the path from the root to x.*

Proof. Let $\mathrm{lss}(\mathfrak{t}, x) = (\mathfrak{t}_1, \ldots, \mathfrak{t}_n)$. We have to show that $g_\mathfrak{t}(x) = \varphi(\mathfrak{t}_1) \cdots \varphi(\mathfrak{t}_n)$. This follows easily from the recursive definition of both *lls* and g. ◀

▶ **Example 41.** The label shown in Figure 3 is obtained by relabeling the internal nodes of the tree $\mathcal{L}(w)$ in Figure 1, using the left foliage function.

Proof of Theorem 36. Let us consider the two trees $\mathcal{L}(w)$ and $\mathcal{C}(w)$. Note first that they have the same foliage. So, in order to prove the equality, it is enough to show that the two trees obtained from $\mathcal{L}(w)$ and $\mathcal{C}(w)$ by removing the leaves (that is, considering only the internal nodes) are equal.

We actually show that by labeling the internal nodes of $\mathcal{L}(w)$ using the function $g_\mathfrak{t}$, we obtain the same tree as $\mathcal{C}(w)$. For this it is enough to show that the labeling $g_\mathfrak{t}$ is decreasing, and that its projection is exactly the sequence $(p_1, \ldots, p_n)$ of nonempty prefixes of w. Indeed, the decreasing tree associated with an injective word on a totally alphabet is unique.

Let us consider two internal nodes x and y of t, such that y is a child of x. We show that $g_{\mathrm{t}}(y) \prec g_{\mathrm{t}}(x)$, i.e., that either

- $g_{\mathrm{t}}(y)^\omega < g_{\mathrm{t}}(x)^\omega$, or
- $g_{\mathrm{t}}(y)^\omega = g_{\mathrm{t}}(x)^\omega$ and $|g_{\mathrm{t}}(y)| > |g_{\mathrm{t}}(x)|$.

Suppose first that y is a right child of x and denote by $h_1, \ldots, h_n$ the foliages of the subtrees in the sequence $\mathrm{lss}(\mathrm{t}, y)$. We have $n \geq 2$, and by Lemma 40 $g_{\mathrm{t}}(x) = h_1 \cdots h_{n-1}$, and $g_{\mathrm{t}}(y) = h_1 \cdots h_n$. Thus $g_{\mathrm{t}}(y) \prec g_{\mathrm{t}}(x)$ follows from Lemmata 37 and 39 and from the fact that $|g_{\mathrm{t}}(y)| > |g_{\mathrm{t}}(x)|$.

Suppose now that y is a left child of x. Let $\mathrm{lss}(\mathrm{t}, x) = (\mathrm{t}_1, \ldots, \mathrm{t}_n)$, with $\mathrm{t}_n = (\mathrm{t}'_n, \mathrm{t}''_n)$. Then $\mathrm{lss}(\mathrm{t}, y) = (\mathrm{t}_1, \ldots, \mathrm{t}_{n-1}, \mathrm{t}'_n)$. Let h_i be the foliage of t_i for each $1 \leq i \leq n$, and h'_n be the foliage of t'_n. Thus $g_{\mathrm{t}}(x) = h_1 \cdots h_n$ and $g_{\mathrm{t}}(y) = h_1 \cdots h_{n-1} h'_n$. Thus $g_{\mathrm{t}}(x) \prec g_{\mathrm{t}}(y)$ by the Lemmata 37 and 39.

It remains to show that the projection is exactly $(p_1, \ldots, p_n)$. This follows from the fact the length of $g_{\mathrm{t}}(x)$ is equal to the number of leaves located at the left of a given node x; hence $g_{\mathrm{t}}(x)$ is the prefix of w of length this number. We conclude because the lengths of the successive projections of the internal nodes increase by 1 from left to right. ◀

▶ **Example 42.** Let $w = aabaacab$. The tree $\mathcal{L}(w)$ with each internal node x labeled by $g_{\mathcal{L}(w)}(x)$ (as described in the proof of Theorem 36) is shown in Figure 3.

References

1. Alberto Apostolico and Maxime Crochemore. Fast parallel Lyndon factorization with applications. *Math. Systems Theory*, 28(2):89–108, 1995.
2. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 562–571. SIAM, Philadelphia, PA, 2015.
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. George M. Bergman. Centralizers in free associative algebras. *Trans. Amer. Math. Soc.*, 137:327–344, 1969.
5. Jean Berstel, Aaron Lauve, Christophe Reutenauer, and Franco V. Saliola. *Combinatorics on words*, volume 27 of *CRM Monograph Series*. American Mathematical Society, Providence, RI, 2009.
6. Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Internat. J. Found. Comput. Sci.*, 25(8):1161–1175, 2014.
7. Christian Choffrut and Juhani Karhumäki. Combinatorics of words. In *Handbook of formal languages, Vol. 1*, pages 329–438. Springer, Berlin, 1997.
8. Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. The maximal number of cubic runs in a word. *J. Comput. System Sci.*, 78(6):1828–1836, 2012.
9. Maxime Crochemore and Luís M.S. Russo. Cartesian and Lyndon trees. *Theoretical Computer Science*, 2018. doi:10.1016/j.tcs.2018.08.011.
10. Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *TCS*, 2018. doi:10.1016/j.tcs.2018.12.015.
11. Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
12. Ira M. Gessel, Antonio Restivo, and Christophe Reutenauer. A bijection between words and multisets of necklaces. *European J. Combin.*, 33(7):1537–1546, 2012.

13 Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *J. Combin. Theory Ser. A*, 64(2):189–215, 1993.

14 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Block sorting-based transformations on words: beyond the magic BWT. In *Developments in language theory*, volume 11088 of *Lecture Notes in Comput. Sci.*, pages 1–17. Springer, Cham, 2018.

15 Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theoret. Comput. Sci.*, 307(1):173–178, 2003.

16 Manfred Kufleinter. On bijective variants of the Burrows-Wheeler transform. In *Proceedings of the Prage Stringology Conference 2009*, pages 65–79, 2009.

17 M. Lothaire. *Combinatorics on words*. Cambridge Mathematical Library. Cambridge University Press, Cambridge, 1997.

18 M. Lothaire. *Algebraic combinatorics on words*, volume 90 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2002.

19 Roger C. Lyndon. On Burnside's problem. *Trans. Amer. Math. Soc.*, 77:202–215, 1954.

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

21 Dominique Perrin and Antonio Restivo. Words. In *Handbook of enumerative combinatorics*, Discrete Math. Appl. (Boca Raton), pages 485–539. CRC Press, Boca Raton, FL, 2015.

22 Christophe Reutenauer. Mots de Lyndon généralisés. *Sém. Lothar. Combin.*, 54:Art. B54h, 16, 2005/07.

23 Victor A. Ufnarovskij. Combinatorial and asymptotic methods in algebra. In *Algebra, VI*, volume 57 of *Encyclopaedia Math. Sci.*, pages 1–196. Springer, Berlin, 1995.

24 Gérard Viennot. *Algèbres de Lie libres et monoïdes libres*, volume 691 of *Lecture Notes in Mathematics*. Springer, Berlin, 1978.

Stringology Combats Microbiological Threats

Michal Ziv-Ukelson
Ben Gurion University of the Negev, Israel
https://www.cs.bgu.ac.il/~negevcb/
michaluz@cs.bgu.ac.il

Abstract

A major concern worldwide is the acquisition of antibiotic resistance by pathogenic bacteria. Genomic elements carrying resistance and virulence function can be acquired through horizontal gene transfer, yielding a broad spread of evolutionary successful elements, both within and in between species, with devastating effect. Recent advances in pyrosequencing techniques, combined with global efforts to study microbial adaptation to a wide range of ecological niches (and in particular to life in host tissues that we perceive as pathogenesis), yield huge and rapidly-growing databases of microbial genomes.

This big new data statistically empowers genomic-context based approaches to functional analysis: the idea is that groups of genes that are clustered locally together across many genomes usually express protein products that interact in the same biological pathway, and thus the function of a new, uncharacterized gene can be deciphered based on the previously characterized genes that are co-localized with it in the same gene cluster. Identifying and interpreting microbial gene context in huge genomic data requires efficient string-based data mining algorithms. Additionally, new computational challenges are raised by the need to study the grammar and evolutionary spreading patterns of microbial gene context.

In this talk, we will review some classical combinatorial pattern matching and data mining problems, previously inspired by this application domain. We will re-examine the biological assumptions behind the previously proposed models in light of some new biological observations. We will consider the computational challenges arising in accomodating the new biological observations, and in exploiting them to scale up the algorithmic solutions to the huge new data. Our goal is to inspire interesting new problems that harness Stringology to the study of microbial adaptation and to the fight against microbiological threats ...

2012 ACM Subject Classification Applied computing → Computational biology

Keywords and phrases comparative genomics, syntenic blocks, gene clusters, reconciliation of gene and species trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.3

Category Invited Talk

Optimal Rank and Select Queries on Dictionary-Compressed Text

Nicola Prezza
Department of Computer Science, University of Pisa, Italy
nicola.prezza@di.unipi.it

Abstract

We study the problem of supporting queries on a string S of length n within a space bounded by the size γ of a string attractor for S. In the paper introducing string attractors it was shown that random access on S can be supported in optimal $O(\log(n/\gamma)/\log\log n)$ time within $O(\gamma \text{ polylog } n)$ space. In this paper, we extend this result to *rank* and *select* queries and provide lower bounds matching our upper bounds on alphabets of polylogarithmic size. Our solutions are given in the form of a space-time trade-off that is more general than the one previously known for grammars and that improves existing bounds on LZ77-compressed text by a $\log\log n$ time-factor in *select* queries. We also provide matching lower and upper bounds for *partial sum* and *predecessor* queries within attractor-bounded space, and extend our lower bounds to encompass navigation of dictionary-compressed tree representations.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Cell probe models and lower bounds

Keywords and phrases Rank, Select, Dictionary compression, String Attractors

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.4

Related Version A full version of the paper is available at `https://arxiv.org/abs/1811.01209`.

Funding The author is supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

1 Related Work

Access, rank, and select queries stand at the core of many tasks on compact and compressed data structures, including compressed indexes, graphs, trees, sets of points, etc. Given a string $S[1..n]$ over an integer alphabet $\Sigma = [0, \sigma - 1]$, these queries are defined as:

- $S.access(i)$, $i = 1, \ldots, n$: return the i-th symbol of S.
- $S.rank_c(i)$, $i = 1, \ldots, n$: return the number of occurrences of symbol c in $S[1..i]$.
- $S.select_c(i)$, $i = 1, \ldots, S.rank_c(n)$: return the position of the i-th occurrence of c in S.

While the problem is essentially solved within entropy-compressed space bounds [2], the increasingly growing production of repetitive datasets in fields such as biology, physics, and storage of software web repositories is raising the problem of extending such functionalities (as well as more complex queries such as indexing) to dictionary-compressed representations. This problem has lately received some attention in the literature, and solutions are known for some (but not all) compression schemes. Belazzougui et al. in [5] provided near-optimal bounds on LZ77-compressed text: letting z be the number of LZ77 phrases, they show how to perform access in $O(z \log^{\epsilon} n \log(n/z)/\log\log n)$ space and $O(\log(n/z)/\log\log n)$ time. *rank* and *select* require a σ-factor more space and are supported in $O(\log(n/z)/\log\log n)$ and $O(\log(n/z))$ time, respectively. Ordóñez et al. [23] and Belazzougui at al. [4] solved the problem on grammar-compressed strings in $O(\sigma g)$ space and $O(\log n)$ time for rank and select queries. The latter paper also provides a trade-off using $O(\tau\sigma g \log_{\tau}(n/g))$ words and supporting queries in $O(\log_{\tau}(n/g))$ time, for $2 \leq \tau \leq \log^{\epsilon} n$ and any constant $\epsilon > 0$. For $\tau = \log^{\epsilon} n$, this

solution yields $O(\sigma g \log^{\epsilon} n \log(n/g)/\log\log n)$ space and $O(\log(n/g)/\log\log n)$ query time. No solutions are known for other dictionary compression schemes such as Macro Schemes [24], Collage Systems [17], or the run-length Burrows-Wheeler transform [9].

In this paper, we use the newborn theory of string attractors [15, 16] to provide a universal solution working simultaneously on all known dictionary compressors. We moreover provide the first lower bounds for these queries and describe matching upper bounds on polylogarithmic alphabets. Our solutions are the first for the run-length Burrows-Wheeler transform, Macro Schemes, Run-length SLPs, and Collage Systems. We obtain the full-spectrum trade-off for grammars, generalizing the result of Belazzougui at al. [4] for all $2 \leq \tau \leq n$, and improve the existing bounds on LZ77-compressed text [5] by a $\log\log n$ time-factor in *select* queries.

We note that the reason why Belazzougui et al. [5, 4] cannot reach the optimal time for *select* queries on LZ77 and cannot get the full spectrum $2 \leq \tau \leq n$ on grammars is the same: they need to perform a predecessor query at each level of their structure. In the first case, they use z-fast tries [3], which introduce a $O(\log\log n)$ multiplicative factor in query times. In the second case, they use fusion trees [12], which support predecessor queries in constant time only for $\tau = O(\log^{\epsilon} n)$. We solve this problem, and obtain optimal query times also for *select* queries, by reducing both *rank* and *select* to partial sum queries:

- $S.psum(i) = \sum_{j=1}^{i} S[j],\ i = 1, \ldots, n.$

We show how to support partial sums in optimal time within attractor-bounded space (optimality follows from a straightforward reduction from *access* queries) by generalizing the strategy used by Belazzougui et al. [5] to solve *rank* queries. Importantly, our reductions to *partial sum* preserve asymptotically the attractor size. This solution allows us to avoid performing expensive predecessor queries at each level of our structure, thus obtaining constant time per level on the whole range $2 \leq \tau \leq n$.

On our way, we extend our lower bounds to operations on dictionary-compressed sequences of balanced parentheses, typically used to support navigation on (compressed) trees. Our lower bounds show that existing solutions [7, Lem. 8.2, 8.3] on trees represented using grammar-compressed sequences of balanced parentheses are far from the optimum by a $O(\log\log n)$-factor.

We also note that our lower- and upper- bounds easily extend to the well-studied *predecessor* problem, which asks to find the largest element x not larger than a given y in an opportunely-encoded set $\{x_1, \ldots, x_m\} \subseteq [1, n]$. A classic result from Beame and Fich [1, Cor 3.10] states that, using words of size $\log^{O(1)} n$, no static data structure of size $m^{O(1)}$ can answer predecessor queries in time $o(\sqrt{\log m/\log\log m})$. First, note that a dictionary compressed representation of the sequence $x_1, x_2 - x_1, \ldots, x_m - x_{m-1}$ always takes at most $O(m)$ words of space, and could take much less if the sequence is repetitive. Indeed, we show that Beame and Fich's lower bound can be improved to $\Omega(\log n/\log\log n)$ when the sequence is dictionary-compressed, and provide a data structure matching this lower bound.

1.1 String Attractors

Let S be a string of length n. Informally, a string attractor [16] for S is a set $\Gamma \subseteq [1..n]$ with the following property: any substring of S has at least one occurrence in S crossing at least one position in Γ. The following definition formalizes this concept.

▶ **Definition 1** (String attractor [16]). *A* string attractor *of a string $S \in \Sigma^n$ is a set of positions $\Gamma \subseteq [1..n]$ such that every substring $S[i..j]$ has at least one occurrence $S[i'..j'] = S[i..j]$ with $j'' \in [i'..j']$ for some $j'' \in \Gamma$.*

String attractors were originally introduced as a unifying framework for known dictionary compressors: Straight-Line programs [18] (context-free grammars generating the string), Collage Systems [17], Macro schemes [24] (a set of substring equations having the string as unique solution; this includes Lempel-Ziv 77 [19]), the run-length Burrows-Wheeler transform [9] (a string permutation whose number of equal-letter runs decreases as the string's repetitiveness increases), and the compact directed acyclic word graph [8, 11] (the minimization of the suffix tree). As shown in [16], any of the above compressed representations induces a string attractor of the same asymptotic size, which for most compressors is also a polylogarithmic approximation to the smallest attractor (NP-hard to find [15, 16]). The other way round also holds for a subset of the above compressors: given a string attractor of size γ, one can build a compressed representation of size $O(\gamma \log(n/\gamma))$. These reductions imply that we can design universal compressed data structures (i.e. working on top of any of the above compressors). In particular, it can be shown that optimal-time random access can be supported within $O(\gamma \text{ polylog } n)$ space [16]. Similarly, optimal text indexing can be achieved within the same space [21, 22].

2 Lower Bounds on Dictionary-Compressed Strings

We start by providing lower bounds for *rank, select* and *partial sum* queries on dictionary-compressed text. We also consider *predecessor* queries on sets represented by dictionary-compressed binary strings. Our lower bounds are shown using grammar compression (Straight-Line Programs, SLPs [18]), and therefore automatically extend to any compression scheme more powerful than SLPs. In the following, we only consider grammars whose right-hand side has size two, and define the grammar's size to be the number of nonterminals. Our lower bounds are proved in Yao's cell-probe model [26] (inherited from the lower bounds [25, 16] that we use as building blocks).

We make an important clarification. Our lower bounds will state that certain operations cannot be done in less than $O(\log n/\log\log n)$ time within $O(g \text{ polylog } n)$ space. This, of course, does not mean that the lower bounds hold whenever g is the size of *any* grammar representation of the sequence (otherwise, the lower bounds would hold for $g = \Theta(n)$ since we can always build a grammar of size $g = \Theta(n)$). What the lower bounds state is that there cannot exist a structure that, given *any* grammar G for the sequence, can *always* answer queries faster than $O(\log n/\log\log n)$ time within $O(g \text{ polylog } n)$ space, where g is the size of G. This means that any data structure of that size could answer queries faster than $O(\log n/\log\log n)$ time for *some* grammar G, but not for *all* of them.

Our starting point is the following theorem from Verbin and Yu [25], in the variant revisited by Kempa and Prezza [16, Thm 5.1]:

▶ **Theorem 2** (Verbin and Yu [25]). *Let g be the size of any Straight-Line Program for a string S of length n over a binary alphabet. Any static data structure taking $O(g \text{ polylog } n)$ space cannot answer random access queries on S in $o(\log n/\log\log n)$ time.*

The idea is to show a reduction from *access* queries to *partial sum*, *predecessor*, and *rank* and from *rank* to *select*, while asymptotically preserving the grammar size. For *rank* and *partial sum* the reduction is straightforward. Let S be a binary string. Then, $S.access(i) = S.rank_1(i) - S.rank_1(i-1) = S.psum(i) - S.psum(i-1)$, where $S.rank(0) = S.psum(0) = 0$ for convenience. It follows that also $rank_1$ and *partial sum* queries cannot break the lower bound of Theorem 2 (note that, since this is a lower bound, we can remove the restriction on the alphabet size).

To extend the lower bound to *select* queries, we build a string $\delta(S)$ such that (i) $\delta(S)$ has a SLP of size at most $g+1$, and (ii) *rank* queries on S can be simulated with a constant number of *select* queries on $\delta(S)$. Then, the result follows from the hardness of *rank*.

▶ **Definition 3.** *Let S be a binary string of length n. With $\delta : \{0,1\} \to \{0,1\}^*$ we denote the function defined as $\delta(0) = 1$ and $\delta(1) = 01$. With $\delta(S)$ we denote the string $\delta(S[1])\delta(S[2])\dots\delta(S[n])$.*

▶ **Lemma 4.** *If S has a Straight Line Program of size g, then $\delta(S)$ has a Straight Line Program of size at most $g+1$.*

Proof. It is sufficient to modify the Straight Line Program G for S as follows. First, we create a new nonterminal X expanding to 01 (this is not necessary if such a nonterminal already exists). Then, in the rules of G, we replace each terminal 1 with X and each terminal 0 with 1. It is easy to see that the resulting SLP – of size at most $g+1$ – generates $\delta(S)$. ◀

▶ **Lemma 5.** $S.rank_1(i) = \delta(S).select_1(i) - i$.

Proof. First, note that each character of S generates exactly one bit set in $\delta(S)$. Then, $\delta(S).select_1(i)$ is the position of the last bit of the encoding of $S[i]$ in $\delta(S)$.

Moreover, each bit equal to 1 in S generates exactly one 0-bit in $\delta(S)$, while 0-bits in S do not generate 0-bits in $\delta(S)$. Then, the number of 0's before position t in $\delta(S)$ – which is $t - i = \delta(S).select_1(i) - i$ – corresponds to $S.rank_1(i)$, i.e. our claim. ◀

The above lemmas imply that also $select_1$ queries cannot break the lower bound within grammar-compressed space. Note that our lower bounds can trivially be extended to $rank_0$ and $select_0$ by simply flipping all bits (this operation does not increase the grammar size as it is sufficient to flip the two grammar's terminals).

▶ **Theorem 6.** *Let S be a string of length n, and let g be the size of a Straight-Line Program for S. Then, $\Omega(\log n / \log\log n)$ time is needed to perform partial sum, rank, and select queries on S within $O(g \text{ polylog } n)$ space.*

We now move to the well-studied *predecessor* problem. Let $U \subseteq [1,n]$ be a set of integers of cardinality m. Beame and Fich [1, Cor 3.10] proved that, using words of size $\log^{O(1)} n$, no static data structure of size $m^{O(1)}$ can answer predecessor queries in time $o(\sqrt{\log m / \log\log m})$. Note that the size g of a straight-line program expanding to the distances between elements of U could be much smaller than m; one of the consequences of this increased compression power is that we can improve Beame and Fich's lower bound within space bounded by g:

▶ **Theorem 7.** *Let $U \subseteq [1,n]$ be a set of size m, and let $S \in \{0,1\}^n$ be the bit-string representing U: $S[i] = 1$ iff $i \in U$. Let moreover g be the size of a Straight-Line Program for S. Then, $\Omega(\log n / \log\log n)$ time is needed to perform predecessor queries on U within $O(g \text{ polylog } n)$ space.*

Proof. The proof is a straightforward reduction from *access* queries. Let S be a binary string, and U be the set defined as $i \in U$ iff $S[i] = 1$. Then, for $i < n$, $S[i] = 1$ if and only if the predecessor of $i+1$ in U is i (we take $i+1$ because, by definition, the predecessor of i is the largest j such that $j < i$). The lower bound follows from Theorem 2. ◀

As noted in [16], all the above lower bounds immediately extend to any compression scheme more powerful than grammars (including string attractors).

▶ **Corollary 8.** *Let S be a string of length n, and let α be any of these measures of repetitivity of S: the size γ of a string attractor, the size g of a SLP, the size g_{rl} of a RLSLP, the size c of a collage system, the size z of the LZ77 parse, the size b of a macro scheme. Then, $\Omega(\log n/\log\log n)$ time is needed to perform partial sum, rank, and select queries on S within $O(\alpha \text{ polylog } n)$ space. The same lower bound holds for* predecessor *queries when S is binary and represents a set of integers.*

3 Lower bounds on Dictionary-Compressed Trees

A space-efficient way to represent trees, that also supports fast navigational operations, is to encode their topology with a (binary) string. There are three main ways to do this: LOUDS [14], DFUDS [6], and BP [14]. LOUDS requires just *rank/select* support on a binary string, for which we already provided lower bounds. We now extend our lower bounds to operations on balanced parentheses (DFUDS/BP), which will show the hardness of navigating a dictionary-compressed tree representation. DFUDS and BP require additional primitives to be supported on the underlying string of balanced parentheses:

(1) `excess(`i`)`: the difference between open and closed parentheses before position i (included).
(2) `findopen(`i`)`: the position j of the open parenthesis matching the close parenthesis in position i.
(3) `findclose(`i`)`: the inverse of `findopen`.
(4) `fwd_search(`i,δ`)`: the first position $j > i$ such that `excess(`j`)` $=$ `excess(`i`)`$+\delta$.
(5) `bwd_search(`i,δ`)`: as `fwd_search`, but looking backwards.
(6) `rmq(`i,j`)`/`RMQ(`i,j`)`: minimum/maximum in `excess(`$i..j$`)`.
(7) `rmqi(`i,j`)`/`RMQi(`i,j`)`: leftmost position of a minimum/maximum in `excess(`$i..j$`)`.

For more details, Navarro [20] gives a complete description of the above operations and explains how tree navigation queries can be reduced to them. Alternative ways to compress trees include *Tree Straight-Line Programs* (TSLPs), which are not covered here[1].

We start by showing a reduction from $rank_1$ to *excess*. The reduction is actually not new, as it already appeared in Chan et al. [10, Sec. 5.3]. The new ingredient we introduce is the hardness of performing *excess* on grammar-compressed strings. Given any binary string S, we show how to build a string $\Delta(S)$ of balanced parentheses such that (i) the grammar-compressed representation of $\Delta(S)$ is not much larger than that of S and (ii) $S.rank_1$ can be solved with a constant number of $\Delta(S).excess$ queries. Note that, in the definition below, we add an extra pair of enclosing parentheses in order to make the sequence a tree (otherwise, the transformed string could represent a forest).

▶ **Definition 9** ([10])**.** *Let $\delta(0) =$ `()` and $\delta(1) =$ `((`. When S is a binary string of length n, we define $\Delta(S) =$ `(` $\cdot\ \delta(S[1])\ \cdots\ \delta(S[n])\ \cdot$ `)`$^{k+1}$, where $k = 2 \cdot S.rank_1(n)$.*

▶ **Example 10.** If $S = 00101$, then $\Delta(S) =$ `(()()(((()(()))))).`

Note that $\Delta(S)$ is always balanced: first, we introduce an open parenthesis, then terms $\delta(0)$ are balanced and terms $\delta(1)$ introduce two unbalanced open parentheses each. Those $k = 2 \cdot S.rank_1(n)$ parentheses, plus the first open parenthesis are balanced in the final suffix `)`$^{k+1}$ of $\Delta(S)$.

[1] It is worth to note, however, that the compression ratio of these representations is not as good as that of an SLP for the DFUDS string of small-degree trees [13, Thms 2, 4].

▶ **Lemma 11.** *If $S \in \{0,1\}^n$ has a SLP of size g, then $\Delta(S)$ has a SLP of size $O(g + \log n)$.*

Proof. Let G be a SLP for S. We replace the terminal '0' with a nonterminal expanding to `()`, and the terminal '1' with a nonterminal expanding to `((`. We finally create at most $O(\log k) = O(\log n)$ new nonterminals to generate the final suffix $)^{k+1}$, and add two more rules to concatenate the resulting SLPs to the additional open parenthesis prefixing $\Delta(S)$. ◀

▶ **Lemma 12.** $S.rank_1(i) = (\Delta(S).excess(2i+1) - 1)/2$.

Proof. Assume $S.rank_1(i) = t$. Then, $S.rank_0(i) = i - t$. Since each '0' in S generates one open and one close parenthesis in $\Delta(S)$, and each '1' in S generates two open parentheses in $\Delta(S)$ and taking into account the extra open parenthesis at the beginning of $\Delta(S)$, the number of open parentheses before $\Delta(S)[2i+1]$ is $1 + 2 \cdot t + 1 \cdot (i - t)$. Similarly, the number of close parentheses before $\Delta(S)[2i+1]$ is $i - t$. Then, by definition *excess* is precisely the difference between these two values: $\Delta(S).excess(2i+1) = 1 + 2 \cdot t + 1 \cdot (i-t) - (i-t) = 2 \cdot t + 1 = 2 \cdot S.rank_1(i) + 1$. Our claim follows. ◀

Suppose, by contradiction, that there is a structure supporting $o(\log n'/\log\log n')$-time *excess* queries on a length-n' sequence B within $O(g'\text{polylog n}')$ space, where g' is the size of any SLP compressing B. Then, given any binary string $S \in \{0,1\}^n$ with SLP of size g, we can build $\Delta(S)$ of length $n' = \Theta(n)$, which by Lemma 11 has a SLP of size $g' = O(g + \log n') = O(g + \log n)$. By Lemma 12 we can use our hypothetical *excess* structure to answer $rank_1$ queries on S in $o(\log n'/\log\log n') = o(\log n/\log\log n)$ time and $O(g'\text{polylog n}') = \text{O(g polylog n)}$ space, which by Theorem 8 is a contradiction. This completes our hardness proof for *excess*.

We now reduce $rank_1$ to *findclose*. Given any binary string S, we show how to build a string $\Delta(S)$ of balanced parentheses such that (i) the grammar-compressed representation of $\Delta(S)$ is not much larger than that of S and (ii) $S.rank_1$ can be solved with a constant number of $\Delta(S).findclose$ queries. The solution for *findopen* is symmetric and is not considered here.

▶ **Definition 13.** *Let $\delta(0) =$ `)` and $\delta(1) =$ `())`. When S is a binary string of length n, we define $\Delta(S) = (^n \cdot \delta(S[1]) \cdots \delta(S[n])$.*

Note that $\Delta(S)$ is always balanced: we first open n parentheses, and then each term $\delta(S[i])$ adds an unmatched closed parenthesis.

▶ **Example 14.** If $S = 00101$, then $\Delta(S) =$ `((((())()))())`.

The proof of the following lemma is analogous to that of Lemma 11.

▶ **Lemma 15.** *If $S \in \{0,1\}^n$ has a SLP of size g, then $\Delta(S)$ has a SLP of size $O(g + \log n)$.*

We obtain the following reduction:

▶ **Lemma 16.** $S.rank_1(i) = \big(\Delta(S).findclose(n - i + 1) - n - i\big)/2$.

Proof. The idea behind the proof is the following. To solve rank, we first "jump" on the i-th last open parentheses $\Delta(S)[n-i+1]$ in the prefix of length n of $\Delta(S)$. Then, the corresponding closed parentheses $\Delta(S).findclose(n-i+1)$ is the last parenthesis of $\delta(S[i])$ (note that each character of S generates exactly one locally-unmatched closed parenthesis in $\Delta(S)$, which is matched in the prefix $(^n$). Let $S.rank_1(i) = t$. It follows that before (and including) position $\Delta(S).findclose(n-i+1)$ there are: n parentheses (the prefix $(^n$ of $\Delta(S)$), plus $3t$ parentheses (three for each '1' in $S[1..i]$), plus $i - t$ parentheses (one for each '0' in $S[1..i]$). We conclude that $\Delta(S).findclose(n-i+1) = n + 3t + (i-t) = n + 2t + i$. Our claim follows. ◀

Note that operation *findopen* is symmetric to *findclose*, so its hardness is immediate. Finally, the lower bounds automatically transfer to `fwd_search` and `bwd_search` since they can be used to implement *findopen* and *findclose* (see also [20]): `findclose(`i`)` = `fwd_search(`$i, -1$`)` and `findopen(`i`)` = `bwd_search(`$i, 0$`)`$+1$.

To prove the hardness of range queries, consider the string $\Delta(S)$ of Definition 9. Clearly, the maximum excess in the length-2 string $\delta(0) =$ `()` is reached in the first parenthesis. On the other hand, the maximum excess in $\delta(1) =$ `((` is reached in the second parenthesis. This shows that `RMQi` and `rmqi` can be used to answer *access* queries: the maximum (resp. minimum) excess in the length-2 substring $\Delta(S)[k, k+1]$ corresponding to $\delta(S[i])$ is in position k (resp. $k+1$) if and only if $S[i] = 0$ (resp. 1). Similarly, we can solve `RMQi` (resp. `rmqi`) by issuing two `RMQ` (resp. `rmq`) in the unary ranges $\Delta(S)[k]$ and $\Delta(S)[k+1]$, and taking the position with the maximum (resp. the minimum) excess. We finally obtain our result, stated in the most general form:

▶ **Theorem 17.** *Let S be a balanced parentheses sequence of length n, and let α be any of these measures of repetitivity of S: the size γ of a string attractor, the size g of a SLP, the size g_{rl} of a RLSLP, the size c of a collage system, the size z of the LZ77 parse, the size b of a macro scheme. Then, $\Omega(\log n/\log\log n)$ time is needed to perform operations (1-7) on S within $O(\alpha \text{ polylog } n)$ space.*

4 Upper Bounds

In this section we provide upper bounds matching our lower bounds on known dictionary-compressed string representations.

4.1 Partial Sums

We support *partial sum* queries by generalizing the *rank* solution presented by Belazzougui et al. [5] (designed on block trees) as follows. We divide the text in γ blocks of length n/γ. We call this the *level 0* of our structure. We keep $\log_\tau(n/\gamma)$ further levels: for each attractor position i, at level $j \geq 1$ we store some information about 2τ non-overlapping and equally-spaced substrings of S (we call them blocks) centered on i and whose length exponentially decreases with the level j (i.e. at level $j = 0$ the block length is n/γ, at level 1 it is $n/(\tau\gamma)$, at level 2 it is $n/(\tau^2\gamma)$, and so on). For each block, we store some partial sum information and a pointer to one of its occurrences crossing an attractor position (which exist by definition of attractor). Then, a *partial sum* query is answered by navigating the structure from the first to last level. All details are reported in the following Theorem.

▶ **Theorem 18.** *Let γ be the size of an attractor for a string S of length n over an integer alphabet. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\gamma\log_\tau(n/\gamma))$ supporting `partial sum` queries on S in $O(\log_\tau(n/\gamma))$ time.*

Proof. For simplicity, we assume that γ divides n and that $\log_\tau(n/\gamma)$ is an integer. Our structure is composed of $\log_\tau(n/\gamma)+1$ levels. Level $j \geq 0$ contains a set of blocks (i.e. text substrings), each of length $\ell_j = n/(\gamma\cdot\tau^j)$ and defined as follows. In the first level, number 0, our blocks are the γ contiguous and non-overlapping S-substrings of length n/γ: $B_{0,k} = S[(k-1)\cdot(n/\gamma)+1..k\cdot(n/\gamma)]$, for $k = 1,\ldots,\gamma$. At level $j \geq 1$, blocks are instead centered around attractor elements. Let i be an attractor position. Then, at level $j \geq 1$ we store the 2τ blocks $\overleftarrow{B}_{i,j,k} = S[i-\ell_j\cdot k..i-\ell_j\cdot(k-1)-1]$, for $k = 1,\ldots,\tau$, and

$\overrightarrow{B}_{i,j,k} = S[i+1+\ell_j\cdot(k-1)..i+\ell_j\cdot k]$, for $k=1,\ldots,\tau$ (note: $\overleftarrow{B}_{i,j,k}$ are on the left of attractor position i, blocks $\overrightarrow{B}_{i,j,k}$ are on its right, and none of the blocks intersects i).

Note: clearly, we do not explicitly store the text substrings associated with each block. Each block will store just a constant amount of information (detailed below) that will be used to answer partial sum queries. However, letting B be a block, to simplify notation in the following we will also use the symbol B to indicate the substring represented by the block B. In this sense, $|B|$ will refer to the substring's length. The use will be clear from the context.

Each block B at level $j\geq 0$ stores a pointer to one of its occurrences (as a string) at level $j+1$ crossing an attractor position (at least one occurrence of this kind exists by definition of 1-adjacent attractor). This pointer is simply a pair (i,v), where i is the attractor position and v is such that $S[i-v..i-v+|B|-1]=B$. Crucially, note that the substring $S[i-v..i-v+|B|-1]$ is completely covered by contiguous and non-overlapping blocks of length $|B|/\tau$ at level $j+1$, except possibly position $S[i]$ that is not included in any of those blocks (this will be used to devise a recursive strategy).

In the last level $j=\log_\tau(n/\gamma)$ (where the block size is $\ell_{\log_\tau(n/\gamma)}=1$) we explicitly store in $2\gamma\tau$ words the strings representing the blocks. We also store the character $S[i]$ under each attractor position i. Note that $S[i]$ can be retrieved in constant time from i using, e.g. perfect hashing.

We associate to each block some partial sum information. To simplify notation, let $sum(B)$ denote the sum of all integers in the string B.

- **(a)** For each block B at any level, let (i,v) be the pointer associated with it. Then, we associate to B the value $sum(B[1..v])$.
- **(b)** At level 0, each block $B_{0,k}$ with $k=1,\ldots,\gamma$, stores $sum(B_{0,1}\cdots B_{0,k-1})$, i.e. the sum of all integers preceding the block in the input string (this value is 0 for $B_{0,1}$).
- **(c)** At levels $j\geq 1$, each block B stores $sum(B)$.
- **(d)** Blocks $\overleftarrow{B}_{i,j,k}$ moreover store the partial sum $sum(\overleftarrow{B}_{i,j,k}\cdots\overleftarrow{B}_{i,j,1})$, for $k=1,\ldots,\tau$, while blocks $\overrightarrow{B}_{i,j,k}$ store the partial sum $sum(\overrightarrow{B}_{i,j,1}\cdots\overrightarrow{B}_{i,j,k})$, for $k=1,\ldots,\tau$.

Overall, we store $O(1)$ words per block. It follows that our structure fits in $O(\tau\gamma\log_\tau(n/\gamma))$ words. We now show how to efficiently answer partial sum queries using this information.

To answer $S.psum(v)$ we proceed as follows. Let $k=\lfloor(v-1)/(n/\gamma)\rfloor+1$ and $v'=((v-1) \bmod (n/\gamma))+1$. Then, $S.psum(v)=sum(B_{0,1},\ldots,B_{0,k-1})+B_{0,k}.psum(v')$. The first term $sum(B_{0,1},\ldots,B_{0,k-1})$ is explicitly stored (read point (b) above). To compute $B_{0,k}.psum(v')$, note that this is a block prefix; we now show how to compute the sum of the integers in the prefix of any block at level $j\geq 0$ by reducing the problem to that of computing the sum of the integers in a prefix of a block at level $j+1$. The answer in the last level $j=\log_\tau(n/\gamma)$ (where the block size is $\ell_{\log_\tau(n/\gamma)}=1$) can be obtained in constant time since we explicitly store the integers contained in the blocks.

Let us show how to compute $sum(B[1..t])$ at level $j\geq 0$, for some $t\leq\ell_j$ and some level-j block B. First, we map $B[1..t]$ to level $j+1$ using its pointer (i,v). We distinguish two cases.

- **(1)** If $t>v$, then $sum(B[1..t])=sum(B[1..v])+sum(B[v+1..t])$. The first term, $sum(B[1..v])$ is explicitly stored (read point (a) above). Note that the string appearing in the second term, $B[v+1..t]$, prefixes $S[i]\cdot\overrightarrow{B}_{i,j+1,1}\cdots\overrightarrow{B}_{i,j+1,\tau}$. We can therefore decompose $B[v+1..t]$ into $S[i]$, followed by a (possibly empty) prefix of $d=\lfloor(t-v-1)/\ell_{j+1}\rfloor$ blocks of length ℓ_{j+1} – i.e. the prefix $\overrightarrow{B}_{i,j+1,1},\ldots,\overrightarrow{B}_{i,j+1,d}$ –, followed by a (possibly empty) suffix of length $l=(t-v-1) \bmod \ell_{j+1}$. We can retrieve in constant time the

sum (explicitly stored, see point (d) above) of the integers contained in the prefix of full blocks, as well as the value $S[i]$. As far as the remaining suffix of $B[v+1..t]$ is concerned, note that it coincides with the block prefix $\overrightarrow{B}_{i,j+1,d+1}[1..l]$ and we can thus compute the corresponding partial sum by recursing our strategy.

(2) If $t \leq v$, then $sum(B[1..t]) = sum(B[1..v]) - sum(B[t+1..v])$. The first term, $sum(B[1..v])$ is explicitly stored (read point (a) above). The second term can be computed with a strategy completely symmetric to that described in point (1). Let $d = \lfloor (v-t)/\ell_{j+1} \rfloor$ and $l = (v-t) \bmod \ell_{j+1}$. We decompose $B[t+1..v]$ into the prefix $\overleftarrow{B}_{i,j+1,d+1}[(\ell_{j+1}-l+1)..\ell_{j+1}]$ (note: this is a block suffix) followed by the suffix $\overleftarrow{B}_{i,j+1,d} \cdots \overleftarrow{B}_{i,j+1,1}$. The sum of integers in the suffix of full blocks is explicitly stored (point (d) above), so we are left with the problem of computing the sum in $\overleftarrow{B}_{i,j+1,d+1}[(\ell_{j+1}-l+1)..\ell_{j+1}]$, which is a block suffix. Since we explicitly store $sum(\overleftarrow{B}_{i,j+1,d+1})$ (point (c) above), we can, also in this case, reduce the problem to that of computing the sum in a prefix of a block at level $j+1$ (i.e. the block prefix $\overleftarrow{B}_{i,j+1,d+1}[1..\ell_{j+1}-l]$) with a simple subtraction.

The strategy above described allows us to compute $S.psum(v)$ with a single descent from the first to last level. At each level we spend constant time. It follows that the overall procedure terminates in $O(\log_\tau(n/\gamma))$ time. ◀

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$ our structure takes $O(\gamma \log^\epsilon n \log(n/\gamma)/\log\log n)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This matches the lower bound stated in Theorem 8.

4.2 Rank

Clearly, on binary strings it holds that $S.rank_1(i) = S.psum(i)$ so our problem is already solved in this case by Theorem 18. Given a string S over a generic alphabet of size σ, we can solve $S.rank_c()$ as follows. We build σ bit-strings S_c, one for each alphabet character c, defined as $S_c[i] = 1$ if and only if $S[i] = c$. It is easy to verify that, if S has an attractor Γ, then Γ is an attractor also for S_c (repetitions are preserved). Then, we build our structure of Theorem 18 on each S_c using Γ as attractor and compute $S.rank_c(i) = S_c.rank_1(i)$ (we can associate each c to its structure on S_c in constant time by using perfect hashing). We obtain:

▶ **Theorem 19.** *Let γ be the size of a string attractor for a string S of length n over an alphabet of size σ. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\sigma\gamma\log_\tau(n/\gamma))$ supporting* rank *queries on S in $O(\log_\tau(n/\gamma))$ time.*

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$, our structure takes $O(\sigma\gamma \log^\epsilon n \log(n/\gamma)/\log\log n)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This running time matches the lower bound stated in Theorem 8 when $\sigma \in O(\text{polylog n})$.

4.3 Select

We first consider the binary case and $select_1$ queries. We use a straightforward reduction from $select_1$ to *partial sum* queries that blows up the attractor size only by a constant factor.

We assume for simplicity that our input bit-vector S ends with bit 1. If this is not the case, removing the trailing zeros from S does not change the answer of any $select_1$ query and does not increase the attractor's size. Let $S = 0^{x_1-1}1\ldots0^{x_m-1}1$, with $x_i \geq 1$ for $i = 1,\ldots,m$, and define $S' = x_1 \ldots x_m$.

▶ **Lemma 20.** *If S has an attractor of size γ, then S' has an attractor of size at most $2\gamma+1$.*

Proof. Let Γ be an attractor for S. We build an attractor Γ' for S' as follows. Note that, to build S', we partition S in blocks: each block is formed by a sequence of zeros terminated by a bit set. Given a position $i \in [1, |S|]$, we say that i' is the *corresponding position* of i in S' iff i belongs to the i'-th block. Starting with $\Gamma' = \{1\}$, for every $i \in \Gamma$ we insert in Γ' the positions i' and $i'+1$ (unless they fall outside the range $[1, |S'|]$), where i' is the corresponding position of i in S'. More formally, if $S[i] = 0$ then we insert $S.rank_1(i)+1$ and $S.rank_1(i)+2$ in Γ'. Otherwise, if $S[i] = 1$ then we insert $S.rank_1(i)$ and $S.rank_1(i)+1$ in Γ'. Clearly, $|\Gamma'| \leq 2|\Gamma| + 1 = 2\gamma + 1$.

Consider any substring $S'[i..j]$. We prove that $S'[i..j]$ has an occurrence $S'[i'..j']$ (possibly, $i' = i$ and $j' = j$) such that $S'[i'..j']$ crosses an element of Γ'. This will prove our claim.

If $i = 1$, then $S'[i..j]$ crosses position $1 \in \Gamma'$ and we are done. Otherwise, we focus on sequence $S'[i-1..j] = y_1 y_2 \dots y_{j-i+2}$. Note that this is the sequence of exponents of zeros in a subsequence of S entirely covered by blocks: $S[S.select_1(i-2)+1..S.select_1(j)] = 0^{y_1}1 \dots 0^{y_{j-i+2}}1$, where we take $S.select_1(0) = 0$. By definition of Γ, this substring of S has an occurrence $S[i'', j'']$ crossing an element of Γ. However, note that this occurrence is not necessarily preceded by a bit set. As a consequence, we can only state that the corresponding subsequence of S', i.e. $S'[t..S.rank_1(j'')]$ with $t = S.rank_1(i'')+1$ if $S[i''] = 0$ and $t = S.rank_1(i'')$ otherwise, is such that $S'[t..S.rank_1(j'')] = w y_2 \dots y_{j-i+2}$, with $w \geq y_1$. Since $S[i'', j'']$ crosses an element of Γ then, by the way we defined Γ', either $S'[S.rank_1(j'')]$ or two adjacent characters $S'[k, k+1]$, with $t \leq k < S.rank_1(j'')$, cross an element of Γ'. Then, this means that $S'[t+1..S.rank_1(j'')] = y_2 \dots y_{j-i+2} = S'[i..j]$ crosses an element of Γ'. This concludes our proof. ◀

At this point, the solution for select is immediate: we build the structure of Theorem 18 on the sequence $S' = x_1 \dots x_m$ using the attractor of Lemma 20, and simply note that $S.select_1(i) = S'.psum(i)$.

Given a string S over a generic alphabet of size σ, we can solve $S.select_c()$ as follows. We build σ bit-strings S_c, one for each alphabet character c, defined as $S_c[i] = 1$ if and only if $S[i] = c$. As seen in the previous section, an attractor for S is also an attractor for S_c. We build our structure solving $select_1$ on each S_c and compute $S.select_c(i) = S_c.select_1(i)$.

▶ **Theorem 21.** *Let γ be the size of a string attractor for a string S of length n over an alphabet of size σ. Then, for all $2 \leq \tau \leq n/\gamma$, we can store a data structure of size $O(\tau\sigma\gamma \log_\tau(n/\gamma))$ supporting* select *queries on S in $O(\log_\tau(n/\gamma))$ time.*

For $\tau = \log^\epsilon n$ and any constant $\epsilon > 0$, our structure takes $O\left(\sigma\gamma \log^\epsilon n \log(n/\gamma)/\log\log n\right)$ words of space and answers queries in $O(\log(n/\gamma)/\log\log n)$ time. This running time matches the lower bound stated in Theorem 8 when $\sigma \in O(\text{polylog n})$.

4.4 Predecessor and Tree Navigation

Let $U \subseteq [1, n]$ be a set of size m, and let S be a binary string of length n such that $S[i] = 1$ iff $i \in U$ with attractor of size γ. Using the solutions for *rank* and *select* seen in the previous sections, we can easily support *predecessor* on U in $O(\gamma$ polylog n) space and $O(\log(n/\gamma)/\log\log n)$ time (optimal by Corollary 8). In an extended version of this paper we will show that we can improve this upper bound (both in space and time) on sparse sets, achieving $O(\gamma \log^{1+\epsilon} m/\log\log m) = O(\gamma$ polylog m) space and $O(\log m/\log\log m)$ query time. This solution is analogous to that used in Theorem 18 but, in addition, uses fusion trees to accelerate local predecessor queries.

To conclude, one can obtain fast navigational queries on attractor-compressed trees by combining the SLP-based implementation of balanced-parentheses operations (1-7) (see Section 3) described by Bille et al. [7, Lem. 8.2, 8.3] with the SLP of [16, Thm. 3.14], built on the balanced-parentheses representation of the tree. This immediately yields $O(\log n)$-time navigation within $O(\gamma \log^2(n/\gamma))$ words of space. To reduce this running time to the optimal $O(\log n/\log\log n)$, we note that one could increase the arity of the SLP to $\log^\epsilon n$ as described in [4, Thm. 2]. Space could be further reduced by employing the RLSLP – of size $O(\gamma \log(n/\gamma))$ – described in [21] and adapting the algorithms to work on run-length SLPs. We will cover these improvements in an extended version of this paper.

References

1 Paul Beame and Faith E Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

2 D. Belazzougui and G. Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

3 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with O (1) accesses. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 785–794. SIAM, 2009.

4 Djamal Belazzougui, Patrick Hagge Cording, Simon J Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Algorithms-ESA 2015*, pages 142–154. Springer, 2015.

5 Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Data Compression Conference (DCC), 2015*, pages 83–92. IEEE, 2015.

6 David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

7 Philip Bille, Gad M Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

8 Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

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

10 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):21, 2007.

11 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching (CPM)*, pages 116–129. Springer, 1997.

12 Michael L Fredman and Dan E Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 1–7. ACM, 1990.

13 Moses Ganardi, Danny Hucke, Markus Lohrey, and Eric Noeth. Tree compression using string grammars. *Algorithmica*, 80(3):885–917, 2018.

14 Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

15 Dominik Kempa, Alberto Policriti, Nicola Prezza, and Eva Rotenberg. String Attractors: Verification and Optimization. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

16 Dominik Kempa and Nicola Prezza. At the Roots of Dictionary Compression: String Attractors. In *Annual Symposium on Theory of Computing (STOC)*, pages 827–840. ACM, 2018.

17 T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 298(1):253–272, 2003.

18 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

19 A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Information Theory*, 22(1):75–81, 1976.

20 Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

21 Gonzalo Navarro and Nicola Prezza. Faster Attractor-Based Indexes. *arXiv preprint*, 2018. `arXiv:1811.12779`.

22 Gonzalo Navarro and Nicola Prezza. Universal Compressed Text Indexing. *Theoretical Computer Science*, 2018.

23 Alberto Ordóñez, Gonzalo Navarro, and Nieves R Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017.

24 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

25 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Annual Symposium on Combinatorial Pattern Matching*, pages 247–258. Springer, 2013.

26 Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM (JACM)*, 28(3):615–628, 1981.

A 2-Approximation Algorithm for the Complementary Maximal Strip Recovery Problem

Haitao Jiang
Department of Computer Science and Technology, Shandong University, China
htjiang@sdu.edu.cn

Jiong Guo
Department of Computer Science and Technology, Shandong University, China
jguo@sdu.edu.cn

Daming Zhu
Department of Computer Science and Technology, Shandong University, China
dmzhu@sdu.edu.cn

Binhai Zhu
Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA
bhz@montana.edu

Abstract

The Maximal Strip Recovery problem (MSR) and its complementary (CMSR) are well-studied NP-hard problems in computational genomics. The input of these dual problems are two signed permutations. The goal is to delete some gene markers from both permutations, such that, in the remaining permutations, each gene marker has at least one common neighbor. Equivalently, the resulting permutations could be partitioned into common strips of length at least two. Then MSR is to maximize the number of remaining genes, while the objective of CMSR is to delete the minimum number of gene markers. In this paper, we present a new approximation algorithm for the Complementary Maximal Strip Recovery (CMSR) problem. Our approximation factor is 2, improving the currently best 7/3-approximation algorithm. Although the improvement on the factor is not huge, the analysis is greatly simplified by a compensating method, commonly referred to as the non-oblivious local search technique. In such a method a substitution may not always increase the value of the current solution (it sometimes may even decrease the solution value), though it always improves the value of another function seemingly unrelated to the objective function.

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

Keywords and phrases Maximal strip recovery, complementary maximal strip recovery, computational genomics, approximation algorithm, local search

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.5

Funding This research is supported by National Natural Science Foundation of China, No. 61472222, 61732009, 61761136017, 61628207.

1 Introduction

Maximal Strip Recovery (MSR) is a problem originally proposed to eliminate noise and ambiguities in genomic maps [4, 15]. In comparative genomics, a genetic map (interchangeably, a signed permutation) is represented by a sequence of n distinct gene markers (interchangeably, letters). A gene marker can appear in two different genomic maps, either positively or negatively. A *strip* (or, syntenic block) is a sequence of distinct markers that appears as subsequences in two maps, either directly or in a reversed and negated form.

Given two genetic maps G_1 and G_2 of length n, the problem *Maximal Strip Recovery* (MSR) [4, 15] is to find two subsequences of d strips (each of length at least two), denoted as $G_i^\star$, for $i = 1, 2$, and find two signed permutations π_i of $\langle 1, \ldots, d\rangle$, such that each sequence

$G_i^\star = S_{\pi_i(1)} \dots S_{\pi_i(d)}$ (here S_{-j} denotes the reversed and negated sequence of S_j) is a subsequence of G_i, and the total length of these S_j's is maximized. Intuitively, those gene markers not included in $G_1^\star$ and $G_2^\star$ are noisy and ambiguous markers. The complementary problem of deleting the minimum number of noisy and ambiguous markers to have a feasible solution (i.e., every remaining marker must be in some strip) is called the *Complementary Maximal Strip Recovery*, which will be abbreviated as CMSR. We illustrate an example in Fig. 1. In this example, each integer in G_1 and G_2 represents a gene marker.

$$
\begin{aligned}
G_1 &= \langle 1,2,3,4,5,6,7,8,9,10,11,12,13\rangle \\
G_2 &= \langle -9,-4,-8,-7,-6,1,2,3,-13,-12,-10,-5,-11\rangle \\
S_1 &= \langle 1,2,3\rangle \\
S_2 &= \langle 6,7,8,9\rangle \\
S_3 &= \langle 11,12,13\rangle \\
\pi_1 &= \langle 1,2,3\rangle \\
\pi_2 &= \langle -2,1,-3\rangle \\
G_1^\star &= \langle 1,2,3,6,7,8,9,11,12,13\rangle \\
G_2^\star &= \langle -9,-8,-7,-6,1,2,3,-13,-12,-11\rangle
\end{aligned}
$$

Figure 1 An example for the problem MSR and CMSR. MSR has a solution size of ten (with $d = 3$ strips in $G_1^\star$ and $G_2^\star$; i.e., (1,2,3),(6,7,8,9) and (11,12,13)). CMSR has a solution size of three: the deleted markers are 4,5 and 10.

It was shown in [14] that both (the decision versions of) MSR and CMSR are NP-complete. (Readers are referred to [5] for the basic concepts in algorithms and NP-completeness.) A bit later, MSR was shown to be APX-hard [2, 9] and CMSR was also shown to be APX-hard [10]. For the positive results, in [4, 15], some heuristic approaches based on Max Independent Set and Max Clique were proposed and shown to be effective. In [3], a factor-4 polynomial-time approximation algorithm was proposed for MSR. In [7], a factor-3 polynomial-time approximation algorithm was proposed for CMSR and an $O^*(3^k)$ FPT algorithm, where k is the parameter representing the minimum number of deleted genes, was also presented for CMSR. Currently, the best approximation factor for CMSR is 2.33 [13] and the best FPT algorithmic bound is $O(2.36^k n^2)$ [1]. In 2014, Jiang and Zhu showed that CMSR admits a kernel of size $78k$ [8]. This kernel was improved to $58k$ more recently by Hu *et al.* [6] and then to $42k$ by Li *et al.* [12]. Combined with these kernel bounds, CMSR can be solved in $O(n^2 + 2.36^k k^2)$ time.

In this paper, we devise a new 2-approximation algorithm for the CMSR problem by a non-oblivious local search technique, initially proposed by Khanna *et al.* [11]. During the non-oblivious local search process, a substitution (or local update) may not always increase the value of the current solution, it sometimes makes the value unchanged or even decreased. (But it always improves the value of a function which is seemingly unrelated to the objective function – that is probably how the algorithm is named "non-oblivious".) In other words, some strips with larger priority are preferred even though they cannot lead to a local minimum solution. While the idea is simple, the analysis is very involved. We hope to see more applications of this technique for problems in computational biology.

2 Preliminaries

We first present some formal definitions. Let $G = \pi_1\pi_2\cdots\pi_n$ be a (signed) permutation and let $\bar{G}$ be the reversed version of G, with all elements negated. A *substring* of G is a consecutive segment in G, $\pi_i\pi_{i+1}\cdots\pi_j$, such that $1 \le i < j \le n$ and its length is denoted as $|\pi_i\pi_{i+1}\cdots\pi_j| = j - i + 1$. A *subsequence* of G is a sequence of letters $\pi_{l_1}\pi_{l_2}\cdots\pi_{l_k}$ such that $1 \le l_1 < l_2 \cdots < l_k \le n$.

Let G_1 and G_2 be two input (signed) permutations over the same set of alphabet (letters). (We assume that G_1 is the identity permutation I_n over $[1..n]$ throughout this paper.) A (common) *strip*, $\pi_{l_1}\pi_{l_2}\cdots\pi_{l_j}$ $(2 \le j \le n)$, is a common subsequence between G_1 and G_2 (or $\bar{G}_2$), with at least two letters, and its length is j. An *existing* common strip is a common substring between G_1 and G_2 (or $\bar{G}_2$), again with length at least two. Sometimes we also use $S = [a, b]$ to denote a common strip, with a and b being its ending letters, and $|S| = |[a, b]|$ to denote its length. An *adjacency* is a strip of length two. Hence, a pair of common strips of length l has $l - 1$ adjacencies. In Figure 1, the strip $(1, 2, 3)$ appears positively in G_2 and the strip $(6, 7, 8, 9)$ appears in reversed and negated form in G_2 (or, appears positively in $\bar{G}_2$); moreover, $(1, 2, 3) = [1, 3]$ is an existing common strip with two adjacencies.

The **Complementary Maximal Strip Recovery** (*CMSR*) problem is formally defined as follows: Given two signed permutations G_1 and G_2 over the same set of alphabet of size n, delete the minimum number of letters from G_1 and G_2 to obtain $G_1^\star$ and $G_2^\star$ such that the remaining letters are all in some common strips; moreover, each common strip is a substring in $G_1^\star$ and $G_2^\star$.

Two letters, a and b, form a *candidate adjacency* (a, b), if ab or $-b - a$ is a subsequence of G_1 and G_2. Let $IN_1(a, b)$ and $IN_2(a, b)$ be the set of (interior) letters that appears in between a and b in G_1 and G_2 respectively. If $|IN_1(a, b)|+|IN_2(a, b)| = i$, then (a, b) is called an *i-candidate* adjacency. Obviously, a 0-candidate adjacency is an adjacency from a pair of existing common strips of G_1 and G_2. An i-candidate adjacency, say (a, b), becomes an adjacency whenever the letters in $IN_1(a, b)$ and $IN_2(a, b)$ are deleted. In Figure 1, $(11, 12)$ is initially a 2-candidate adjacency and after the two markers 5 and 10 are deleted it becomes a strip (or a valid adjacency). The main idea of our algorithm is to identify some candidate adjacencies and delete the letters in between them.

3 Algorithm Description

In this section, we show the details of our algorithms. Since candidate adjacencies eventually form the final common strips after deleting the interior letters, the main idea of our algorithm is to identify some candidate adjacencies.

3.1 Preprocessing

Firstly, the algorithm preprocesses the two input permutations to identify some special subsequences, which could be put into the common strip components directly.

▶ **Definition 1** (*r*-candidate subsequence). A subsequence $a_1a_2\cdots a_{r-1}a_r$ of G_1 and G_2 (or $\bar{G}_2$), $r \ge 3$, is an *r-candidate subsequence,* if (a_i, a_{i+1}) is a 1-candidate adjacency, $1 \le i \le r-1$.

A letter c can *left-attach* (resp. *right-attach*) to a common strip $[a, b]$, if (c, a) (resp. (b, c)) is a candidate adjacency, then a new common strip $[c, b]$ (resp. $[a, c]$) is generated, while deleting the letters in $In_1(c, a) \cup In_2(c, a)$ (resp. $In_1(b, c) \cup In_2(b, c)$). Throughout Algorithm 1, let CS_0 be the current set of common strips. Initially, CS_0 is empty.

Algorithm 1 Preprocessing.

1: Put all the existing common strips into CS_0.
2: Compute a maximal set of non-overlapping r-candidate subsequences for $r \geq 3$ and put them in CS_0.
3: **for** each common strip $[a, b] \in CS_0$ **do**
4: **while** there exists a letter c, such that (c, a) or (b, c) is a 1-candidate adjacency **do**
5: Delete the letter in $In_1(c, a) \cup In_2(c, a)$ (or $In_1(b, c) \cup In_2(b, c)$) from G_1 and G_2, delete $[a, b]$ from CS_0, and add $[c, b]$ (or $[a, c]$) to CS_0.
6: **end while**
7: **end for**

Note that keeping existing common strips immediately implies that the factor 2 is the best we could have. For example, $G_1 = uabv \cdot xy$ and $G_2 = -v - u \cdot -yab - x$, keeping ab can only have an approximation factor 2 (as the optimal solution for CMSR is to delete a and b). The reason why we cannot keep 2-candidate subsequences is that they could force the approximation factor to be beyond 2. For instance, $G_1 = uv \cdot xa_1ya_2z$ and $G_2 = -va_1a_2 - u \cdot -z - y - x$, keeping a_1a_2 would give us an approximation factor of 2.5 (as we need to delete 5 letters, while the optimal solution is to delete a_1 and a_2). We now use CS_0 to denote the set of common strips in CS_0 found by the preprocessing Algorithm 1 and will not alter CS_0 henceforth. We then assign $CS \leftarrow CS_0$ and try to improve CS. Next, we show how to obtain more common strips.

3.2 How to generate a common strip

A letter is *matched* if it belongs to a common strip in CS, otherwise it is *unmatched*. A matched letter could have either one or two adjacent letters in a common strip. For a matched letter b, which has exactly one adjacent letter, say a, such that (a, b) is a p-candidate adjacency, then b is of *type-p*. For a matched letter b, which has exactly two adjacent letters, say a and c, such that (a, b) is a p-candidate adjacency and (b, c) is a q-candidate adjacency, then b is of *type*-$\min\{p, q\}$.

The letters in $In_1(a, b) \cup In_2(a, b)$ of a candidate adjacency (a, b) are *caught* in both G_1 and G_2, if a and b do form an adjacency; otherwise they are *released*. There are three ways to generate a common strip in our algorithm.

(I) ***Join***: generate a common strip $[a, b]$ from a p-candidate adjacency (a, b), where $p \leq 3$.

(II) ***Attach***: generate a common strip $[a, c]$ of length 3 by right-attaching a letter c to a common strip $[a, b]$ of length 2, such that (b, c) is a p-candidate adjacency, where $p \leq 2$. Moreover, if (a, b) is a 3-candidate adjacency, then delete a from $[a, c]$ to obtain the strip $[b, c]$, to release the 3 letters in $IN_1(a, b) \cup IN_2(a, b)$. The case that (c, a) is a p-candidate adjacency, where $p \leq 3$, is similar.

(III) ***Split***: generate two common strips $[a, b]$ and $[c, d]$, both of length 2, by right-attaching a letter d to a common strip $[a, b, c]$ of length 3, such that (c, d) is a p-candidate adjacency, where $p \leq 3$. The case that (d, a) is a p-candidate adjacency, where $p \leq 3$, is similar.

▶ **Property 1.** *Each common strip generated by Join, Attach, and Split operations is of length 2 or 3. Moreover, if* $[a, b]$ *is a common strip of length 2, then* (a, b) *is a* p*-candidate adjacency with* $p \leq 3$*; if* $[a, b, c]$ *is a common strip of length 3, then* (a, b) *is a* p*-candidate adjacency and* (b, c) *is* q*-candidate adjacency with* $p, q \leq 2$.

3.3 The Non-oblivious Local Search Algorithm

The main idea of our algorithm is a non-oblivious local search, which is outlined as Algorithm 2. The algorithm improves CS iteratively according to the objective function F. Let b_p be the number of type-p letters in the current solution, where $p \in \{1, 2, 3\}$, Define

$$F = 15b_1 + 5b_2 + b_3.$$

Generally, the algorithm adopts local substitutions, which means substituting exactly one current strip, say S, in CS with some other strips, provided that the value of F could be increased.

Let $CS = \{S_1, S_2, \cdots, S_m\}$ be the current set of common strips, and G'_1 and G'_2 be the current common strip components. For a common strip, say S_j, let S_i and S_k be its proceeding and following common strips in G'_1, and $S_{i'}$ and $S_{k'}$ be its proceeding and following common strips in G'_2 respectively. Define four letters to be $L^1_{S_j}$, $L^2_{S_j}$, $R^1_{S_j}$, $R^2_{S_j}$ as follows. If the number of letters appear in between S_i and S_j is less than 4 in G_1, then $L^1_{S_j}$ is the letter to the immediate right of S_i; otherwise, $L^1_{S_j}$ is the fourth letter to the left of S_j. The other three letters can be defined similarly.

CS could be improved by performing the following three operations iteratively.

1. *0-substitution*: Generate new common strips according to (I), (II) and (III), provided that the value of F can be increased.
2. *1-substitution*: Substitute a common strip S by some other strip, which can be searched from L^1_S to R^1_S in G_1 and from L^2_S to R^2_S in G_2 and according to (I), (II) and (III), provided that the value of F can be increased.
3. *2-substitution*: Substitute two consecutive common strips S_u and S_v in G'_1 (resp. G'_2) by two other strips, which can be searched from $L^1_{S_u}$ to $R^1_{S_v}$ in G_1 and from $L^2_{S_u}$ to $R^2_{S_u}$ as well as $L^2_{S_v}$ to $R^2_{S_v}$ in G_2 according to (I), (II) and (III), provided that the value of F can be increased.

The pseudo-code of our algorithm is shown in **Algorithm 2**.

Algorithm 2 CMSR by Non-oblivious-Local-Search.

```
Call Algorithm 1: Preprocessing.
while (0-substitution, 1-substitution, or 2-substitution can be applied) do
    Apply a 0-substitution, 1-substitution, or 2-substitution to improve CS.
end while
```

▶ **Theorem 2.** *The algorithm* CMSR by Non-oblivious-Local-Search *runs in* $O(n^2)$ *time.*

Proof. First of all, it is easy to see that Algorithm 1 runs in $O(n^2)$ time. It takes at most $O(n)$ time to perform a 0-substitution. A 1-substitution on a common strip S tries to identify all possible $1, 2, 3$-candidate adjacencies formed by letters from L^1_S to R^1_S in G_1 and from L^2_S to R^2_S in G_2. There are at most 21 distinct letters in this range if S is of length 2, and at most 23 distinct letters if S is of length 3. Hence the number of possible $1, 2, 3$-candidate adjacencies is bounded by a constant, and the number of its combination is also bounded by a constant. In this case, a candidate adjacency can be found in constant time. Similar argument holds for a 2-substitution.

The number of common strips in CS is obviously bounded by n. It remains to count the time spent on the While-Loop. Note that the While-Loop runs while CS is updated/refreshed, in other words, each round of the While-Loop implies a feasible substitution, and F must be

increased. Since the value of F increases by at least one each time, and the maximum value of F is bounded by $15n$, the While-Loop runs at most $15n$ times.

Consequently, it takes $O(n)$ time to find/perform a feasible substitution. And this procedure loops $O(n)$ times. Therefore the time complexity of **Algorithm 2** is $O(n^2)$. ◀

4 Performance Analysis

In this section, we show that **Algorithm 2** returns a 2-approximation for the complementary maximum strip recovery problem. To analyze the performance of **Algorithm 2**, we should compare it with the optimal solution. Here, we adopt the amortized analysis method.

Let $G_1 = [\pi_1\pi_2\cdots\pi_n] = I_n$ and $G_2 = [\tilde{\pi}_1\tilde{\pi}_2\cdots\tilde{\pi}_n]$ be two (signed) permutations over $[1..n]$. Let G_1^* and G_2^* be the optimal common strip components, each of which is composed of the set CS^* of common strips. Let OPT be the set of letters which are deleted from both G_1 and G_2 to obtain G_1^* and G_2^* respectively. As aforementioned, CS is the set of common strips computed by **Algorithm 2**, which constitute two feasible common strip components G_1' and G_2'. Let ALG be the set of letters which are deleted to obtain G_1' and G_2'. Without causing confusion, CS and CS^* can also be viewed as the sets of adjacencies, as well as the sets of preserved letters. Then we have, $|OPT| + |CS^*| = |ALG| + |CS| = n$.

Next, we review a property of the optimal solution presented in [7].

▶ **Lemma 3.** *There exists an optimal solution OPT, such that, for every existing common strip S of length 2 or greater, (1) S is either totally contained in OPT, or totally disjoint with OPT; (2) if S is of length 4 or greater, then it is totally disjoint with OPT.*

This lemma and its proof also appear in [7], so we omit the proof here. The idea, following the example right after Algorithm 1, is that while existing length-2 or length-3 common strips could be deleted in any optimal solution, when the existing common strips are of length at least 4 then there is some optimal solution which keeps them. For example, $G_1 = xabcdy \cdot uv$ and $G_2 = -y - x \cdot -vabcd - u$, then the optimal solution for MSR could either be $abcd$ (by keeping the existing length-4 common strip $abcd$, the corresponding CMSR solution is to delete u, v, x and y), or with two strips xy and uv (the corresponding CMSR solution is to delete a, b, c and d).

Now, consider the imaginary permutations whenever we reinsert the letters of $OPT \cap CS$ back into G_1^* and G_2^*. Some common strips could be broken into either shorter blocks or *isolates* (e.g., a single letter in either G_1 or G_2 which does not form adjacencies with its neighbors). For instance, the letters 9 and 11 in Fig. 1 are both isolates.

Without of loss of generality, we only focus on the case when the adjacency (a, b) (and (x, y)) appears positively in G_1 and G_2. The other case when it appears in reversed and negated form in G_2 is similar. But we omit the details.

▶ **Definition 4.** *Let (a, b) be an adjacency of a common strip in CS, where $a = \pi_{i_1} = \tilde{\pi}_{i_2}$ and $b = \pi_{j_1} = \tilde{\pi}_{j_2}$. Let (x, y) be an adjacency of a common strip in CS^*, where $x = \pi_{i_1^*} = \tilde{\pi}_{i_2^*}$ and $y = \pi_{j_1^*} = \tilde{\pi}_{j_2^*}$.*

*When $(a, b) = (x, y)$, we say that (a, b) **fully occupies** (x, y).*

*Let $\sqcap$ be the operation for computing the intersection of two intervals, which could be closed (i.e., in the form $[i, j]$) or open (i.e., in the form (i, j)). If $[i_1, j_1] \sqcap [i_1^*, j_1^*] = [i_1^*, i_1^*] = [j_1, j_1]$ or $[i_2, j_2] \sqcap [i_2^*, j_2^*] = [i_2^*, i_2^*] = [j_2, j_2]$, we say that (a, b) **half-occupies** (x, y) via the letter b, then y belongs to the **occupying set** of (a, b), denoted as $O[(a, b)]$. (The case (a, b) **half-occupies** (x, y) via a can be defined symmetrically, in which x belongs to $O[(a, b)]$.)*

Otherwise, if $(i_1, j_1) \sqcap (i_1^*, j_1^*) \neq \emptyset$ *or* $(i_2, j_2) \sqcap (i_2^*, j_2^*) \neq \emptyset$*, we say that* (a, b) ***breaks*** (x, y)*, then both* x *and* y *belong to the* ***breaking set*** *of* (a, b)*, denoted as* $B[(a, b)]$*.*

▶ **Lemma 5.** *Let* (x, y) *be an adjacency in* CS^**, but not in* CS*. Then* (x, y) *is either broken or half-occupied by some adjacencies in* CS*, or* (x, y) *is a p-candidate adjacency with* $p \geq 4$*.*

Proof. Since otherwise, **Algorithm 2** would keep running as the current solution can be improved. ◀

The ***occupying set*** of a common strip $S = [a, \ldots, b]$ in CS, denoted as $O[S]$, is the set of all letters from the adjacencies in CS^* which are half-occupied by $(a, \bullet)$ via a or by $(\bullet, b)$ via b, i.e., $O[S] = O[(a, \bullet)] \cup O[(\bullet, b)]$.

The ***breaking set*** of a common strip S in CS, denoted by $B[S]$, is the set of all letters from adjacencies in CS^*, which are broken by adjacencies of S, i.e., $B[S] = \cup_{(a,b)\subseteq S} B[(a, b)]$. It follows from the definition that, for each common strip S in CS, $O[S] \cap B[S] = \emptyset$.

The ***auxiliary set*** of a common strip S in CS, denoted by $U[S]$, is the set of all letters such that (1) not in CS, (2) appearing in between adjacencies of CS^*, which are broken or half-occupied by adjacencies of S.

Let $CS^*_{\geq 4}$ be the set of letters from p-adjacencies of CS^* with $p \geq 4$, which are not broken or half-occupied by any common strip of CS. Let $U^*_{\geq 4}$ be the set of letters which appear in between adjacencies formed by letters in $CS^*_{\geq 4}$. Viewing CS and CS^* as set of letters, we have $ALG \cup CS = OPT \cup CS^*$. Consequently

$$ALG - OPT = CS^* - CS = \cup_{S\in CS}(B[S] \cup O[S]) \bigcup CS^*_{\geq 4}.$$

From the definition, we know that the union of the auxiliary sets are letters in $OPT \cap ALG$ which appear in between adjacencies of $CS^* - CS^*_{\geq 4}$, i.e.,

$$OPT \cap ALG \supseteq (\cup_{S\in CS} U[S]) \bigcup U^*_{\geq 4}.$$

We also have,

$$OPT - ALG = CS - CS^* = \cup_{S\in CS}(S - CS^*).$$

The approximation factor can be described as

$$\frac{|ALG|}{|OPT|} = \frac{|ALG - OPT| + |OPT \cap ALG|}{|OPT - ALG| + |OPT \cap ALG|} \leq 2$$

which is equivalent to,

$$\begin{aligned}
\frac{|ALG|}{|OPT|} &\leq \frac{|ALG - OPT| - |OPT \cap ALG|}{|OPT - ALG|} \\
&\leq \frac{|\cup_{S\in CS}(B[S] \cup O[S]) \bigcup CS^*_{\geq 4}| - |(\cup_{S\in CS} U[S]) \bigcup U^*_{\geq 4}|}{|\cup_{S\in CS}(S - CS^*)|} \\
&= \frac{|\cup_{S\in CS}(B[S] \cup O[S])| + |\bigcup CS^*_{\geq 4}| - |(\cup_{S\in CS} U[S]) \bigcup U^*_{\geq 4}|}{|\cup_{S\in CS}(S - CS^*)|} \leq 2 \qquad (1)
\end{aligned}$$

Since each letter in $OPT \cap ALG$ appears in G_1 and G_2 exactly once respectively, we have,

$$|(\cup_{S\in CS} U[S]) \bigcup U^*_{\geq 4}| = \frac{\sum_{S\in CS} |U[S]| + |U^*_{\geq 4}|}{2}.$$

Then, it is sufficient to show that

$$(1) = \frac{|\cup_{S\in CS}(B[S] \cup O[S])| - \frac{\sum_{S\in CS} |U[S]|}{2} + |\bigcup CS^*_{\geq 4}| - \frac{|U^*_{\geq 4}|}{2}}{|\cup_{S\in CS}(S - CS^*)|} \leq 2 \qquad (2)$$

Note that a letter may belong to more than one breaking and half-occupying sets. If we assign a weight $\omega(-)$ to each letter of each set, then (2) can be rewritten as

$$(2) \quad = \quad \frac{\sum_{S\in CS}(\omega(B[S]) + \omega(O[S]) - \frac{|U[S]|}{2}) + |\bigcup CS^*_{\geq 4}| - \frac{|U^*_{\geq 4}|}{2}}{\sum_{S\in CS}|S - CS^*|} \leq 2 \tag{3}$$

▶ **Theorem 6.** *The approximation factor will not be greater than two, if the following two conditions hold:*

1. $|\bigcup CS^*_{\geq 4}| - |U^*_{\geq 4}|/2 \leq 0$, *and*
2. *for each* $S \in CS$, $(\omega(B[S]) + \omega(O[S]) - |U[S]|/2)/|S - CS^*| \leq 2$.

The former condition holds from the following lemma straightforwardly.

▶ **Lemma 7.** *Let* (x, y) *be a* p*-adjacency of* CS^* *with* $p \geq 4$*, which is not broken or half-occupied by any adjacency of* CS*, being not able to return* (x, y) *as an adjacency will not result in an approximation factor greater than 2.*

Proof. Since the optimal solution includes the p letters appearing in between x and y, besides these p letters, the approximated solution also includes x and y. Then we have $2 - p/2 \leq 0$, in light of $p \geq 4$. ◀

It remains to assign weights to the letter of the breaking and half-occupying sets. Note that a letter cannot appear in two half-occupying sets.

Weight Assignment.

- **(I)** For each $S \in CS_0$, each letter of $B[S]$ bears a weight of 1; each letter of $O[S]$ bears a weight of 1 if it does not appear in any other set, and bears a weight of 0 if it also appears in some other breaking sets.
- **(II)** If a letter appears in exactly one breaking set, then this appearance bears a weight of 1; if it appears in two or more breaking sets, each of its appearance bears a weight of $\frac{1}{2}$.
- **(III)** If a letter appears in exactly one half-occupying set, then this appearance bears a weight of 1.
- **(IV)** If a letter appears in the half-occupying set $O[S](S \notin CS_0)$, as well as in the breaking set $B[S']$, then $O[S]$ bears a weight of $-\frac{1}{2}$, and $B[S']$ bears a weight of $3/2$.
- **(V)** If a letter appears in the half-occupying set $O[S](S \notin CS_0)$, as well as two breaking sets $B[S']$ and $B[S'']$, then $O[S]$ bears a weight of $-\frac{1}{2}$, and $B[S']$ bears a weight of 1 and $B[S'']$ bears a weight of $\frac{1}{2}$.

▶ **Lemma 8.** $\sum_{S\in CS}(\omega(B[S]) + \omega(O[S])) \geq |\cup_{S\in CS}(B[S] \cup O[S])|$.

Proof. It can be verified that, under the above weight assignment, each letter of $\cup_{S\in CS}(B[S] \cup O[S])$ has a total weight of at least 1. ◀

Next, we show that condition (2) is satisfied. We say that a common strip $S \in CS$ is *safe*, if $\frac{|B[S]|}{|S-CS^*|} \leq 2$. An algorithm is *safe* if all the common strips generated at the end of the algorithm are safe.

4.1 Algorithm 1 is Safe

We first try to show that all the common strips founded by Algorithm 1 are safe.

▶ **Lemma 9.** *The existing common strips are safe.*

Proof. Assume that S is an existing common strip of G_1 and G_2. If $S \subseteq CS^*$, we are done. If not, according to Lemma 3, it satisfies that $|S - CS^*| = |S| \geq 2$. In this case, reinserting S back into G_1^* and G_2^*, would break at most 2 adjacencies in CS^*, thus $|B(S)| \leq 4$. then we have $|B[S]|/|S - CS^*| \leq 4/2 = 2$. ◀

The following example shows that (keeping) the existing common strips would be safe even for the worst case. Let $G_1 = 1ab2 \cdot 34$ and $G_2 = -2 - 1 \cdot -4ab - 3$. Keeping $S = ab$ would imply deleting the four letters $\{1, 2, 3, 4\}$, while the optimal solution is to delete $\{a, b\}$. Note that in this example, $B[S] = \{1, 2, 3, 4\}$, $S - CS^* = \{a, b\}$, hence $|B[S]| = 4$ and $|S - CS^*| = 2$; moreover,

$$|B[S]|/|S - CS^*| = 2.$$

▶ **Lemma 10.** *Every 3-candidate subsequence is safe.*

Proof. Let $T = abc$ be a 3-candidate subsequence. From **Definition 1**, both (a, b) and (b, c) are 1-candidate adjacencies. Assume that the letter x appears in between a and b in G_1 or G_2, and the letter y appears in between b and c in G_1 or G_2. There are 8 cases according to whether or not a, b, c belong to OPT.

1. $a, b, c \notin OPT$. $[a, b, c]$ becomes a common strip after deleting x and y from both G_1 and G_2, according to Lemma 9, $[a, b, c]$ is safe.
2. $\{a, b, c\} \subseteq OPT$. Reinserting $\{a, b, c]\}$ back into G_1^* and G_2^* would break at most 4 adjacencies, thus $|B([a, b, c])| \leq 6$, and $|B([a, b, c])|/|(\{a, b, c\} - CS^*)| \leq 6/3{=}2$.
3. $\{a, b\} \subseteq OPT$. Reinserting $[a, b]$ back into G_1^* and G_2^* would break at most 2 adjacencies, thus $|B([a, b, c])| \leq 4$, and $|B([a, b, c])|/|(\{a, b, c\} - CS^*)| \leq 4/2{=}2$.
4. The case when $\{b, c\} \subseteq OPT$ is symmetric to the case $\{a, b\} \subseteq OPT$.
5. $a \in OPT$, then $y \in OPT$. Reinserting a back into G_1^* and G_2^* would break at most one adjacency, thus $|B([a, b, c])| \leq 2$, and $|B([a, b, c])|/|(\{a, b, c\} - CS^*)| \leq 2/1{=}2$.
6. The case when $c \in OPT$ is symmetric to the case $a \in OPT$.
7. $b \in OPT$. Reinserting b back into G_1^* and G_2^* would break at most one adjacency (x, y), thus $|B([a, b, c])| \leq 2$, and $|B([a, b, c])|/|(\{a, b, c\} - CS^*)| \leq 2/1{=}2$.
8. $\{a, c\} \subseteq OPT$ but not b, then there exists a p-adjacency $(b, d) \in CS^*$. Since $In_1(b, d) \cup In_2(b, d) \supseteq In_1(b, c) \cup In_2(b, c)| = \{y\}$, we could obtain another optimal solution by replacing c by d in CS^*. ◀

▶ **Lemma 11.** *Generate a common strip $S' = [a, \ldots, b, c]$ by attaching a letter c to a safe common strip $S = [a, \ldots, b]$ via a 1-candidate adjacency (b, c), then S' is safe.*

Proof. Since $S = [a, \ldots, b]$ is safe, we have $\frac{|B(S)|}{|S - CS^*|} \leq 2$. Assume that the letter x appears in between b and c in G_1 or G_2. There are 2 cases according to whether or not $c \in OPT$:

(1) $c \in OPT$. Reinserting c back into G_1^* and G_2^* would break at most one adjacency (x, y), which is not broken by a, b. Thus, $|B(S') - B(S)| \leq 2$. Note that $|S' - CS^*| = |S - CS^*| + 1$, then $\frac{|B(S')|}{|S' - CS^*|} \leq \frac{|B(S)|+2}{|S - CS^*|+1} \leq 2$.

(2) $c \notin OPT$. In this case, either $x \in OPT$ or $x \in B(S)$. In either case, $B(S') = B(S)$ and $S - CS^* = S' - CS^*$. ◀

From Lemma 9,10,11, we know that all the common strips of CS_0 are safe.

▶ **Lemma 12.** *For each* $S \in CS_0$, $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{|S-CS^*|} \leq 2$.

Proof. It suffices to show that $\omega(O[S]) - |U[S]|/2 \leq 0$. Assume that $S = [a, \cdots, b]$, (b, c) is an adjacency in CS^*, which means $c \in O[S]$. By the **Weight Assignment** scheme, if (b, c) is broken by some other adjacencies in CS, then c bears a weight of 0 in $O[S]$. If (b, c) is not broken by any other adjacency of CS, then c bears a weight of 1 in $O[S]$. According **Algorithm 1**, (b, c) is a p-candidate adjacency with $p \geq 2$, and these p letters are all in $U[S]$. Thus, $1 - p/2 \leq 0$. ◀

4.2 Algorithm 2 is Safe

Aside from the common strips in CS_0, we now focus on the other common strips in CS which are of length either 2 or 3. If it is of length 2 and it is also an i-adjacency, we say that it is an *i-common strip*; if its length is 3 and it has two consecutive adjacencies: an i-adjacency and a j-adjacency, we say that it is an *$i \bowtie j$-common strip*. From Property 1, there are five types of common strips: 1-common strip, 2-common strip, 3-common strip, $1 \bowtie 2$-common strip, $2 \bowtie 2$-common strip.

Before showing that the common strips found by **Algorithm 2** fulfills condition 2, we first show some properties. The key idea is that, when **Algorithm 2** terminates, the value of F will not be increased by applying more 0,1,2-substitutions.

▶ **Lemma 13.** *At the termination of **Algorithm 2**, a 1-candidate adjacency will either become an adjacency in CS or be broken by some 1-adjacency in CS.*

Proof. Firstly, a 1-adjacency cannot be half-occupied by a 1-adjacency in CS. The reason is that, in this case, the two 1-adjacencies form a 3-candidate subsequence, which would have been handled by **Algorithm 1**.

Then we show that a 1-adjacency cannot only be half-occupied by adjacencies in CS. Assume that $S = [a, \cdots, b] \in CS$ and (b, c) is a 1-adjacency. Obviously, $S \notin CS_0$, thus $|S| = 2$ or $|S| = 3$, and $(\bullet, b)$ is a p-adjacency with $p \geq 2$. If $|S|$=2, then left-attaching c to S will increase the value of F definitely, if $|S| = 3$, then substituting S with $[a, \cdots]$ and $[b, c]$ will also increase the value of F. In case that $S' = [c, \cdots, d] \in CS$, then $|S'| = 2$ or $|S'| = 3$, and $(c, \bullet)$ is a q-adjacency with $q \geq 2$. keeping the common strip $[b, c]$ will increase the value of F.

If a 1-adjacency (b, c) is half-occupied by some adjacency A in CS and is also broken by another t-adjacencies A' of CS, then for $t \geq 2$, keeping the common strips $[b, c]$ will increase the value of F by at least $15 - 2 \times 5 = 5$. ◀

From Lemma 13 and the **Weight Assignment** scheme, if an adjacency is multiply broken, then each common strip breaking it bears a weight of 1 from it. If an adjacency is singly broken by $S \in CS$ and is also half-occupied by $S' \in CS$, then the common strip breaking it bears a weight of 3/2 from it, and the common strip half-occupying it bears a weight of $-1/2$ from it; moreover, this adjacency must be a p-adjacency with $p \geq 2$, which means that there could be a letter in $U[S]$.

▶ **Lemma 14.** *At the termination of **Algorithm 2**, a 2-adjacency will (1) either become an adjacency of CS, (2) or be broken by some adjacencies of CS, (3) or be half-occupied by two adjacencies of CS, (4) or be half-occupied by only one 1-adjacency of CS, which is in a common strip of CS_0 or a $1 \bowtie 2$-common strip.*

Proof. From Lemma 5, a 2-adjacency will either become an adjacency of CS or be broken or be half-occupied. Note that an adjacency can be half-occupied at most twice, then it is

sufficient to show that (4) holds. The reason is that if a 2-adjacency is only half-occupied by a 1-common strip, 2-common strip, 3-common strip, or a 2-adjacency in either a $1 \bowtie 2$-common strip or a $2 \bowtie 2$-common strip, then value of F will be increased by keeping it. ◀

▶ **Lemma 15.** *At the termination of **Algorithm 2**, if a p-adjacency in CS^* is broken by a q-adjacency in CS, where $q \leq 3$, then either $p \geq q$ or it is also broken or half-occupied by some other adjacencies in CS.*

Proof. Since otherwise, **Algorithm 2** will keep running as the current solution can be improved. ◀

▶ **Lemma 16.** *A p-common strip $S = [a, b]$, where $p = 1, 2, 3$, guarantees that* $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{|S-CS^*|} \leq 2$.

Proof. There are three cases: $|S - CS^*| = 2$, $|S - CS^*| = 1$, $|S - CS^*| = 0$.

(1) $|S - CS^*| = 2$. $|B[S]| \leq p + 4$ and $|O[S]| = 0$.

- **(1.1)** $p = 1$, there are $p + 2 = 3$ adjacencies broken by S. Since S is a common strip at the termination of **Algorithm 2** (i.e., no more local improvement is possible), at most one of these three adjacencies is a 1-candidate adjacency, which is singly broken by S. The other adjacencies must either be broken or half-occupied by some other adjacency in CS, or be p-candidate adjacencies with $p \geq 4$. For each of them, if it is half-occupied, $|B[S]| + |O[S]| \leq 4$. According to the **Weight Assignment** scheme, $B[S]$ bears a weight of $3/2$, which means some other adjacency in CS bears a weight of $-1/2$. Moreover, if it is broken, $B[S]$ bears a weight of 1; if it is a p-candidate adjacency with $p \geq 4$, then three letters are added to $U[S]$. In the worst case, we have $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{2} \leq \frac{p+4-p}{2} = 2$.
- **(1.2)** $p = 2$, there are at least $p + 1 = 3$ candidate adjacencies broken by S. As no local improvement is available, at most one of them could be a 2-candidate adjacency, which is singly broken by S. The others must either be broken or half-occupied by other adjacencies in CS, or be p-candidate adjacencies with $p \geq 4$. By an argument similar to (1.1), in the worst case, we have $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{2} \leq \frac{p+4-p}{2} = 2$.
- **(1.3)** $p = 3$, there are at least $p + 2 = 5$ candidate adjacencies broken by S. Similar to the above argument, as no local improvement is available, at least $p + 1$ candidate adjacencies must either be broken or half-occupied by other adjacencies in CS, or be p-candidate adjacencies with $p \geq 4$. Similar to the previous arguments, we have $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{2} \leq \frac{p+4-(p+1)}{2} \leq 2$.

(2) $|S - CS^*| = 1$. $|B[S]| \leq p + 2$ and $|O[S]| = 1$. In this case, according to the **Weight Assignment** scheme, $O[S]$ bears a weight of $-1/2$.

- **(2.1)** $p = 1$, then we are done, since $|B[S]| \leq 2$.
- **(2.2)** $p = 2$, there are two candidate adjacencies broken by S. As local improvement is not possible, at most one of them could be a 2-candidate adjacency. The 2-candidate adjacency bears a weight 2, while the other one bears a weight 1; moreover, the 2-candidate adjacency also implies that a letter is in $U[S]$. So we have, $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{2} \leq \frac{p+2-1/2-1-1/2}{1} = 2$.
- **(2.3)** $p = 3$, there are three candidate adjacencies broken by S. As local improvement cannot be performed further, at most one of them could be a 3-candidate adjacency. The 3-candidate adjacency bears a weight 2, while the other one bears a weight 1; moreover, the 3-candidate adjacency also implies that two letters are in $U[S]$. So we have, $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{2} \leq \frac{p+2-1/2-1-1-1/2}{1} = 2$.

(3) $|S - CS^*| = 0$. $|B[S]| \leq p$ and $|O[S]| = 2$. In this case, according to the **Weight Assignment** procedure, $O[S]$ bears a weight of -1.

(3.1) $p = 1$, then we are done, since $\omega(B[S]) + \omega(O[S]) \leq 0$.

(3.2) $p = 2$, there is one candidate adjacency broken by S, which could be multiply broken or be a p-candidate adjacency with $p \geq 2$. The former means that $B[S]$ bears a weight of 1, the latter means that there are two letters in $U[S]$. So we have, $\omega(B[S]) + \omega(O[S]) - |U[S]|/2 \leq 0$.

(3.3) $p = 3$, if there is one candidate adjacency broken by S, then it becomes the case (3.2). If there are two candidate adjacencies composed of three letters broken by S, then they could be multiply broken or be a p-candidate adjacency with $p \geq 3$. For each of them, if it is also broken by some common strips of CS_0, then $B[S]$ bears a weight of 0, and we are done; if it is a p-candidate adjacency with $p \geq 3$, then there are three letters in $U[S]$. If both of them are also broken by common strips not in CS_0, both adjacencies are not 1-candidate adjacencies (since otherwise, they become a 3-candidate subsequence). Thus, a new letter appears in $U[S]$. So we have, $\omega(B[S]) + \omega(O[S]) - |U[S]|/2 \leq 0$. ◀

▶ **Lemma 17.** *A $p \bowtie q$-common strip $S = [a, b, c]$, where $p = 1, 2$ and $q = 2$, guarantees that* $\frac{\omega(B[S])+\omega(O[S])-|U[S]|/2}{|S-CS^*|} \leq 2$.

Proof. Similar to Lemma 16, hence the details are omitted. ◀

We summarize the main result of this paper as follows.

▶ **Theorem 18.** *The algorithm* CMSR by Nonoblivious-Local-Search *approximates CMSR with a factor of 2, and it runs in $O(n^2)$ time.*

5 Concluding Remarks

We show a non-trivial application of non-oblivious local search for the CMSR problem. The local update step does not always increase the objective function. The difficulty has been assigning different weights for some potential common strips. We hope that this technique might be useful to other optimization problems in computational biology.

References

1 Laurent Bulteau, Guillaume Fertin, Minghui Jiang, and Irena Rusu. Tractability and approximability of maximal strip recovery. *Theoretical Computer Science*, 440-441:14–28, 2012. doi:10.1016/j.tcs.2012.04.034.

2 Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Maximal Strip Recovery Problem with Gaps: Hardness and Approximation Algorithms. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 710–719, 2009. doi:10.1007/978-3-642-10631-6_72.

3 Zhixiang Chen, Bin Fu, Minghui Jiang, and Binhai Zhu. On recovering syntenic blocks from comparative maps. *Journal of Combinatorial Optimization*, 18(3):307–318, 2009. doi:10.1007/s10878-009-9233-x.

4 Vicky Choi, Chunfang Zheng, Qian Zhu, and David Sankoff. Algorithms for the Extraction of Synteny Blocks from Comparative Maps. In *Algorithms in Bioinformatics, 7th International Workshop, WABI 2007, Philadelphia, PA, USA, September 8-9, 2007, Proceedings*, pages 277–288, 2007. doi:10.1007/978-3-540-74126-8_26.

5 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

6 Shuai Hu, Wenjun Li, and Jianxin Wang. An Improved Kernel for the Complementary Maximal Strip Recovery Problem. In *Computing and Combinatorics - 21st International Conference, COCOON 2015, Beijing, China, August 4-6, 2015, Proceedings*, pages 601–608, 2015. doi:10.1007/978-3-319-21398-9_47.

7 Haitao Jiang, Zhong Li, Guohui Lin, Lusheng Wang, and Binhai Zhu. Exact and approximation algorithms for the complementary maximal strip recovery problem. *Journal of Combinatorial Optimization*, 23(4):493–506, 2012. doi:10.1007/s10878-010-9366-y.

8 Haitao Jiang and Binhai Zhu. A linear kernel for the complementary maximal strip recovery problem. *Journal of Computer and System Sciences*, 80(7):1350–1358, 2014. doi:10.1016/j.jcss.2014.03.005.

9 Minghui Jiang. Inapproximability of Maximal Strip Recovery. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, pages 616–625, 2009. doi:10.1007/978-3-642-10631-6_63.

10 Minghui Jiang. Inapproximability of Maximal Strip Recovery: II. In *Frontiers in Algorithmics, 4th International Workshop, FAW 2010, Wuhan, China, August 11-13, 2010. Proceedings*, pages 53–64, 2010. doi:10.1007/978-3-642-14553-7_8.

11 Sanjeev Khanna, Rajeev Motwani, Madhu Sudan, and Umesh V. Vazirani. On Syntactic versus Computational Views of Approximability. *SIAM Journal on Computing*, 28(1):164–191, 1998. doi:10.1137/S0097539795286612.

12 Wenjun Li, Haiyan Liu, Jianxin Wang, Lingyun Xiang, and Yongjie Yang. A 42k Kernel for the Complementary Maximal Strip Recovery Problem. In *Frontiers in Algorithmics - 11th International Workshop, FAW 2017, Chengdu, China, June 23-25, 2017, Proceedings*, pages 175–186, 2017. doi:10.1007/978-3-319-59605-1_16.

13 Guohui Lin, Randy Goebel, Zhong Li, and Lusheng Wang. An improved approximation algorithm for the complementary maximal strip recovery problem. *Journal of Computer and System Sciences*, 78(3):720–730, 2012. doi:10.1016/j.jcss.2011.10.014.

14 Lusheng Wang and Binhai Zhu. On the Tractability of Maximal Strip Recovery. *Journal of Computational Biology*, 17(7):907–914, 2010. (Correction: 18(1):129, Jan, 2011). doi:10.1089/cmb.2009.0084.

15 Chunfang Zheng, Qian Zhu, and David Sankoff. Removing Noise and Ambiguities from Comparative Maps in Rearrangement Analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(4):515–522, 2007. doi:10.1145/1322075.1322077.

Sufficient Conditions for Efficient Indexing Under Different Matchings

Amihood Amir
Department of Computer Science, Bar-Ilan University, Israel
https://u.cs.biu.ac.il/~amir
amir@esc.biu.ac.il

Eitan Kondratovsky
Department of Computer Science, Bar-Ilan University, Israel
kondrae@cs.biu.ac.il

Abstract

The most important task derived from the massive digital data accumulation in the world, is *efficient access* to this data, hence the importance of indexing. In the last decade, many different types of *matching relations* were defined, each requiring an efficient indexing scheme. Cole and Hariharan in a ground breaking paper [Cole and Hariharan, SIAM J. Comput., 33(1):26–42, 2003], formulate *sufficient conditions* for building an efficient indexing for *quasi-suffix collections*, collections that behave as suffixes. It was shown that known matchings, including parameterized, 2-D array and order preserving matchings, fit their indexing settings. In this paper, we formulate more basic sufficient conditions based on the *order relation* derived from the *matching relation* itself, our conditions are more general than the previously known conditions.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases off-the-shelf indexing algorithms, general matching relations, weaker sufficient conditions for indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.6

Funding *Amihood Amir*: Partly supported by ISF grant 1475/18 and BSF grant 2014028.
Eitan Kondratovsky: Partly supported by ISF grant 1475/18.

1 Introduction

Having grown accustomed to the presence of all the web data at our fingertips, it is easy to forget that a mere 25 years ago, an encyclopedia on a CD was the height of information technology. At the root of efficient access of this staggering amount of data is *indexing*. The concept of indexing is that one spends time and effort preprocessing the entire data and constructing auxiliary data structures that will make it possible to efficiently answer future queries of the form: "does input instance I appear in our data?".

The term "appear" is application dependent. It may be an exact matching of a word in a text, it may be searching for a picture of a face in photographs, it may be seeking a gene in the DNA sequence, looking for a tune in a music database, or many other needs in various areas.

The simplest and earliest form of the problem is indexing texts for exact matching. Two strings are said to be matched if and only if they are exactly the same string. This problem has been heavily researched over the last decades and led to the development of two important data structures – the *suffix tree* and the *suffix array*.

The suffix tree is a compacted trie of all the suffixes of T. Its size is $O(n)$ and, starting with Weiner's algorithm [28], various efficient construction algorithms with numerous properties have been proposed (e.g. [23, 26, 11, 4, 18]).

The suffix array [22, 14] is the lexicographically sorted array of suffixes.

The next step was considering more complex matching relations. Two such relations are *parameterized matching* [5, 6, 3] and *order-preserving matching* [17, 19, 16].

Parameterized matching was introduced by Baker [5, 6]. Her main motivation lay in software maintenance, where program fragments are to be considered "identical" even if variable names are different. Baker developed an algorithm to build parameterized suffix trees. Her algorithm essentially takes $O(n(|\Pi| + \log(|\Sigma| + |\Pi|)))$ preprocessing time and $O(m + occ)$ subsequent query time, where Σ and Π are alphabets of fixed symbols and parameter symbols respectively, m is the query's length and occ is the number of pattern occurrences in the text.

Order Preserving matching has been introduced independently by Kubica et al. [19] and by Kim et al. [17]. Crochemore et al. [10] developed a direct suffix tree construction for the order preserving matching, that works in $O(n\sqrt{\log n})$ preprocessing time and $O(m)$ query time if the pattern alphabet is of size $poly(m)$, where m is the pattern length. Otherwise, it can be made by sorting, i.e. in $O(m\sqrt{\log\log m})$ randomized time or in $O(m \log\log m)$ deterministic time. Reporting the pattern occurrences requires adding an extra $O(occ)$ time.

Another matching relation whose indexing problem has been rigorously researched lately is the *histogram matching* (also called *jumbled matching*, *Parikh matching*, or *permuted matching*). In that problem a text T is to be preprocessed. In subsequent queries, a pattern P is given. All locations of T where a permutation of P occurs are sought. Amir et al. [2] showed that this problem is unlikely to have an efficient indexing algorithm. By *efficient indexing*, we mean that preprocessing the text is done in $O(n^{2-\epsilon})$ time for some $\epsilon > 0$, which then allowing for subsequent queries in time $\tilde{O}(m + occ)$, where m is the length of P and occ is the number of locations in T that match P. Recently, the *block mass* [24, 1] problem was considered. It turns out to have an efficient indexing algorithm for constant-size alphabets, but unlikely to have one for unbounded alphabets.

The history of indexing suggests a major difficulty: every matching relation required a *new, ad-hoc* consideration for its indexing version. The desired situation is one where: (1) There are necessary and sufficient conditions on the matching relation for having an efficient indexing algorithm, and (2) when the conditions are sufficient, to have a generic indexing algorithm, so that there will be no need to propose a data structure for every new relation. Finding necessary conditions for efficient indexing is, perhaps, one of the biggest challenges in the area, when the quest is for a generic indexing algorithm.

Cole and Hariharan [9] presented the first such algorithm. They showed that if the suffixes satisfy some conditions, then an index can be built efficiently. In Sect. 3, we present these conditions in more details.

Both parameterized matching [9] and order preserving matching [10] satisfy these conditions, thus Cole and Hariharan made an important step on the way to generalizing indexing algorithms. They showed a randomized algorithm that can construct the suffix tree in linear time, for alphabets of polynomial size. In addition to being *randomized*, their algorithm is based on McCreight's suffix tree construction algorithm [23] and thus, is *not online*. The online situation was solved by Lee et al. [21]. In this paper, we go a step further. Our contributions are:

1. There is a more natural way of characterizing whether indexing is possible, is by considering the **matching relation**. Thus we present a set of sufficient conditions on the matching relation that, if satisfied, allow efficient indexing. Consequently, when presented with a relation, all that is necessary is to check whether that relation satisfies these conditions. We suggest sufficient conditions that are more natural and less strict than the Cole and Hariharan conditions. We then show a reduction from Cole and Hariharan conditions to our conditions. Moreover, we show a matching relation that satisfies our conditions but not the Cole and Hariharan conditions.

2. We present a very simple generic **online, deterministic** indexing algorithm whose worst case times are $\tilde{O}(n)$ time for preprocessing, and $\tilde{O}(m + occ)$ time for query, where occ is the number of pattern occurrences in the text, and $\tilde{O}$ stands for the complexity up to polynomial logarithmic factor, i.e. $\log^c n$ for some constant c.
3. Our generic scheme allows for an easy algorithm for indexing a new matching which we call *jump forward* for string matching. This will be seen in detail in Sect. 4.3. This matching does not satisfy the Cole and Hariharan conditions.

The novelty of our algorithm is using the matching relation properties rather than using the specific matching to enable us to construct the indexing. These properties allow us to *sort suffixes efficiently*. Our algorithm is slower by a polylog factor than the specialized indexing algorithms for the known *non exact* matching relations, e.g. order preserving matching or parameterized matching. However, we were not driven by the goal of improving the indexing complexity of a given matching relation. Our interest is in presenting a small number of basic sufficient conditions that allows the construction of a **generic** algorithm that can be used as an **off-the-shelf** efficient indexing algorithm (up to polylog factors) for **any** matching relation that satisfies the conditions. An example of the effectiveness of our algorithm was recently demonstrated. Some colleagues were discussing indexing a new matching relation – *Cartesian Tree Matching* [25]. A quick check of the matching relation properties convinced us that an efficient indexing exists. What remained was only shaving some powers off the polylogarithmic factor.

2 Preliminaries

Let Σ be an alphabet. A *string* T over Σ is a finite sequence of letters from Σ. By $T[i]$, for $1 \leq i \leq |T|$, we denote the i^{th} letter of T. The *empty string* is denoted by ϵ. By $T[i..j]$ we denote the string $T[i] \dots T[j]$ called a *substring* of T (if $i > j$, then the substring is an empty string). A substring is called a *prefix* if $i = 1$ and a *suffix* if $j = |T|$. The prefix of length j is denoted by $T[..j]$. While by $S_i = T[i..]$ we denote the suffix which starts from index i in T. By $lcp(S, S')$ we denote the *longest common prefix (lcp)* of S and S'. In case S and S' are suffixes S_i and S_j, respectively, we denote $lcp(i, j) = lcp(S_i, S_j)$ for short. By T^R we denote the reversal (the mirror image) of T.

A *suffix tree* of a string T, denoted $ST(T)$, is a compacted trie containing all the suffixes of $T\$$, where $\$$ is a unique character not occurring in Σ. In a compacted trie we define the *depth* of a node to be its number of explicit ancestors, and the *string depth* to be the length of the string it represents. The string of a node x, denoted by $str(x)$, represents the result of reading the letters in path between trie's root and x. In a suffix tree we define the *suffix link* of a node representing the string aS to be a pointer to the node representing S, where S is a string and a is a character. Every explicit node v stores such a link $sl(v)$.

A matching $M \subseteq \Sigma^* \times \Sigma^*$ is a binary relation between strings, where Σ^l is the set of all l-length strings and $\Sigma^* = \bigcup_{l=0}^{\infty} \Sigma^l$. Two strings A and B are said to be matched if and only if $(A, B) \in M$. We consider matchings in which two matched strings are of the same length.

▶ **Example 1** (Exact Matching). Two strings are matched, if they are exactly the same string. Formally, $M_E = \{(A, A) \mid \forall A \in \Sigma^*\}$.

An order relation $R \subseteq \Sigma^* \times \Sigma^*$ is a binary relation between strings. String A is said to be less or equal to string B if and only if $(A, B) \in R$, ARB in short.

▶ **Definition 2** (Total Order for M). *Let R be an order relation, and M a matching. $R \subseteq \Sigma^* \times \Sigma^*$ is* total order for M *if the following properties hold.*

- Transitivity: $\forall A, B, C \quad ARB \wedge BRC \Rightarrow ARC$
- Antisymmetry: $\forall A, B \quad ARB \wedge BRA \iff (A, B) \in M$
- Completeness: $\forall A, B \quad ARB \vee BRA$

An order relation R is said to be *consistent with prefixes* (*consistent with suffixes*) if for all A, B such that ARB then $\forall\ l = 1, \ldots, \min\{|A|, |B|\} \quad A[..l]\ R\ B[..l]$ ($A[l..]\ R\ B[l..]$).

▶ **Definition 3** (Lexicographic Order). *Given two different strings $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_m$, the first one is smaller than the second if $a_i < b_i$ for the first i where a_i and b_i differ, otherwise if no such i exists (happens when one of the strings is a prefix of the other) the shorter string is the smaller. The lexicographic order is a total order for M_E and consistent with prefixes.*

▶ **Definition 4** (Quasi-Suffix Collections (QSC) [9]). *An ordered collection of strings $s_1, s_2, ..., s_n$ is called a quasi-suffix collection if and only if the following conditions hold.*

1. $|s_1| = n, |s_2| = n - 1, \cdots, |s_n| = 1$.
2. *No s_i is prefix of another s_j.*
3. *If $lcp(s_i, s_j) = l > 0$ then $lcp(s_{i+1}, s_{j+1}) \geq l - 1$.*

▶ **Definition 5** (Character Oracles [9]). *A data structure which supplies the j^{th} character of the i^{th} string of the quasi-suffix collection, in $O(1)$ time, is called character oracle. Note that the total length of the explicit strings for a quasi-suffix collection of n strings is $O(n^2)$. While character oracle allows holding the collection in smaller space and reading it on demand. Cole and Hariharan reduce the amount of character oracle calls to be linear during the suffix tree construction.*

Our main contributions are the following.

▶ **Theorem 6** (Order Relation based Sufficient Conditions). *Let M be a matching. There is an efficient indexing construction for the matching if there exists an order relation R and the following conditions hold.*

- *R is a* total order for *M and* consistent with prefixes.
- Order derivation: *given $lcp(i, j)$, compute the order between suffix i and j in $\tilde{O}(1)$ time.*
- Lcp derivation: *assuming oracle access to $lcp(i', j')$ for $i', j' > 1$ compute $lcp(1, i)$ where i', j' and i are indexes in the text in $\tilde{O}(1)$ time.*
- Efficient extension by one character: *maintaining lcp derivation and order derivation, to support text extension by one new letter at the beginning. This operation takes amortized $\tilde{O}(1)$ time per extension.*
- Efficient extension by a pattern: *extending the lcp derivation and order derivation, to support lcp derivation of $lcp(P, S_i)$ and order between the pattern and a suffix, takes overall $\tilde{O}(|P|)$ time and space.*

$lcp(i, j)$ is well defined when R is *consistent with prefixes*, because then M is hereditary from right, i.e. dropping same length suffixes of matched strings leads to a matched pair of strings. That is, the prefixes of S_i and S_j from length 1 to length $lcp(i, j)$ are equal. And $lcp(i, j)$ is the length of the longest common prefix.

▶ **Theorem 7.** (The Reduction) *Let M be a matching. Assume a character oracle f such that for every string S, $\{s_i\}_{i=1}^n = \{f(S_i)\}_{i=1}^n$ is a quasi-suffix collection. Then the conditions of Theorem 6 hold for M.*

3 Previous Results

Cole and Hariharan paper [9] guarantees the following. When one wants to index a string T under some matching M, a quasi-suffix collection is required. A compacted trie is then constructed for the supplied collection in $O(n)$ time and space with a failure probability close to inverse exponential. Later, this construction was generalized to be online using the same time and space complexities by Lee et al. [21], by adding letters to the end.

Character oracles are a general abstraction for accessing the quasi-suffix collection's elements. Previously studied character oracles are for parameterized matching [5, 6, 9, 21], 2D matching [13, 9] and order preserving matching [19, 7, 10]. In practice, they are built directly from some transformation f. Each suffix of T might be transformed to a string which is not related to the other suffixes' transformations, these are the $\{s_i\}_{i=1}^{n}$. Based on these transformations the quasi-suffix collection is defined as $s_i = f(S_i)$. It is necessary that the prefixes of any string S be consistent after the transformation, i.e. $f(S[..i]) = f(S)[..i]$, to ensure that the search is well defined, locating the pattern as a prefix of some s_i.

▶ **Example 8** (Parameterized Matching)**.** Let Σ and Π be the alphabets of fixed symbols and parameters symbols respectively. Two strings are said to *parameterized match*, if one string can be transformed into the other by renaming the characters via a *one to one* function on their parameter alphabet.

A predecessor string of a string T has at each location i the distance between i and the location containing the previous appearance of the symbol $T[i]$. The first appearance of each symbol is replaced with a 0. The transformation is defined by the predecessor string and denoted by $f(T)$. The predecessor string is consistent with prefixes because dropping letters from the end of the string does not affect the distances to previous occurrences of the remaining letters.

Let $S = ababb$, where $\Pi = \{a, b\}$, then the quasi-suffix collection is $\{00221\$, 0021\$, 001\$, 01\$, 0\$, \$\}$. A special \$ is appended to each transformed suffix to fit Condition 2 of QSC.

The character oracle is based on an observation about the predecessor string. Each time a character is removed from the beginning of S_i it might affect at most one character in the transformation of S_{i+1}. This happens because at most one position distance is replaced by 0. Therefore, by maintaining for each index the distance to the previous occurrence of the same character it is possible to answer $f(S_i)[j]$ in constant time. If $j - pv[j] \geq i$ then $f(S_i)[j] = pv[j]$, otherwise $f(S_i)[j] = 0$, where pv is the distance of each character to its previous occurrence in T.

The power behind the previous results is the optimization of character oracle calls. The amount of oracle calls turns out to be linear even though the quasi-suffix collection size might be quadratic. The revolutionary idea is to create additional nodes during the construction. These nodes are used for every new suffix insertion to locate where to insert the new suffix in efficient time. Their implementation uses amortized constant time per suffix insertion. Condition 3 of QSC guarantees the correctness of suffix links. Let x be a node which is an ancestor of both s_i and s_j and $|str(x)| = lcp(s_i, s_j)$. Removing the first character results in equal strings. By Condition 3 of QSC, $lcp(s_{i+1}, s_{j+1}) \geq |str(x)| - 1 = |str(y)|$ where $y = sl(x)$. However, without this condition it is impossible to define correctly the suffix link because we might have two suffixes whose beginnings are equal, but dropping their first letters would not preserve the equality and we would not be able to search down from the node pointed to by the suffix link.

Two major problems that had been solved by the Cole and Hariharan algorithm are *the branching problem* and *the missing suffix links problem*. In the first, we might have a logarithmic factor due to linear number of children for a certain internal node in the suffix

tree. Thus, a perfect hash was introduced in Cole and Hariharan to overcome this logarithmic factor. Next, some internal node suffix links might point to a middle of an edge. This was solved by introducing new nodes called *imaginary nodes* which are added to allow explicit pointing by the suffix links.

4 Our Conditions

We wish to consider the problem of supplying an indexing construction for different matchings from a more general perspective. We would like to know whether, just by looking at the matching relation, it is possible to decide whether it can be indexed efficiently. We decided to go about this by first observing existent indexing constructions and then trying to identify a set of matching properties that are sufficient for constructing an indexing data structure. Afterwords, we introduce two additional conditions which are necessary for construction efficiency. Our conditions are proven to be more general than the ones previously studied. We discuss the connection between sufficient conditions used in previous papers for efficient indexing construction compared to our proposed conditions.

4.1 The Intuition Behind The Choice of Sufficient Conditions

Known indexing data structures, including the Suffix Tree (ST), the Suffix Array (SA) and the Burrows Wheeler Transform (BWT), sort the suffixes lexicographically in their construction procedures, resulting in an index data structure that answers queries using efficient search algorithms. Examples of these algorithms are: a binary search in the SA, search by first letter in the edges of a suffix tree, and search by reducing ranges in BWT using the *FM index* [12]. We observe that all efficient indexes needed both a fast sorting technique and an efficient search algorithm.

For the sake of being able to sort the suffixes, we need to have some order defined on them. This order is required to be a *total order*. *Transitivity* allows sorting the suffixes in a well-defined order. *Antisymmetry* is the property that if a pair of elements are related to each other then they are matched. Hence, *antisymmetry* connects the *order relation* to its matching. Finally, *completeness* allows comparing the pattern with the prefixes of any suffix.

The ability to search efficiently for a pattern P relies on comparison of the pattern with $|P|$-length prefixes of the suffixes. This is possible if the sorting is *consistent with prefixes*, otherwise a pre-sorting of the suffixes will not aid in finding prefixes of a certain length $|P|$. Formally, let a and b be two strings such that $\min\{|a|, |b|\} \geq |P|$. If w.l.o.g. aRb then $a[1..|P|]\ R\ b[1..|P|]$.

It is not enough to have a total order consistent with prefixes for a fast indexing construction. The reason is that the algorithm needs a way to quickly check if xRy is true in order to sort the suffixes. In addition, it is necessary to know during the searching if a pattern P is located at the beginning of some suffix. We use two properties to fill this gap, *order derivation* and *lcp derivation*. *Order derivation* inputs two suffixes i and j and outputs the order between them. *Lcp derivation* receives as input two suffixes i and j and outputs their longest common prefix length. We also want these derivations to be implementable online, supporting extensions by letters added to the beginning of the text. Using the lcp derivation we extend the text from the left by P. Then we can use lcp derivation queries to check if P occurs at any suffix. When finishing the search we undo all the changes that we made to the construction. Note that the extension by P should be achieved in $\tilde{O}(|P|)$ time and space to ensure pattern search efficiency.

However, implementing lcp and order derivation in sublinear time is not a trivial task even for exact matching.

▶ **Example 9.** Let i, j be two suffixes. The order derivation of those suffixes can be determined from comparing $T[i + lcp(i, j) + 1]$ with $T[j + lcp(i, j) + 1]$, while answering $lcp(i, j)$ might require a complex data structure such as Range-Minimum queries [15] on the LCP Array or LCA construction [20] on a suffix tree.

Indeed, computing the *lcp* without any previous knowledge is a hard task. But a modification of the properties can simplify the calculation to be neat and efficient.

▶ **Example 10.** *Lcp derivation* for exact matching can be calculated in constant time based on previous knowledge of $lcp(i + 1, j + 1)$.

$$lcp(i,j) = \begin{cases} lcp(i+1,j+1)+1, & \text{if } T[i] = T[j] \\ 0, & \text{otherwise.} \end{cases}$$

Moreover, *order derivation* can be answered in constant time when $lcp(i, j)$ is supplied.

The above intuition results in Theorem 6 - our sufficient conditions for efficient indexing. We prove Theorem 6 in Sect. 5, by providing an indexing construction.

▶ **Theorem 11.** *Let M be a matching relation and R a total order that is* consistent with prefixes. *Based on R's properties, the following properties hold for M.*

- Reflexivity*:* $\forall a \quad (a, a) \in M$
- Symmetry*:* $\forall a, b \quad (a, b) \in M \iff (b, a) \in M$
- Transitivity*:* $\forall a, b, c \quad (a, b) \in M \wedge (b, c) \in M \Rightarrow (a, c) \in M$
- Heredity from right (prefix consistency)*:* $\forall x, y$ *n-length strings* $\quad (x, y) \in M \Rightarrow (x[1..n-1], y[1..n-1]) \in M$

Proof. *Reflexivity* is hold, from *completeness*, aRa and from *antisymmetry*, $(a, a) \in M$.

Symmetry is hold, from *antisymmetry*, $aRb \wedge bRa$, thus from *antisymmetry*, $(b, a) \in M$.

Transitivity is hold, from *antisymmetry*, $aRb \wedge bRa \wedge bRc \wedge cRb$, thus from *transitivity*, $aRc \wedge cRa$, and from *antisymmetry*, $(a, c) \in M$.

Hereditary from right is hold, from *antisymmetry*, $xRy \wedge yRx$, from *consistent with prefixes*, $x[..n-1]Ry[..n-1] \wedge y[..n-1]Rx[..n-1]$, and finally, from *antisymmetry*, $(x[..n-1], y[..n-1]) \in M$. ◀

When presented with a new matching relation, we recommend checking these conditions first because they do not need to use the total order but are depended on the matching itself. If the matching M is reflexive, symmetric, transitive and hereditary from the right, devise a total order for the matching. Finally, provide *order and lcp derivation* properties.

4.2 The Connection To Quasi-Suffix Collections

▶ **Lemma 12.** *Let M be a matching which is an equivalence relation, i.e. reflective, symmetric and transitive. Then the following conditions are equivalent.*

1. *There* exists *a transformation f, consistent with prefixes, such that* $\forall S, T \quad f(S) = f(T) \iff (S, T) \in M$.
2. *The matching is* hereditary from right.

Proof. $1 \Rightarrow 2$. It is trivial because the transformation f is consistent with prefixes. Thus, the matching is also consistent with prefixes.

$2 \Rightarrow 1$. We will show a construction of f. It is still an open problem if it is possible to calculate a character in f in $\tilde{O}(1)$ time. As a consequence of the reflexive, symmetric, and transitive properties of M, an equivalence relation provides a partition of Σ^* into disjoint equivalence classes. The empty string is transformed into an empty string. We observe the equivalence classes of single characters. Each of those class is transferred to an identical number. This identical number is chosen from $[1, n]$, because there are at most n identical characters. Then we observe substrings of length equals to 2. Each disjoint equivalence classes of those strings is marked with an identical number from $[1, n]$. And the transformation is defined to be the transformation of the string without the last character to which the identical number that corresponds to the full string equivalence class is appended at the end. It is easy to see that this transformation is consistent with prefixes. To support the QSC definition it is required to append \$ at the end to each transformed suffix. ◀

▶ **Lemma 13.** *Let M be a matching which is an equivalence relation. Then the following two conditions are equivalent.*

1. *For every string S, $\forall i, j \;\; lcp(i, j) = l > 0 \;\Rightarrow\; lcp(i + 1, j + 1) \geq l - 1$.*
2. *The matching is* hereditary from left*, i.e. dropping some amount of characters from the beginning of two matched strings results in a matched pair.*

4.3 The Jump Forward Matching

In this section we show a matching that does not satisfy the Cole and Hariharan conditions, yet satisfies our properties.

From Lemma 12, *heredity from right* implies that a character oracle exists, although it does not necessarily follow that such an oracle answers the query in sublinear time. On the other hand, Condition 3 of QSC stipulates that

$$\forall i, j \;\; lcp(i, j) = l > 0 \;\Rightarrow\; lcp(i + 1, j + 1) \geq l - 1 \tag{1}$$

By Lemma 13, this condition is equivalent to *heredity from left*, i.e. dropping the characters from the beginnings of two matched strings results in matched pair of strings.

Thus, we wish to create a matching where *heredity from the left* is not satisfied but our conditions hold. This leads to the following matching definition.

▶ **Definition 14** (Jump Forward Matching)**.** *Let $S \in N^n$, and let $i \in \{1, ..., n\}$. A* forward jump of index i *is index $j = i + S[i]$, if $i + S[i] \leq n$, and is* out *if $i + S[i] > n$.*

The jumps sequence till out *of index i is the sequence $i_1, ..., i_k$, where $i_1 = i$, i_j is the forward jump of i_{j-1}, for $j = 2, ..., k$, and the forward jump of i_k is out. We denote the main jumps sequence to be the jumps sequence till out of index 1 by $f(S)$. Two strings are said to be matched if their main jumps sequences are equal.*

The search of a pattern turns to be a search of the main jumps sequence of the pattern.

A normalization of string S is the replacement of bigger jumps than n by jumps of size n. The normalized string is defined on $\Sigma = \{1, \ldots, n\}$ and is equivalent to the original string.

▶ **Example 15.** Let $S = 2\ 2\ 2\ 2$ and $T = 2\ 3\ 2\ 2$. These two strings are matched because their main jumps sequences are $f(S) = 1\ 3$ and similarly, $f(T) = 1\ 3$.

In this example, $S[2..]$ and $T[2..]$ are not matched because $f(S[2..]) = 1\ 3$ while $f(T[2..]) = 1$. Thus, *heredity from left* does not hold and the matching does not fit the Cole and Hariharan conditions.

▶ **Lemma 16.** *Theorem 6 conditions hold for the jump forward matching.*

Proof. Let $R = \{(X,Y)|f(X) \leq f(Y)\}$ be the order relation, where the comparison between $f(X)$ and $f(Y)$ is using the lexicographic order. This order relation is a total order, it is transitive, reflexive and complete. Moreover, it is consistent with prefixes because the transformation is consistent with prefixes, i.e. $\forall S \quad f(S[..i])$ is a prefix of $f(S)$, where i is an index in the string S.

Note that the *lcp* is defined to be the longest prefix for which the main sequences are equal. A straightforward approach can be used to implement the lcp derivation property.

$$lcp(i,j) = \begin{cases} lcp(i + S[i], j + S[j]) + 1, & \text{if } S[i] = S[j] \\ 1, & \text{otherwise.} \end{cases}$$

The order derivation property is satisfied by returning $f(S[i..])[lcp + 1]$ compared to $f(S[j..])[lcp + 1]$, where $lcp = lcp(i,j)$. We construct a forest based on the forward jumps, each character is represented by a node in the forest. Its value, the jump, is represented by an edge to its parent node in a tree. In such a way, the maximal jumps sequences are beginning from the leafs, where maximal stands for a jumps sequence that cannot be extended to a longer jumps sequence. Each leaf stores the jumps sequence that begins from its index in the text. For a node, which corresponds to index i, we store a pointer to one the leafs in its subtree. Then $f(S_i)[j]$ is calculated by the leaf's jumps sequence. First, we search to find the position *pos* in which i is located. Afterwards, we return the content at position $pos + j$. The search is implemented by *van-Emde Boas* data structure [27] in $O(\log \log n)$ time and $O(n)$ preprocessing. The indices are stored in the data structure related to the end position of the text, i.e. $n - i$ for an index i. That is, extending the text at the beginning does not affect on existent indices.

Extension of the *order and lcp derivations* by the pattern are done by constructing of the above data structure on the pattern in $O(m)$ time. ◀

4.4 The Reduction

To complete the claim that our conditions are more general than Cole and Hariharan's, we present a reduction from their conditions to our conditions.

▶ **Theorem 7.** (The Reduction) *Let M be a matching. Assume a character oracle f such that for every string S, $\{s_i\}_{i=1}^n = \{f(S_i)\}_{i=1}^n$ is a quasi-suffix collection. Then the conditions of Theorem 6 hold for M.*

Proof.

Order definition. The order relation is the lexicographic order and is a *total order*, it is transitive, reflexive and complete. The quasi-suffix collection is a collection of strings between which the comparison is done using the lexicographic order.

Order consistence with prefixes. Immediate from the order definition.

Lcp derivation. Let S be some string, we create an indexing construction based on Cole and Hariharan [9]. This construction results in a suffix tree of the quasi-suffix collection $\{s_i\}_{i=1}^n$. We then process the suffix tree to support LCA queries [8]. These queries allow the calculation of $lcp(s_i, s_j)$. Thus the lcp query is supported in constant time.

Order derivation. Order derivation is supported based on the character oracle. We compare $s_i[l+1]$ with $s_j[l+1]$ to determine the order, where $l = lcp(i,j)$.

Efficient extension by P. We add P to the suffix tree construction. $lcp(P, S_i)$ is calculated using an LCA query between the pattern and the leaf represented by the suffix S_i. The order is computed by a comparison between $P[l+1]$ and $s_i[l+1]$, where $l = lcp(P, S_i)$. Both *lcp and order derivations* are supplied in constant time. ◀

5 The Construction

In this section, we prove Theorem 6 by providing an efficient indexing construction for a matching M which satisfies the theorem's conditions. Without consistency of the suffixes, suffix links are not well defined during the suffix tree construction and a different structure is required. We use a balanced binary search tree (BST) to store the suffixes in sorted order defined by R. At each phase, a new suffix is inserted from the shortest to the longest. We can also handle text symbols that arrive by prepending to the beginning of the text. The calculation of *lcp derivation* relies on previous knowledge about other lcp values between suffixes which have already been inserted to the BST. We maintain at each node both the lcp of its suffix with the next suffix in the sort, and its lcp with the previous suffix in the sort. The following lemma helps calculate the lcp between every two suffixes in logarithmic time and constant space.

▶ **Lemma 17.** *Let $S_{i_1}, \ldots, S_{i_k}$ be a sorted group of suffixes by some total order R which is consistent with prefixes. Assume w.l.o.g. $i < j$, and $i = i_p$, $j = i_q$, then*

$$lcp(i,j) = \min_{p \le r < q} \{lcp(S_{i_r}, S_{i_{r+1}})\} \tag{2}$$

An identical lemma was proven for exact matching by Manber and Myers [22]. The order there is assumed to be the lexicographic order. Their proof is correct for our case without any modification because it is based on transitivity and consistency with prefixes.

Answering $lcp(i,j)$ for every two suffixes that exist in the BST is possible based on the above lemma.

At each node, we maintain the minimal value of the lcp values in its rooted subtree, by taking into account both lcp with previous and next suffixes. When observing the path between suffixes i and j in the BST, all the suffixes $\{S_{i_r}\}_{r=p}^{q}$ could be iterated to calculate $\min_{p \le r < q}\{lcp(i_r, i_{r+1})\}$. The iteration over the whole subtree can be replaced by constant time accessing of the minimal lcp value stored at each node. This improvement implies a logarithmic time calculation because the BST height is $O(\log n)$.

We are now ready to present how to search for a pattern P. First, we extend the lcp and order derivation, using the efficient extension property, to support *lcp derivation* of $lcp(P, S_i)$ and order between P and S_i. Then, similarly to binary search, we wish to find the range bounds S_{i_p} and S_{i_q}, where $S_{i_p} < S_{i_q}$, inside the sorted group of suffixes $\{S_{i_k}\}_{k=1}^{n}$ in which P is located at the beginning of all elements, and P matches neither a prefix of $S_{i_{p-1}}$ nor a prefix of $S_{i_{q+1}}$.

W.l.o.g. assume that we search for the left corner suffix $S_{i_{p-1}}$. Let S_{i_j} be the root of the BST. If $lcp(P, S_{i_j}) < |P|$ then the next node in the search is the root child followed to the direction of the order between P and S_{i_j}. Otherwise, if $lcp(P, S_{i_j}) = |P|$, P is a prefix of S_{i_j}, and we should check if S_{i_j} is the left corner of the range. Similarly, we check also for $lcp(P, S_{i_{j-1}})$. If this value is smaller than $|P|$ then we finish, otherwise we continue to search to the left. We store a bi-linked list between all suffixes in the sort. It allows to access predecessor and successor suffixes in the sort in constant time.

Note that $lcp(P, S_{i_j})$ is calculated in the following way. Its value is derived from a constant number of lcp values of the form $lcp(P[i'..], S_{i_{j+j'}})$ for some offsets i', j'. If $lcp(P, S_{i_j})$ value is derived from a single lcp value, its calculation is done in $O(m)$ time and constant space by a recursion, where $m = |P|$. Otherwise, because the dependency on i', j' might cause a quadratic amount of lcp calculations, another search algorithm is purposed. We insert each suffix of P to the sorted group of suffixes. In such a way $lcp(P, S_i)$ is calculated in logarithmic time based on Lemma 17.

Time.

Search for matchings with simple lcp derivation. The search time complexity is $O(\log n \cdot (T_{od} + m \cdot T_{ld})) + \tilde{O}(m)$.

Search for matchings with complex lcp derivation. The search time complexity is $O(m \log n \cdot (T_{od} + T_{ld} + \log n) + \log n \cdot (T_{od} + T_{ld} + \log n)) + \tilde{O}(m)$.

Construction time. $O(n \log n \cdot (T_{od} + T_{ld} + \log n))$ using $O(n)$ space.

In all above time formulae, T_{od} and T_{ld} are the time to perform a single order derivation and an lcp derivation query, respectively. Simple lcp derivation stands for lcp derivation calculation which requires at most a single lcp query, otherwise the lcp derivation property is called complex.

6 Conclusions and Open Problems

Assuming a matching relation on strings and an order relation of the form smaller or equal ($\leq$) which holds $A \leq B$ and $B \leq A \iff A = B$. We identified a small set of sufficient conditions on the order relation that allows indexing: (1) is total order (2) consistent with prefixes, i.e. the order between two strings is preserved for any pair of equal-length prefixes of them. We presented an efficient indexing algorithm for any matching that fulfills these conditions. Our algorithm relies on four properties: *lcp derivation*, the ability to calculate $lcp(i, j)$ based on other lcp values; *order derivation*, the ability to retrieve the order between suffixes i and j based on $lcp(i, j)$; *efficient extension by one letter*, the ability to extend both *lcp and order derivation* after text extension by one letter at the beginning, in amortized $\tilde{O}(1)$ time; and *efficient extension by the pattern*, the ability to extend both *lcp and order derivation*, to support $lcp(P, S_i)$ and order between the pattern and a suffix, in overall $\tilde{O}(m)$ time and space. An important open problem is to also provide the *necessary* conditions for indexing.

We show that our conditions for indexing are more general than previous known conditions. We prove that the consistency with prefixes is equivalent to a character oracle existence.

We also define a new matching where the previous conditions, in particular *consistency with suffixes*, do not hold but where our conditions hold. When *consistency with suffixes* means that dropping equal length prefixes from two matched strings result in a matched pair of strings. We show an efficient indexing construction which does not rely on suffix links, which cannot be used when there is no heredity from left. An interesting future direction is to define suffix trees that support these kinds of matchings.

References

1. A. Amir, A. Butman, and E. Porat. On the Relationship between Histogram Indexing and Block-Mass Indexing. *Philosophical Transactions of the Royal Society A: Mathematical Physical and Engineering Sciences*, 372(2016), 2014. URL: `http://doi.org/10.1098/rsta.2013.0132`.
2. A. Amir, T.M. Chan, M. Lewenstein, and N. Lewenstein. On the Hardness of Jumbled Indexing. In *Proc. 41st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 114–125, 2014.
3. A. Amir, M. Farach, and S. Muthukrishnan. Alphabet Dependence in Parameterized Matching. *Information Processing Letters*, 49:111–115, 1994.
4. A. Amir and I. Nor. Real-Time Indexing over Fixed Finite Alphabets. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1086–1095, 2008.
5. B. S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

6 B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.

7 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.

8 R. Cole and R. Hariharan. Dynamic LCA Queries in Trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 235–244, 1999.

9 R. Cole and R. Hariharan. Faster Suffix Tree Construction with Missing Suffix Links. *SIAM J. Comput.*, 33(1):26–42, 2003.

10 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving Indexing. *Theoretical Computer Science*, 613:122–135, 2016.

11 M. Farach. Optimal Suffix Tree Construction with Large Alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

12 P. Ferragina and G. Manzini. Indexing Compressed Texts. *J. of the ACM*, 52(4):552–581, 2005.

13 R. Giancarlo. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM J. Comput.*, 24(3):520–562, 1995.

14 J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.

15 T. Kasai, G. Lee, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

16 J. Kim, A. Amir, J. C. Na, K. Park, and J. S. Sim. On Representations of Ternary Order Relations in Numeric Strings. *Mathematics in Computer Science*, 11(2):127–136, 2017.

17 J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order Preserving Matching. *Theoretical Computer Science*, 525:68–79, 2014.

18 T. Kopelowitz. On-line Indexing for General Alphabets. In *Proc. 53rd IEEE Symposium on the Foundation of Computer Science (FOCS)*, 2012.

19 M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

20 G.M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

21 T. Lee, J.C. Na, and K. Park. On-Line Construction of Parameterized Suffix Trees. *Information Processing Letters*, 111(5):201–207, 2011.

22 U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 319–327, 1990.

23 E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

24 J. Ng, A. Amir, and P. A. Pevzner. Blocked Pattern Matching Problem and its Applications in Proteomics. In *Proc. 15th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 298–319, 2011.

25 S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian Tree Matching and Indexing. In *Proc. 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2019. To appear.

26 E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

27 P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Systems Theory*, 10:99–127, 1977.

28 P. Weiner. Linear Pattern Matching Algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Space-Efficient Computation of the LCP Array from the Burrows-Wheeler Transform

Nicola Prezza
Department of Computer Science, University of Pisa, Italy
nicola.prezza@di.unipi.it

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy
giovanna.rosone@unipi.it

Abstract

We show that the Longest Common Prefix Array of a text collection of total size n on alphabet $[1, \sigma]$ can be computed from the Burrows-Wheeler transformed collection in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input and output. Our result improves (on small alphabets) and generalizes (to string collections) the previous solution from Beller et al., which required $O(n)$ bits of extra working space. We also show how to merge the BWTs of two collections of total size n within the same time and space bounds. The procedure at the core of our algorithms can be used to enumerate suffix tree intervals in succinct space from the BWT, which is of independent interest. An engineered implementation of our first algorithm on DNA alphabet induces the LCP of a large (16 GiB) collection of short (100 bases) reads at a rate of 2.92 megabases per second using in total 1.5 Bytes per base in RAM. Our second algorithm merges the BWTs of two short-reads collections of 8 GiB each at a rate of 1.7 megabases per second and uses 0.625 Bytes per base in RAM. An extension of this algorithm that computes also the LCP array of the merged collection processes the data at a rate of 1.48 megabases per second and uses 1.625 Bytes per base in RAM.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Burrows-Wheeler Transform, LCP array, DNA reads

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.7

Related Version A full version of the paper is available at `https://arxiv.org/abs/1901.05226`.

Funding GR is partially, and NP is totally, supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

1 Introduction

The increasingly-growing production of huge datasets composed of short strings – especially in domains such as biology, where new generation sequencing technologies can nowadays generate Gigabytes of data in few hours – is lately generating much interest towards fast and space-efficient algorithms able to index this data. The Burrows-Wheeler Transform [7] and its extension to sets of strings [16, 1] is becoming the gold-standard in the field: even when not compressed, its size is one order of magnitude smaller than classic suffix arrays (while preserving many of their indexing capabilities). The functionalities of this transformation can be extended by computing additional structures such as the LCP array [9]; see, e.g. [19, 20] for a bioinformatics application where this component is needed. To date, several practical algorithms have been developed to solve the task of merging or building *de novo* such components [1, 9, 13, 14, 6, 10], but little work has been devoted to the task of computing the LCP array from the BWT of string collections in little space (internal and external working space). The only existing work we are aware of in this direction is from Beller et al. [5], who show how to build the LCP array from the BWT of a single text in $O(n \log \sigma)$

time and $O(n)$ bits of working space on top of the input and output. Other works [17, 2] show how to build the LCP array directly from the text in $O(n)$ time and $O(n \log \sigma)$ bits of space (compact). In this paper, we combine Beller et al.'s algorithm with a recent suffix-tree enumeration procedure of Belazzougui [2] and reduce this working space to $o(n \log \sigma)$ while also generalizing the algorithm to string collections. As a by-product, we show an algorithm able to merge the BWTs of two string collections using just $o(n \log \sigma)$ bits of working space. An efficient implementation of our algorithms on DNA alphabet uses (in RAM) as few as n bits on top of a packed representation of the input/output, and can process data as fast as 2.92 megabases per second.

2 Basic Concepts

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet of size σ with $c_1 < c_2 < \ldots < c_\sigma$, where $<$ denotes the standard lexicographic order. Given a text $T = t_1 t_2 \cdots t_n \in \Sigma^*$ we denote by $|T|$ its length n. We use ϵ to denote the empty string. A *factor* (or *substring*) of T is written as $T[i, j] = t_i \cdots t_j$ with $1 \leq i \leq j \leq n$. A *right-maximal* substring W of T is a string for which there exist at least two distinct characters a, b such that Wa and Wb occur in T.

Let $\mathcal{C} = \{T_1, \ldots, T_m\}$ a string collection of total length n, where each T_i is terminated by a character # (the terminator) lexicographically smaller than all other alphabet's characters. In particular, a collection is an ordered multiset, and we denote $\mathcal{C}[i] = T_i$.

The *generalized suffix array* $GSA[1..n]$ (see [21, 9, 15]) of $\mathcal{C}$ is an array of pairs $GSA[i] = \langle j, k \rangle$ such that $\mathcal{C}[j][k..]$ is the i-th lexicographically smallest suffix of strings in $\mathcal{C}$, where we break ties by input position (i.e. j in the notation above). We denote by $\texttt{range}(\texttt{W}) = \langle \texttt{left}(\texttt{W}), \texttt{right}(\texttt{W}) \rangle$ the maximal pair $\langle L, R \rangle$ such that all suffixes in $GSA[L, R]$ are prefixed by W. Note that the number of suffixes lexicographically smaller than W in the collection is $L - 1$. We extend this definition also to cases where W is not present in the collection: in this case, the (empty) range is $\langle L, L - 1 \rangle$ and we still require that $L - 1$ is the number of suffixes lexicographically smaller than W in the collection.

The *extended Burrows-Wheeler Transform* $BWT[1..n]$ [16, 1] of $\mathcal{C}$ is the character array defined as $BWT[i] = \mathcal{C}[j][k - 1 \bmod |\mathcal{C}[j]|]$, where $\langle j, k \rangle = GSA[i]$.

The *longest common prefix* (LCP) array of a collection $\mathcal{C}$ of strings (see [9, 15, 10]) is an array storing the length of the longest common prefixes between two consecutive suffixes of $\mathcal{C}$ in lexicographic order (with $LCP[1] = 0$).

Given two collections $\mathcal{C}_1, \mathcal{C}_2$ of total length n, the Document Array of their union is the binary array $DA[1..n]$ such that $DA[i] = 0$ if and only if the i-th smallest suffix comes from $\mathcal{C}_1$. When merging suffixes of the two collections, ties are broken by collection number (i.e. suffixes of $\mathcal{C}_1$ are smaller than suffixes of $\mathcal{C}_2$ in case of ties).

The C-array of a string (or collection) S is an array $C[1..\sigma]$ such that $C[i]$ contains the number of characters lexicographically smaller than i in S, plus one (S will be clear from the context). Alternatively, $C[c]$ is the starting position of suffixes starting with c in the suffix array of the string. When S (or any of its permutations) is represented with a balanced wavelet tree, then we do not need to store explicitly C, and $C[c]$ can be computed in $O(\log \sigma)$ time with no space overhead on top of the wavelet tree [18]. $S.rank_c(i)$ is the number of characters equal to c in $S[1, i - 1]$.

Function $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$, where BWT is the extended Burrows-Wheeler transform of a string collection and $\langle L, R \rangle$ is the suffix array interval of some string W appearing in the collection, returns all suffix array intervals of strings cW, with $c \neq \#$, that occur in the collection. When BWT is represented with a balanced wavelet tree, we can implement

this function so that it terminates in $O(\log \sigma)$ time per returned interval [5]. The function can be made to return the output intervals on-the-fly, one by one (in an arbitrary order), without the need to store them all in an auxiliary vector, with just $O(\log n)$ bits of additional overhead in space [5] (this requires to DFS-visit the sub-tree of the wavelet tree induced by $BWT[L, R]$; the visit requires only $\log \sigma$ bits to store the current path in the tree).

We note that an extension of the above function that navigates in parallel two BWTs is immediate. Function $\texttt{getIntervals}(\texttt{L}_1, \texttt{R}_1, \texttt{L}_2, \texttt{R}_2, \texttt{BWT}_1, \texttt{BWT}_2)$ takes as input two ranges of a string W on the BWTs of two collections, and returns the pairs of ranges on the two BWTs corresponding to all left-extensions cW of W ($c \neq \#$) such that cW appears in at least one of the two collections. To implement this function, it is sufficient to navigate in parallel the two wavelet trees as long as at least one of the two intervals is not empty.

The function $S.\texttt{rangeDistinct}(\texttt{i}, \texttt{j})$ returns the set of distinct alphabet characters *different than the terminator* # in $S[i, j]$. Also this function can be implemented in $O(\log \sigma)$ time per returned element when S is represented with a wavelet tree (again, this requires a DFS-visit of the sub-tree of the wavelet tree induced by $S[i, j]$).

$BWT.\texttt{bwsearch}(\langle \texttt{L}, \texttt{R} \rangle, \texttt{c})$ is the procedure that, given the suffix array interval $\langle L, R \rangle$ of a string W, returns the suffix array interval of cW by using the BWT [12]. This function requires access to array C and *rank* support on BWT, and runs in $O(\log \sigma)$ time when BWT is represented with a balanced wavelet tree.

3 Our Contributions

Our work builds upon the following two results from Belazzougui[1] [2] and Beller et al. [5].

▶ **Theorem 1** (Belazzougui [2]). *Given the Burrows-Wheeler Transform of a text $T \in [1, \sigma]^n$ represented with a wavelet tree, we can solve the following problem in $O(n \log \sigma)$ time using $O(\sigma^2 \log^2 n)$ bits of working space on top of the BWT. Enumerate the following information for each distinct right-maximal substring W of T: (i) $|W|$, and (ii) $range(Wc_i)$ for all $c_1 < \cdots < c_k$ such that Wc_i occurs in T.*

▶ **Theorem 2** (Beller et al. [5]). *Given the Burrows-Wheeler Transform of a text T represented with a wavelet tree, we can compute the LCP array of T in $O(n \log \sigma)$ time using $4n$ bits of working space on top of the BWT and the LCP.*

Theorem 2 represents the state of the art for computing the LCP array from the BWT. Our first observation is that Theorem 1 can be directly used to induce the LCP array of T using just $O(\sigma^2 \log^2 n)$ bits of working space on top of the input and output (proof in Section 6.1). In Section 6.1 we combine this result with Theorem 2 and obtain our first theorem:

▶ **Theorem 3.** *Given the Burrows-Wheeler Transform of a text $T \in [1, \sigma]^n$, we can compute the LCP array of T in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP.*

Proof. First, we replace T by its wavelet matrix [8] – of size $n \log \sigma + o(n \log \sigma)$ bits – in $O(n \log \sigma)$ time using just n bits of additional working space as shown in [8]. Wavelet matrices support the same set of operations of wavelet trees in the same running times (indeed, they

[1] While the original theorem [2, Sec. 5.1] is general and uses the underlying rank data structure as a black box, in our case we strive for succinct space (not compact as in [2]) and stick to wavelet trees. All details on how to achieve the claimed running time and space are described in Section 4.

can be considered as a wavelet tree representation). We re-use a portion (n bits) of the LCP array's space (which always takes $\geq n$ bits) to accommodate the extra n bits required for building the wavelet matrix, so the overall working space does not exceed $o(n \log \sigma)$ bits on top of the BWT and LCP. In the rest of the paper we will simply assume that the input is represented by a wavelet tree.

At this point, if $\sigma < \sqrt{n}/\log^2 n$ then $\sigma^2 \log^2 n = o(n)$ and our extension of Theorem 1 gives us $o(n \log \sigma)$ additional working space. If $\sigma \geq \sqrt{n}/\log^2 n$ then $\log \sigma = \Theta(\log n)$ and we can use Theorem 2, which yields extra working space $O(n) = o(n \log n) = o(n \log \sigma)$. Note that, while we used the threshold $\sigma < \sqrt{n}/\log^2 n$, any threshold of the form $\sigma < \sqrt{n}/\log^{1+\epsilon} n$, with $\epsilon > 0$ would work. The only constraint is that $\epsilon > 0$, since otherwise for $\epsilon = 0$ the working space would become $O(n \log \sigma)$ for constant σ (not good since we aim at $o(n \log \sigma)$). ◀

We proceed by extending Theorem 1 to enumerate also the intervals corresponding to leaves of the generalized suffix tree of a collection (Theorem 1 enumerates internal nodes). In Section 6.1 we show that this simple modification, combined again with the strategy of Theorem 2 (generalized to collections), can be used to extend Theorem 3 to text collections:

▶ **Theorem 4.** *Given the Burrows-Wheeler Transform of a collection $\mathcal{C} = \{T_1, \ldots, T_m\}$ of total length n on alphabet $[1, \sigma]$, we can compute the LCP array of $\mathcal{C}$ in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP.*

In [2, 3], Belazzougui et al. show that Theorem 1 can be adapted to merge the BWTs of two texts T_1, T_2 and obtain the BWT of the collection $\{T_1, T_2\}$ in $O(nk)$ time and $n \log \sigma(1 + 1/k) + 11n + o(n)$ bits of working space for any $k \geq 1$ [3, Thm. 7]. We show that our strategy enables a more space-efficient algorithm for the task of merging BWTs of collections. The following theorem, proved in Section 6.2, merges two BWTs by computing the binary DA of their union. After that, the merged BWT can be streamed to external memory (the DA tells how to interleave characters from the input BWTs) and does not take additional space in internal memory. Similarly to what we did in the proof of Theorem 3, this time we re-use the space of the Document Array to accommodate the extra n bits needed to replace the BWTs of the two collections with their wavelet matrices.

▶ **Theorem 5.** *Given the Burrows-Wheeler Transforms of two collections $\mathcal{C}_1$ and $\mathcal{C}_2$ of total length n on alphabet $[1, \sigma]$, we can compute the Document Array of $\mathcal{C}_1 \cup \mathcal{C}_2$ in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input BWTs and the output DA.*

When $k = \log \sigma$, the running time of [3, Thm. 7] is the same as our Theorem 5 but the working space is higher: $n \log \sigma + O(n)$ bits. We also briefly discuss how to extend Theorem 5 to build the LCP array of the merged collection. In Section 7 we present an implementation of our algorithms and an experimental comparison with `eGap` [11], the state-of-the-art tool designed for the same task of merging BWTs while inducing the LCP of their union.

4 Belazzougui's Enumeration Algorithm

In [2], Belazzougui showed that a BWT with *rank* and *range distinct* functionality (see Section 2) is sufficient to enumerate in small space a rich representation of the internal nodes of the suffix tree of a text T. In this section we describe his algorithm.

Remember that explicit suffix tree nodes correspond to right-maximal text substrings. By definition, for any right-maximal substring W there exist at least two distinct characters $c_1, \ldots, c_k$ such that Wc_i is a substring of T, for $i = 1, \ldots, k$. The first idea is to represent any text substring W (not necessarily right-maximal) as follows. Let $\texttt{chars}_{\texttt{W}}[1] < \cdots < \texttt{chars}_{\texttt{W}}[\texttt{k}_{\texttt{W}}]$

be the (lexicographically-sorted) character array such that $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$ is a substring of T for all[2] $i = 1, \ldots, k_W$, where k_W is the number of right-extensions of W. Let moreover $\texttt{first}_\texttt{W}[\texttt{1..k}_\texttt{W} + \texttt{1}]$ be the array such that $\texttt{first}_\texttt{W}[\texttt{i}]$ is the starting position of (the range of) $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$ in the suffix array of T for $i = 1, \ldots, k_W$, and $\texttt{first}_\texttt{W}[\texttt{k}_\texttt{W} + \texttt{1}]$ is the end position of W in the suffix array of T. The representation for W is (differently from [2], we omit $\texttt{chars}_\texttt{W}$ from the representation and we add $|W|$; these modifications will turn useful later): $\texttt{repr}(\texttt{W}) = \langle \texttt{first}_\texttt{W}, |\texttt{W}| \rangle$. Note that, if W is not right-maximal and is not a text suffix, then W is followed by $k_W = 1$ distinct characters in T and the above representation is still well-defined. When W is right-maximal, we will also say that $\texttt{repr}(\texttt{W})$ is the representation of a suffix tree explicit node (i.e. the node reached by following the path labeled W from the root). At this point, the enumeration algorithm works by visiting the Weiner Link tree of T starting from the root's representation $\texttt{repr}(\epsilon) = \langle \texttt{first}_\epsilon, \texttt{0} \rangle$, where $\texttt{first}_\epsilon = \langle C[c_1], \ldots, C[c_\sigma], n \rangle$ (see Section 2 for a definition of the C-array) and $c_1, \ldots, c_\sigma$ are all (and only) the sorted alphabet's characters. The visit uses a stack storing representations of suffix tree nodes, initialized with $\texttt{repr}(\epsilon)$. At each iteration, we pop the head $\texttt{repr}(\texttt{W})$ from the stack and we push $\texttt{repr}(\texttt{cW})$ such that cW is right-maximal in T. If nodes are pushed on the stack in decreasing order of interval length, then the stack's size never exceeds $O(\sigma \log n)$. In Appendix A we describe in detail how Weiner links are computed, and show that with this strategy we visit all suffix tree nodes in $O(n \log \sigma)$ time using overall $O(\sigma^2 \log^2 n)$ bits of additional space (for the stack). In Section 6.1 we show that this enumeration algorithm can be used to compute the LCP array from the BWT of a collection.

5 Beller et al.'s Algorithm

Also Beller et al.'s algorithm [5] works by enumerating a (linear) subset of the BWT intervals. LCP values are induced from a particular visit of those intervals. Belazzougui's and Beller et al.'s algorithms have, however, two key differences which make the former more space-efficient on small alphabets, while the latter more space-efficient on large alphabets: (i) Beller et al. use a queue (FIFO) instead of a stack (LIFO), and (ii) they represent W-intervals with just the pair of coordinates $\texttt{range}(\texttt{W})$ and the value $|W|$. In short, while Beller et al.'s queue might grow up to size $\Theta(n)$, the use of intervals (instead of the more complex representation used by Belazzougui) makes it possible to represent it using $O(1)$ bitvectors of length n. On the other hand, the size of Belazzougui's stack can be upper-bounded by $O(\sigma \log n)$, but its elements take more space to be represented.

Beller et al.'s algorithm starts by initializing all LCP entries to $\perp$ (an undefined value), and by inserting in the queue the triple $\langle 1, n, 0 \rangle$, where the first two components are the BWT interval of ϵ (the empty string) and the third component is its length. From this point, the algorithm keeps performing the following operations until the queue is empty. We remove the first (i.e. the oldest) element $\langle L, R, \ell \rangle$ from the queue, which (by induction) is the interval and length of some string W: $\texttt{range}(\texttt{W}) = \langle L, R \rangle$ and $|W| = \ell$. Using operation $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$ [5] (see Section 2) we left-extend the BWT interval $\langle L, R \rangle$ with the characters $c_1, \ldots, c_k$ in $\texttt{rangeDistinct}(\texttt{L}, \texttt{R})$, obtaining the triples $\langle L_1, R_1, \ell + 1 \rangle, \ldots, \langle L_k, R_k, \ell + 1 \rangle$ corresponding to the strings $c_1 W, \ldots, c_k W$. For each such triple $\langle L_i, R_i, \ell + 1 \rangle$, if $R_i \neq n$ and $LCP[R_i + 1] = \perp$ then we set $LCP[R_i + 1] \leftarrow \ell$ and push $\langle L_i, R_i, \ell + 1 \rangle$ on the queue. Importantly, note that we can push the intervals returned by $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$ in the queue in any order; as discussed in Section 2, this step can

[2] We require $\texttt{chars}_\texttt{W}$ to be also complete: if Wc is a substring of T, then $c \in \texttt{chars}_\texttt{W}$.

Algorithm 1: `Node-Type(BWT,LCP)`.

input : Wavelet tree of the Burrows-Wheeler transformed collection `BWT` $\in [1,\sigma]^n$ and empty array `LCP[1..n]`.
behavior : Fills node-type `LCP` values.

```
1  if σ > √n / log² n then
2  |  BGOS(BWT, LCP);                                   /* Run Beller et al.'s algorithm */
3  else
4  |  P ← new_stack();                                  /* Initialize new stack */
5  |  P.push(repr(ϵ));                                 /* Push representation of ϵ */
6  |  while not P.empty() do
7  |  |  ⟨first_W, ℓ⟩ ← P.pop();                       /* Pop highest-priority element */
8  |  |  t ← |first_W| − 1;                            /* Number of children of ST node */
9  |  |  for i = 2, ..., t do
10 |  |  |  LCP[first_W[i]] ← ℓ;                      /* Set LCP value */
11 |  |  x_1, ..., x_k ← BWT.Weiner(⟨first_W, ℓ⟩);    /* Follow Weiner Links */
12 |  |  x'_1, ..., x'_k ← sort(x_1, ..., x_k);       /* Sort by interval length */
13 |  |  for i = k ... 1 do
14 |  |  |  P.push(x'_i);                             /* Push representations */
15 LCP[0] ← 0;
```

be implemented with just $O(\log n)$ bits of space overhead with a DFS-visit of the wavelet tree's sub-tree induced by $BWT[L,R]$ (i.e. the intervals are not stored temporarily anywhere: they are pushed as soon as they are generated). To limit space usage, Beller et al. use two different queue representations. As long as there are $O(n/\log n)$ elements in the queue, they use a simple vector. When there are more intervals, they switch to a representation based on four bitvectors of length n that still guarantees constant amortized operations. All details are described in Appendix B. Beller et al. [5] show that the above algorithm correctly computes the LCP array of a text. In the next section we generalize the algorithm to text collections.

6 Our Algorithms

We describe our algorithms directly on string collections. This will include, as a particular case, inputs formed by a single text. Procedure `BGOS(BWT,LCP)` in Line 2 of Algorithm 1 is a call to Beller et al.'s algorithm, modified as follows. First, we set `LCP[C[c]]` $\leftarrow 0$ for all $c \in \Sigma$. Then, we push in the queue $\langle$`range(c)`$, 1\rangle$ for all $c \in \Sigma$ and start the main algorithm. Note moreover that (see Section 2) from now on we never left-extend ranges with #.

6.1 Computing the LCP From the BWT

Let $\mathcal{C}$ be a text collection where each string is ended by a terminator # (common to all strings). Consider now the LCP and GSA (generalized Suffix Array) arrays of $\mathcal{C}$. We divide LCP values in two types. Let $GSA[i] = \langle j,k\rangle$, with $i > 1$, indicate that the i-th suffix in the lexicographic ordering of all suffixes of strings in $\mathcal{C}$ is $\mathcal{C}[j][k..]$. A LCP value `LCP[i]` is of *node type* when the i-th and $(i-1)$-th suffixes are distinct: $\mathcal{C}[j][k..] \neq \mathcal{C}[j'][k'..]$, where $GSA[i] = \langle j,k\rangle$ and $GSA[i-1] = \langle j',k'\rangle$. Those two suffixes differ before the terminator is reached in both suffixes (it might be reached in one of the two suffixes, however); we use the name *node-type* because $i-1$ and i are the last and first suffix array positions of the ranges of two adjacent children of some suffix tree node, respectively (i.e. the node corresponding to string $\mathcal{C}[j][k..k+LCP[i]-1]$). Note that it might be that one of the two suffixes, $\mathcal{C}[j][k..]$ or $\mathcal{C}[j'][k'..]$, is the empty string (followed by the terminator) #. Similarly, a *leaf-type* LCP value `LCP[i]` is such that the i-th and $(i-1)$-th suffixes are equal: $\mathcal{C}[j][k..] = \mathcal{C}[j'][k'..]$. We use

the name *leaf-type* because, in this case, it must be the case that $i \in [L+1, R]$, where $\langle L, R\rangle$ is the suffix array range of some suffix tree leaf (it might be that $R > L$ since there might be repeated suffixes in the collection). Note that, in this case, $\mathcal{C}[j][k..] = \mathcal{C}[j'][k'..]$ could coincide with #. Entry $LCP[0]$ escapes the above classification, so we will set it separately.

Our idea is to compute first node-type and then leaf-type LCP values. We argue that Beller et al.'s algorithm already computes the former kind of LCP values. When this algorithm uses too much space (i.e. on small alphabets), we show that Belazzougui's enumeration strategy can be adapted to reach the same goal: by the very definition of node-type LCP values, they lie between children of some suffix tree node x, and their value corresponds to the string depth of x. This strategy is described in Algorithm 1. Function `BWT.Weiner(x)` in Line 11 takes as input the representation of a suffix tree node x and returns all explicit nodes reached by following Weiner links from x (an implementation of this function is described in Appendix A. Leaf-type LCP values, on the other hand, can easily be computed by enumerating intervals corresponding to suffix tree leaves. To reach this goal, it is sufficient to enumerate ranges of suffix tree leaves starting from `range`(#) and recursively left-extending with backward search with characters different from # whenever possible. For each range $\langle L, R\rangle$ obtained in this way, we set each entry $LCP[L+1, R]$ to the string depth (terminator excluded) of the corresponding leaf. This strategy is described in Algorithm 2. In order to limit space usage, we use again a stack or a queue to store leaves and their string depth (note that each leaf takes $O(\log n)$ bits to be represented): we use a queue when $\sigma > n/\log^3 n$, and a stack otherwise. The queue is the same used by Beller et al.[5] and described in Appendix B. This guarantees that the bit-size of the queue/stack never exceeds $o(n \log \sigma)$ bits: since leaves take just $O(\log n)$ bits to be represented and the stack's size never contains more than $O(\sigma \cdot \log n)$ leaves, the stack's bit-size never exceeds $O(n/\log n) = o(n)$ when $\sigma \leq n/\log^3 n$. Similarly, Beller et al's queue always takes at most $O(n)$ bits of space, which is $o(n \log \sigma)$ for $\sigma > n/\log^3 n$. Note that in Lines 13-16 we can afford storing temporarily the k resulting intervals since, in this case, the alphabet's size is small enough. To sum up, our full procedure works as follows: (1) we initialize an empty array `LCP[1..n]`, (2), we fill node-type entries using procedure `Node-Type(BWT, LCP)` described in Algorithm 1, and (3) we fill leaf-type entries using procedure `Leaf-Type(BWT, LCP)` described in Algorithm 2.

Algorithm 2: `Leaf-Type(BWT,LCP)`.

input : Wavelet tree of the Burrows-Wheeler transformed collection `BWT` $\in [1, \sigma]^n$ and array `LCP[1..n]`.
behavior : Fills leaf-type `LCP` values.

```
1  if σ > n/log³ n then
2  |  P ← new_queue();                                   /* Initialize new queue */
3  else
4  |  P ← new_stack();                                   /* Initialize new stack */
5  P.push(BWT.range(#), 0);                  /* Push range of terminator and LCP value 0 */
6  while not P.empty() do
7  |  ⟨⟨L, R⟩, ℓ⟩ ← P.pop();                            /* Pop highest-priority element */
8  |  for i = L + 1 ... R do
9  |  |  LCP[i] ← ℓ;                                   /* Set LCP inside range of ST leaf */
10 |  if σ > n/log³ n then
11 |  |  P.push(getIntervals(L, R, BWT), ℓ + 1);       /* Pairs ⟨interval, ℓ + 1⟩ */
12 |  else
13 |  |  ⟨L_i, R_i⟩_{i=1,...,k} ← getIntervals(L, R, BWT);
14 |  |  ⟨L'_i, R'_i⟩_{i=1,...,k} ← sort(⟨L_i, R_i⟩_{i=1,...,k});   /* Sort by interval length */
15 |  |  for i = k ... 1 do
16 |  |  |  P.push(⟨L'_i, R'_i⟩, ℓ + 1);              /* Push in order of decreasing length */
```

Theorems 3 and 4 follow from the correctness of our procedure, which for space reasons is reported in Appendix C as Lemma 6. As a by-product, in Appendix D we note that Algorithm 1 can be used to enumerate suffix tree intervals in succinct space from the BWT, which could be of independent interest.

6.2 Merging BWTs in Small Space

The procedure of Algorithm 2 can be extended to merge BWTs of two collections $\mathcal{C}_1$, $\mathcal{C}_2$ using $o(n \log \sigma)$ bits of working space on top of the input BWTs and output Document Array (here, n is the cumulative length of the two BWTs). The idea is to simulate a navigation of the *leaves* of the generalized suffix tree of $\mathcal{C}_1 \cup \mathcal{C}_2$ (note: for us, a collection is an ordered multi-set of strings). Our procedure differs from that described in [3, Thm. 7] in two ways. First, they navigate a subset of the suffix tree *nodes* (so-called *impure* nodes, i.e. the roots of subtrees containing suffixes from distinct strings), whereas we navigate leaves. Second, their visit is implemented by following Weiner links. This forces them to represent the nodes with the "heavy" representation `repr` of Section 4, which is not efficient on large alphabets. On the contrary, leaves can be represented simply as ranges and allow for a more space-efficient queue/stack representation.

We represent each leaf by a pair of intervals, respectively on $BWT(\mathcal{C}_1)$ and $BWT(\mathcal{C}_2)$, of strings of the form $W\#$. Note that: (i) the suffix array of $\mathcal{C}_1 \cup \mathcal{C}_2$ is covered by the non-overlapping intervals of strings of the form $W\#$, and (ii) for each such string $W\#$, the interval $\texttt{range}(\texttt{W}\#) = \langle L, R\rangle$ in $GSA(\mathcal{C}_1 \cup \mathcal{C}_2)$ can be partitioned as $\langle L, M\rangle \cdot \langle M+1, R\rangle$, where $\langle L, M\rangle$ contains only suffixes from $\mathcal{C}_1$ and $\langle M+1, R\rangle$ contains only suffixes from $\mathcal{C}_2$ (one of these two intervals could be empty). It follows that we can navigate in parallel the leaves of the suffix trees of $\mathcal{C}_1$ and $\mathcal{C}_2$ (using again a stack or a queue containing pairs of intervals on the two BWTs), and fill the Document Array $DA[1..n]$, an array that will tell us whether the i-th entry of $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$ comes from $BWT(\mathcal{C}_1)$ ($DA[i] = 0$) or $BWT(\mathcal{C}_2)$ ($DA[i] = 1$). To do this, let $\langle L_1, R_1\rangle$ and $\langle L_2, R_2\rangle$ be the ranges on the suffix arrays of $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, of a suffix $W\#$ of some string in the collections. Note that one of the two intervals could be empty: $R_j < L_j$. In this case, we still require that $L_j - 1$ is the number of suffixes in $\mathcal{C}_j$ that are smaller than $W\#$. Then, in the collection $\mathcal{C}_1 \cup \mathcal{C}_2$ there are $L_1 + L_2 - 2$ suffixes smaller than $W\#$, and $R_1 + R_2$ suffixes smaller than or equal to $W\#$. It follows that the range of $W\#$ in the suffix array of $\mathcal{C}_1 \cup \mathcal{C}_2$ is $\langle L_1 + L_2 - 1, R_1 + R_2\rangle$, where the first $R_1 - L_1 + 1$ entries correspond to suffixes of strings from $\mathcal{C}_1$. Then, we set $DA[L_1 + L_2 - 1, L_2 + R_1 - 1] \leftarrow 0$ and $DA[L_2 + R_1, R_1 + R_2] \leftarrow 1$. The procedure starts from the pair of intervals corresponding to the ranges of the string "#" in the two BWTs, and proceeds recursively by left-extending the current pair of ranges $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$ with the symbols in $\texttt{BWT}_1.\texttt{rangeDistinct}(\texttt{L}_1, \texttt{R}_1) \cup \texttt{BWT}_2.\texttt{rangeDistinct}(\texttt{L}_2, \texttt{R}_2)$. The detailed procedure is reported in Algorithm 3. The leaf visit is implemented, again, using a stack or a queue; this time however, these containers are filled with pairs of intervals $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$. We implement the stack simply as a vector of quadruples $\langle L_1, R_1, L_2, R_2\rangle$. As far as the queue is concerned, some care needs to be taken when representing the pairs of ranges using bitvectors as seen in Appendix B with Beller et al.'s representation. Recall that, at any time, the queue can be partitioned in two sub-sequences associated with LCP values ℓ and $\ell + 1$ (we pop from the former, and push in the latter). This time, we represent each of these two subsequences as a vector of quadruples (pairs of ranges on the two BWTs) as long as the number of quadruples in the sequence does not exceed $n/\log n$. When there are more quadruples than this threshold, we switch to a bitvector representation defined as follows. Let $|BWT(\mathcal{C}_1)| = n_1$, $|BWT(\mathcal{C}_2)| = n_2$, and $|BWT(\mathcal{C}_1 \cup \mathcal{C}_2)| = n = n_1 + n_2$. We keep two

bitvectors $\texttt{Open}[1..n]$ and $\texttt{Close}[1..n]$ storing opening and closing parentheses of intervals in $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$. We moreover keep two bitvectors $\texttt{NonEmpty}_1[1..n]$ and $\texttt{NonEmpty}_2[1..n]$ keeping track, for each i such that $\texttt{Open}[i] = 1$, of whether the interval starting in $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)[i]$ contains suffixes of reads coming from $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Finally, we keep four bitvectors $\texttt{Open}_j[1..n_j]$ and $\texttt{Close}_j[1..n_j]$, for $j = 1, 2$, storing non-empty intervals on $BWT(\mathcal{C}_1)$ and $BWT(\mathcal{C}_2)$, respectively. To insert a pair of intervals $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$ in the queue, let $\langle L, R\rangle = \langle L_1 + L_2 - 1, R_1 + R_2\rangle$. We set $\texttt{Open}[L] \leftarrow 1$ and $\texttt{Close}[R] \leftarrow 1$. Then, for $j = 1, 2$, we set $\texttt{NonEmpty}_j[L] \leftarrow 1$, $\texttt{Open}_j[L_j] \leftarrow 1$ and $\texttt{Close}_j[R_j] \leftarrow 1$ if and only if $R_j \geq L_j$. This queue representation takes $O(n)$ bits. By construction, for each bit set in $\texttt{Open}$ at position i, there is a corresponding bit set in $\texttt{Open}_j$ if and only if $\texttt{NonEmpty}_j[i] = 1$ (moreover, corresponding bits set appear in the same order in $\texttt{Open}$ and $\texttt{Open}_j$). It follows that a left-to-right scan of these bitvectors is sufficient to identify corresponding intervals on $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$, $BWT(\mathcal{C}_1)$, and $BWT(\mathcal{C}_2)$. By packing the bits of the bitvectors in words of $\Theta(\log n)$ bits, the t pairs of intervals contained in the queue can be extracted in $O(t + n/\log n)$ time (as described in [5]) by scanning in parallel the bitvectors forming the queue. Particular care needs to be taken only when we find the beginning of an interval $\texttt{Open}[L] = 1$ with $\texttt{NonEmpty}_1[L] = 0$ (the case $\texttt{NonEmpty}_2[L] = 0$ is symmetric). Let L_2 be the beginning of the corresponding non-empty interval on $BWT(\mathcal{C}_2)$. Even though we are not storing L_1 (because we only store nonempty intervals), we can retrieve this value as $L_1 = L - L_2 + 1$. Then, the empty interval on $BWT(\mathcal{C}_1)$ is $\langle L_1, L_1 - 1\rangle$.

The same arguments used in the previous section show that the algorithm runs in $O(n \log \sigma)$ time and uses $o(n \log \sigma)$ bits of space on top of the input BWTs and output Document Array. This proves Theorem 5. To conclude, we note that the algorithm can be extended to compute the LCP array of the merged collection while merging the BWTs. This requires adapting Algorithm 1 to work on pairs of suffix tree nodes (as we did in Algorithm 3 with pairs of leaves), but for space reasons we do not describe all details here. Results on an implementation of the extended algorithm are discussed in the next section. From the practical point of view, note that it is more advantageous to induce the LCP of the merged collection while merging the BWTs (rather than first merging and then inducing the LCP using the algorithm of the previous section), since leaf-type LCP values can be induced directly while computing the document array.

Note that Algorithm 3 is similar to Algorithm 2, except that now we manipulate pairs of intervals. In Line 22, we sort quadruples according to the length $R_1^i + R_2^i - (L_1^i + L_2^i) + 2$ of the combined interval on $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$. Finally, note that Backward search can be performed correctly also when the input interval is empty: $\texttt{BWT}_j\texttt{.bwsearch}(\langle L_j, L_j - 1\rangle, c)$, where $L_j - 1$ is the number of suffixes in $\mathcal{C}_j$ smaller than some string W, correctly returns the pair $\langle L', R'\rangle$ such that L' is the number of suffixes in $\mathcal{C}_j$ smaller than cW: this is true when implementing backward search with a $rank_c$ operation on position L_j; then, if the original interval is empty we just set $R' = L' - 1$ to keep the invariant that $R' - L' + 1$ is the interval's length.

7 Implementation and Experimental Evaluation

We implemented our algorithms on DNA alphabet in `https://github.com/nicolaprezza/bwt2lcp` using the language C++. Thanks to the small alphabet size, it was actually sufficient to implement our extension of Belazzougui's enumeration algorithm (and not the strategy of Beller et al., which is more suited to large alphabets). The repository features a new packed string on DNA alphabet $\Sigma_{DNA} = \{A, C, G, T, \#\}$ using 4 bits per character and able to

Algorithm 3: `Merge(BWT_1, BWT_2, DA)`.

input : Wavelet trees of the Burrows-Wheeler transformed collections $\texttt{BWT}_1 \in [1,\sigma]^{n_1}$, $\texttt{BWT}_2 \in [1,\sigma]^{n_2}$ and empty document array `DA[1..n]`, with $n = n_1 + n_2$.
behavior : Computes Document Array DA.

```
1  if σ > n/log^3 n then
2  |  P ← new_queue();                                  /* Initialize new queue of interval pairs */
3  else
4  |  P ← new_stack();                                  /* Initialize new stack of interval pairs */
5  P.push(BWT_1.range(#), BWT_2.range(#));              /* Push SA-ranges of terminator */
6  while not P.empty() do
7  |  ⟨L_1, R_1, L_2, R_2⟩ ← P.pop();                   /* Pop highest-priority element */
8  |  for i = L_1 + L_2 − 1 ... L_2 + R_1 − 1 do
9  |  |  DA[i] ← 0;                                    /* Suffixes from C_1 */
10 |  for i = L_2 + R_1 ... R_1 + R_2 do
11 |  |  DA[i] ← 1;                                    /* Suffixes from C_2 */
12 |  if σ > n/log^3 n then
13 |  |  P.push(getIntervals(L_1, R_1, L_2, R_2, BWT_1, BWT_2));   /* New intervals */
14 |  else
15 |  |  c^1_1, ..., c^1_{k_1} ← BWT_1.rangeDistinct(L_1, R_1);
16 |  |  c^2_1, ..., c^2_{k_2} ← BWT_2.rangeDistinct(L_2, R_2);
17 |  |  {c_1 ... c_k} ← {c^1_1, ..., c^1_{k_1}} ∪ {c^2_1, ..., c^2_{k_2}};
18 |  |  for i = 1 ... k do
19 |  |  |  ⟨L^i_1, R^i_1⟩ ← BWT_1.bwsearch(⟨L_1, R_1⟩, c_i);   /* Backward search step */
20 |  |  for i = 1 ... k do
21 |  |  |  ⟨L^i_2, R^i_2⟩ ← BWT_2.bwsearch(⟨L_2, R_2⟩, c_i);   /* Backward search step */
22 |  |  ⟨L̂^i_1, R̂^i_1, L̂^i_2, R̂^i_2, ⟩_{i=1,...,k} ← sort(⟨L^i_1, R^i_1, L^i_2, R^i_2, ⟩_{i=1,...,k});
23 |  |  for i = k ... 1 do
24 |  |  |  P.push(L̂^i_1, R̂^i_1, L̂^i_2, R̂^i_2);   /* Push in order of decreasing length */
```

compute the quintuple $\langle rank_c(i)\rangle_{i\in\Sigma_{DNA}}$ with just one cache miss. This is crucial for our algorithms, since at each step we need to left-extend ranges by all characters. This class divides the text in blocks of 128 characters. Each block is stored using 512 cache-aligned bits (the typical size of a cache line), divided as follows. The first 128 bits store four 32-bits counters with the partial ranks of A, C, G, and T before the block (if the string is longer than 2^{32} characters, we further break it into superblocks of 2^{32} characters; on reasonably-large inputs, the extra rank table fits in cache and does not cause additional cache misses). The following three blocks of 128 bits store the first, second, and third bits, respectively, of the characters' binary encodings (each character is packed in 3 bits). Using this layout, the rank of each character in the block can be computed with at most three masks, a bitwise AND (actually less, since we always compute the rank of all five characters and we re-use partial results whenever possible), and a `popcount` operation. We also implemented a packed string on the augmented alphabet $\Sigma^+_{DNA} = \{A, C, G, N, T, \#\}$ using 4.38 bits per character and offering the same cache-efficiency guarantees. In this case, a 512-bits block stores 117 characters, packed as follows. As before, the first 128 bits store four 32-bits counters with the partial ranks of A, C, G, and T before the block. Each of the following three blocks of 128 bits is divided in a first part of 117 bits and a second part of 11 bits. The first parts store the first, second, and third bits, respectively, of the characters' binary encodings. The three parts of 11 bits, concatenated together, store the rank of N's before the block. This layout minimizes the number of bitwise operations (in particular, shifts and masks) needed to compute a parallel rank.

Several heuristics have been implemented to reduce the number of cache misses in practice. In particular, we note that in Algorithm 2 we can avoid backtracking when the range size becomes equal to one; the same optimization can be implemented in Algorithm 3 when also

Table 1 Datasets used in our experiments. Size accounts only for the alphabet's characters. The alphabet's size σ includes the terminator.

Name	Size GiB	σ	N. of reads	Max read length	Bytes for lcp values
NA12891.8	8.16	5	85,899,345	100	1
shortreads	8.0	6	85,899,345	100	1
pacbio	8.0	6	942,248	71,561	4
pacbio.1000	8.0	6	8,589,934	1000	2
NA12891.24	23.75	6	250,000,000	100	1
NA12878.24	23.75	6	250,000,000	100	1

Table 2 In this experiment, we merge pairs of BWTs and induce the LCP of their union using `eGap` and `merge`. We also show the resources used by the pre-processing step (building the BWTs) for comparison. Wall clock is the elapsed time from start to completion of the instance, while RAM (in GiB) is the peak Resident Set Size (RSS). All values were taken using the `/usr/bin/time` command. During the preprocessing step on the collections pacBio.1000 and pacBio, the available memory in MB (parameter m) of `eGap` was set to 32000 MB. In the merge step this parameter was set to about to the memory used by `merge`. `eGap` and `merge` take as input the same BWT file.

	Preprocessing		`eGap`		`merge`	
Name	Wall Clock (h:mm:ss)	RAM (GiB)	Wall Clock (h:mm:ss)	RAM (GiB)	Wall Clock (h:mm:ss)	RAM (GiB)
NA12891.8	1:15:57	2.84	10:15:07	18.09 (-m 32000)	3:16:40	26.52
NA12891.8.RC	1:17:55	2.84				
shortreads	1:14:51	2.84	11:03:10	16.24 (-m 29000)	3:36:21	26.75
shortreads.RC	1:19:30	2.84				
pacbio.1000	2:08:56	31.28	5:03:01	21.23 (-m 45000)	4:03:07	42.75
pacbio.1000.RC	2:15:08	31.28				
pacbio	2:27:08	31.25	2:56:31	33.40 (-m 80000)	4:38:27	74.76
pacbio.RC	2:19:27	31.25				
NA12878.24	4:24:27	7.69	31:12:28	47.50 (-m 84000)	6:41:35	73.48
NA12891.24	4:02:42	7.69				

computing the LCP array, since leaves of size one can be identified during navigation of internal suffix tree nodes. Overall, we observed (using a memory profiler) that in practice the combination of Algorithms 1-2 generates at most $1.5n$ cache misses; the extension of Algorithm 3 that computes also LCP values generates twice this number of cache misses (this is expected, since it navigates two BWTs).

We now report some preliminary experiments on our algorithms: `bwt2lcp` (Algorithms 1-2) and `merge` (Algorithm 3, extended to compute also the LCP array). All tests were done on a DELL PowerEdge R630 machine, used in non exclusive mode. Our platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GiB of shared memory and 1TB of SSD. The system is Ubuntu 14.04.2 LTS. The code was compiled using gcc 8.1.0 with flags `-Ofast -fstrict-aliasing`.

Table 3 In this experiment, we induced the LCP array from the BWT of a collection (each collection is the union of two collections from Table 2). We also show pre-processing requirements (i.e. building the BWT) of the better performing tool between BCR and `eGap`.

	Preprocessing		`bwt2lcp`	
Name	Wall Clock (h:mm:ss)	RAM GiB	Wall Clock (h:mm:ss)	RAM (GiB)
NA12891.8 ∪ NA12891.8.RC (BCR)	2:43:02	5.67	1:40:01	24.48
shortread ∪ shortread.RC (BCR)	2:47:07	5.67	2:14:41	24.75
pacbio.1000 ∪ pacbio.1000.RC (`eGap`-m 32000)	7:07:46	31.28	1:54:56	40.75
pacbio ∪ pacbio.RC (`eGap`-m 80000)	6:02:37	78.125	2:14:37	72.76
NA12878.24 ∪ NA12891.24 (BCR)	8:26:34	16.63	6:41:35	73.48

Table 1 summarizes the datasets used in our experiments. "NA12891.8"[3] contains Human DNA reads on the alphabet Σ_{DNA} where we have removed reads containing the nucleotide N. "shortreads" contains Human DNA short reads on the extended alphabet Σ^{+}_{DNA}. "pacbio" contains PacBio RS II reads from the species *Triticum aestivum* (wheat). "pacbio.1000" are the strings from "pacbio" trimmed to length 1,000. All the above datasets except the first have been download from `https://github.com/felipelouza/egap/tree/master/dataset`. To conclude, we added two collections, "NA12891.24" and "NA12878.24" obtained by taking the first $250,000,000$ reads from individuals NA12878[4] and NA12891. All datasets except "NA12891.8" are on the alphabet Σ^{+}_{DNA}. In Tables 2 and 3, the suffix ".RC" added to a dataset's name indicates the reverse-complemented dataset.

We compare our algorithms with `eGap`[5] and BCR[6], two tools designed to build the BWT and LCP of a set of DNA reads. Since no tools for inducing the LCP from the BWT of a set of strings are available in the literature, in Table 3 we simply compare the resources used by `bwt2lcp` with the time and space requirements of `eGap` and BCR when building the BWT. In [10], experimental results show that BCR works better on short reads and collections with a large average LCP, while `eGap` works better when the datasets contain long reads and relatively small average LCP. For this reason, in the preprocessing step we have used BCR for the collections containing short reads and `eGap` for the other collections. `eGap`, in addition, is capable of merging two or more BWTs while inducing the LCP of their union. In this case, we can therefore directly compare the performance of `eGap` with our tool `merge`; results are reported in Table 2. Since the available RAM is greater than the size of the input, we have used the semi-external strategy of `eGap`. Notice that an entirely like-for-like comparison between our tools and `eGap` is not completely feasible, being `eGap` a semi-external memory tool (our tools, instead, use internal memory only). While in our tables we report RAM usage only, it is worth to notice that `eGap` uses a considerable amount of disk working space. For example, the tool uses 56GiB of disk working space when run on a 8GiB input (in general, the disk usage is of $7n$ bytes).

Our tools exhibit a dataset-independent linear time complexity, whereas `eGap`'s running time depends on the average LCP. Table 3 shows that our tool `bwt2lcp` induces the LCP from the BWT faster than building the BWT itself. When N's are not present in the dataset,

[3] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12891/sequence_read/SRR622458_1.filt.fastq.gz`
[4] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12878/sequence_read/SRR622457_1.filt.fastq.gz`
[5] `https://github.com/felipelouza/egap`
[6] `https://github.com/giovannarosone/BCR_LCP_GSA`

`bwt2lcp` processes data at a rate of 2.92 megabases per second and uses 0.5 Bytes per base in RAM in addition to the LCP. When N's are present, the throughput decreases to 2.12 megabases per second and the tool uses 0.55 Bytes per base in addition to the LCP. As shown in Table 2, our tool `merge` is from 1.25 to 4.5 times faster than `eGap` on inputs with large average LCP, but 1.6 times slower when the average LCP is small (dataset "pacbio"). When N's are not present in the dataset, `merge` processes data at a rate of 1.48 megabases per second and uses 0.625 Bytes per base in addition to the LCP. When N's are present, the throughput ranges from 1.03 to 1.32 megabases per second and the tool uses 0.673 Bytes per base in addition to the LCP. When only computing the merged BWT (results not shown here for space reasons), `merge` uses in total 0.625/0.673 Bytes per base in RAM (without/with N's) and is about 1.2 times faster than the version computing also the LCP.

References

1. M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013.
2. D. Belazzougui. Linear time construction of compressed text indices in compact space. In *STOC*, pages 148–193. ACM, 2014.
3. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Linear-time string indexing and analysis in small space. *arXiv preprint arXiv:1609.06378*, 2016.
4. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *TALG*, 10(4):23, 2014.
5. T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows–Wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.
6. P. Bonizzoni, G. Della Vedova, S. Nicosia, Y. Pirola, M. Previtali, and R. Rizzi. Divide and Conquer Computation of the Multi-string BWT and LCP Array. In *CiE*, LNCS, pages 107–117. Springer, 2018.
7. M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DEC Systems Research Center, 1994.
8. F. Claude, G. Navarro, and A. Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.
9. A.J. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37:17–33, 2016.
10. L. Egidi, F.A. Louza, G. Manzini, and G.P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6, 2019.
11. L. Egidi and G. Manzini. Lightweight BWT and LCP merging via the Gap algorithm. In *SPIRE*, LNCS, pages 176–190. Springer, 2017.
12. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE, 2000.
13. J. Holt and L. McMillan. Constructing Burrows-Wheeler transforms of large string collections via merging. In *ACM-BCB*, pages 464–471. ACM, 2014.
14. J. Holt and L. McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014.
15. F.A. Louza, G.P. Telles, S. Hoffmann, and C.D.A. Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26, 2017.
16. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.
17. J.I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *SODA*, pages 408–424. SIAM, 2017.
18. Gonzalo N. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.
19. N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Detecting Mutations by eBWT. In *WABI 2018*, volume 113 of *LIPIcs*, pages 3:1–3:15, 2018.

20 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms Mol. Biol.*, 14(1):3, 2019.

21 F. Shi. Suffix Arrays for Multiple Strings: A Method for On-Line Multiple String Searches. In *ASIAN*, volume 1179 of *LNCS*, pages 11–22. Springer, 1996.

A Notes on Belazzougui's Algorithm

The enumeration algorithm works by visiting the Weiner link tree of the text. While this guarantees that we will visit all and only the suffix tree's explicit nodes (see [2]), there are two main issues that need to be addressed. First, the stack's size may grow in an uncontrolled way. The solution to this problem is simple: once computed $\texttt{repr}(\texttt{cW})$ for the right-maximal left-extensions cW of W, we push them on the stack in decreasing order of range length $range(cW)$ (i.e. the node with the smallest range is pushed last). This guarantees that the stack will always contain at most $O(\sigma \log n)$ elements [2]. Since each element takes $O(\sigma \log n)$ bits to be represented, the stack's size never exceeds $O(\sigma^2 \log^2 n)$ bits.

The second issue that needs to be addressed is how to efficiently compute $\texttt{repr}(\texttt{cW})$ from $\texttt{repr}(\texttt{W})$ for the characters c such that cW is right-maximal in T. In [2, 3] this operation is supported efficiently by first enumerating all *distinct* characters in each range $BWT[\texttt{first}_\texttt{W}[\texttt{i}]..\texttt{first}_\texttt{W}[\texttt{i}+1]]$ for $i = 1, \ldots, k_W$. Using the notation of [2], let us call $\texttt{rangeDistinct}(\texttt{i}, \texttt{j})$ the operation that returns all distinct characters in $BWT[i, j]$. Equivalently, for each $a \in \texttt{chars}_\texttt{W}$ we want to list all distinct left-extensions cWa of Wa. Note that, in this way, we may also visit implicit suffix tree nodes (i.e. some of these left-extensions could be not right-maximal). Stated otherwise, we are traversing all explicit *and* implicit Weiner links. Since the number of such links is linear [2, 4] (even including implicit Weiner links[7]), globally the number of distinct characters returned by `rangeDistinct` operations is $O(n)$. An implementation of `rangeDistinct` on wavelet trees is discussed in [5] with the procedure `getIntervals` (this procedure actually returns more information: the suffix array range of each cWa). This implementation runs in $O(\log \sigma)$ time per returned character. Globally, we therefore spend $O(n \log \sigma)$ time using a wavelet tree. We now need to compute $\texttt{repr}(\texttt{cW})$ for all left-extensions of W and keep only the right-maximal ones. Let $x = \texttt{repr}(\texttt{W})$ and $\texttt{BWT.Weiner}(\texttt{x})$ be the function that returns the representations of such strings (used in Line 11 of Algorithm 1). This function can be implemented by observing that

$$\texttt{range}(\texttt{cWa}) = \langle\ \texttt{C}[\texttt{c}] + \texttt{BWT.rank}_\texttt{c}(\texttt{left}(\texttt{Wa})), \texttt{C}[\texttt{c}] + \texttt{BWT.rank}_\texttt{c}(\texttt{right}(\texttt{Wa}) + 1) - 1\ \rangle$$

where $a = \texttt{chars}_\texttt{W}[\texttt{i}]$ for $1 \leq i < |\texttt{first}_\texttt{W}|$, and noting that $\texttt{left}(\texttt{Wa})$ and $\texttt{right}(\texttt{Wa})$ are available in $\texttt{repr}(\texttt{W})$. Note also that we do not actually need to know the value of characters $\texttt{chars}_\texttt{W}[\texttt{i}]$ to compute the ranges of each $cW \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$; this is the reason why we can omit $\texttt{chars}_\texttt{W}$ from $\texttt{repr}(\texttt{W})$. Using a wavelet tree, the above operation takes $O(\log \sigma)$ time. By the above observations, the number of strings cWa such that W is right-maximal is bounded by $O(n)$. Overall, computing $\texttt{repr}(\texttt{cW}) = \langle \texttt{first}_\texttt{cW}, |W| + 1 \rangle$ for all left-extensions cW of all right-maximal strings W takes therefore $O(n \log \sigma)$ time. Within the same running time, we can check which of those extensions is right maximal (i.e. those such that $|\texttt{first}_\texttt{cW}| \geq 2$), sort them by interval length (we always sort at most σ node representations, therefore also sorting takes globally $O(n \log \sigma)$ time), and push them on the stack.

[7] To see this, first note that the number of right-extensions Wa of W that have only one left-extension cWa is at most equal to the number of right-extensions of W; globally, this is at most the number of suffix tree's nodes (linear). Any other right-extension Wa that has at least two distinct left-extensions cWa and bWa is, by definition, left maximal and corresponds therefore to a node in the suffix tree of the reverse of T. It follows that all left-extensions of Wa can be charged to an edge of the suffix tree of the reverse of T (again, the number of such edges is linear).

B Notes on Beller et al.'s Algorithm

Time complexity. It is easy to see that the algorithm inserts in total a linear number of intervals in the queue since an interval $\langle L_i, R_i, \ell+1\rangle$ is inserted only if $LCP[R_i+1] = \perp$, and successively $LCP[R_i+1]$ is set to a value different than $\perp$. Clearly, this can happen at most n times. In [5] the authors moreover show that, even counting the left-extensions of those intervals (that we compute after popping each interval from the queue), the total number of computed intervals stays linear. Overall, the algorithm runs therefore in $O(n \log \sigma)$ time (as discussed in Section 2, `getIntervals` runs in $O(\log \sigma)$ time per returned element).

Queue implementation. To limit space usage, Beller et al. use the following queue representations. First note that, at each time point, the queue's triples are partitioned in a (possibly empty) sequence with associated LCP value (i.e. the third element in the triples) $\ell+1$, followed by a sequence with associated LCP value ℓ, for some ℓ. We can therefore store the two sequences (with associated LCP value) independently, and there is no need to store the LCP values in the triples themselves (i.e. the queue's elements become just ranges). Note also that we pop elements from the sequence with associated LCP value ℓ, and push elements in the sequence with associated LCP value $\ell+1$. When the former sequence is empty, we create a new sequence with associated LCP value $\ell+2$ and start popping from the sequence with associated LCP value $\ell+1$ (and so on). Beller et al. represent each of the two sequences separately as follows. While inserting elements in a sequence, as long as the sequence's length does not exceed $n/\log n$ we represent it as a vector of pairs (of total size at most $O(n)$ bits). This representation supports push/pop operations in (amortized) constant time. As soon as the sequence's length exceeds $n/\log n$, we switch to a representation that uses two packed bitvectors of length n storing, respectively, the left- and right-most boundaries of the ranges in the sequence. Note that this representation can be used because the sequence of intervals corresponds to suffix array ranges of strings of some fixed length ℓ, therefore there cannot be overlapping intervals. Pushing an interval in this new queue's representation takes constant time. Popping all the t intervals from one of the two sequences, on the other hand, can be implemented in $O(t + n/\log n)$ time by scanning the bitvectors (this requires using simple bitwise operations on words, see [5] for all details). Since at most $O(\log n)$ sequences will exceed size $n/\log n$, overall pop operations take amortized constant time.

C Proofs

▶ **Lemma 6.** *Algorithms 1 and 2 correctly compute the LCP array of the collection in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input and output.*

Proof.

Correctness and completeness - Algorithm 1. We start by proving that Beller et al.'s procedure in Line 2 of Algorithm 1 (procedure `BGOS(BWT,LCP)`) fills all the node-type LCP entries correctly. The proof proceeds by induction on the LCP value ℓ and follows the original proof of [5]. At the beginning, we insert in the queue all c-intervals, for $c \in \Sigma$. For each such interval $\langle L, R\rangle$ we set $LCP[R+1] = \ell = 0$. It is easy to see that after this step all and only the node-type LCP values equal to 0 are correctly set. Assume, by induction, that all node-type LCP values less than or equal to ℓ have been correctly set, and we are about to extract from the queue the first triple $\langle L, R, \ell+1\rangle$ having length $\ell+1$. For each extracted triple with length $\ell+1$ associated to a string W, consider the triple $\langle L', R', \ell+2\rangle$ associated to one of its left-extensions cW. If $LCP[R'+1] \neq \perp$, then we have nothing to do. However, if $LCP[R'+1] = \perp$, then it must be the cases that (i)

the value to write in this cell satisfies $LCP[R'+1] \geq \ell+1$, since by induction we have already filled all node-type LCP values smaller than or equal to ℓ, and (ii) $LCP[R'+1]$ is of node-type, since otherwise the BWT interval of cW would also include position $R'+1$. On the other hand, it cannot be the case that $LCP[R'+1] > \ell+1$ since otherwise the cW-interval would include position $R'+1$. We therefore conclude that $LCP[R'+1] = \ell+1$ must hold.

The above argument settles correctness; to prove completeness, assume that, at some point, $LCP[i] = \perp$ and the correct value to be written in this cell is $\ell+1$. We want to show that we will pull a triple $\langle L, R, \ell+1\rangle$ from the queue corresponding to a string W (note that $\ell+1 = |W|$ and, moreover, W could end with #) such that one of the left-extensions aW of W satisfies $\texttt{range}(\texttt{aW}) = \langle L', i-1\rangle$, for some L'. This will show that, at some point, we will set $LCP[i] \leftarrow \ell+1$. We proceed by induction on $|W|$. Note that we separately set all LCP values equal to 0. The base case $|W| = 1$ is easy: by the way we initialized the queue, $\langle \texttt{range}(\texttt{c}), 1\rangle$, for all $c \in \Sigma$, are the first triples we pop. Since we left-extend these ranges with all alphabet's characters except #, it is easy to see that all LCP values equal to 1 are set. From now on we can therefore assume that we are setting LCP-values equal to $\ell+1 > 1$, i.e. $W = b \cdot V$, for $b \in \Sigma - \{\#\}$ and $V \in \Sigma^+$. Let abV be the length-$(\ell+2)$ left-extension of $W = bV$ such that $\texttt{right}(\texttt{abV}) + \texttt{1} = i$. Since, by our initial hypothesis, $\texttt{LCP}[\texttt{i}] = \ell+1$, the collection contains also a suffix aU lexicographically larger than abV and such that $\texttt{LCP}(\texttt{aU}, \texttt{abV}) = \ell+1$. But then, it must be the case that $\texttt{LCP}(\texttt{right}(\texttt{bV}) + \texttt{1}) = \ell$ (it cannot be smaller by the existence of U and it cannot be larger since $|bV| = \ell+1$). By inductive hypothesis, this value was set after popping a triple $\langle L'', R'', \ell\rangle$ corresponding to string V, left-extending V with b, and pushing $\langle \texttt{range}(\texttt{bV}), \ell+1\rangle$ in the queue. This completes the completeness proof since we showed that $\langle \texttt{range}(\texttt{bV}), \ell+1\rangle$ is in the queue, so sooner or later we will pop it, extend it with a, and set $\texttt{LCP}[\texttt{right}(\texttt{abV}) + \texttt{1}] = \texttt{LCP}[\texttt{i}] \leftarrow \ell+1$.

If the queue uses too much space, then Algorithm 1 switches to a stack and Lines 4-14 are executed instead of Line 2. Note that this pseudocode fragment corresponds to Belazzougui's enumeration algorithm, except that now we also set LCP values in Line 10. By the enumeration procedure's correctness, we have that, in Line 10, $\langle \texttt{first}_\texttt{W}[\texttt{1}], \texttt{first}_\texttt{W}[\texttt{t}+\texttt{1}]\rangle$ is the SA-range of a right-maximal string W with $\ell = |W|$, and $\texttt{first}_\texttt{W}[\texttt{i}]$ is the first position of the SA-range of Wc_i, with $i = 1, \ldots, t$, where $c_1, \ldots, c_2$ are all the (sorted) right-extensions of W. Then, clearly each LCP value in Line 10 is of node-type and has value ℓ, since it is the LCP between two strings prefixed by $W \cdot \texttt{chars}_\texttt{W}[\texttt{i} - \texttt{1}]$ and $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$. Similarly, completeness of the procedure follows from the completeness of the enumeration algorithm. Let $LCP[i]$ be of node-type. Consider the prefix Wb of length $LCP[i]+1$ of the i-th suffix in the lexicographic ordering of all strings' suffixes. Since $LCP[i] = |W|$, the $(i-1)$-th suffix is of the form Wa, with $b \neq a$, and W is right-maximal. But then, at some point our enumeration algorithm will visit the representation of W, with $|W| = \ell$. Since i is the first position of the range of Wb, we have that $i = \texttt{first}_\texttt{W}[\texttt{j}]$ for some $j \geq 2$, and Line 10 correctly sets $LCP[\texttt{first}_\texttt{W}[\texttt{j}]] = LCP[i] \leftarrow \ell = |W|$.

Correctness and completeness - Algorithm 2. Proving correctness and completeness of this procedure is much easier. It is sufficient to note that the `while` loop iterates over all ranges $\langle L, R\rangle$ of strings ending with # and not containing # anywhere else (note that we start from the range of # and we proceed by recursively left-extending this range with symbols different than #). Then, for each such range we set $LCP[L+1, R]$ to ℓ, the string depth of the corresponding string (excluding the final character #). It is easy to see that each leaf-type LCP value is correctly set in this way.

Complexity - Algorithm 1. If $\sigma > \sqrt{n}/\log^2 n$, then we run Beller et al's algorithm, which terminates in $O(n \log \sigma)$ time and uses $O(n) = o(n \log \sigma)$ bits of additional working space. Otherwise, we perform a linear number of operations on the stack since, as observed in Section 4, the number of Weiner links is linear. By the same analysis of Section 4, the operation in Line 11 takes $O(k \log \sigma)$ amortized time on wavelet trees, and sorting in Line 12 (using any comparison-sorting algorithm sorting m integers in $O(m \log m)$ time) takes $O(k \log \sigma)$ time. Note that in this sorting step we can afford storing in temporary space nodes $x_1, \ldots, x_k$ since this takes additional space $O(k\sigma \log n) = O(\sigma^2 \log n) = O(n/\log^3 n) = o(n)$ bits. All these operations sum up to $O(n \log \sigma)$ time. Since the stack always takes at most $O(\sigma^2 \log^2 n)$ bits and $\sigma \leq \sqrt{n}/\log^2 n$, the stack's size never exceeds $O(n/\log^2 n) = o(n)$ bits.

Complexity - Algorithm 2. Note that, in the `while` loop, we start from the interval of # and recursively left-extend with characters different than # until this is possible. It follows that we visit the intervals of all strings of the form $W\#$ such that # does not appear inside W. Since these intervals form a cover of $[1, n]$, their number (and therefore the number of iterations in the `while` loop) is also bounded by n. This is also the maximum number of operations performed on the queue/stack. Using Beller et al.'s implementation for the queue and a simple vector for the stack, each operation takes constant amortized time. Operating on the stack/queue takes therefore overall $O(n)$ time. For each interval $\langle L, R\rangle$ popped from the queue/stack, in Line 9 we set $R - L - 2$ LCP values. As observed above, these intervals form a cover of $[1, n]$ and therefore Line 9 is executed no more than n times. Line 13 takes time $O(k \log \sigma)$. Finally, in Line 14 we sort at most σ intervals. Using any fast comparison-based sorting algorithm, this costs overall at most $O(n \log \sigma)$ time.

As far as the space usage of Algorithm 2 is concerned, note that we always push just pairs interval/length ($O(\log n)$ bits) in the queue/stack. If $\sigma > n/\log^3 n$, we use Beller et al.'s queue, taking at most $O(n) = o(n \log \sigma)$ bits of space. Otherwise, the stack's size never exceeds $O(\sigma \cdot \log n)$ elements, with each element taking $O(\log n)$ bits. This amounts to $O(\sigma \cdot \log^2 n) = O(n/\log n) = o(n)$ bits of space usage. Moreover, in Lines 13-14 it holds $\sigma \leq n/\log^3 n$ so we can afford storing temporarily all intervals returned by `getIntervals` in $O(k \log n) = O(\sigma \log n) = O(n/\log^2 n) = o(n)$ bits. ◀

D Enumerating Suffix Tree Intervals in Succinct Space

We note that Algorithm 1 can be used to enumerate suffix tree intervals using just $o(n \log \sigma)$ space on top of the input BWT of a single text, when this is represented with a wavelet tree. This is true by definition in Belazzougui's procedure (Lines 4-14), but a closer look reveals that also Beller et al's procedure (Line 2) enumerates suffix tree intervals. At each step, we pop from the queue an element $\langle\langle L, R\rangle, |W|\rangle$ with $\langle L, R\rangle =$ `range(W)` for some string W, left-extend the range with all $a \in$ `BWT.rangeDistinct(L,R)`, obtaining the ranges `range(aW)` $= \langle L_a, R_a\rangle$ and, only if $LCP[R_a + 1] = \perp$, set $LCP[R_a + 1] \leftarrow |W|$ and push $\langle\langle L_a, R_a\rangle, |W| + 1\rangle$ on the stack. But then, since $LCP[R_a + 1] = |W|$ we have that the R_a-th and $(R_a + 1)$-th smallest suffixes start, respectively, with aUc and aUd for some $c < d \in \Sigma$, where $W = Uc$. This implies that aU is right-maximal, and the corresponding suffix tree node has at least two children labeled c and d; in particular, $\langle L_a, R_a\rangle$ is the range of $aW = aUc$, that is, one of these two children. Since we assume that we are working with a single text, $\langle L_a, R_a\rangle$ is the range of a suffix tree node if and only if $R_a > L_a$: in this case, we return this range. Completeness of the visit (i.e. we return all suffix tree nodes' intervals)

follows from the completeness of the LCP-array construction procedure (i.e. we fill all LCP values). To conclude note that, to perform our visit, we are using the array LCP to store null/non-null entries (i.e. $\perp$ or any other number); this shows that we do not actually need the LCP array: a bitvector of length $n = o(n \log \sigma)$ is sufficient (remember that Beller et al.'s strategy is used on large alphabets, so $n = o(n \log \sigma)$ holds).

Safe and Complete Algorithms for Dynamic Programming Problems, with an Application to RNA Folding

Niko Kiirala
Department of Computer Science and Helsinki Institute for Information Technology HIIT, University of Helsinki, Finland
kiirala@cs.helsinki.fi

Leena Salmela[1]
Department of Computer Science and Helsinki Institute for Information Technology HIIT, University of Helsinki, Finland
leena.salmela@helsinki.fi

Alexandru I. Tomescu[1]
Department of Computer Science and Helsinki Institute for Information Technology HIIT, University of Helsinki, Finland
alexandru.tomescu@helsinki.fi

Abstract

Many bioinformatics problems admit a large number of solutions, with no way of distinguishing the correct one among them. One approach of coping with this issue is to look at the partial solutions common to all solutions. Such partial solutions have been called *safe*, and an algorithm outputting all safe solutions has been called *safe and complete*. In this paper we develop a general technique that automatically provides a safe and complete algorithm to problems solvable by dynamic programming. We illustrate it by applying it to the bioinformatics problem of RNA folding, assuming the simplistic folding model maximizing the number of paired bases. Our safe and complete algorithm has time complexity $O(n^3M(n))$ and space complexity $O(n^3)$ where n is the length of the RNA sequence and $M(n) \in \Omega(n)$ is the time complexity of arithmetic operations on $O(n)$-bit integers. We also implement this algorithm and show that, despite an exponential number of optimal solutions, our algorithm is efficient in practice.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Applied computing → Life and medical sciences; Applied computing → Molecular structural biology; Theory of computation → Design and analysis of algorithms

Keywords and phrases RNA secondary structure, RNA folding, Safe solution, Safe and complete algorithm, Counting problem

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.8

Funding *Leena Salmela*: Supported by Academy of Finland (grants 308030 and 314170).

1 Introduction

Many bioinformatics problems ask to reconstruct a biological object based on some data observed from it. However, in many cases, such data is incomplete or erroneous, and the bioinformatics problem admits many solutions, with no way of distinguishing the correct one among them. One approach is to enumerate all solutions, or only the first k best solutions to the problem. Then, "one can apply more sophisticated quality criteria, wait for data to become available to choose among them, or present them all to human decision-makers" [11].

[1] Equal contribution

For example, this approach was applied to the reconstruction and analysis of metabolic pathways [5, 4] and gene regulation networks [24]. See the surveys [11, 12] for more details on best-k enumeration. However, in many other cases this approach is infeasible because there may be a large number of solutions, even exponential in the input length. In such cases, another approach is to consider only the partial solutions common to all optimal or near-optimal solutions. Due to the large number of solutions, it is often infeasible to enumerate all solutions and then find the parts common to all of them.

In the past this approach has been applied e.g. to sequence alignment. The aligned symbols common to all optimal and near-optimal alignments of two biological sequences were shown to be efficiently retrievable in [29, 28, 9, 14, 36]. In [29] these were called "reliable regions" and were shown to match in a significant proportion the true ones determined experimentally from tertiary structure superpositions.

More recently, this approach was applied to the contig assembly stage of the genome assembly problem. This requires to reconstruct strings as long as possible that are guaranteed to occur in any genome that could have generated the sequenced reads. In [26] such a contig assembly algorithm outputting strings common to all genome assembly solutions was called *safe*. A safe algorithm outputting *all* such safe strings was called *complete*. For the theoretical assembly model where the assembly solution is an Eulerian cycle, a safe and complete algorithm was suggested in [20]. Safe and complete algorithms were also studied for the gap filling problem [22], which is another stage of the genome assembly problem.

1.1 Contribution

In this paper we study another bioinformatics problem which admits a large number of optimal solutions in practice: the RNA folding problem. (We will formally introduce it and discuss it in greater detail in Sec. 1.2.) We will consider a basic model of RNA folding, namely the one maximizing the number of paired RNA bases, solvable by dynamic programming.

As opposed to the previous safe and complete algorithms tailored for each separate problem, we develop here a technique having the advantage that it can produce safe and complete algorithms for many problems solvable by dynamic programming. The only restriction is that the safe partial solutions it can find are required to be identifiable from the dynamic programming recurrence.

Our technique is based on counting solutions and works as follows: (i) for every substring of the input string, it counts the number of optimal solutions for it; (ii) for every substring of the input string, it counts the number of full optimal solutions that also use an optimal solution of the substring; (iii) for every possible partial solution identifiable from the dynamic programming recurrence (i.e., base pair, or unpaired base), it counts in how many full optimal solutions it appears. This base-pair maximization model for RNA folding can be solved in time $O(n^3)$, where n is the length of the RNA molecule. Our safe and complete algorithm for it reports the bases paired in all optimal solutions, and the bases unpaired in all optimal solutions. It has time complexity $O(n^3 M(n))$ and space complexity $O(n^3)$, where $M(n) \in \Omega(n)$ is the time complexity of arithmetic operations on $O(n)$-bit integers. Thus, our algorithm is also efficient, in the sense that its time complexity has no additional overhead with respect to computing a single optimal solution, except for the $M(n)$ factor.[2]

[2] Note that computing an optimal folding under the base-pair maximization model can be sped up to $O(n^3/\log n)$ using the Four Russians technique, see [13]. It would be interesting to study if this technique can be applied to speed up also our algorithm.

Our technique generalizes, and has some similarities with existing algorithms. For example, it generalizes the previous safe and complete algorithm [29] reporting aligned symbols appearing in all optimal alignments. However, our algorithm is strictly more general: in the sequence alignment problem each case of the recurrence depends only on one previously computed value, whereas our technique encompasses problems where several previously computed values are combined. Our algorithm is also similar to the inside-outside algorithm [6] used e.g. in EM estimation of probabilistic context-free grammars. The first step of our algorithm, counting the number of optimal solutions for any input substring, resembles the inside algorithm, whereas the second step is similar to the outside algorithm. Another approach whose aim is also to provide general solutions to problems solvable by dynamic programming is the Algebraic Dynamic Programming approach of Giegerich et al. [15]. This also provides a general algorithm for counting full solutions to dynamic programming algorithms. However, it does not count full solutions containing a given partial solution, and thus does not provide insights about safety.

Other dynamic programming problems where our techniques provides a safe and complete algorithm include, just to name a few: finding the parenthesization of an arithmetic expression [15] maximizing or minimizing its value, finding an optimal order in which to multiply a chain of matrices [15], finding a maximum-weight independent set in a node-weighted tree. In each such case the partial solutions that can be found as safe are the ones that are identifiable from the dynamic programming recurrences. For example, for the first problem, the parentheses common to all optimal solutions, and for the last problem, the nodes common to all optimal solutions. In Sec. 5 we will explain this more formally, in detail.

1.2 RNA folding

RNA is a single-stranded molecule that consists of bases `A`, `C`, `G`, and `U`. RNA folds upon itself so that base `A` pairs with `U` and `C` pairs with `G`. Base `G` can also pair with `U`, forming a wobble pair. The pairing of the bases forms the secondary structure of an RNA molecule. This structure is important for the biological function of many classes of RNA molecules, such as ribosomal RNA (rRNA) and transfer RNA (tRNA). Thus, in order to predict the biological function of an RNA molecule, we first need to predict its secondary structure.

When folding upon itself, an RNA molecule attempts to find a state which is energetically optimal. Traditional RNA secondary structure prediction algorithms, such as the Nussinov algorithm [21], have formulated the problem as finding a secondary structure which maximizes the number of paired bases. Zuker [35] extended this model to minimize free energy, which is biologically more accurate. Later, the free energy model has been refined when the contribution of different substructures has been better understood [30, 18]. However, none of these models is fully accurate or complete. For example, although some dynamic programming algorithms allow a limited form of pseudoknots (see e.g. [2]), many other dynamic programming solutions do not attempt to predict pseudoknots. Additionally, RNA does not always fold in the globally optimal shape, but instead sometimes takes a locally optimal shape, for example in the presence of other molecules.

The solution to the RNA secondary structure prediction problem is very rarely unique. In the maximum-pairs formulation, a large number of optimal solutions can usually be found. The minimum free energy formulation is more fine grained, but still, if one looks at optimal *and* near-optimal solutions, then again their number is large. There are several established ways of coping with the issue of multiple solutions. The algorithm by Wuchty et al. [33] enumerates all optimal and suboptimal solutions, and therefore its output can be too large to utilize. The Zuker algorithm [35] considers each possible pair of bases at

a time, and computes the secondary structure with best score that contains that pair of bases. However, this method might not find all optimal or close to optimal solutions, since the number of solutions can be exponential in the length of the RNA sequence, whereas the maximal number of different base pairs is only quadratic. When assuming the free energy model, McCaskill [19] calculates the equilibrium partition function for an RNA secondary structure. Based on this, the binding probabilities under the free energy model are computed for each pair of bases, over all possible secondary structures, not only the optimal ones. This can be done in cubic time, see [19] or [10]. More recently, Zakov et al. [34] showed how to reduce the time complexity of computing the binding probabilities under the free energy model to sub-cubic time, using ideas from Valiant's sub-cubic algorithm for context free grammar recognition, see [27, 1].

In order to clearly illustrate our technique for obtaining general safe and complete algorithms for dynamic programming problems, in this paper we focus on the simplistic model of RNA folding maximizing the number of paired bases. We implement our safe and complete algorithm for it and run experiments on real RNA molecules to show how it performs in practice.

In the Appendix, we also study experimentally the biological relevance of the notion of safety for the RNA folding problem. For this purpose, we also implement a trivial safe and complete algorithm, which, given all optimal and sub-optimal RNA foldings under the minimum free energy model reported by the ViennaRNA package [17], it finds the base pairs common to all such foldings. We observe that for the maximum pairs formulation, the precision of the safe sub-solutions are improved by up to 72% (median value), respectively, as compared to a single, entire, solution. For the minimum free energy formulation, the increases of precision of the safe sub-solutions are up to 22% (median value).

However, the running time of this naive safe and complete algorithm for the free energy model is prohibitively large for longer RNA sequences, whereas our algorithm for the maximum pairs formulation is efficient in practice. Since also the minimum free energy model is solvable by dynamic programming (using four dynamic programming tables), it is relevant as future work to apply our technique also to this more complex dynamic programming algorithm.

2 Preliminaries

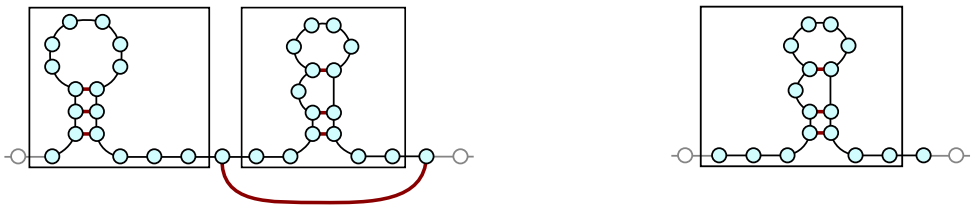
In this section we introduce the basic notation and definitions. We then describe Nussinov's dynamic programming algorithm for RNA folding, which is the starting point of our safe and complete algorithm.

In this paper we assume that $R \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}^n$ is a string over the RNA alphabet, indexed from 1 to n. We denote by $R(i, j)$ its substring starting at position i and ending at position j, inclusively ($1 \leq i \leq j \leq n$). The following RNA symbols *can be paired*: $\{\mathtt{A}, \mathtt{U}\}$, $\{\mathtt{C}, \mathtt{G}\}$, $\{\mathtt{G}, \mathtt{U}\}$. A *folding* of R is a set of pairs of indices in R (also called *base pairs*) $\{\{i_1, j_1\}, \{i_2, j_2\}, \ldots, \{i_t, j_t\}\}$ such that:

- $i_1, j_1, i_2, j_2, \ldots, i_t, j_t$ are all distinct, and
- $R(i_k)$ can be paired with $R(j_k)$, for all $k \in \{1, \ldots, t\}$.

A *pseudoknot* in such a folding consists of two pairs of indices, say $\{x, y\}$ and $\{z, w\}$ ($x < y$ and $z < w$), such that z is in the interval (x, y), but w is outside the interval (x, y). We next introduce the main problem tackled in this paper.

▶ **Problem 1** (Maximum pairs formulation). *Given a string $R \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}^n$, find a folding of R without pseudoknots maximizing the number of base pairs.*



(a) Pair $R(k)$ and $R(j)$. **(b)** Append $R(j)$ as unpaired.

Figure 1 Cases in Nussinov's algorithm.

A well-known algorithm for solving Problem 1 is Nussinov's algorithm [21]. This is based on dynamic programming, and runs in time $O(n^3)$ and requires $O(n^2)$ space. This algorithm is conceptually simple, and thus a good candidate to be extended into a safe and complete algorithm for Problem 1. Since all algorithms considered in this paper will be about foldings without pseudoknots, from now onwards by "folding" we mean one without pseudoknots.

Nussinov's algorithm defines $V(i,j)$ as the number of base pairs in an optimal folding of the substring $R(i,j)$. The maximum number of base pairs in a folding for R will be $V(1,n)$, and such a folding can be obtained by standard backtracking through the table V.

We initialize $V(i,j) = 0$ for $i \geq j$, since a sequence of length at most one cannot have any paired bases. The values $V(i,j)$ are computed with the following recurrence:

$$V(i,j) = \max \begin{cases} \max \Big\{ V(i,k-1) + V(k+1,j-1) + 1 \mid \\ \qquad i \leq k < j \text{ and } R(k) \text{ can be paired with } R(j) \Big\}, \\ V(i,j-1). \end{cases} \tag{1}$$

The two cases of this recurrence are also shown in Figure 1. In the first case, Nussinov's algorithm attempts to pair any base $R(k)$, where $k \in \{i, \ldots, j-1\}$, with the base $R(j)$. Whenever this pairing is possible, it uses the previously computed answers for the shorter substrings $R(i,k-1)$ and $R(k+1,j-1)$. The second case applies when $R(j)$ is not paired with any base $R(k)$: the algorithm takes the optimal folding of $R(i,j-1)$ and appends the base $R(j)$ to it as non-paired.

Nussinov's algorithm generates only a single optimal folding. However, it can be modified to output all optimal and suboptimal foldings, through a general technique introduced by [31] for the shortest path problem. This technique was applied to the RNA folding problem by Wuchty et al. [33]. In fact, [33] applies this technique to both models for RNA folding, maximum pairs and minimum free energy. The minimum free energy version is included in the RNAsubopt program contained in the ViennaRNA package [17].

3 A safe and complete algorithm for RNA folding

We say that a base pair $\{i,j\}$ is *safe*, if for any optimal folding $F = \{\{i_1,j_1\},\{i_2,j_2\},\ldots,\{i_t,j_t\}\}$ of R, we have that $\{i,j\} \in F$. Likewise, we say that an (unpaired) base i is *safe*, if for any such optimal folding F, we have that i is not paired with any other base. Formally, if for all optimal foldings F and all $\{x,y\} \in F$, we have $i \notin \{x,y\}$.

A safe and complete algorithm for Problem 1 is one outputting all safe base pairs and safe unpaired bases. We start by describing a trivial safe and complete algorithm. In Sections 3.1–3.3 we describe our efficient algorithm.

As mentioned above, Nussinov's algorithm can be extended to generate all optimal foldings. Let us assume that we have generated all of them, and that their number is s. We can then naively go through each optimal folding, and for each of the $O(n^2)$ possible base pairs of R, and each unpaired base, keep track if it appears in all optimal foldings. This trivial algorithm takes overall time $O(sn + n^2)$. The $O(sn)$ term comes from reading through every folding, and the $O(n^2)$ term comes from initializing the base-pairs matrix. Note however that this complexity is not polynomial in n, as there can be RNA strings of length n admitting an exponential number s of foldings (see our experimental results). Also in practice this complexity is prohibitive; for example, even some tRNA samples that are around 80-bases long have hundreds of thousands of solutions.

We will next describe a polynomial-time safe and complete algorithm, based on Nussinov's algorithm. The main idea is to derive, for every base pair, the number of optimal foldings in which it appears; and analogously, for every base, the number of optimal foldings in which it appears as unpaired. If in addition we also know the total number of optimal foldings, then we can decide safety based on checking if these numbers are equal or not.

These numbers will be derived in a sequence of steps, each computing a numeric table by dynamic programming. The table from each step will be used in the computation of the table for the next steps. These steps are:

1. Compute the number of optimal foldings of any substring $R(i, j)$ (Section 3.1)
2. Compute the number of optimal foldings of R, that use an optimal folding of the substring $R(i, j)$ (Section 3.2)
3. Compute the number of optimal foldings of R containing a given base pair, or a given unpaired base. (Section 3.3)

When analyzing the time complexity of this algorithm, we have to take into account that the number of foldings may be exponential in the input size. As such, we cannot assume constant-time integer arithmetic operations.

It is not hard to see that one upper bound on the total number of foldings of any RNA string of length n is 3^n. Any possible folding corresponds to a string of balanced parentheses (the paired bases), in which we have interspersed a third character standing for the non paired bases. The number of such ternary strings of length n is trivially at most 3^n. Such numbers take $O(\log_2(3^n)) = O(n)$ bits to represent. Tighter bounds on the number of optimal RNA foldings under the base-pair maximization model exist, see e.g. [16]. We note that additions on n-bit numbers can be performed in time $O(n)$, and we denote by $M(n) \in \Omega(n)$ the time needed to multiply or divide two $O(n)$ bit numbers.[3] Our safe and complete algorithm for Problem 1 will take $O(n^3 M(n))$ time and $O(n^3)$ space.

3.1 The number of optimal foldings of any subsequence

Given $i \leq j \in \{1, \ldots, n\}$, we let $S(i, j)$ be the number of optimal foldings of $R(i, j)$. The total number of optimal foldings of R will thus be obtained as $S(1, n)$. Computing $S(i, j)$ can be done by dynamic programming, with a recurrence analogous to Nussinov's recurrence (1), and also using Nussinov's table V.

As base case, observe that any sequence containing at most one base clearly has exactly one optimal folding. In other words, $S(i, j) = 1$ whenever $i \geq j$.

[3] For example, with the Schönhage-Strassen method [23], two n-bit numbers can be multiplied in time $M(n) = O(n \log n \log \log n)$.

As in Nussinov's recurrence, there are two ways of producing an optimal folding of $R(i,j)$: by pairing $R(j)$ with some $R(k)$, for some $k \in \{i,\ldots,j-1\}$, or appending $R(j)$ at the end of the optimal folding of $R(i,j-1)$. For each of these cases, and for each value of k, we obtain a distinct folding. As such, to obtain $S(i,j)$ we can sum up the number of optimal foldings in each case.

In the first case, when pairing $R(j)$ with $R(k)$, there are two substrings to consider: $R(i,k-1)$ and $R(k+1,j)$. Every possible combination of optimal foldings of these substrings will produce a unique optimal folding to the larger substring $R(i,j)$. Such combination thus contributes $S(i,k-1) \times S(k+1,j)$ optimal foldings to $R(i,j)$. In the second case, when subsequence $R(i,j)$ is produced by appending $R(j)$ to the optimal folding of $R(i,j-1)$, it can be produced in $S(i,j-1)$ ways.

This is summarized in the recurrence below. For conciseness, we use the Kronecker delta notation $\delta(x,y) = 0$ if $x \neq y$, and $\delta(x,y) = 1$ otherwise.

$$S(i,j) = \sum \begin{cases} \sum\limits_{i \leq k < j \mid R(k) \text{ can be paired with } R(j)} S(i,k-1) \times S(k+1,j-1) \times \\ \qquad \times \delta(V(i,j), V(i,k-1) + V(k+1,j-1) + 1), \\ S(i,j-1) \times \delta(V(i,j), V(i,j-1)). \end{cases} \tag{2}$$

The time complexity of computing all $S(i,j)$ values is $O(n^3 M(n))$: the computation goes over $O(n^2)$ substrings of R, tries $O(n)$ possible ways to construct each and performs at most a multiplication, requiring $O(M(n))$ time. The space complexity for computing $S(i,j)$ is $O(n^3)$, because we have a matrix with $O(n^2)$ cells, and each cell uses $O(n)$ bits.

3.2 The number of optimal global foldings that use an optimal folding of a subsequence

We will now compute a more subtle quantity. Given $i \leq j \in \{1,\ldots,n\}$, we denote by $T(i,j)$ the number of optimal foldings of R that, when restricted to the range from i to j, they are also an optimal folding of the substring $R(i,j)$. In other words, $T(i,j)$ is the total number of global optimal extensions that the optimal foldings of $R(i,j)$ have.

Since we are not considering pseudoknots, bases of R outside of $R(i,j)$ cannot pair with bases in $R(i,j)$. As such, each optimal folding of $R(i,j)$ is independent of the optimal folding of the rest of the string R. As such, if we know $T(i,j)$ and $S(i,j)$, then we can deduce that each optimal folding of $R(i,j)$ can be extended in $T(i,j)/S(i,j)$ ways into a global optimal folding of R. Let us denote the ratio $T(i,j)/S(i,j)$ by $c(i,j)$, and note that, by the above observation, $c(i,j)$ is integer.

The table T will be constructed starting from the full string $R(1,n)$ and moving towards single base substring. By definition, the number of optimal solutions for the full string is the same as the number of optimal solutions that use the full string. Thus, the base case for this algorithm is $T(1,n) = S(1,n)$. All other values of T will be initialized as 0.

The dynamic programming algorithms considered so far have been of collecting type: the required values for computing each dynamic programming cell have been computed earlier in the algorithm. The algorithm for computing T will be of distributing type instead. When this algorithm enters a cell, the value of that cell has already been computed correctly. This value is then used to incrementally derive the correct values of the cells that will be entered later.

When entering a cell (i,j), the algorithm checks all distinct ways how an optimal folding of $R(i,j)$ can be generated from optimal foldings of its substrings. This is done with the same logic as in Nussinov's recurrence (1). Since $T(i,j)$ is already computed correctly, then

Algorithm 1: Computing the table T.

```
1  for each substring R(i,j) in decreasing order of length do
2      c ← T(i,j)/S(i,j);
3      for each way of constructing an optimal folding of R(i,j) do
4          if this optimal folding pairs R(k) with R(j) then
5              a ← c × S(i,k−1) × S(k+1,j−1);
6              T(i,k−1) ← T(i,k−1) + a;
7              T(k+1,j−1) ← T(k+1,j−1) + a;
8          else
               // the optimal folding appends R(j) at the end of the optimal
                  folding for R(i,j−1)
9              a ← c × S(i,j−1);
10             T(i,j−1) ← T(i,j−1) + a;
11             T(j,j) ← T(j,j) + a;
```

we know $c(i,j)$. Namely, we know that each optimal folding of $R(i,j)$ can be extended in $c(i,j)$ ways into a global optimal folding of R. We can then add these ways to each substring of $R(i,j)$ used for generating an optimal folding of $R(i,j)$. Processing the substrings in decreasing order on length guarantees that the final correct value of T for any substring is available by the time that substring is processed.

See Algorithm 1 for the computation of T. For simplicity, in line 3 we write "iterate over each way of constructing an optimal folding of $R(i,j)$" meaning that we iterate over all cases from Nussinov's recurrence (1). That is, we consider all $k \in \{i, \ldots, j-1\}$ such that $R(k)$ can be paired with $R(j)$ and this folding gives an optimal value (i.e., $V(i,j) = V(i,k-1) + V(k+1,j-1) + 1$). This loop also considers the second case of Nussinov's recurrence in which the optimal folding of $R(i,j)$ appends $R(j)$ to the optimal folding for $R(i,j-1)$.

The time complexity of this algorithm is $O(n^3 M(n))$. It goes over $O(n^2)$ subsequences and tries $O(n)$ possible ways to construct each. For each such try, the algorithm may perform a division and a few multiplications requiring $O(M(n))$ time each. As before, the space complexity for computing $T(i,j)$ is $O(n^3)$, because we have a matrix with $O(n^2)$ cells, and each cell uses $O(n)$ bits.

3.3 The number of optimal foldings containing a base pair, or an unpaired base

An interesting property of the table T is that its diagonal values $T(i,i)$ represent how many of all optimal foldings contain that single-base substring. In other words, in how many foldings that base appears unpaired. This is useful for computing safety, so let us create a new array U out of the single-base values on the diagonal, having values $U(i) = T(i,i)$.

We will also create a table P, such that $P(i,j)$ is the number of optimal foldings of R that use the pair (i,j). This is not directly available from T, but it can be computed by going over all substrings of R, determining if their optimal foldings use the pair (i,j), and summing the number of optimal global foldings that extend this local folding. This is shown in Algorithm 2. We assume that all $P(i,j)$ cells are initialized as 0.

Algorithm 2: Computing the table R.

```
for each substring R(i, j) do
    c ← T(i, j)/S(i, j);
    for each way of constructing an optimal folding of R(i, j) do
        if this optimal folding pairs R(k) with R(j) then
            P(k, j) ← P(k, j) + c × S(i, k − 1) × S(k + 1, j − 1);
```

The time complexity of this algorithm is $O(n^3 M(n))$. It goes over $O(n^2)$ subsequences and tries $O(n)$ possible ways to construct each. For each such try, it performs one or two multiplications requiring $O(M(n))$ time each. As before, the total space complexity is $O(n^3)$.

It can be noted that this algorithm has no requirements on the order of iteration over the subsequences, and it only uses the values $T(i, j)$ and $S(i, j)$ for any given substring $R(i, j)$. Due to these properties, it is possible to join this and the algorithm for computing table T, so that both tables are computed in one pass.

With the tables U and P in place, it is straightforward to determine the safety of base pairs and unpaired bases in any given folding. A base pair $\{i, j\}$ $(i < j)$ is safe if and only if $P(i, j)$ is equal to the total number of optimal foldings, namely $S(1, n)$. Similarly, an unpaired base i is safe if and only if $U(i) = S(1, n)$.

4 Experimental results

We implemented the safe and complete algorithm of Section 3 in the Go language. To represent big integers we used the standard `math/big` library. Our implementation is available at `https://github.com/kiirala/rna-safe-complete` or through Go environment with command `go get keltainen.duckdns.org/rnafolding`.

We performed experiments to test this implementation on real RNA sequences, mainly focusing on the number of optimal solutions and the running time. Experiments about the biological relevance of the safe solutions can be found in the Appendix. To have a baseline for comparison, we also performed experiments with ViennaRNA's program RNAsubopt [17] implementing the minimum free energy formulation. RNAsubopt computes all optimal and suboptimal foldings that are in a certain range of the optimum. We chose this range to be 1 kcal/mol (option `-e 1`). In order to obtain a safe and complete algorithm for this minimum free energy formulation, given all s optimal and suboptimal solutions reported by RNAsubopt, we applied the trivial safe and complete algorithm mentioned at the beginning of Sec. 3, running in time $O(sn + n^2)$. RNAsubopt's output was piped to this trivial algorithm, so there was no overhead in writing / reading from disk.

We used data from the STRAND RNA database [3]. We analyzed tRNA [7, 32, 25], 5S ribosomal RNA [8, 32], 16S ribosomal RNA [8, 32] and 23S ribosomal RNA [8, 32]. When multiple database entries have exactly the same sequence, only one of such entries is analyzed. Any sequences that are marked as sequence fragments in the database are ignored and only complete sequences are included in the analysis. See Table 1 for the number of sequences analyzed and their average length. Out of the 117 sequences from the 23S rRNA dataset, only on 58 of them RNAsubopt finished running in the allocated time (on some of these 58, it ran for more than 96 hours). As such, all table rows "23S rRNA*" contain results only for these 58 sequences, and the rows "23S rRNA" contain results for all 117 sequences.

Table 1 Characteristics of the four datasets, and statistics on the number of reported solutions by each method. Large values are shown without decimal digits.

		Avg	#Solutions of Max Pairs			#Solutions of RNAsubopt		
	#Inputs	length	Median	Avg	Max	Median	Avg	Max
tRNA	633	78	975	$7 \cdot 10^{11}$	$4 \cdot 10^{14}$	3	6	421
5S rRNA	136	118	37301	$4 \cdot 10^{6}$	$2 \cdot 10^{8}$	10	15	109
16S rRNA	647	1536	$9 \cdot 10^{45}$	$3 \cdot 10^{93}$	$2 \cdot 10^{96}$	$4 \cdot 10^{5}$	$1 \cdot 10^{6}$	$4 \cdot 10^{9}$
23S rRNA*	58	2439	$1 \cdot 10^{86}$	$4 \cdot 10^{99}$	$2 \cdot 10^{101}$	$1 \cdot 10^{7}$	$8 \cdot 10^{7}$	$8 \cdot 10^{8}$
23S rRNA	117	2726	$4 \cdot 10^{89}$	$6 \cdot 10^{130}$	$7 \cdot 10^{132}$	–	–	–

Table 2 Running time (seconds) and average memory usage (MB) by each method. Columns "RNAsubopt – All" refer to RNAsubopt outputting all solutions only. Columns "RNAsubopt – Safe&Compl." include also the trivial safe and complete algorithm. Column "Med" means median.

	Max Pairs – Safe&Compl.				RNAsubopt – All				RNAsubopt – Safe&Compl.			
	Time			Mem.	Time			Mem.	Time			Mem.
	Med	Avg	Max	Max	Med	Avg	Max	Max	Med	Avg	Max	Max
tRNA	0.01	0.01	0.78	41	0.01	0.01	0.37	12	0.01	0.01	0.39	12
5S rRNA	0.04	0.05	0.11	10	0.03	0.04	0.08	10	0.03	0.04	0.1	10
16S rRNA	47	55	310	1964	108	2223	440727	175	131	3199	753752	175
23S rRNA*	256	205	327	2488	4180	25194	281149	273	5454	31457	348989	273
23S rRNA	240	242	931	3561	–	–	–	–	–	–	–	

In Table 1 we also show statistics on the number of optimal solutions to the maximum pairs formulation and the number of optimal and suboptimal solutions reported by RNAsubopt. In both cases, the number of solutions increases exponentially, with a much more rapid growth for the maximum pairs formulation. This also shows that the minimum free energy formulation is more stable, and thus more accurate.

In Table 2 we show the running time and memory usage for: our implementation of the safe and complete algorithm for the maximum pairs formulation, RNAsubopt reporting all optimal and suboptimal solutions, RNAsubopt plus the trivial safe and complete algorithm checking all RNAsubopt's solutions. The datasets tRNA, 5S rRNA and 16S rRNA were run on a machine with an Intel Xeon E5-2697 (2.7 GHz) CPU, with each process limited to 2 GiB memory. The dataset 23S rRNA was run on a machine with an Intel Xeon E3-1220 (3.1 GHz) CPU, with each process limited to 4 GiB memory.

For short RNAs (tRNA and 5S rRNA), the used resources are very similar, with RNAsubopt running slightly faster, likely because it outputs very few solutions. However, for longer RNAs (16S rRNA and 23S rRNA), our safe and complete algorithm for the maximum pairs formulation is significantly faster, even though in our case the number of optimal solutions is significantly larger. Moreover, the running time of RNAsubopt is much more variable for longer RNAs, with some inputs taking more than 96 hours (this particular RNA string had 726 Million solutions reported by RNAsubopt). Our safe and complete algorithm for the maximum pairs formulation finished in all cases in under 16 minutes, even though the numbers of optimal solutions can be of the order 10^{130}. We also observe a larger amount of memory used by our algorithm (up to 3.5GB for 23S rRNA), which is likely due to the fact that the numbers of solutions are much larger. However, this is still not prohibitive on modern machines.

5 Discussion and conclusions

To conclude, we would like to sketch how our technique generalizes to other problems solvable by dynamic programming. Precise algorithms need to be derived for each problem, but the technique introduced in this paper gives a blueprint for obtaining such algorithms. To make this more formal, suppose we have a dynamic programming recurrence of the form:

$$V(i,j) = \bigoplus_{\ell \in L_{i,j}} \left\{ f_\ell\big(V(i_1^\ell, j_1^\ell), \ldots, V(i_{k_\ell}^\ell, j_{k_\ell}^\ell)\big) \mid \varphi_\ell(i_1^\ell, j_1^\ell, \ldots, i_{k_\ell}^\ell, j_{k_\ell}^\ell) \right\} \tag{3}$$

where $L_{i,j}$ is some set of possible cases to consider in computing $V(i,j)$, $\bigoplus$ is an operation of these $|L_{i,j}|$ cases, for example min or max, values $V(i_1^\ell, j_1^\ell), \ldots, V(i_{k_\ell}^\ell, j_{k_\ell}^\ell)$ have previously been computed when computing $V(i,j)$, f_ℓ is a function on these previously computed values (for example sum), and φ_ℓ is a condition that must hold in order to consider the ℓ-th case in the recurrence. We also require that each of these $|L_{i,j}|$ cases leads to a different optimal solution (we observed at the beginning of Sec. 3.1 that this also holds for Nussinov's recurrence). It is immediate to verify that Nussinov's recurrence (1) fits into the above recurrence scheme (3). Note also that, even though recurrence (3) has only two parameters (i,j), our technique works for an arbitrary number of parameters.

Having all values $V(i,j)$ computed, one can obtain a safe and complete algorithm that can detect which cases allowed by the sets $L_{i,j}$ and the functions φ_ℓ are present in all optimal solutions. We need to apply exactly the same three steps outlined in Sec. 3. First, we need to compute the matrix $S(i,j)$ counting the number of optimal solutions of the partial input (i,j), then we need to compute the matrix $T(i,j)$ counting the number of optimal full solutions that, when restricted to (i,j), they are also an optimal solution for the partial input (i,j). Finally, one can compute a matrix analogous to our $P(i,j)$, which counts, for every possible partial solution that is a candidate to be safe, in how many cases of the recurrence it appears, and how many full optimal solutions these give rise to (information available from matrix T). These counts then indicate in how many full optimal solutions each partial solution appears, and thus indicate safety.

In this paper we applied our technique to the classical RNA folding recurrence of Nussinov. The choice of this problem was motivated by two factors: (1) it is simple enough to illustrate our technique and its general applicability, and (2) it is also a problem which on real data admits a large number of solutions, making the notion of safety practically relevant. For example, we observed RNA sequences admitting an exponential number of solutions. Despite this, our implementation of the safe and complete algorithm for it was efficient and ran in 16 minutes at most. Moreover, our experiments from the Appendix also show that safe sub-solutions for the maximum pairs formulation match the true biological folding with a precision of more than 40%.

However, we should also note that in the case of RNA folding, the minimum free energy model and its refinements are more biologically accurate. Our experiments on combining the output of RNAsubopt and the trivial algorithm for computing safety (in the Appendix) indicate that the safe sub-solutions for the minimum energy formulation have a significantly higher precision than for the maximum pairs formulation, reflecting the more biological accurate problem formulation. Thus, it would be interesting to derive efficient safe and complete algorithms for the minimum free energy model. Such algorithms would also need to be compared to McCaskill [19] approach deriving "predominance" of RNA folding substructures based on partition-function based methods.

References

1 Tatsuya Akutsu. Approximation and Exact Algorithms for RNA Secondary Structure Prediction and Recognition of Stochastic Context-free Languages. *Journal of Combinatorial Optimization*, 3(2):321–336, July 1999. doi:10.1023/A:1009898029639.

2 Tatsuya Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104(1):45–62, 2000. doi:10.1016/S0166-218X(00)00186-4.

3 Mirela Andronescu, Vera Bereg, Holger H Hoos, and Anne Condon. RNA STRAND: the RNA secondary structure and statistical analysis database. *BMC Bioinformatics*, 9(1):340, 2008. doi:10.1186/1471-2105-9-340.

4 Masanori Arita. Graph modeling of metabolism. *Journal-Japanese Society for Artificial Intelligence*, 15(4):703–710, 2000.

5 Masanori Arita. Metabolic reconstruction using shortest paths. *Simulation Practice and Theory*, 8(1-2):109–125, 2000.

6 J. K. Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132–S132, 1979. doi:10.1121/1.2017061.

7 Helen M. Berman, Wilma K. Olson, David L. Beveridge, John Westbrook, Anke Gelbin, Tamas Demeny, Shu-Hsin Hsieh, A.R. Srinivasan, and Bohdan Schneider. The nucleic acid database. A comprehensive relational database of three-dimensional structures of nucleic acids. *Biophys J*, 63(3):751–759, 1992.

8 Jamie J. Cannone, Sankar Subramanian, Murray N. Schnare, James R. Collett, Lisa M. D'Souza, Yushi Du, Brian Feng, Nan Lin, Lakshmi V. Madabusi, Kirsten M. Müller, Nupur Pande, Zhidi Shang, Nan Yu, and Robin R. Gutell. The Comparative RNA Web (CRW) Site: an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BMC Bioinformatics*, 3(1):2, January 2002. doi:10.1186/1471-2105-3-2.

9 Kun-Mao Chao, Ross C. Hardison, and Webb Miller. Locating well-conserved regions within a pairwise alignment. *CABIOS*, 9(4):387–396, 1993. doi:10.1093/bioinformatics/9.4.387.

10 Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

11 David Eppstein. K-Best Enumeration. *Bulletin of the EATCS*, 115, 2015. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/322.

12 David Eppstein. k-Best Enumeration. In *Encyclopedia of Algorithms*. Springer, Berlin, Heidelberg, 2015. doi:10.1007/978-3-642-27848-8_733-1.

13 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, January 2010. doi:10.1186/1748-7188-5-13.

14 Andreas Friemann and Stefan Schmitz. A new approach for displaying identities and differences among aligned amino acid sequences. *Comput Appl Biosci*, 8(3):261–265, June 1992.

15 Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004. doi:10.1016/j.scico.2003.12.005.

16 Ivo L. Hofacker, Peter Schuster, and Peter F. Stadler. Combinatorics of RNA secondary structures. *Discrete Applied Mathematics*, 88(1):207–237, 1998. Computational Molecular Biology DAM - CMB Series. doi:10.1016/S0166-218X(98)00073-0.

17 Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(26), November 2011. doi:10.1186/1748-7188-6-26.

18 David H Mathews, Jeffrey Sabina, Michael Zuker, and Douglas H Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of Molecular Biology*, 288(5):911–940, 1999. doi:10.1006/jmbi.1999.2700.

19 John S. McCaskill. The Equilibrium Partition Function and Base Pair Binding Probabilities for RNA Secondary Structure. *Biopolymers*, 29(6-7):1105–1119, 1990. doi:10.1002/bip.360290621.

20 Niranjan Nagarajan and Mihai Pop. Parametric Complexity of Sequence Assembly: Theory and Applications to Next Generation Sequencing. *Journal of Computational Biology*, 16(7):897–908, 2009.

21 Ruth Nussinov and Ann B. Jacobson. Fast Algorithm for Predicting the Secondary Structure of Single-Stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 77(11):6309–6313, November 1980. doi:10.1073/pnas.77.11.6309.

22 Leena Salmela and Alexandru I. Tomescu. Safely Filling Gaps with Partial Solutions Common to All Solutions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2019. doi:10.1109/TCBB.2017.2785831.

23 Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971. doi:10.1007/BF02242355.

24 Yu-Keng Shih and Srinivasan Parthasarathy. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12):i49–i58, 2012.

25 Mathias Sprinzl and Konstantin S. Vassilenko. Compilation of tRNA sequences and sequences of tRNA genes. *Nucleic Acids Research*, 33(Database issue):139–140, 2005.

26 Alexandru I. Tomescu and Paul Medvedev. Safe and Complete Contig Assembly Through Omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017. doi:10.1089/cmb.2016.0141.

27 Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975. doi:10.1016/S0022-0000(75)80046-8.

28 Martin Vingron. Near-optimal sequence alignment. *Curr. Opinion in Structural Biol.*, 6(3):346–352, June 1996. doi:10.1016/s0959-440x(96)80054-6.

29 Martin Vingron and Patrick Argos. Determination of reliable regions in protein sequence alignments. *Prot. Engin.*, 3(7):565–569, 1990. doi:10.1093/protein/3.7.565.

30 Amy E Walter, Douglas H Turner, James Kim, Matthew H Lyttle, Peter Müller, David H Mathews, and Michael Zuker. Coaxial stacking of helixes enhances binding of oligoribonucleotides and improves predictions of RNA folding. *Proceedings of the National Academy of Sciences*, 91(20):9218–9222, 1994. doi:10.1073/pnas.91.20.9218.

31 Michael S. Waterman and Thomas H. Byers. A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Mathematical Biosciences*, 77(1):179–188, 1985. doi:10.1016/0025-5564(85)90096-3.

32 John Westbrook, Zukang Feng, Li Chen, Huanwang Yang, and Helen M. Berman. The Protein Data Bank and structural genomics. *Nucleic Acids Research*, 31(1):489–491, January 2003. doi:10.1093/nar/gkg068.

33 Stefan Wuchty, Walter Fontana, Ivo L Hofacker, and Peter Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49(2):145–165, 1999. doi:10.1002/(SICI)1097-0282(199902)49:2<145::AID-BIP4>3.0.CO;2-G.

34 Shay Zakov, Dekel Tsur, and Michal Ziv-Ukelson. Reducing the worst case running times of a family of RNA and CFG problems, using Valiant's approach. *Algorithms for Molecular Biology*, 6(1):20, August 2011. doi:10.1186/1748-7188-6-20.

35 Michael Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244(4900):48–52, April 1989. doi:10.1126/science.2468181.

36 Michael Zuker. Suboptimal sequence alignment in molecular biology: Alignment with error analysis. *J Mol Biol*, 221(2):403–420, September 1991.

A Biological relevance of the notion of safety

We also compared how well the safe sub-solutions match the reference biological foldings from the STRAND RNA database. Each possible $\binom{n}{2}$ base pairs $\{i, j\}$, with $1 \leq i < j \leq n$, and each unpaired base i, with $1 \leq i \leq n$, was classified as:

- true positive (TP), if it is declared safe, and it appears in the reference folding,
- false positive (FP), if it is declared safe, but it does not appear in the reference folding,
- false negative (FN), if it is not declared safe, but it appears in the reference folding.

For each folding and each method (maximum pairs and minimum free energy), we computed the numbers TP, FP, FN. Then, we computed precision, as TP/(TP+FP), and recall, as TP/(TP+FN). Intuitively, precision measures how correct the safe sub-solutions are, and recall measures how much of the reference folding is correctly classified as safe. We also computed the proportion of all bases in the input RNA strings that are classified as safe. We report summary statistics on these metrics in Table 3. In Figure 2 we plot the precision and recall for each individual molecule.

In order to analyze how much the notion of safe sub-solution improves the correctness of the solutions, we also computed the precision and recall metrics when considering an entire solution as "safe" in the above definitions of TP, FP and FN. For this experiment, we ran our maximum pairs implementation and RNAsubopt so that they report a single optimal solution. We show these results in Table 4 and in Figure 2. Note that Figure 2 shows that all points for the single solutions lie close to the diagonal. This is due to the predicted solutions having roughly the same amount of paired bases as the biologically correct solutions and thus TP+FP is roughly equal to TP+FN.

Comparing Tables 3 and 4, we observe that for both methods, the notion of safe sub-solution is relevant, since it generally improves precision at the cost of recall. Figure 2 shows that the safety notion moves the points towards right. The increase in precision is more pronounced for the maximum pairs formulations, e.g., with an increase in median precision between 37% and 72% in the four data sets as shown in Table 5.

In our experiments, precision is not affected by the proportion of safe decisions. However, Figure 3 shows that high recall correlates with a high proportion of safe decisions. These observations hold for both our safe and complete algorithm and for RNAsubopt. For the sequences with a high proportion of safe decisions, the optimal folding tends to be unambiguous, which results in increased recall.

The maximum pairs model has a smaller proportion of safe sub-solutions than the minimum free energy model, because it admits significantly more solutions. As such, also its recall is lower. Table 3 and Figure 2 show that the precision and recall metrics of the maximum pairs model are lower than for the minimum free energy model. This is a consequence of the fact that the minimum free energy model is more biologically accurate.

Table 3 The precision and recall of the safe sub-solutions reported by each method, for all molecules in the datasets. We also show the proportion of bases of the RNA strings that are classified as safe.

	Max Pairs – Safe and Complete						RNAsubopt – Safe and Complete					
	Precision		Recall		Prop. Safe		Precision		Recall		Prop. Safe	
	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg	Med	Avg
tRNA	0.68	0.67	0.19	0.21	0.33	0.39	0.89	0.86	0.75	0.72	0.87	0.83
5S rRNA	0.58	0.53	0.11	0.15	0.24	0.28	0.84	0.79	0.65	0.59	0.83	0.76
16S rRNA	0.37	0.42	0.07	0.07	0.20	0.23	0.60	0.60	0.33	0.33	0.59	0.58
23S rRNA*	0.40	0.42	0.06	0.07	0.17	0.20	0.66	0.64	0.38	0.38	0.62	0.61
23S rRNA	0.41	0.43	0.06	0.06	0.15	0.18	–	–	–	–	–	–

Table 4 The precision and recall of a single solution reported by each method, for all molecules in the datasets.

	Max Pairs – Single				RNAsubopt – Single			
	Precision		Recall		Precision		Recall	
	Med	Avg	Med	Avg	Med	Avg	Med	Avg
tRNA	0.45	0.45	0.41	0.41	0.83	0.80	0.85	0.82
5S rRNA	0.34	0.36	0.30	0.33	0.76	0.71	0.75	0.70
16S rRNA	0.27	0.27	0.23	0.24	0.49	0.49	0.47	0.47
23S rRNA	0.28	0.29	0.25	0.25	0.54	0.55	0.53	0.52

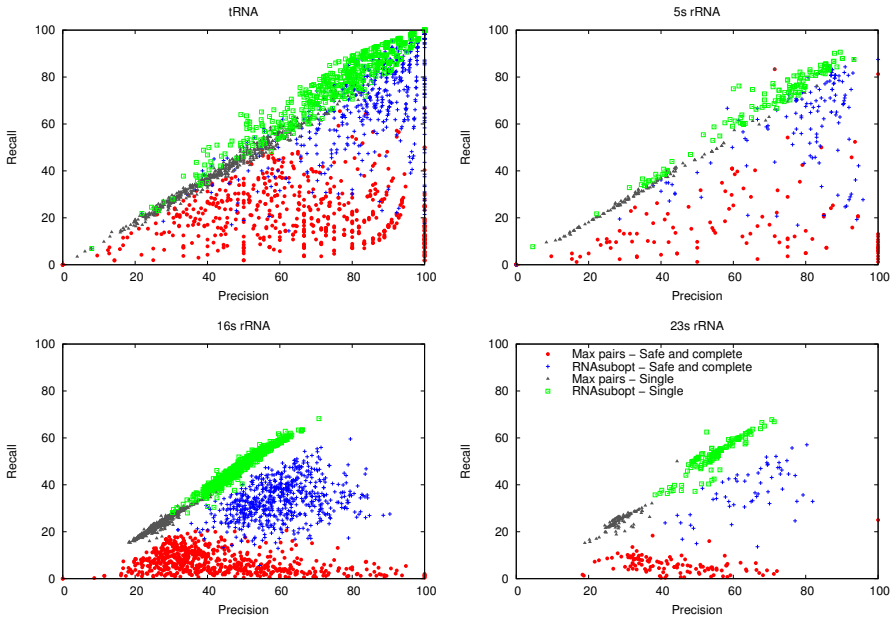


Figure 2 The precision and recall of each molecule in the datasets.

Table 5 The relative increase of precision and recall of the safe sub-solutions, as compared to a single solution, for all molecules in the datasets.

	Max Pairs				RNAsubopt			
	Precision		Recall		Precision		Recall	
	Med	Avg	Med	Avg	Med	Avg	Med	Avg
tRNA	52%	50%	-52%	-48%	8%	6%	-12%	-12%
5S rRNA	72%	49%	-65%	-54%	10%	11%	-13%	-16%
16S rRNA	37%	54%	-72%	-70%	22%	22%	-30%	-30%
23S rRNA	41%	47%	-75%	-73%	22%	17%	-28%	-28%

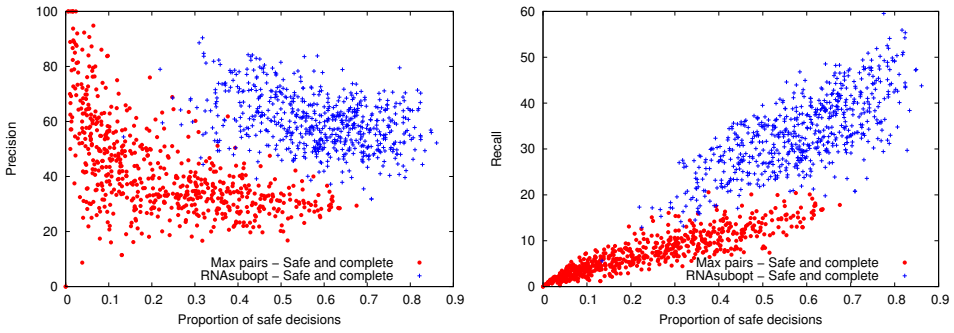


Figure 3 Precision and recall as a function of the proportion of safe decisions for each molecule in the 16S rRNA dataset.

Conversion from RLBWT to LZ77

Takaaki Nishimoto
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan
takaaki.nishimoto@riken.jp

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project, Tokyo, Japan
yasuo.tabei@riken.jp

Abstract

Converting a compressed format of a string into another compressed format without an explicit decompression is one of the central research topics in string processing. We discuss the problem of converting the run-length Burrows-Wheeler Transform (RLBWT) of a string into Lempel-Ziv 77 (LZ77) phrases of the reversed string. The first results with Policriti and Prezza's conversion algorithm [Algorithmica 2018] were $O(n \log r)$ time and $O(r)$ working space for length of the string n, number of runs r in the RLBWT, and number of LZ77 phrases z. Recent results with Kempa's conversion algorithm [SODA 2019] are $O(n/\log n + r \log^9 n + z \log^9 n)$ time and $O(n/\log_\sigma n + r \log^8 n)$ working space for the alphabet size σ of the RLBWT. In this paper, we present a new conversion algorithm by improving Policriti and Prezza's conversion algorithm where dynamic data structures for general purpose are used. We argue that these dynamic data structures can be replaced and present new data structures for faster conversion. The time and working space of our conversion algorithm with new data structures are $O(n \min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$ and $O(r)$, respectively.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Burrows-Wheeler Transform, Lempel-Ziv Parsing, Lossless Data Compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.9

1 Introduction

Converting a compressed format of a string into another compressed format without an explicit decompression is one of the central research topics in string processing. Examples are conversions from the Lempel-Ziv 77 (LZ77) Phrases of a string into a grammar-based encoding [10, 16], from a grammar-based encoding of a string into LZ78 phrases [2, 1] and from a grammar-based encoding of a string into another grammar-based encoding [17]. Such conversion is beneficial when one intends to process a compressed string in a different compressed format without decompressing it.

LZ77 parsing, proposed in 1976 [13], is one of the most popular lossless data compression algorithms and is a greedy partition of a string such that each phrase is a previous occurrence of a substring or a character not occurring previously in the string. The *run-length Burrows-Wheeler transform (RLBWT)* [5] is a recent popular lossless data compression algorithm with a run-length encoded permutation of a string.

Policriti and Prezza [15] proposed the first conversion algorithm from the RLBWT of an input string into the LZ77 phrases of the reversed string. The basic idea with this algorithm is to carry out backward searches on the RLBWT and find a previous occurrence of each phrase using red-black trees storing a sampled suffix array and dynamic data structure for solving *the searchable partial sums with the indels problem* (e.g., [4, 9]). Since these data structures are updated frequently for scanning the string in the RLBWT format, the running time and working space are $O(n \log r)$ and $O(r)$ words, respectively, for string length n and number of runs r (i.e., the number of continuous occurrences of the same characters).

Table 1 Summary of conversion algorithms from the RLBWT to LZ77.

Algorithm	Conversion time	Working space (words)
Policriti and Prezza (Thm. 7) [15]	$O(n\sqrt{\frac{\log r}{\log\log r}})$ or expected $O(n \log\log n)$	$O(r)$
Kempa (Thm. 7.3) [12]	$O(n/\log n + r\log^9 n + z\log^9 n)$	$O(n/\log_\sigma n + r\log^8 n)$
This study	$O(n\min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$	$O(r)$

Their running time can be improved using a more faster dynamic predecessor instead of the red-black trees. As a result, we can achieve expected $O(n \log\log n)$ time and $O(n\sqrt{\frac{\log r}{\log\log r}})$ time using *y-fast trie* [18] and Beame and Fich's dynamic predecessor [3], respectively.

Kempa [12] recently presented a conversion algorithm from the RLBWT to LZ77, which runs in $O(n/\log n + r\log^9 n + z\log^9 n)$ time and $O(n/\log_\sigma n + r\log^8 n)$ working space for the number z of LZ77 phrases and the alphabet size σ of the string in the RLBWT format. While the algorithm runs in $o(n)$ time and working space, especially when r and z are small (e.g., $r, z = O(n/\log^9 n)$), the working space of the algorithm is larger than that of Policriti and Prezza's algorithm in many cases.

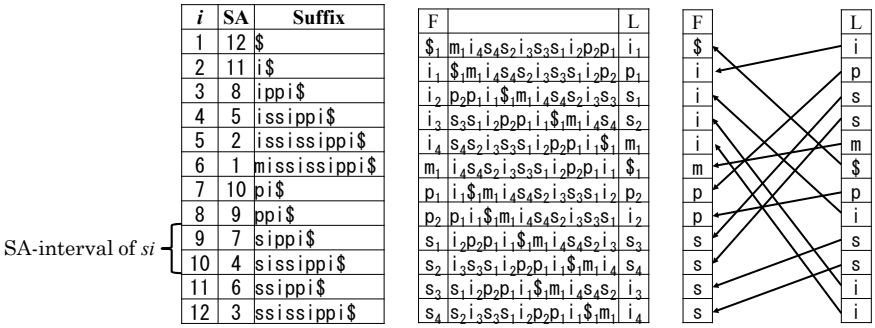
In this paper, we present a new conversion algorithm from the RLBWT to LZ77 by improving Policriti and Prezza's algorithm. Their algorithm adopts dynamic data structures for four queries comprising backward search, LF function, access queries on the RLBWT, and so-called *range more than query (RMTQ)*. We argue that these dynamic data structures can be replaced for answering those queries and present new data structures for faster conversion. Our algorithm runs in $O(n\min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$ deterministic time and $O(r)$ working space, which improves their algorithm (see Table 1 for a summary of conversion algorithms).

2 Preliminaries

Let Σ be an ordered alphabet of size σ, T be a string of length n over Σ and $|T|$ be the length of T. Let $T[i]$ be the i-th character of T and $T[i..j]$ be the substring of T that begins at position i and ends at position j. The $T[i..]$ denotes the suffix of T beginning at position i, i.e., $T[i..n]$. Let T^R be the reversed string of T, i.e., $T^R = T[n]T[n-1]\cdots T[1]$. For two integers i and j $(i \leq j)$, $[i, j]$ represents $\{i, i+1, \ldots, j\}$. For two strings T and P, $T \prec P$ is that T is lexicographically smaller than P. $Occ(T, P)$ denotes all the occurrence positions of string P in string T, i.e., $Occ(T, P) = \{i \mid P = T[i..(i+|P|-1)], i \in [1, n-|P|+1]\}$. *Right occurrence* p of substring $T[i..j]$ is a subsequent occurrence position of $T[i..j]$ in T, i.e., any $p \in Occ(T[i+1..], T[i..j])$.

For a string T, character c, and integer i, $\mathsf{rank}(T, c, i)$ returns the number of a character c in $T[..i]$, i.e., $\mathsf{rank}(T, c, i) = |Occ(T[..i], c)|$. $\mathsf{access}(T, i)$ returns $T[i]$. $\mathsf{select}(T, c, i)$ returns the position of the i-th occurrence of a character c in T. If the number of occurrences of c in T is smaller than i, it returns $n+1$, i.e., $\mathsf{select}(T, c, i) = \min(\{j \mid |Occ(T[..j], c)| \geq i, j \in [1, n]\} \cup \{n+1\})$ where $\min\{S\}$ returns the minimum value in a given set S.

For an integer x and set S of integers, a *predecessor* query $\mathsf{pred}(S, x)$ returns the number of elements that are no more than x in S, i.e., $\mathsf{pred}(S, x) = |\{y \mid y \leq x, y \in S\}|$. A *predecessor data structure* of S supports predecessor queries on S. For an integer array D and two positions i, j $(i \leq j)$ on D, a *range maximum query* (RMQ) $\mathsf{RMQ}(D, i, j, k)$ returns the maximum value in $D[i..j]$, i.e., $\mathsf{RMQ}(D, i, j) = \max D[i..j]$, where $\max\{S\}$ returns the maximum value in a given set S.



i	SA	Suffix
1	12	$
2	11	i$
3	8	ippi$
4	5	issippi$
5	2	ississippi$
6	1	mississippi$
7	10	pi$
8	9	ppi$
9	7	sippi$
10	4	sissippi$
11	6	ssippi$
12	3	ssissippi$

F		L
$_1$	m$_1$i$_4$s$_4$s$_2$i$_3$s$_3$s$_1$i$_2$p$_2$p$_1$	i$_1$
i$_1$	$$_1m_1i_4s_4s_2i_3s_3s_1i_2p_2$	p$_1$
i$_2$	p$_2$p$_1$i$_1$$$_1m_1i_4s_4s_2i_3s_3$	s$_1$
i$_3$	s$_3$s$_1$i$_2$p$_2$p$_1$i$_1$$$_1m_1i_4s_4$	s$_2$
i$_4$	s$_4$s$_2$i$_3$s$_3$s$_1$i$_2$p$_2$p$_1$i$_1$$$_1$	m$_1$
m$_1$	i$_4$s$_4$s$_2$i$_3$s$_3$s$_1$i$_2$p$_2$p$_1$i$_1$	$$_1$
p$_1$	i$_1$$$_1m_1i_4s_4s_2i_3s_3s_1i_2$	p$_2$
p$_2$	p$_1$i$_1$$$_1m_1i_4s_4s_2i_3s_3s_1$	i$_2$
s$_1$	i$_2$p$_2$p$_1$i$_1$$$_1m_1i_4s_4s_2i_3$	s$_3$
s$_2$	i$_3$s$_3$s$_1$i$_2$p$_2$p$_1$i$_1$$$_1m_1i_4$	s$_4$
s$_3$	s$_1$i$_2$p$_2$p$_1$i$_1$$$_1m_1i_4s_4s_2$	i$_3$
s$_4$	s$_2$i$_3$s$_3$s$_1$i$_2$p$_2$p$_1$i$_1$$$_1m_1$	i$_4$

F
$
i
i
i
i
m
p
p
s
s
s
s

L
i
p
s
s
m
$
p
i
s
s
i
i

Figure 1 Example for SA (left), F, L (center), and LF function (right) of $T = mississippi\$$.

Our computation model is a unit-cost word RAM with a machine word size of $\Omega(\log_2 n)$ bits. We evaluate the space complexity in terms of the number of machine words. A bitwise evaluation of space complexity can be obtained with a $\log_2 n$ multiplicative factor. Throughout this paper, logarithm to base 2 is used if the logarithmic base is not indicated.

2.1 Suffix Array (SA) and SA interval

The suffix array (SA) [14] of string T is an integer array of size n such that $\mathsf{SA}[i]$ stores the starting position of i-th suffix of T in lexicographical order. Formally, SA is the permutation of $[1..n]$ such that $T[SA[1]..] \prec \cdots \prec T[SA[n]..]$ holds. For example, $T =$ **mississippi$** and $\mathsf{SA} = 12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3$. The left figure in Figure 1 depicts the sorted suffixes of T and SA of T.

Since suffixes in the suffix array are sorted in the lexicographical order, suffixes with prefix Y occur continuously on an interval in the suffix array. We call this interval *SA interval* of Y. Formally, the SA interval of string Y is interval $[b, e]$ such that $p \in \mathsf{SA}[b..e]$ holds for all $p \in Occ(T, Y)$. For the above example, the SA intervals of **si** and **p** are $[9, 10]$ and $[7, 8]$, respectively.

2.2 BWT and backward search

The Burrows Wheeler Transform (BWT) [5] of string T is a permutation of T obtained as follows. We sort all the n rotations of T in lexicographical order and take the last character at each rotation in sorted order. L is the permutation of T such that for all $i \in [1..n]$, $L[i] = T[\mathsf{SA}[i] - 1]$ holds if $\mathsf{SA}[i] \neq 1$ and $L[i] = T[n]$ holds otherwise. Similarly, let F be a permutation of T that consists of the first characters in rotations in sorted order, i.e., $F[i] = T[\mathsf{SA}[i]]$ for all $i \in [1..n]$. The middle table in Figure 1 represents F, L, and sorted rotations of $T =$ **mississippi$**.

A property of BWT is that the i-th occurrence of character c in F corresponds to the i-th occurrence of c in L. In other words, let x and y be the positions of the i-th occurrence of c in F and L, respectively. Then $F[x]$ is a character of position p in T when $L[y]$ is a character of the same position in T. The LF function receives a position y in L as input and returns such corresponding position x in F. Since F consists of the sorted characters, $\mathsf{LF}(y) = C[L[y]] + \mathsf{rank}(L, L[y], i)$ holds for all $y \in [1, n]$, where $C[c]$ is the number of occurrences of characters lexicographically less than $c \in \Sigma$ in L.

Backward search computes the SA interval of cY for a given SA interval of Y and character c using the BWT of T. Function $\mathsf{backward\_search}(T, b, e, c)$ takes SA interval $[b, e]$ of Y and character $c \in \Sigma$ as input and returns SA interval $[b', e']$ of cY. We compute backward

search using LF, rank, and select queries on L. The LF function receives the first and last occurrences of c in $L[b..e]$ and returns the first and last positions of the SA interval of cY because $L[i]$ represents the character preceding $F[i]$ on T for an integer $i \in [2..n]$. We can compute the first and last occurrences of c using rank and select queries on L.

Formally, let $x = \mathsf{rank}(L, c, b-1)$ and $y = \mathsf{rank}(L, c, e)$; $b' = \mathsf{LF}(\mathsf{select}(L, c, x+1))$ and $e' = \mathsf{LF}(\mathsf{select}(L, c, y))$ hold if the length of the SA interval for cY is not zero.

2.2.1 Run-length encoding and RLBWT

For a string T, *Run-length encoding* $\mathsf{RLE}(T)$ is a partition of T into substrings $f_1, f_2, \ldots, f_r$ such that each f_i is a maximal repetition of the same character in T. We call each f_i a *run*.

The RLBWT of T is the BWT encoded by the run-length encoding, i.e., $\mathsf{RLE}(L)$. The RLBWT is stored in $O(r)$ space because each run in the RLBWT can be encoded into a pair of integers c and ℓ, where c is the character and ℓ is the length of the run. We call such a representation the *compressed form* of the RLBWT.

2.3 LZ77

For a string T, LZ77 parsing [13] of the reversed T greedily partitions T into substrings (phrases) $f_z, f_{z-1}, \ldots, f_1$ in right-to-left order such that each phrase is either (i) copied from a subsequent substring in T (target phrase) or (ii) an explicit character (character phrase). We denote LZ77 phrases of the reversed T as $\mathsf{LZ}(T^R)$.

Formally, let i' be the ending position of f_i for $i \in [1, z]$, i.e., $i' = |f_{i'-1} \cdots f_1| + 1$. Then $f_1 = T[n]$, and f_i is $T[i']$ for $i \in [2, z]$ if $T[i']$ is a new character (i.e., $Occ(T[i'+1..], T[i']) = \emptyset$); otherwise, f_i is the longest suffix P of $T[..i']$, which has right occurrences in T (i.e., $|P| = \max\{\ell \mid Occ(T[i'-\ell+2..], T[i'-\ell+1..i']) \neq \emptyset, \ell \in [1, i']\}$).

We can store LZ77 phrases in $O(z)$ space because we encode a target phrase into the pair $\langle p, \ell \rangle$ of the right occurrence p and length ℓ of the phrase. We call such representation the *compressed form* of LZ77. For example, let $T = cbbbbbabaababa$. Then $\mathsf{LZ}(T^R) = c, bbbb, baba, aba, b, a$, and the compressed form of $\mathsf{LZ}(T^R)$ is $c, \langle 3, 4\rangle, \langle 11, 4\rangle, \langle 12, 3\rangle, b, a$.

3 Policriti and Prezza's conversion algorithm

Policriti and Prezza's conversion algorithm [15] converts a compressed string of T in the RLBWT format to another compressed string of T^R in the LZ77 format while using data structures in $O(r)$ space. The data structures support four queries: backward search, the LF function, access queries on the RLBWT L, and RMTQ on the suffix array of T. The $\mathsf{RMTQ}(D, i, j, k)$ takes value k, interval $[b, e]$, and array D as inputs and returns a value larger than k on interval $[b, e]$ in D.

The algorithm extracts the original string from L in right-to-left order using the LF function and access queries on L and computes LZ77 phrases sequentially using backward search and RMTQ. In each step, the algorithm extracts a suffix of the original string (i.e., current extracted string) and it outputs the LZ77 phrase called *current pattern* in the suffix. In each step, the following two conditions for the current pattern are guaranteed: (i) the current pattern has at least one right occurrence or is the string of length 0; (ii) the length of the current pattern is no less than that of the following current pattern (i.e., the next computed LZ77 phrase).

For computing the current extracted string in each step, the algorithm computes (i) the next character preceding the current extracted string, (ii) computes the SA interval corresponding to the current pattern using the backward search, and (iii) finds any right

Algorithm 1: Policriti and Prezza's conversion algorithm.

```
Data: RLBWT L of T and position y of T[n] on L
Result: LZ(T^R)
(b, e, p, ℓ, x) ← (1, n, −1, 1, n);                          /* Initialization */
while x ≥ 1 do
    /* Let P be T[x..x+l-1]                                                    */
    c ← access(L, y);                                          /* Access T[x] */
    [b′, e′] ← backward_search(T, b, e, c);    /* Compute the SA interval of P */
    p′ ← RMTQ(SA, b′, e′, x + 1);
    if p′ = −1 then          /* Any right occurrence of P was not found */
        if ℓ > 1 then
            output ⟨p, ℓ − 1⟩;
        else
            output c;
            (x, y) = (x − 1, LF(y));
        (b, e, p, ℓ) ← (1, n, p′, 1);
    else
        (b, e, p, ℓ, x, y) ← (b′, e′, p′, ℓ + 1, x − 1, LF(y));
```

occurrence of the current pattern in the SA interval using RMTQ. If such right occurrence of the extended pattern does not exist, the algorithm outputs the current pattern as the next LZ77 phrase. When the length of the current pattern is zero, the next phrase is the next character. The algorithm repeats the above step until it extracts the whole string and outputs LZ77 phrases of T^R. Algorithm 1 shows a pseudo code of the algorithm.

The algorithm uses two dynamic data structures: one for supporting backward search, the LF function, and access queries on L in $O(\log r)$ time and $O(r)$ space; the other for supporting RMTQ in $O(\log r)$ time and $O(r)$ space, which is detailed in the next subsection.

3.1 RMTQ data structure

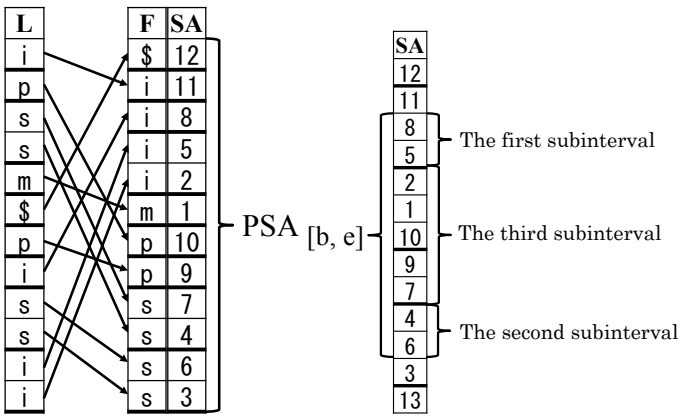


Figure 2 Left figure illustrates partitioned suffix array (PSA). Bold horizontal lines on suffix array represent partitions on PSA; hence, PSA is $(12), (11), (8), (5, 2), (1), (10), (9), (7, 4), (6, 3)$. Right figure illustrates three subintervals used in Section 4.1.

Policriti and Prezza presented an RMTQ data structure for fixed k, which can be updated when k is decremented. The construction for RMTQ data structure partitions the suffix array of T into subarrays for every run in L by the LF function, resulting in r subarrays in total. Since the i-th occurrence of any character c in F corresponds to the i-th occurrence of c in L, the characters on every run in L also occur continuously on F. The F can be partitioned into r substrings such that each substring corresponds to a run in L. We call such subarrays *partitioned suffix arrays* (PSAs). The left in Figure 2 shows an example for the PSA of T in Figure 1.

The data structure does not store the whole PSA. Instead, it stores only the first and last values larger than k on each subarray of the PSA and their corresponding positions on it. The red-black tree is used to store those positions. We call such a first position (respectively, last position) on the i-th subarray for fixed k a *k-open position* (respectively, *k-close position*) on the i-th subarray, which is denoted as $\mathsf{open}(i,k)$ (respectively, $\mathsf{close}(i,k)$).

$\mathsf{RMTQ}(D,b,e,k)$ using the data structure is divided into two cases according to the relationship between the query interval $[b,e]$ and PSA: (A) there exists a subarray $\mathsf{SA}[p..q]$ of the PSA completely including interval $[b,e]$ (i.e., $p<b\leq e<q$ holds); and (B) such a subarray does not exist. Both cases are detailed as follows.

For case (B), the query interval is partitioned into several subarrays of the PSA and contains either a prefix or suffix of each subarray. Therefore, the interval contains at least one of the k-open and k-close positions if and only if the interval contains a value larger than k. We select an answer of RMTQ from the k-open and k-close positions on subarrays in the PSA in $O(\log r)$ time using the red-black tree storing the set of k-open and k-close positions.

For case (A), $\mathsf{RMTQ}(D,i,j,k)$ is computed using its computation result in the previous iteration of Algorithm 1. The query interval $[b,e]$ represents the SA interval of string P, and the length of P is at least two because the SA interval of a character does not satisfy case (A). This means that the query interval in the previous iteration represents the SA interval of $P[2..]$, and Algorithm 1 computes the answer p ($\neq -1$) in the previous iteration. Thus, in case (A), $(p-1)$ is the answer for P because $P[1]$ is the character on L such that p is at the same position on the suffix array.

Formally, let $\mathsf{RLE}(L)=L_1,L_2,\ldots,L_r$, $p(i)$ be the starting position of L_i in L (i.e., $p(i)=|L_1\cdots L_i|-|L_i|+1$), and X be the permutation of $[1..r]$ such that $\mathsf{LF}(p(X[1]))<\ldots<\mathsf{LF}(p(X[r]))$ holds. Then the PSA of T is r subarrays $s_1,\ldots,s_r$ such that $s_i=\mathsf{SA}[\mathsf{LF}(p(X[i]))..\mathsf{LF}(p(X[i]))+|L_{X[i]}|-1]$ holds for all $i\in[1,r]$. Let $\mathsf{open}(i,k)=\min(\{n+1\}\cup\{j\mid \mathsf{SA}[j]\geq k, j\in[s_i..s_{i+1}-1]\})$ and $\mathsf{close}(i,k)=\max(\{0\}\cup\{j\mid \mathsf{SA}[j]\geq k, j\in[s_i..s_{i+1}-1]\})$ for $k\in[1,n]$ and $i\in[1,r]$. Let $\mathcal{T}_k$ be the set of k-open and close positions in the suffix array of T, i.e, $\mathcal{T}_k=\{\mathsf{open}(1,k),\mathsf{close}(1,k)\ldots,\mathsf{open}(r,k),\mathsf{close}(r,k)\}$. Then the following lemma holds.

▶ **Lemma 1** ([15]). *Let P be a substring of T starting at a position k and $\mathsf{SA}[b..e]$ be the SA interval of P. (1) In case (A), the length of P is at least 2 and $\mathsf{RMTQ}(SA,b,e,k)$ can return $\mathsf{RMTQ}(SA,b',e',k+1)-1$, where $\mathsf{SA}[b..e]$ is the SA interval of $P[2..]$. (2) In case (B), if $\mathcal{T}_k\cap[b..e]\neq\emptyset$ holds, then $\mathsf{RMTQ}(SA,b,e,k)$ can return the value at any position in $\mathcal{T}_k\cap[b..e]$; otherwise $\mathsf{RMTQ}(\mathsf{SA},b,e,k)=-1$ holds.*

4 Data structures for faster conversion

We improve the query time in the data structure for backward search, the LF function, access query on L, and RMTQ for case (B) by presenting two novel data structures: one supports the RMTQ for case (B); and the other supports backward search, the LF function, and access

Algorithm 2: Our $\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$ algorithm for case (B).

Result: $\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$
$\hat{b} \leftarrow \mathsf{pred}(Z, b)$; `/* Get the index` $\hat{b}$ `of the subarray on the position b */`
$\hat{e} \leftarrow \mathsf{pred}(Z, e)$; `/* Get the index` $\hat{e}$ `of the subarray on the position e */`
$(x^{close}, v^{close}) \leftarrow M_k^{close}[\hat{b}]$;
$(x^{open}, v^{open}) \leftarrow M_k^{open}[\hat{e}]$;
$v^\circ \leftarrow \mathsf{RMQ}(M, \hat{b}+1, \hat{e}-1)$;
if $x^{close} \in [b, e]$ **then**
  return v^{close};
else if $x^{open} \in [b, e]$ **then**
  return v^{open};
else if $v^\circ > k$ *and* $\hat{b}+1 \leq \hat{e}-1$ **then**
  return v°;
else
  return -1;

query on L. Our data structures use static predecessor data structures internally and improve four query times by choosing the predecessor data structure with the fastest (estimated) query time. Finally, we show the results of our data structures, which are summarized in Table 1.

4.1 RMTQ data structure in case (B)

Our RMTQ data structure for fixed k consists of four arrays of length r: *k-open array* M_k^{open}, *k-close array* M_k^{close}, *max value array* M, and *starting position array* Z. Data structures for the RMQ and predecessor query are built on M and Z, respectively.

The i-th element of the k-open array (respectively, k-close array) stores the pair of k-open position (respectively, k-close position) and its corresponding value on the i-th subarray of the PSA. The k-open and k-close arrays can be updated when k is decremented. The i-th element of the max value array M stores the maximum value on the i-th subarray of the PSA. The i-th element of the starting position array stores the starting position of the i-th subarray on the PSA (i.e., $Z[i] = p(X[i])$).

Our algorithm for RMTQ ($\mathsf{RMTQ}(\mathsf{SA}, b, e, x)$ algorithm) consists of three parts: (i) it divides a given query interval into at most three subintervals; (ii) computes the RMTQ for each subinterval; and (iii) returns the final answer of the RMTQ for a given interval using answers for subintervals. Those three subintervals are defined as follows: the first subinterval is on the first subarray of the PSA in the query interval, the second subinterval is on the last subarray of the PSA in the query interval, and the third subinterval is on the remaining subarrays. The right figure in Figure 2 illustrates those three subintervals. We compute the three subintervals using predecessor queries on Z for a given query interval.

The first subinterval is on a suffix of the first subarray; hence, the first subinterval has a value larger than k if and only if the subinterval contains the k-close position of the first subarray. Similarly, the second subinterval has a value larger than k if and only if the subinterval contains the k-open position of the last subarray. The third subinterval is on middle subarrays; hence, the third subinterval has a value larger than k if and only if the

maximal value is larger than k on the subarrays. Therefore, we compute the RMTQ for the first subinterval (respectively, the second subinterval) by accessing the element of the first subarray on the k-close array (respectively, the element of the last subarray on the k-open array). We also compute the RMTQ for the third subinterval using the RMQ whose query interval covers the third subinterval on M.

Algorithm 2 shows a pseudo code of our RMTQ algorithm. Since we can compute the RMQ in constant time (e.g., [7]), the query time of our RMTQ algorithm depends on the performance of predecessor queries on Z. We can also convert k-open and close arrays into $(k-1)$-open and close arrays by changing at most two elements because the conversion can change the only element of the subarray containing k.

Formally, let $\hat{b}$ and $\hat{e}$ be the ranks of the subarray of the PSA of T which contains the positions b and e, respectively, i.e., $\hat{b} = \mathsf{pred}(Z, b)$ and $\hat{e} = \mathsf{pred}(Z, e)$. For $k \in [1, n]$, let M_k^{open} be the array of size r such that for $i \in [1, r]$, $M_k^{open}[i] = (\mathsf{SA}[\mathsf{open}(i, k)], \mathsf{open}(i, k))$ if $\mathsf{open}(i, k) \neq n + 1$ holds; otherwise $M_k^{open}[i] = (-1, \mathsf{open}(i, k))$. Similarly, let M_k^{close} be the array of size r such that for $i \in [1, r]$, $M_k^{close}[i] = (\mathsf{SA}[\mathsf{close}(i, k)], \mathsf{close}(i, k))$ if $\mathsf{close}(i, k) \neq 0$; otherwise $M_k^{close}[i] = (-1, \mathsf{close}(i, k))$. Let M be the integer array of length r such that $M[i]$ stores the maximal value in the i-th subarray of the PSA of T, i.e., $M[i] = \mathsf{RMQ}(\mathsf{SA}, p(i), p(i+1) - 1)$ for all $i \in [1, r]$. Let $Q(m, u)$ be the query time of the predecessor query on a set S of size m from a universe $[1, u]$ by a predecessor data structure of $O(m)$ space. Then the following lemmas hold.

▶ **Lemma 2.** *In case (B),* $\mathsf{RMTQ}(\mathsf{SA}, b, e, k) \neq -1$ *holds if and only if there exists an answer of* $\mathsf{RMTQ}(\mathsf{SA}, b, e, k)$ *in* $M_k^{open}[\hat{b}], M_k^{close}[\hat{e}]$*, or* $\mathsf{RMQ}(M, \hat{b} + 1, \hat{e} - 1, k)$*.*

▶ **Lemma 3.** *Our data structure for case (B) can compute* $\mathsf{RMTQ}(SA, b, e, k)$ *in* $O(1+Q(r, n))$ *time. The space usage is* $O(r)$ *space.*

Proof. We construct the predecessor data structure for Z, which supports predecessor queries in $Q(r, n)$ time, and the RMQ data structure for M which supports the RMQ in $O(1)$ time using [7]. Then Lemma 3 holds by Algorithm 2. ◀

▶ **Lemma 4.** *For a given position x of k on the suffix array of T (i.e., $k = \mathsf{SA}[x]$), we can convert M_k^{open} and M_k^{close} into M_{k-1}^{open} and M_{k-1}^{close} in $O(1 + Q(r, n))$ time using the predecessor data structure for Z.*

Proof. $M_k^{open}[i] = M_{k-1}^{open}[i]$ and $M_k^{close}[i] = M_{k-1}^{close}[i]$ hold for all $i \in ([1, r] \setminus \{p\})$, where $p = \mathsf{pred}(Z, x)$. Therefore, we appropriately update $M_k^{open}[p]$ and $M_k^{open}[p]$ for $k - 1$. ◀

4.2 Data structure for backward search, LF, and access queries

We leverage the static data structures presented by Gagie et al. [8] for backward search, the LF function, and access queries on L instead of Policriti and Prezza's dynamic data structure. The static data structures compute three queries by executing only a constant number of predecessor queries, and the three queries using the static data structures can be faster than those using Policriti and Prezza's dynamic data structure.

Since the time for the predecessor query on the static data structures is proportional to the alphabet size of the input RLBWT, the alphabet size slightly increases the query times. For faster queries, we replace one of the static data structures for the predecessor queries depending on the alphabet size with an array of size σ. We obtain the following lemma.

▶ **Lemma 5.** *Let $C(m,u)$ be the construction time for the predecessor data structure of $O(m)$ space, which supports predecessor queries in $Q(m,u)$ time. We can construct the data structure of $O(r+\sigma)$ space, which supports* backward_search, LF, *and* access *queries for L in $O(1+Q(r,n))$ time, by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. See Appendix. ◀

4.3 Improved Policriti and Prezza's algorithm

Algorithm 1 using our data structures requires $O(r+\sigma)$ space, which can be $\omega(r)$ when $\sigma \geq r$, e.g., $\sigma = n^2$. To bound the space usage to $O(r)$, we reduce the alphabet size of the RLBWT L to at most r by renumbering characters in L.

We modify Algorithm 1 as follows: (i) We replace each character c in L with the rank of c in L (i.e., $|\{L[i] \mid L[i] \leq c, i \in [1,n]\}|$) and construct the new RLBWT L' over the alphabet of at most r. We call the converted string the *shrunk string* of L. (ii) We convert L' into LZ77 phrases of the string T'^R recovered from L' using Algorithm 1. (iii) We recover the LZ77 phrases of T^R from that of T'^R using the *inverse array* W, where the i-th element of the array stores the original character of rank i (i.e., $W[L'[i]] = L[i]$ holds for any $i \in [1,n]$). The modified algorithm works correctly because (1) our backward search queries receive only characters that appear in the RLBWT, (2) L' is the RLBWT of the shrunk string of T, and (3) the form of LZ77 phrases is independent of the alphabet, i.e., we obtain the i-th LZ77 phrase of T^R by mapping characters in the i-th LZ77 phrase of T'^R into the original characters.

Finally, we obtain a conversion algorithm from RLBWT into LZ77 in $O(n(1+Q(r,n)) + C(r,n))$ time and $O(r)$ space, and the algorithm depends on the performance of the static predecessor data structure. There exist two predecessor data structures such that (1) $Q(r,n) = O(\sqrt{\log r / \log\log r})$ and $C(r,n) = O(r\sqrt{\log r / \log\log r})$ hold [3] and (2) $Q(r,n) = O(\log\log n)$ and $C(r,n) = O(r)$ hold [6]. Since we can compute r and n by processing the RLBWT in $O(r)$ time, we choose the faster predecessor data structure between those predecessor data structures. Therefore, we obtain the result of our data structures, is listed in Table 1.

Formally, the following lemmas and theorem hold.

▶ **Lemma 6.** *The following statements hold. (1) We can compute the shrunk string L' of L and the inverse array W in $O(r)$ time and working space. (2) The L' is the RLBWT of the shrunk string T' of T. (3) The $|\mathsf{LZ}(T'^R)| = |\mathsf{LZ}(T^R)|$ and $\mathsf{LZ}(T^R)[i][j] = W[\mathsf{LZ}(T'^R)[i][j]]$ hold for $i \in [1,z]$ and $j \in [1, |\mathsf{LZ}(T^R)[i]|]$, where z is the number of LZ77 phrase of T^R. (4) We can convert the compressed form of $\mathsf{LZ}(T'^R)[i]$ into that of $\mathsf{LZ}(T^R)[i]$ in constant time using W for all $i \in [1,z]$.*

Proof. (1) We construct the string E that consists of r first characters in the run-length encoding of L (i.e., $E = \mathsf{RLE}(L)[1][1], \mathsf{RLE}(L)[2][1], \ldots, \mathsf{RLE}(L)[r][1]$) and construct the suffix array of E in $O(r)$ time and working space [11]. We construct the shrunk string E' of E and W using the suffix array and construct L' using E'. (2) Let SA and SA' be the suffix arrays of T and T'. Then $\mathsf{SA}[i] = \mathsf{SA}'[i]$ holds for all $i \in [1,n]$. Therefore, the RLBWT of T' is L'. (3) This holds because $Occ(T, T[x..y]) = Occ(T', T'[x..y])$ holds for two integers $1 \leq x \leq y \leq n$. (4) If $\mathsf{LZ}(T'^R)[i]$ is a target phrase, we return the phrase as $\mathsf{LZ}(T^R)[i]$. Otherwise, we return $W[\mathsf{LZ}(T'^R)[i]]$ as $\mathsf{LZ}(T^R)[i]$. ◀

▶ **Lemma 7.** *We can construct the data structure of Lemma 3 in* $O(n(1+Q(r,n))+C(r,n)+\sigma)$ *time and* $O(r+\sigma)$ *working space using the RLBWT data structure of Lemma 5.*

Proof. See Appendix. ◀

▶ **Theorem 8.** *There exists a conversion algorithm from RLBWT to LZ77 in* $O(n(1+Q(r,n))+C(r,n))$ *time and* $O(r)$ *space.*

Proof. We already have described our algorithm in Section 4.3. Each step of Algorithm 1 additionally needs to update M_k^{open}, M_k^{close} and determine either case (A) or (B) for a given query interval. The total time is $O(1+Q(r,n))$ using predecessor queries on L and Lemma 4. Therefore, Theorem 8 holds by Lemmas 3, 5, 6, and 7. ◀

5 Conclusion

We presented a new conversion algorithm from RLBWT into LZ77 in $O(n(1+Q(r,n))+C(r,n))$ time and $O(r)$ space. By leveraging the fastest static predecessor data structure using $O(r)$ space, we obtain the conversion algorithm that runs in $O(n\min\{\log\log n, \sqrt{\frac{\log r}{\log\log r}}\})$ time. This result improves the previous result in $O(n\log r)$ time and $O(r)$ space.

We have the following open problem: can we achieve the conversion from RLBWT into LZ77 in $O(n)$ time and $O(r)$ space? It is difficult to achieve the $O(n)$ time complexity with our approach because any predecessor data structure for a set using $m^{O(1)}$ words of $(\log|U|)^{O(1)}$ bits requires $\Omega(\sqrt{\log m/\log\log m})$ query time in the worst case [3], where m is the number of elements in the set and U is the universe of elements. Kempa's conversion algorithm can run faster than our algorithm, but it requires $\omega(r)$ working space in the worst case. Thus, a new approach is required to solve this open problem.

References

1 Hideo Bannai, Pawel Gawrychowski, Shunsuke Inenaga, and Masayuki Takeda. Converting SLP to LZ78 in almost Linear Time. In *Proceedings of CPM*, pages 38–49, 2013.

2 Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 Factorization of Grammar Compressed Text. In *Proceedings of SPIRE*, pages 86–98, 2012.

3 Paul Beame and Faith E. Fich. Optimal Bounds for the Predecessor Problem and Related Problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

4 Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj, and Søren Vind. Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation. *Algorithmica*, 80(11):3207–3224, 2018.

5 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical report*, 1994.

6 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proceedings of CPM*, pages 160–171, 2015.

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.

8 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-Time Text Indexing in BWT-runs Bounded Space. In *Proceedings of SODA*, pages 1459–1477, 2018.

9 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for Searchable Partial Sums with optimal worst-case performance. *Theor. Comput. Sci.*, 412(39):5176–5186, 2011.

10 Artur Jez. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016.

11 Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of ICALP*, pages 943–955, 2003.

12 Dominik Kempa. Optimal Construction of Compressed Indexes for Highly Repetitive Texts. In *Proceedings of SODA*, pages 1344–1357, 2019.

13 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

14 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

15 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018.

16 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

17 Kensuke Sakai, Tatsuya Ohno, Keisuke Goto, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. RePair in Compressed Space and Time. *CoRR*, abs/1811.01472, 2018.

18 Dan E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983.

A The proof of Lemma 5

We can compute LF and backward_search queries using C, rank, select, access queries for L and construct C by processing the RLBWT of T in $O(r+\sigma)$ time and working space. Therefore, we give the data structures for rank, select, and access queries for L by the following lemmas.

▶ **Lemma 9.** *We can construct the data structure of $O(r+\sigma)$ space that supports* rank *queries for L in $O(1+Q(r,n))$ time by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. We compute rank queries for a character c by the constant number of accessing elements on two arrays B_c and V_c and the constant number of predecessor queries on B_c. Array B_c is the array storing sorted starting positions of runs of c in L, and V_c is the integer array such that $V_c[i]$ stores the rank of the first character of the i-th run of character c. We can construct $B_1,\ldots,B_\sigma$ and $V_1,\ldots,V_\sigma$ in $O(r+\sigma)$ time and working space by processing the RLBWT of T and predecessor data structures for rank queries in $O(1+C(r,n))$ time. Therefore Lemma 9 holds.

Formally, let $\mathsf{run}(c)$ be the number of runs of character c in L (i.e., $\mathsf{run}(c)=|\{i \mid L_i[1]=c, i\in[1,r]\}|$). Array $B_c[i]$ stores the starting position of the i-th run of character c for all $c\in\Sigma$ and $i\in[1,\mathsf{run}(c)]$, $V_c[i]=\mathsf{rank}(L,c,B_c[i])$ for $c\in\Sigma$ and $i\in[1,\mathsf{run}(c)]$ and $V_c[\mathsf{run}(c)+1]=\mathsf{rank}(L,c,n)+1$. If $L[x]=c$ holds, then $\mathsf{rank}(L,c,x)=V_c[t]+x-B_c[t]$ holds; otherwise, $\mathsf{rank}(L,c,x)=V_c[t+1]-1$ holds, where $t=\mathsf{pred}(B_c,x)$. Since $V_c[i+1]-V_c[i]$ represents the length of the i-th run of character c, we can compute $L[x]=c$ using predecessor queries, i.e., $x-B_c[t]+1\le\ell$ holds if and only if $L[x]=c$ holds, where $\ell=V_c[\mathsf{pred}(B_c,x)+1]-V_c[\mathsf{pred}(B_c,x)]$. ◀

▶ **Lemma 10.** *We can construct the data structure of $O(r+\sigma)$ space that supports* select *queries for L in $O(1+Q(r,n))$ time by processing the RLBWT of T in $O(C(r,n)+\sigma+r)$ time and $O(r+\sigma)$ working space.*

Proof. We compute select queries for a character c using three arrays C, V_c, and B_c and the predecessor on V_c. When $\mathsf{select}(L,c,x)\neq n+1$ holds, $\mathsf{select}(L,c,x)=B_c[\mathsf{pred}(V_c,x)]+x-V_c[\mathsf{pred}(V_c,x)]$ holds. Since $C[c+1]-C[c]$ represents the number of cs in L, we can compute

$\mathsf{select}(L, c, x) \neq n+1$ using C. We can construct $B_1, \ldots, B_\sigma$ and $V_1, \ldots, V_\sigma$ in $O(r+\sigma)$ time and working space by processing the RLBWT of T and predecessor data structures for select queries in $O(1 + C(r, n))$ time. Therefore Lemma 10 holds. ◀

▶ **Lemma 11.** *We can construct the data structure of $O(r)$ space that supports access queries for L in $O(1+Q(r,n))$ time by processing the RLBWT of T in $O(C(r,n)+r)$ time and $O(r)$ working space.*

Proof. We compute access queries using two arrays B and E and the predecessor on B. Array B is the sorted starting positions of runs in L (i.e., $B = p(1), p(2), \ldots, p(r)$), and E is the first characters of runs in L (i.e., $E = \mathsf{RLE}(L)[1][1], \mathsf{RLE}(L)[2][1], \ldots, \mathsf{RLE}(L)[r][1]$). We can construct B and E in $O(r)$ time by processing the RLBWT. Since we can compute $L[i]$ by $E[\mathsf{pred}(B, i)]$ for a given integer i, Lemma 11 holds. ◀

Therefore Lemma 5 holds by Lemmas 9, 10, and 11.

B The proof of Lemma 7

Proof. Our data structure for RMTQ consists of $M, M_n^{open}, M_n^{close}, Z$, the RMQ data structure for M, and the predecessor data structure for Z. We construct Z using *c-integer sequence*. The c-integer sequence is the subarray of Z such that subarray $Z[b..e]$ is on the run of a character c in F, i.e., $b = \min\{X[u] \mid L[p(u)] = c, u \in [1, r]\}$ and $e = \max\{X[u] \mid L[p(u)] = c, u \in [1, r]\}$. We construct 1-integer sequence, ..., σ-integer sequence in $O(r(1 + Q(r, n)) + \sigma)$ using the LF function since $\mathsf{LF}(p(i)) < \mathsf{LF}(p(j))$ holds for two positions i and j such that $L[i] = L[j]$ and $i < j$ hold. We obtain Z by concatenating the sequences. The total time is $O(r(1 + Q(r, n)) + \sigma)$ and the working space is $O(r + \sigma)$.

We construct M using the LF function and predecessor on Z in $O(1 + Q(r, n))$ time since $\mathsf{LF}(i)$ represents a position on the suffix array of T for $i \in [1, n]$. We construct M_n^{open} and M_n^{close} in $O(r)$ time since $M_n^{open} = (-1, n+1), \ldots, (-1, n+1)$ and $M_n^{close} = (-1, 0), \ldots, (-1, 0)$. We construct the RMQ data structure for M in $O(r)$ time and working space using [7]. Therefore Lemma 7 holds since $r \leq n$. ◀

Fully-Functional Bidirectional Burrows-Wheeler Indexes and Infinite-Order De Bruijn Graphs

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

Fabio Cunial
Max Planck Institute for Molecular Cell Biology and Genetics (MPI-CBG), Dresden, Germany
Center for Systems Biology Dresden (CSBD), Dresden, Germany
cunial@mpi-cbg.de

Abstract

Given a string T on an alphabet of size σ, we describe a bidirectional Burrows-Wheeler index that takes $O(|T| \log \sigma)$ bits of space, and that supports the addition *and removal* of one character, on the left or right side of any substring of T, in constant time. Previously known data structures that used the same space allowed constant-time addition to any substring of T, but they could support removal only from specific substrings of T. We also describe an index that supports bidirectional addition and removal in $O(\log \log |T|)$ time, and that takes a number of words proportional to the number of left and right extensions of the maximal repeats of T. We use such fully-functional indexes to implement bidirectional, frequency-aware, variable-order de Bruijn graphs with no upper bound on their order, and supporting natural criteria for increasing and decreasing the order during traversal.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Design and analysis of algorithms

Keywords and phrases BWT, suffix tree, CDAWG, de Bruijn graph, maximal repeat, string depth, contraction, bidirectional index

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.10

Acknowledgements We thank Martin Bundgaard for motivating the contract operation, Rodrigo Canovas for discussions about bidirectional indexes, Gene Myers for discussions about PacBio CCS reads, and German Tischler for help with k-mer counting.

1 Introduction

A *bidirectional index* on a string T is a data structure that represents any substring W of T as a constant-size descriptor that recapitulates the set of all starting positions of W in T, and the set of all ending positions of W in T. Such a representation allows extending W with a character in both directions, enumerating the distinct characters that occur after W in both directions, and switching direction during extension. All existing bidirectional indexes can be seen as updating positions in the suffix tree of T and in the suffix tree of the reverse of T, either literally, as in the *affix tree* [30, 49], or in compact representations, like the *affix array* [50] and the *bidirectional Burrows-Wheeler transform* (BWT) [47]. *Synchronous* bidirectional indexes maintain a position in both trees at every extension step, whereas *asynchronous* indexes maintain a position in just one tree, and compute the position in the other only when the user needs to change direction [18]. Applications of bidirectional indexes to bioinformatics, like read mapping with mismatches and searching for RNA secondary structures, have used until now the ability of bidirectional indexes to *add* characters both to the left and to the right of a string (an operation called *extension*: see e.g. [25, 28, 34, 45, 47, 50] for a small sampler), whereas *removing* characters from the left and from the right (called *contraction*) has only been conjectured to be useful [13, 18], and it has been supported efficiently just

for right-maximal and left-maximal substrings of T, respectively (defined in Section 2), or for strings that occur just once in T, for which the implementation is straightforward (see e.g. [11, 38]).

In this paper we describe a simple method for removing characters from the left or from the right of any substring of T, based just on the ability to measure the length of the *maximal repeats* of T (defined in Section 2). Using the recent observation that all such lengths can be represented in $O(|T|)$ bits of space [6], we show that bidirectional contraction can be supported in constant time with the bidirectional BWT index described in [11], within the same space budget and without changing the complexity of its construction. Our contraction algorithm can also be implemented on top of an existing representation of the suffix tree, based on the *Compact Directed Acyclic Word Graph* (CDAWG), that takes a number of words proportional just to the number of left and right extensions of the maximal repeats of T [8]: this yields an index that supports, in the same asymptotic space, bidirectional extension and contraction of any substring of T in $O(\log\log|T|)$ time.

Having both bidirectional extension and contraction enables several applications, among which a de Bruijn graph that stores the frequency of its k-mers, allows for bidirectional navigation, and supports any value of k, as well as increasing and decreasing the value of k, *with no limit on the maximum k allowed*. We call such a data structure an *infinite-order de Bruijn graph*, and we describe an implementation that takes $O(|T|\log\sigma)$ bits of space (where σ is the size of the alphabet), and that supports all operations in constant time, as well as another implementation that takes a number of words proportional to the left an right extensions of the maximal repeats of T, and that supports all operations in $O(\log\log|T|)$ time. The latter representation establishes a connection between de Bruijn graphs and CDAWGs that was not known before. Our query times are comparable to those of the variable-order, bidirectional representation described in [13], which supports navigation and changing order in $O(\log K)$ time (assuming constant σ), but is frequency-oblivious and requires a maximum order K to be specified during construction. This competitor has the advantage of taking just $O(m\log K)$ bits of space, where m is the number of distinct K-mers, and of allowing the user to specify by how much the order should be changed in each query (the changes in order supported by our index are detailed in Sections 3 and 4). The variable-order representation described in [22] takes constant time (assuming constant σ) to implement changes in order that are similar to those supported by our index, and uses just $O(m)$ bits of space; however, it is unidirectional, frequency-oblivious, and it requires again a maximum K to be known at construction time.

We conjecture that a de Bruijn graph representation based on the CDAWG might be useful for assembling the recently introduced PacBio CCS reads, which have the same 2% error rate as Illumina short reads but an average length of 15 kilobases (see e.g. [51]). Such read sets contain long exact repeats, of length up to ten thousand, so it might be desirable to set k to large values and to decrease it dynamically, down to a minimum value τ. Moreover, most maximal repeats are short (Figure 1, bottom right), and we can remove from the CDAWG all maximal repeats shorter than τ, and all arcs adjacent to them, while still being able to represent all de Bruijn graphs of order at least τ (see Section 4). For practical values of k, the number of nodes and arcs in such a pruned CDAWG grows more slowly than the number of distinct k-mers (Figure 1, top right; reads from the Genome in a Bottle consortium[1]), suggesting that our data structure might be competitive in space with the

[1] `ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/AshkenazimTrio/HG002_NA24385_son/PacBio_CCS_15kb/`

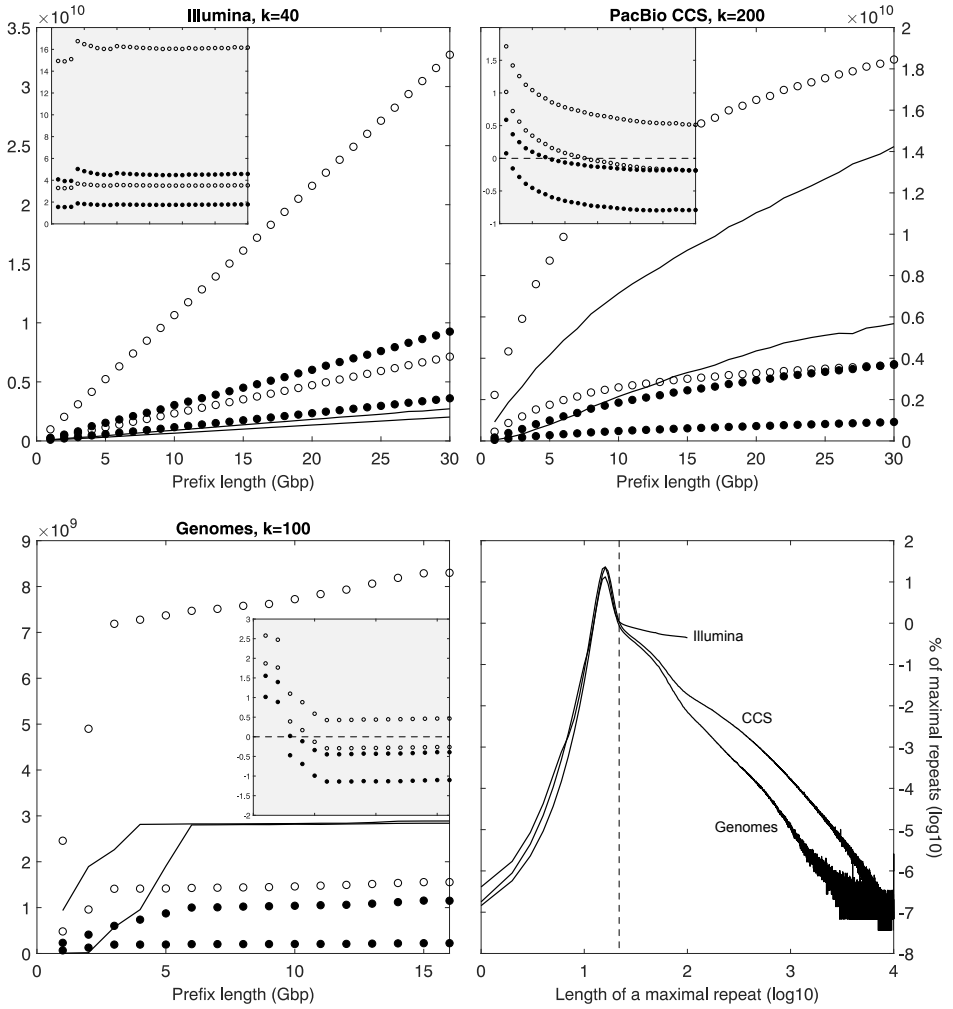


Figure 1 Number of k-mers and repeated k-mers (lines), maximal repeats and bidirectional extensions of maximal repeats (white circles), and maximal repeats of length at least 20 and the bidirectional extensions connecting them (black circles), for prefixes of a human read dataset produced with Illumina and PacBio CCS technologies. Bottom left: prefixes of the concatenation of 5 human genome assemblies. Bottom right: fraction of maximal repeats of each length in the three datasets (the vertical line is at length 20). Inserts show the number of maximal repeats and extensions, divided by the number of repeated k-mers (in $\log_{10}$ scale for CCS and genomes). Decreasing k down to 20 (Illumina), 50 (CCS) and 25 (genomes) yields similar plots. k-mers are counted with KMC 3 [27], and are considered distinct from their reverse complements.

state of the art, whose size is proportional to the number of k-mers for a specific value of k. The same observation applies to repetitive datasets: for example, the de Bruijn graph of a set of individuals from the same species has applications in population genomics, and the de Bruijn graph of a set of genomes from related species is used in comparative genomics [35, 36]. In Figure 1, bottom left, we experiment with the concatenation of assemblies `hg16`, `hg17`, `hg18`, `hg19` and `hg38` of the human genome from the UCSC Genome Browser[2] (a

[2] `http://hgdownload.soe.ucsc.edu/downloads.html#human`

benchmark dataset from [3, 36]), and we observe exact repeats of length up to 489 million. Our data structure might also be useful with noisy long reads after error correction. Even in short-read Illumina datasets, the number of maximal repeats and of their extensions after pruning is just a small multiple of the number of distinct k-mers (Figure 1, top left; reads from the Illumina Platinum project[3]).

Finally, recall that our de Bruijn graph representations allow access to the frequency of a node or arc: this might be useful for avoiding repetitive regions during assembly, or for reconstructing only those [26], for assembling metagenomes with non-uniform sequencing depths [29], or for inferring transcripts with different expression levels [42].

2 Preliminaries

2.1 Strings

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0$ be a separator not in Σ, and let $T = [1..\sigma]^{n-1}$ be a string. We denote by $\overline{W}$ the reverse of a string W, i.e. string W written from right to left, and we call W a k-mer iff $|W| = k$. We denote by $f_T(W)$ the number of (possibly overlapping) occurrences of a string W in the circular version of T. A *repeat* W is a string that satisfies $f_T(W) > 1$. We denote by $\Sigma_T^\ell(W)$ the set of *left-extensions of* W, i.e. the set of characters $\{a \in [0..\sigma] : f_T(aW) > 0\}$. Symmetrically, we denote by $\Sigma_T^r(W)$ the set of *right-extensions of* W, i.e. the set of characters $\{b \in [0..\sigma] : f_T(Wb) > 0\}$. A repeat W is *right-maximal* (respectively, *left-maximal*) iff $|\Sigma_T^r(W)| > 1$ (respectively, iff $|\Sigma_T^\ell(W)| > 1$). It is well-known that T can have at most $n-1$ right-maximal substrings and at most $n-1$ left-maximal substrings. A *maximal repeat* of T (called *balanced substring* in [50]) is a repeat that is both left- and right-maximal.

The *unidirectional de Bruijn graph* of order k of T is a directed graph (V, E) whose node set V is in one-to-one correspondence with the set of distinct k-mers that occur in T; there is an arc $(v, w) \in E$ for every distinct $(k+1)$-mer W such that both $W[1..k]$ and $W[2..k+1]$ occur in T, and such arc is labelled with character $W[k+1]$. In some formulations, E contains just those arcs that correspond to $(k+1)$-mers that occur in T: in this case, a k-mer is right-maximal (respectively, left-maximal) in T iff its corresponding node in V has at at least two outgoing (respectively, incoming) arcs. The *bidirectional* de Bruijn graph is defined symmetrically.

We denote by ST_T the *suffix tree* of $T\#$, and by $\overline{\mathsf{ST}}_T$ the suffix tree of $\overline{T}\#$. We assume the reader to be already familiar with the basics of suffix trees, including suffix links, which we do not further describe here. We denote by $\ell(v)$ the label of a node v of a suffix tree, and we say that v is the *locus* of all substrings $W[1..k]$ of T where $|\ell(u)| < k \leq |\ell(v)|$, u is the parent of v, and $W = \ell(v)$. It is well-known that a substring W of T is right-maximal (respectively, left-maximal) iff $W = \ell(v)$ for some internal node v of ST_T (respectively, for some internal node v of $\overline{\mathsf{ST}}_T$). Suffix links and internal nodes of ST_T form a tree, called the *suffix-link tree* of T and denoted by SLT_T, and inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given an internal node v and a character $a \in [0..\sigma]$, it might happen that string $a\ell(v)$ occurs in T but is not right-maximal, i.e. it is not the label of any internal node of ST_T: all such left extensions of internal nodes that end in the middle of an edge are called *implicit Weiner links*. An internal node v of ST_T can have more than

[3] `https://www.ebi.ac.uk/ena/data/view/PRJEB3381`, run `ERR194146`, file `ERR194146_1.fastq.gz`, read length 101.

one outgoing Weiner link, and all such Weiner links have distinct labels: in this case, $\ell(v)$ is a maximal repeat, as well as the label of a node in $\overline{\mathsf{ST}}_T$. Maximal repeats and implicit Weiner links are related by the following simple property, which was already hinted at in [2]:

▶ **Property 1.** *Let v be an internal node of ST_T. If there is an implicit Weiner link from v, then $\ell(v)$ is a maximal repeat of T.*

It is known that the number of suffix links (or, equivalently, of explicit Weiner links) is upper-bounded by $2n-2$, and that the number of implicit Weiner links can be upper-bounded by $2n-2$ as well. We call SLT^*_T a version of SLT_T augmented with implicit Weiner links and with nodes corresponding to their destinations. We say that a maximal repeat W of T is *rightmost* if no string WV with $V \in [0..\sigma]^+$ is left-maximal in T. Symmetrically, we say that a maximal repeat W of T is *leftmost* if no string VW with $V \in [0..\sigma]^+$ is right-maximal in T. Since left-maximality is closed under prefix operation, it is easy to see that the maximal repeats of T are all and only the nodes of ST_T that lie on paths that start from the root and that end at nodes labelled by rightmost maximal repeats. We call this the *maximal repeat subgraph of* ST_T (Figure 2b). Clearly the maximal repeats of T coincide with the branching nodes of $\overline{\mathsf{SLT}}^*_T$ (Figure 2a), and the rightmost maximal repeats of T coincide with the leaves of $\overline{\mathsf{SLT}}_T$. Thus, it is easy to see that $\overline{\mathsf{SLT}}_T$ (a trie) is a subdivision of the maximal repeat subgraph of ST_T (a compact trie), and that the nodes in the unary paths of $\overline{\mathsf{SLT}}_T$ are in one-to-one correspondence with the internal nodes of ST_T that are not maximal repeats (see Figures 2a and 2b for an example, and see Section 2.1 in [6] for an extended explanation). The following property is thus immediate (and symmetrical notions hold for $\overline{\mathsf{ST}}_T$, SLT^*_T, and leftmost maximal repeats):

▶ **Property 2.** *Let v be an internal node of ST_T. The locus w of $\overline{\ell(v)}$ in $\overline{\mathsf{ST}}_T$ is such that $\ell(w)$ is the reverse of a maximal repeat of T.*

The *compact directed acyclic word graph* of a string T (denoted by CDAWG_T in what follows) is the minimal compact automaton that recognizes the suffixes of T [16, 20]. We denote by $\overline{\mathsf{CDAWG}}_T$ the CDAWG of the reverse of T, by e_T the number of arcs in CDAWG_T, and by $\overline{e}_T$ the number of arcs in $\overline{\mathsf{CDAWG}}_T$. The CDAWG of T can be seen as the minimization of ST_T, in which all leaves are merged to the same node (the sink, that represents T itself), and in which all nodes except the sink are in one-to-one correspondence with the maximal repeats of T [44]. Every arc of CDAWG_T is labeled by a substring of T, and the out-neighbors $w_1, \ldots, w_k$ of every node v of CDAWG_T are sorted according to the lexicographic order of the distinct labels of arcs $(v, w_1), \ldots, (v, w_k)$. Since there is a bijection between the nodes of CDAWG_T and the maximal repeats of T, the node v' of CDAWG_T with $\ell(v') = W$ is the equivalence class of the nodes $\{v_1, \ldots, v_k\}$ of ST_T such that $\ell(v_i) = W[i..|W|]$ for all $i \in [1..k]$, and such that $v_k, v_{k-1}, \ldots, v_1$ is a maximal unary path of explicit Weiner links. The subtrees of ST_T rooted at all such nodes are isomorphic. It follows that a right-maximal string can be identified by the maximal repeat W it belongs to, and by the length of the corresponding suffix of W (see [8] for an extended explanation).

We assume the reader to be familiar with the Burrows-Wheeler transform of T, which we denote by BWT_T (we use $\overline{\mathsf{BWT}}_T$ to denote the BWT of the reverse of T) and we don't further describe here. We say that $\mathsf{BWT}_T[i..j]$ is a *run* iff: (1) $\mathsf{BWT}_T[k] = c \in [0..\sigma]$ for all $k \in [i..j]$; (2) every substring $\mathsf{BWT}_T[i'..j']$ such that $i' \leq i$, $j' \geq j$, and $[i'..j'] \neq [i..j]$, contains at least two distinct characters. We denote by $\mathcal{R}_T$ the set of all triplets (c, i, j) such that $\mathsf{BWT}_T[i..j]$ is a run of character c, and we use $\overline{\mathcal{R}}_T$ to denote the set of runs of $\overline{\mathsf{BWT}}_T$. It is known that $|\mathcal{R}_T|$ is at most equal to the number of arcs in CDAWG_T [10].

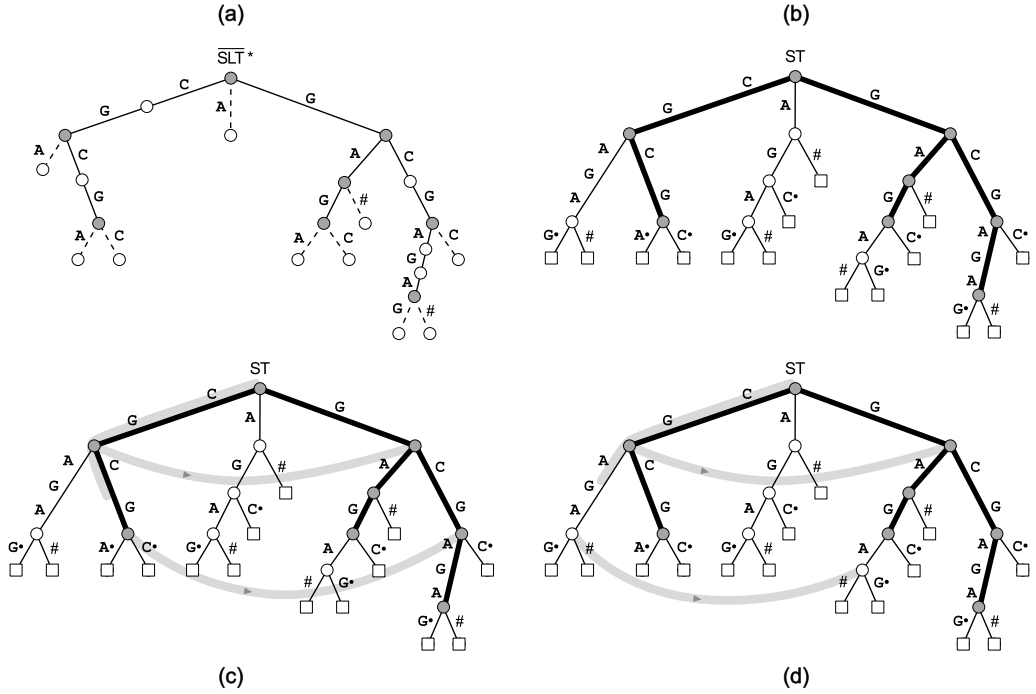


Figure 2 Left-contraction of a substrings that is not right-maximal. (a) The extended suffix-link tree $\overline{\mathsf{SLT}}^*_T$ of string $T = \texttt{CGCGCGAGAGCGAGA\#}$. Nodes that correspond to maximal repeats are highlighted in grey. Implicit Weiner links are dashed. (b) ST_T (thin lines) with $\overline{\mathsf{SLT}}_T$ overlaid (thick lines). Nodes that correspond to maximal repeats are in grey. Labels of edges to leaves are shortened. (c) Left-contraction of substring $aW = \texttt{CGC}$. The edge to which aW belongs is projected to another edge by suffix links (thick grey lines). (d) Left-contraction of substring $aW = \texttt{CGA}$. The edge to which aW belongs is projected to a path by suffix links.

Given a second string $S \in [1..\sigma]^+$, the *matching statistics array* $\mathsf{MS}_{S,T}$ of S with respect to T is an array of length $|S|$ such that $\mathsf{MS}_{S,T}[i]$ is the largest j such that $S[i..i+j-1]$ occurs in T.

In the rest of the paper we drop subscripts whenever they are clear from the context.

2.2 String indexes

A *bidirectional index* is a data structure that, given a constant-space descriptor $\texttt{id}(W)$ of a substring W of T, supports the following operations: $\texttt{extendRight}(\texttt{id}(W), a) = \texttt{id}(Wa)$ if $f(Wa) > 0$, or an error otherwise; $\texttt{enumerateRight}(\texttt{id}(W)) = \{\texttt{id}(Wa) : a \in \Sigma, f(Wa) > 0\}$; $\texttt{isRightMaximal}(\texttt{id}(W)) = \texttt{true}$ iff $|\texttt{enumerateRight}(\texttt{id}(W))| > 1$. Operations `extendLeft`, `enumerateLeft` and `isLeftMaximal` are defined symmetrically. We consider bidirectional indexes based on the BWT: specifically, we denote with $\mathbb{I}(W,T)$ the function that maps a substring W of T to the interval of W in BWT, i.e. to the interval of all suffixes of $T\#$ that start with W, and we use $\mathsf{id}(W) = (\mathbb{I}(W,T), \mathbb{I}(\overline{W},\overline{T}), |W|)$ as a constant-space descriptor of W. A number of bidirectional BWT indexes have been described in the literature; in this paper we are just interested in the data structure from [11], which supports all operations in constant time in the size of their output, takes $O(|T|\log\sigma)$ bits of space, and can be built in randomized $O(|T|)$ time and $O(|T|\log\sigma)$ bits of working space.

Given a string $T \in [1..\sigma]^{n-1}\#$, we call *run-length encoded BWT* (RLBWT_T) any representation of BWT_T that takes $O(|\mathcal{R}_T|)$ words of space and supports the well-known rank and select operations (see e.g. [31, 32, 48]). It is easy to implement a version of RLBWT_T that supports rank and select in $O(\log\log n)$ time [10]. In this paper we use the representation of the suffix tree based on the CDAWG described in [8], which takes just $O(e+\overline{e})$ words of space by augmenting CDAWG and $\overline{\mathsf{CDAWG}}$ with the RLBWT of T and $\overline{T}$. Such a data structure describes a node v of ST as a tuple $\texttt{id}(v) = (v', |\ell(v)|, i, j)$, where v' is the node in CDAWG that corresponds to the equivalence class of v, and $[i..j]$ is the interval of $\ell(v)$ in BWT. For every node v of CDAWG, the index stores, among other things: $|\ell(v)|$ in a variable $v.\texttt{length}$; the number $v.\texttt{size}$ of right-maximal strings that belong to its equivalence class; and the interval $[v.\texttt{first}..v.\texttt{last}]$ of $\ell(v)$ in BWT_T. For every arc $\gamma = (v, w)$ of CDAWG, the index stores the first character of $\ell(\gamma)$ in a variable $\gamma.\texttt{char}$, and the number of characters of the right extension implied by γ in a variable $\gamma.\texttt{right}$. Finally, we add to the CDAWG all arcs (v, w, c) such that w is the equivalence class of the destination of a Weiner link from v labeled by character c in ST_T, as well as the reverse of all explicit Weiner link arcs. See [8] for an extended description of the data structure and of the complexity of its operations. Here we just mention that the index supports operations $\texttt{stringDepth}(\texttt{id}(v))$ and $\texttt{child}(\texttt{id}(v), c)$ in constant time, and $\texttt{parent}(\texttt{id}(v))$, $\texttt{suffixLink}(\texttt{id}(v))$, $\texttt{weinerLink}(\texttt{id}(v), c)$ in $O(\log\log|T|)$ time.

In this paper we need to store the topology of $\overline{\mathsf{SLT}}$ and the topology of ST efficiently. It is well-known that the topology of an ordered tree of n nodes can be represented using $2n + o(n)$ bits, as a sequence of $2n$ balanced parentheses [39]. Let $\texttt{id}(v)$ be the rank of a node v in the preorder traversal of the tree. Given the balanced parentheses representation of the tree encoded in $2n + o(n)$ bits, it is also well-known that one can build a data structure that takes $2n + o(n)$ bits, and that supports several common operations in constant time [40, 41, 46], among which: $\texttt{parent}(\texttt{id}(v))$, which returns $\texttt{id}(u)$, where u is the parent of v, or an error if v is the root; $\texttt{lca}(\texttt{id}(v), \texttt{id}(w))$, which returns $\texttt{id}(u)$, where u is the lowest common ancestor of nodes v and w; $\texttt{leftmostLeaf}(\texttt{id}(v))$ and $\texttt{rightmostLeaf}(\texttt{id}(v))$, which return one plus the number of leaves that, in the preorder traversal of the tree, are visited before the first (respectively, the last) leaf that belongs to the subtree rooted at v; $\texttt{depth}(\texttt{id}(v))$, which returns the distance of v from the root. This data structure can be built in $O(n)$ time and in $O(n)$ bits of working space. Moreover, given a node v and a length d, a *level-ancestor query* asks for the ancestor u of v such that the path from the root to u contains exactly d nodes. The level ancestor data structure described in [14, 15] takes $O(n)$ words of space and answers queries in constant time. Assuming that some nodes of the tree are marked, a *lowest marked ancestor* data structure allows one to move in constant time from any node, to its lowest ancestor that is marked [33].

We use the tree data structures described above to store the topology of ST and of $\overline{\mathsf{SLT}}$. Moreover, we mark in two bitvectors the nodes of $\overline{\mathsf{SLT}}$ and of ST that are maximal repeats (in preorder), and we index such bitvectors to support constant-time rank and select queries. Since $\overline{\mathsf{SLT}}$ is a subdivision of the subgraph of ST induced by maximal repeats, the i-th one in the two bitvectors correspond to the same maximal repeat. Thus, if node v is a maximal repeat, and if we know its preorder position in ST, we can compute the length of $\ell(v)$ by moving to the corresponding node v' in $\overline{\mathsf{SLT}}$ and by computing the depth of v' in the topology of $\overline{\mathsf{SLT}}$ (see [6] for an extended explanation).

The rest of the paper focuses on representations of variable-order, bidirectional de Bruijn graphs that support the following primitives (for brevity we list here just operations in one direction). Let k be the current order of the de Bruijn graph. Operation $v = \texttt{node}(W)$, called *membership*, returns the identifier of the node associated with k-mer W, or an error

if W does not occur in T. Operation $C = \texttt{arcLabels}(v)$ returns the set of characters C that label all arcs from node v in the right direction, and operation $\texttt{degree}(v)$ returns the number of such arcs. Query $e = \texttt{arc}(v, c)$ returns the identifier of the arc that corresponds to string $\ell(v) \cdot c$, if any, where v is a node in the current de Bruijn graph, $\ell(v)$ is the k-mer that corresponds to node v, and c is a character; it returns an error if no such arc exists. Operation $w = \texttt{followArc}(v, c)$ is similar, but returns the identifier of the node w reached by the arc, if any. Queries $\texttt{freq}(v)$ and $\texttt{freq}(e)$ return the number of occurrences of the k-mer associated with node v and of the $(k+1)$-mer associated with arc e (the number of occurrences of an arc might be zero). Representations that support such queries are called *frequency-aware* or *weighted* (see e.g. [42]). Operation $v' = \texttt{increaseK}(v, c)$ for $c \in [0..\sigma]$ returns the node v' associated with string $\ell(v) \cdot c$ in the de Bruijn graph of order $k+1$, if any, or an error otherwise. Operation $v' = \texttt{decreaseK}(v)$ returns the node v' associated with the prefix of length $k-1$ of $\ell(v)$ in the de Bruijn graph of order $k-1$.

In addition to increasing and decreasing the order by one unit, some variable-order representations allow the user to specify the desired amount of change [13, 17]. In the rest of the paper we argue that it is more natural to change the order based on the frequency or on the extensions of k-mers, as proposed in [22]. Specifically, given a node v of the current de Bruijn graph, let $\ell(v) \cdot W$, $W \in \Sigma^*$, be the longest string with the same frequency as $\ell(v)$ in T. Operation $(v', k') = \texttt{increaseK}(v)$ returns the node v' associated with $\ell(v) \cdot W$ in the de Bruijn graph of order $k + |W|$, and sets k' to the new order $k + |W|$. Given a node v of the current de Bruijn graph, let W be the longest prefix of $\ell(v)$ that has a different frequency from $\ell(v)$ in T. Operation $(v', k') = \texttt{decreaseK}(v)$ returns the node v' associated with W in the de Bruijn graph of order $|W|$, and sets k' to $|W|$. Alternatively, one might want W to be the longest prefix of $\ell(v)$ such that the left-extensions of W are a superset of the left-extensions of $\ell(v)$. A de Bruijn graph that supports such operations without returning the value of the new order is called *hidden-order* [22].

3 Contracting in constant time

As mentioned, existing bidirectional BWT indexes support left-contraction just from right-maximal substrings (and symmetrically, they support right-contraction just from left-maximal substrings). Specifically, if the substring aW is right-maximal and labels a node v of ST, then $\mathbb{I}(W, T)$ is the interval of node $\texttt{suffixLink}(v)$ in ST, and since we are removing one character from the right of $\overline{aW}$, the locus of $\overline{W}$ in $\overline{\mathsf{ST}}$ is either the same as the locus w of $\overline{aW}$, or it is $\texttt{parent}(w)$, whichever has the same frequency as $\mathbb{I}(W, T)$ [11, 38].

To support left-contraction from a substring that is not right-maximal, it is enough to have access to the topology of $\overline{\mathsf{SLT}}$:

▶ **Theorem 1.** *Let T be a string on alphabet Σ. There is a data structure that supports operations* `extendRight`*,* `extendLeft`*,* `contractRight` *and* `contractLeft` *in constant time and in $O(n \log \sigma)$ bits of space. Such a data structure can be built in randomized $O(n)$ time and $O(n \log \sigma)$ bits of working space.*

Proof. We use the data structures described in [11], augmented with the topology of SLT and with a bitvector to commute between the topology of ST and the topology of SLT (see [6] for details on commuting). Such data structures take $O(n \log \sigma)$ bits of space, and they can be built in randomized $O(n)$ time using the algorithms in [4, 12]. They support operations $\texttt{extendRight}(\texttt{id}(W), a) = \texttt{id}(Wa)$ and $\texttt{extendLeft}(\texttt{id}(W), a) = \texttt{id}(aW)$, where $\texttt{id}(W) = (\mathbb{I}(W, T), \mathbb{I}(\overline{W}, \overline{T}))$. We additionally assume the knowledge of $|W|$, i.e. $\texttt{id}(W) =$

$(\mathbb{I}(W,T), \mathbb{I}(\overline{W},\overline{T}), |W|)$. We only show how to support $\texttt{contractLeft}(\texttt{id}(aW)) = \texttt{id}(W)$, since supporting $\texttt{contractRight}(\texttt{id}(Wa)) = \texttt{id}(W)$ is symmetric. Since [11] already supports $\texttt{contractLeft}(\texttt{id}(aW))$, we assume for now that aW is not right-maximal. Note that we can decide whether aW is right-maximal or not by using $\mathbb{I}(\overline{aW},\overline{T})$, and, if W is right-maximal, we can just use the contraction algorithm described above. Let v be the locus of aW in ST: this can be computed from $\mathbb{I}(aW,T)$ using $\texttt{lca}$ queries on ST. Since aW is not right maximal, $aW \neq \ell(v)$ and aW ends in the middle of edge (u,v) of ST. We take in constant time the suffix link (u,u') from u and the suffix link (v,v') from v, and we decide whether (u',v') is an edge or a path of ST by comparing u' to $\texttt{parent}(v')$, which can be computed in constant time. If (u',v') is an edge of ST (Figure 2c), then v' is the locus of W and we compute $\mathbb{I}(\ell(v'),T)$ in constant time. Otherwise (Figure 2d), we compute in constant time $z = \texttt{parent}(v')$: this node is a maximal repeat by Property 1, since it is an internal node of ST with an implicit Weiner link whose destination falls inside (u,v). We use the data structures in Section 2.2 to measure the length of $\ell(z)$ in constant time. If $|W| > |\ell(z)|$, the locus of W is again v'. Otherwise, since z is a maximal repeat, we move in constant time to the node z' of $\overline{\mathsf{SLT}}$ that corresponds to $\ell(z)$, we issue a constant-time level ancestor query from z' on $\overline{\mathsf{SLT}}$ with length $|W|$, and, from the destination x' of such a level ancestor query, we move in constant time to the first branching descendant y' of x', by using $\texttt{leftmostLeaf}$, $\texttt{rightmostLeaf}$, and $\texttt{lca}$ queries on $\overline{\mathsf{SLT}}$. Finally, we move in constant time to the node y of ST that corresponds to y', and we compute $\mathbb{I}(\ell(y),T)$ in constant time. We compute $\mathbb{I}(\overline{W},\overline{\mathsf{ST}})$ as described at the beginning of Section 3. ◀

Note that the algorithm in Theorem 1 works even when aW is right-maximal; moreover, if the information on whether aW is right maximal or not is given in input, the algorithm can decide whether W is right maximal or not. In a practical implementation, once we have taken the suffix link (v,v') from v, we could check whether v' is a maximal repeat, and in the positive case we could immediately commute to SLT and issue level ancestor queries. If v' is not a maximal repeat, we could move in constant time to the lowest ancestor v'' of v' that is a maximal repeat, using a lowest marked ancestor data structure on ST, we could measure $|\ell(v'')|$, and if $|\ell(v'')| \geq |W|$, we could again issue level ancestor queries in $\overline{\mathsf{SLT}}$ (otherwise, the locus of W is again v').

A bidirectional index on T that supports extension and contraction in constant time, can be used to implement in linear time several applications that slide a window $S[i..j]$ of fixed length over a query string S, and that compute the frequency of every $S[i..j]$ in T, *without the size of the window being known during construction*[4]. For example, measuring the frequency of windows of fixed length for read correction [43], computing the inner product between the k-mer composition vectors of S and T (a step in k-mer kernels), estimating the probability of S according to a fixed-order Markov model trained on T, and checking whether S is a path in the de Bruijn graph of T. Our index enables also applications in which *the sliding window needs to be extended or contracted during the scan*, like variable-order and interpolated Markov models (see [21] for an overview). A fully-functional bidirectional index is not needed for computing the matching statistics array between S and T, in linear time and in $O(|T| \log \sigma)$ bits of space, since one can use the algorithms in [5] on top of the data structures in [4]. However, achieving such bounds with our bidirectional index becomes trivial.

[4] If the size k of the window is fixed and known during construction, most such applications do not need the contract operation, and can be made to work using just one BWT and a bitvector of length $|T|$ that marks the boundaries of k-mer intervals in the BWT.

In practical applications of matching statistics, one typically needs to maintain the intervals in both BWT and $\overline{\mathsf{BWT}}$ just after every successful right extension, and, when the current match $S[i..j]$ cannot be extended with $S[j+1]$ in T any longer, one might need both BWT intervals just for the proper suffixes $S[k..j]$ such that $\Sigma_T^r(S[i..j]) \subset \Sigma_T^r(S[k..j])$, i.e. just for the suffixes of $S[i..j]$ from which a right-extension with $S[j+1]$ is attempted again. Every such suffix is a maximal repeat ancestor of $\overline{S[i..j]}$ in $\overline{\mathsf{ST}}$ [9], thus, once we reach the locus of such a suffix in $\overline{\mathsf{ST}}$ with `parent` operations, we can compute its interval in $\overline{\mathsf{BWT}}$, we can measure its string length p, and we can compute its interval in BWT by issuing $\mathsf{MS}[i] - p$ contract operations from the locus of $S[i..j]$ in ST, but without updating the interval in $\overline{\mathsf{BWT}}$ after each contraction. Even more aggressively, we can just issue $\mathsf{MS}[i] - p$ suffix links from the locus of $S[i..j]$ in ST. Note that such a locus might correspond to the right-maximal string $S[i..j] \cdot V$ for some nonempty V, thus taking $\mathsf{MS}[i] - p$ suffix links might lead to a node of ST that corresponds to the right-maximal string $S[k..j] \cdot V$: thus, we need to move in constant time from such a node, to its lowest ancestor in ST that is a maximal repeat; from there, we can then issue a level ancestor query with value p. Such a lazy synchronization might be faster than issuing $\mathsf{MS}[i] - p$ full contract operations in practice.

Our index can be seen as a representation of a de Bruijn graph that supports bidirectional navigation, that allows access to the frequency of every k-mer and $(k+1)$-mer, and that has no upper bound on the order: we call *infinite-order* such a de Bruijn graph. Note that, for a given order k, we can support both the variant in which arcs must occur in T (calling `extendRight` and then `contractLeft` to implement `arc` and `followArc`), and the variant in which arcs do not have to occur in T (calling `contractLeft` and then `extendRight`). Membership queries reduce to backward searches, and we can move from a higher to a lower order using the same algorithm as in matching statistics. Indeed, one typically wants to switch to a suffix of the current k-mer whenever there is only one arc in the graph of the current order, and this arc is labelled with the terminator character [22]; or, more generally, whenever one needs to increase the number of outgoing arcs from the current k-mer (for example because the existing ones have already been explored [37]), or to increase the frequency of the current right-maximal k-mer. In all such cases, one wants to switch to the largest order with the desired property, and the corresponding suffix is always a maximal repeat (for example, the longest suffix, of the current right-maximal k-mer, that has strictly greater frequency, is a maximal repeat). Symmetrically, when increasing the order, one may want to switch e.g. from the current k-mer W that is left-maximal but not right-maximal, to the maximal repeat WV with shortest V. Clearly $\mathbb{I}(WV, T) = \mathbb{I}(W, T)$, we know $|V|$ since we can access $|WV|$, and we can compute $\mathbb{I}(\overline{WV}, \overline{T})$ by taking $|V|$ Weiner links from $\mathbb{I}(\overline{W}, \overline{T})$. All such Weiner links are implicit, so in practice we can just update the first position of the interval at every step.

In the next section, we describe a representation of an infinite-order de Bruijn graph in which the time to decrease or increase the order does not depend on the difference between the source and the destination order.

4 Implementing de Bruijn graphs with CDAWGs

An *affix link* $\mathbb{A}(w)$ is a mapping from a node w of ST, to the locus of $\overline{\ell(w)}$ in $\overline{\mathsf{ST}}$ (we use $\overline{\mathbb{A}}(w)$ to denote the symmetrical mapping from a node w of $\overline{\mathsf{ST}}$, to the locus of $\overline{\ell(w)}$ in ST) [49, 50]. We use $\mathbb{A}(W)$ as a shorthand for $\mathbb{A}(w)$ where w is the locus of W. In asynchronous bidirectional indexes, affix links are used to switch direction when the user desires [50]. In this section we are more interested in their ability to extend a non-maximal repeat in a

bidirectional index: for example, if W is right-maximal but not left-maximal, and if it has loci (v, w) in ST and $\overline{\mathsf{ST}}$, respectively, then its shortest left-maximal extension VW with $|V| \geq 0$, i.e. the shortest maximal repeat that contains W as a (not necessarily proper) suffix, has loci $(\overline{\mathbb{A}}(w), w)$; and if W is neither left- nor right-maximal, then the shortest maximal repeat UWV with the same frequency as W has loci $(\overline{\mathbb{A}}(\mathbb{A}(v)), \mathbb{A}(v)) = (\overline{\mathbb{A}}(w), \mathbb{A}(\overline{\mathbb{A}}(w)))$ [50]. Thus, in what follows we ignore affix links from leaves.

Rather than storing $\mathbb{A}(w)$ for every internal node w of ST, it has been proposed to sample $\mathbb{A}(w)$ every p suffix links [18]: indeed, $\mathbb{A}(w)$ is either $v = \mathbb{A}(\texttt{suffixLink}(w))$, if $|\ell(v)| \geq |\ell(w)|$, or it is the child of v obtained by following the first character of $\ell(w)$ [50]. This allows one to compute $\mathbb{A}(w)$ in $O(p)$ time, paying $O((|T|/p) \log n)$ bits of space. We briefly observe that, compared to existing sampling schemes for bidirectional indexes, we can further reduce space to $O((|T|/p) \log m)$ bits, where m is the number of maximal repeats of T, since, by Property 2, $\mathbb{A}(v)$ is a maximal repeat of T for every internal node v of ST_T. In practice following Weiner links is faster than following suffix links: thus, one could sample the value of $\mathbb{A}(w)$ for every maximal repeat, and then sample every p characters inside an edge of $\overline{\mathsf{ST}}$ that connects two maximal repeats, i.e. every p explicit Weiner links. If $\mathbb{A}(w)$ is not sampled, then $\ell(w)$ is not left-maximal, so we take the only possible Weiner link from it and we repeat the search from there, returning the value of the first sampled node we find. This sampling scheme takes $O((m + (|T| - m)/p) \log m)$ bits of space. One could even waive sampling the nodes of ST that are not maximal repeats, but to retrieve their value one would have to pay a number of Weiner links that is at most equal to the length of the longest edge of $\overline{\mathsf{ST}}$ connecting two maximal repeats. Clearly, sampling just maximal repeats works also for the scheme based on suffix links.

In this section we store $\mathbb{A}(w)$ and $\overline{\mathbb{A}}(w)$ explicitly, but just for maximal repeats, together with CDAWG_T and $\overline{\mathsf{CDAWG}}_T$, to implement an infinite-order de Bruijn graph in which the time to increase or decrease the order does not depend on the difference between the source and the destination order:

▶ **Theorem 2.** *Given a string T, there are a fully-functional bidirectional index, and an infinite-order representation of the de Bruijn graph of T, that take space proportional to the number of left and right extensions of the maximal repeats of T, and that support all queries in $O(\log \log |T|)$ time.*

Proof. We represent ST and $\overline{\mathsf{ST}}$ using CDAWGs, as described in [8] and summarized in Section 2.2 of this paper. In addition to RLBWT, $\overline{\mathsf{RLBWT}}$, CDAWG and $\overline{\mathsf{CDAWG}}$, to support Theorem 1 we store also a weighted level ancestor data structure on the maximal repeat subgraph of ST and $\overline{\mathsf{ST}}$, which takes $O(m)$ space and answers queries in $O(\log \log |T|)$ time [1, 24], and we store $\mathbb{A}$ and $\overline{\mathbb{A}}$ to support changes in the order of the de Bruijn graph. We represent an arbitrary substring W of T as a triple $(\texttt{id}(v), \texttt{id}(w), |W|)$, where v is the locus of W in ST, w is the locus of $\overline{W}$ in $\overline{\mathsf{ST}}$, and $\texttt{id}$ is the identifier of a node in the CDAWG-based representation of a suffix tree, i.e. $\texttt{id}(v) = (v', |\ell(v)|, i, j)$ where v' is a node of a CDAWG and $[i..j]$ is a BWT interval.

To implement $\texttt{extendRight}(W, c)$, where Wc is assumed to occur in T, we first check whether W is right-maximal, by comparing $|W|$ to $|\ell(v)|$: if W is not right-maximal, then the representation of Wc is $(\texttt{id}(v), \texttt{weinerLink}(\texttt{id}(w), c), |W| + 1)$. Otherwise, the representation is $(\texttt{child}(\texttt{id}(v), c), \texttt{weinerLink}(\texttt{id}(w), c), |W| + 1)$. If we assume that procedure $\texttt{extendRight}(W, c)$ can be called with an invalid c, we first have to check whether Wc occurs in T using the interval of W in $\overline{\mathsf{BWT}}$. To implement $\texttt{contractLeft}(aW)$, we first check whether aW is right-maximal, by comparing $|aW|$ to $|\ell(v)|$: if so, the representation of W is $(\texttt{suffixLink}(\texttt{id}(v)), \texttt{id}(w'), |W|)$, where w' is either the parent of w or w itself,

depending on which one of them has the same frequency as the locus of W in ST. If aW is not right-maximal, we run the algorithm in Theorem 1 using the `suffixLink` and `parent` operations provided by the CDAWG-based representation of ST.

To implement `decreaseOrder` and `increaseOrder` in the de Bruijn graph, we proceed as follows. If the current k-mer W is right-maximal, the representation of the longest suffix of W that is a maximal repeat is clearly $(\mathtt{id}(z), \mathtt{id}(\mathbb{A}(z)), |\ell(z)|)$, where z is the maximal repeat reached by taking a suffix link arc from the node of the CDAWG pointed by $\mathtt{id}(v)$. One could further move to a suitable ancestor of such a maximal repeat, by marking the topology of the maximal repeat subgraph of ST. If the current W is left-maximal but not right-maximal, the representation of the shortest maximal repeat of the form WV for some nonempty V is $(\mathtt{id}(z), \mathtt{id}(\mathbb{A}(z)), |\ell(z)|)$, where z is the node of the CDAWG pointed by $\mathtt{id}(v)$. The same holds if W is neither left- nor right-maximal, and if we want to move to the shortest k-mer that contains W and is both left- and right-maximal. Implementing the other operations of a bidirectional de Bruijn graph is straightforward and is left to the reader. We use data structures from [7] to answer the membership query $\mathtt{node}(W)$ in $O(|W|)$ time. ◀

Our construction based on two CDAWGs is reminiscent of the *symmetric compact DAWG* described in [16], which was used however just for bidirectional extension. Theorem 2 could be simplified in several ways for a practical implementation. For example, as noted already in [16], since CDAWG and $\overline{\mathsf{CDAWG}}$ share the same set of nodes, every such node could be stored only once, in which case $\mathbb{A}$ and $\overline{\mathbb{A}}$ would not need to be represented explicitly. If the descriptor of a substring W is $(\mathtt{id}(v), \mathtt{id}(w), |W|)$ with $\mathtt{id}(v) = (v', |\ell(v)|, i, j)$ and $\mathtt{id}(w) = (w', |\ell(w)|, i', j')$, then v' and w' would become pointers to the same node, $|\ell(w)|$ could be derived from $|\ell(v')| - |\ell(v)| + |W|$, and rather than storing i, j and i', j', we could just store $i, i', f(W)$. Our representation collapses to the sink of a CDAWG all k-mers that occur just once in the dataset, which are likely induced by sequencing errors and are thus not useful for most applications: in this case, we don't even need to store left and right extensions of maximal repeats directed to the sink. If the target application never uses orders smaller than a threshold τ, we could remove from the index all maximal repeats of length smaller than τ and prune the top part of the corresponding tree data structures, as described in [22]. We could proceed in a similar way when the user specifies a lower bound on the frequency of k-mers (called *solid*, see e.g. [29, 37]).

5 Discussion and extensions

Our CDAWG-based representation of the de Bruijn graph might be practical: a full experimental study and a careful implementation of each primitive would be an interesting research direction. Given a node v in the de Bruijn graph, it would also be interesting to know if we can traverse an entire maximal non-branching path, i.e. a path in which no k-mer except for v and the destination has more than one arc to the left and to the right, without taking time proportional to the length of such a path: this would provide a fast implementation of the *compacted* de Bruijn graph (see e.g. [19, 36] and references therein). It is natural to wonder whether one can support the operations of an infinite-order de Bruijn graph in less space than our indexes. Another open question is whether the CDAWG can be used as a substrate for implementing the *string graph* as well, and whether we can design a single compact index, as wished by [23], that supports both the primitives of a string graph and of an infinite-order de Bruijn graph efficiently, allowing the user to take advantage of both approaches in genome assembly.

References

1 Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms (TALG)*, 3(2):19, 2007.

2 Alberto Apostolico and Gill Bejerano. Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. *Journal of Computational Biology*, 7(3-4):381–393, 2000.

3 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform. *Bioinformatics*, 32(4):497–504, 2015.

4 Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proceedings of the forty-sixth Annual ACM Symposium on Theory of Computing*, pages 148–193. ACM, 2014.

5 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *International Symposium on String Processing and Information Retrieval*, pages 179–190. Springer, 2014.

6 Djamal Belazzougui and Fabio Cunial. A Framework for Space-Efficient String Kernels. *Algorithmica*, 79(3):857–883, 2017.

7 Djamal Belazzougui and Fabio Cunial. Fast label extraction in the CDAWG. In *International Symposium on String Processing and Information Retrieval*, pages 161–175. Springer, 2017.

8 Djamal Belazzougui and Fabio Cunial. Representing the Suffix Tree with the CDAWG. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 78. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

9 Djamal Belazzougui, Fabio Cunial, and Olgert Denas. Fast matching statistics in small space. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 103. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

10 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Annual Symposium on Combinatorial Pattern Matching*, pages 26–39. Springer, 2015.

11 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile Succinct Representations of the Bidirectional Burrows-Wheeler Transform. In *21st Annual European Symposium on Algorithms (ESA 2013)*, volume 8125 of *Lecture Notes in Computer Science*, pages 133–144, France, 2013. Springer.

12 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *arXiv preprint*, 2016. `arXiv:1609.06378`.

13 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J Puglisi. Bidirectional variable-order de Bruijn graphs. In *Latin American Symposium on Theoretical Informatics*, pages 164–178. Springer, 2016.

14 Michael A Bender and Martın Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

15 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

16 Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

17 Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *2015 Data Compression Conference*, pages 383–392. IEEE, 2015.

18 Rodrigo Cánovas and Eric Rivals. Full Compressed Affix Tree Representations. In *Data Compression Conference (DCC), 2017*, pages 102–111. IEEE, 2017.

19 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

20 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In Alberto Apostolico and Jotun Hein, editors, *CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1997.

21 Fabio Cunial, Jarno Alanko, and Djamal Belazzougui. A framework for space-efficient variable-order Markov models. *bioRxiv preprint*, page 443101, 2018.

22 Diego Díaz-Domínguez, Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Gonzalo Navarro, and Simon J Puglisi. Assembling Omnitigs using Hidden-Order de Bruijn Graphs. *arXiv preprint*, 2018. `arXiv:1805.05228`.

23 Diego Díaz-Domínguez, Travis Gagie, and Gonzalo Navarro. Simulating the DNA String Graph in Succinct Space. *arXiv preprint*, 2019. `arXiv:1901.10453`.

24 Martin Farach and S Muthukrishnan. Perfect hashing for strings: formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140. Springer, 1996.

25 Simon Gog, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, and Niko Välimäki. Multi-pattern matching with bidirectional indexes. *Journal of Discrete Algorithms*, 24:26–39, 2014.

26 Philipp Koch, Matthias Platzer, and Bryan R Downie. RepARK — de novo creation of repeat libraries from whole-genome NGS reads. *Nucleic Acids Research*, 42(9):e80–e80, 2014.

27 Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

28 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on*, pages 31–36. IEEE, 2009.

29 Dinghua Li, Ruibang Luo, Chi-Man Liu, Chi-Ming Leung, Hing-Fung Ting, Kunihiko Sadakane, Hiroshi Yamashita, and Tak-Wah Lam. MEGAHIT v1.0: a fast and scalable metagenome assembler driven by advanced methodologies and community practices. *Methods*, 102:3–11, 2016.

30 Moritz G Maaß. Linear bidirectional on-line construction of affix trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 320–334. Springer, 2000.

31 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching*, pages 45–56. Springer, 2005.

32 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.

33 Shoshana Marcus and Dina Sokol. Engineering Small Space Dictionary Matching. *arXiv preprint*, 2013. `arXiv:1301.6428`.

34 Giancarlo Mauri and Giulio Pavesi. Pattern discovery in RNA secondary structure using affix trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 278–294. Springer, 2003.

35 Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. *bioRxiv preprint*, page 548123, 2019.

36 Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, 2016.

37 Pierre Morisse, Thierry Lecroq, and Arnaud Lefebvre. Hybrid correction of highly noisy long reads using a variable-order de Bruijn graph. *Bioinformatics*, 34(24):4213–4222, 2018.

38 J Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 408–424. SIAM, 2017.

39 J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

40 Gonzalo Navarro. *Compact data structures: a practical approach.* Cambridge University Press, 2016.

41 Gonzalo Navarro and Kunihiko Sadakane. Fully Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, 2014.

42 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, 2017.

43 Nicolas Philippe, Mikaël Salson, Thérèse Commes, and Eric Rivals. CRAC: an integrated approach to the analysis of RNA-seq reads. *Genome Biology*, 14(3):R30, 2013.

44 Mathieu Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165–169, 2001.

45 Luís Russo, Gonzalo Navarro, Arlindo L Oliveira, and Pedro Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

46 K. Sadakane and G. Navarro. Fully-Functional Succinct Trees. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 134–149, Austin, Texas, USA, 2010. ACM-SIAM.

47 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012.

48 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008.*, pages 164–175, 2008.

49 Jens Stoye. Affix trees. Master's thesis, Universität Bielefeld, 2000.

50 Dirk Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theoretical Computer Science*, 389(1-2):278–294, 2007.

51 Aaron M Wenger et al. Highly-accurate long-read sequencing improves variant detection and assembly of a human genome. *bioRxiv preprint*, 2019. doi:10.1101/519025.

Entropy Lower Bounds for Dictionary Compression

Michał Gańczorz
Institute of Computer Science, University of Wrocław, Poland
mga@cs.uni.wroc.pl

Abstract

We show that a wide class of dictionary compression methods (including LZ77, LZ78, grammar compressors as well as parsing-based structures) require $|S|H_k(S) + \Omega\left(|S|k \log \sigma / \log_\sigma |S|\right)$ bits to encode their output. This matches known upper bounds and improves the information-theoretic lower bound of $|S|H_k(S)$. To this end, we abstract the crucial properties of parsings created by those methods, construct a certain family of strings and analyze the parsings of those strings. We also show that for $k = \alpha \log_\sigma |S|$, where $0 < \alpha < 1$ is a constant, the aforementioned methods produce an output of size at least $\frac{1}{1-\alpha}|S|H_k(S)$ bits. Thus our results separate dictionary compressors from context-based one (such as PPM) and BWT-based ones, as the those include methods achieving $|S|H_k(S) + \mathcal{O}(\sigma^k \log \sigma)$ bits, i.e. the redundancy depends on k and σ but not on $|S|$.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases compression, empirical entropy, parsing, lower bounds

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.11

Funding Supported under National Science Centre, Poland project number 2017/26/E/ST6/00191.

1 Introduction

Dictionary compression. Dictionary compression is one of the most known and intensively studied area in data compression. As a result, there are many both simple and efficient methods that are commonly used in practice, those include algorithms from Lempel-Ziv family: LZ77 [34] and LZ78 [35], grammar compressors: Re-Pair [23], Greedy [4, 5], Sequitur [28], Sequential [33] as well as others [20, 32]. On top of that, there are also compressed data structures based on dictionary compression [16, 13, 8, 3, 22, 25, 7, 8, 19, 26].

Many methods mentioned above share a common feature – they induce a *parsing* of the input string. This is explicit for LZ77 and LZ78, as they both output a series of phrases, for grammar compressors the parsing is not always explicit, but it can be done fairly easy. For the mentioned data structures this is also either explicit [16, 13, 3, 22, 25] or easily done [7, 8]. The actual encodings of the phrases differ between the methods, but often we can lower bound their size by (zeroth order) entropy of the induced parsing. For example, in LZ78, for parsing with c phrases each phrase is assigned different (prefix-free) bit code, thus (by the properties of the prefix-free codes) this claim trivially holds (note also that all phrases in parsing induced by LZ78 are different). Another good example is grammar compressors: Re-Pair explicitly encodes the starting string of the grammar using entropy coder [23], Sequential uses its own encoding [33], which can be lower bounded by zeroth order entropy, etc.

Higher order empirical entropy. The k-th order empirical entropy, denoted $H_k(S)$ for a string S, is one of the most widely used measure of compressibility of texts, as on one hand in practice it is a good estimation of the "compressibility" of the text and on the other hand there are compressors that roughly achieve it. Surprisingly, for many dictionary compression methods it was shown that on a text S over an alphabet of size σ their output

is of size at most

$$\beta|S|H_k(S) + \mathcal{O}(k|S|\log\sigma/\log_\sigma|S|) + \mathcal{O}(|S|\log\log_\sigma|S|/\log_\sigma|S|) \quad (1)$$

bits, for some constant β (ideally $\beta = 1$) [16, 13, 21, 27, 15, 30]. When $\beta = 1$ then the terms other than $|S|H_k(S)$ are often called the *redundancy* of the encoding. The summands under $\mathcal{O}$ notation are in $o(|S|\log\sigma)$ for $k = o(\log_\sigma|S|)$, such term is often treated as "lower order term". To capture this phenomenon, Kosaraju and Manzini introduced the notion of *coarse optimality* [21]: an algorithm is (β) coarse optimal if it achieves $\beta|S|H_k(S) + o(|S|\log\sigma)$ bits (our definition is slightly different than the original one, in which it was assumed that k and σ are constant). However, for larger k the additional term in (1) is in $\Omega(|S|\log\sigma)$, which is not satisfactory, as $|S|H_k(S) \leq |S|\log\sigma$ and so the additional term dominates the main one.

The bound in (1) for particular methods is in fact connected to a recently proved general upper bound on (zeroth-order) entropy of *arbitrary* parsing:

▶ **Theorem 1** ([15, Theorem 7]). *For any parsing $Y_S = y_1y_2\cdots y_c$ of S:*

$$cH_0(Y_S) \leq |S|H_k(S) + ck\log\sigma + |L|H_0(L) \ , \quad (2)$$

where L is string whose letters are lengths of consecutive phrases in Y_S, i.e. $L = |y_1|\,|y_2|\cdots|y_c|$.

To see the connection, observe that parsings produced by considered methods are usually of size at most $\mathcal{O}(|S|/\log_\sigma|S|)$, thus $ck\log\sigma$ turns to $\mathcal{O}(k|S|\log\sigma/\log_\sigma|S|)$, and for such sizes the $|L|H_0(L)$ can be estimated by $\mathcal{O}(|S|\log\log_\sigma|S|/\log_\sigma|S|)$. Of course, the considered methods encode also other information, which increases the constants in $\mathcal{O}$ notation or increases the constant in front of $|S|H_k(S)$.

Our results. The main result of this paper is a proof that the estimation (1) is tight for $k < \log_\sigma|S|$, at least for "natural" algorithms. Since estimating the size of the produced output is hard due to different encodings, we prove that the estimation on parsing entropy given by Theorem 1 is tight for such "natural" algorithms; note that this means that we disregard the size of other parts of the output. This is the reason, why we have to restrict the considered parsings, and so also the algorithms producing them, as it is impossible to, say, show any lower bound for a trivial parsing which consists of one phrase. To this end we define the class of *natural parsers*; this definition covers most of the dictionary based compressors, in particular it contains LZ77, LZ78, Re-Pair and grammar compressors producing irreducible grammars (such as Greedy or Sequitur). It also covers succinct data structures which parse the string into short phrases. We require that each phrase $y = wa$ in a parsing of the input string S induced by an algorithm is short (i.e. $|y| < \log_\sigma|S|$) or w occurs in S more than once. Note that the seemingly artificial condition on w and not on $y = wa$ is used in order to capture LZ77 and LZ78; the condition on short phrases ensures that some structures based on parsing into equal-length blocks [16, 13] are also covered.

Theorem 1 holds for any parsing, and it is known that the bound can be made more subtle when we can choose a specific parsing: at least one of l trivial parsings into equal-length phrases of length l achieves the mean of the first l entropies (plus smaller order term).

▶ **Theorem 2** ([15, Theorem 2]). *Let S be a string over an alphabet of size σ. Then for any integer l we can construct a parsing Y_S of size $|Y_S| \leq \left\lceil \frac{|S|}{l} \right\rceil + 1$ satisfying:*

$$|Y_S|H_0(Y_S) \leq |S|\frac{\sum_{i=0}^{l-1} H_i(S)}{l} + \mathcal{O}(\log|S|) \ .$$

All phrases except the first and last one have length l.

It is easy to give examples, for which Theorem 2 is tight, for instance de Bruijn sequence for $l \leq \log_\sigma |S|$ are examples of Theorem 2 tightness. On the other hand, natural parsers can perform better on de Bruijn strings than the bound in (1): we can show that they achieve $|S|H_k(S) + \mathcal{O}(|S| \log \log_\sigma |S| / \log_\sigma |S|) = |S| \log \sigma + \mathcal{O}(|S| \log \log_\sigma |S| / \log_\sigma |S|)$ for $k < \log_\sigma |S|$. This is because we can assign to each factor of length l a prefix-free code of length $\log \log_\sigma |S| + l \log \sigma$ bits.

We construct nontrivial examples on which bound from Theorem 2 is tight for any parsing produced by natural parsers for $l \in \Theta(\log_\sigma |S|)$. Moreover, each constructed string S satisfies

$$\log \sigma = H_0(S) \approx H_1(S) \ldots \approx H_{k-1}(S) \approx c \cdot H_k(S) \approx \ldots \approx c \cdot H_{l-1}(S) \ , \tag{3}$$

where k can be any number such that $k \leq \log_\sigma |S|$, c is some constant greater than 1, and $\approx$ means that any two values from the sequence differ by at most $\mathcal{O}(\text{polylog}|S|/|S|) = o(1)$. In particular, in the limit for $|S| \to \infty$ they all tend to the same value. Those strings demonstrate that the first $\mathcal{O}(\cdot)$ summand in (1) is asymptotically optimal:

▶ **Corollary 3** (short version of Corollary 12). *Let* A *be a natural parser. Then for large enough σ there exists an infinite family of strings $\{S_n\}_{n=1}^{\infty}$ over σ-size alphabet such that the size of the output generated by* A *on $S \in \{S_n\}_{n=1}^{\infty}$ is at least $|S|H_k(S) + \Omega\left(\frac{|S|k \log \sigma}{\log_\sigma |S|}\right)$, for $k \leq \log_\sigma |S| - \frac{1}{2}$*

We can generalize the construction so that c in (3) is arbitrarily large (it depends on l, though), as a result we are able to prove that for $k = \alpha \log_\sigma |S|$, where $0 < \alpha < 1$, natural parsers output cannot be bounded with respect to $H_k(S)$.

▶ **Corollary 4** (short version of Corollary 14). *Let $0 < \alpha < 1$ be a rational constant and* A *be a natural parser. Then for large enough σ there exists an infinite family of strings $\{S_n\}_{n \in \mathbb{N}}$ over σ-size alphabet such that if* A *achieves $\beta|S|H_k(S) + o(|S| \log \sigma)$ bits on each $S \in \{S_n\}_{n \in \mathbb{N}}$ for $k = \alpha \log_\sigma |S|$, then $\beta \geq \frac{1}{1-\alpha}$.*

The bounds provided in Corollary 3 and Corollary 4 give higher lower bounds than information-theoretic ones, i.e. those based on counting the number of strings satisfying certain conditions. In fact, our proofs do not use information-theoretic arguments, instead, we give explicit construction of a string and analyze how natural parsers can parse it.

Consequences. It was asked [27], whether one can bound the size of Re-Pair's output by $c|S|H_k(S) + o(|S| \log \sigma)$, for some constant c, when $k = \alpha \log_\sigma |S|$. Corollary 4 settles this question in the negative.

The provided bounds can be used to compare the parsing-based methods with context-based ones (like PPM [9]) or ones based on BWT [11, 24]. It is known that their output is of size at most $|S|H_k(S) + \mathcal{O}(\sigma^k \cdot \log \sigma)$, even for $k = \alpha \log_\sigma |S|$, i.e. their redundancy is $o(|S|)$ [11, 24, 10]. Thus our theoretical bound backs a practically observed phenomenon of superiority of those methods over dictionary-based ones, at least for non-repetitive texts [9, 2, 1].

Lastly, we believe that the strings constructed in this paper are interesting on their own and can become "benchmark strings" for data compression algorithms.

Value of k. In the paper we consider $k = o(\log_\sigma |S|)$ and $k = \alpha \log_\sigma |S|$ for a constant $0 < \alpha < 1$. Let us comment, why those are the reasonable values of k (as a function of $|S|$ and σ). The value of k cannot be too large: for $k \geq \log_\sigma |S|$ the H_k can be smaller than

information-theoretic bound [14]. The case of "small" k is clearly $k = o(\log_\sigma |S|)$: on one side it is traditionally thought to be small, on the other Corollary 3 shows that it is the necessary assumption to obtain reasonable bounds for a wide range of compression algorithms. Lastly, the intermediate case of $k = \alpha \log_\sigma k$ for some constant $0 < \alpha < 1$ considered in Corollary 4 is also well-motivated: the already mention context or BWT-based methods have good theoretical bounds for such k.

Related results. Kosaraju and Manzini considered also stronger notions than mentioned above: an algorithm is λ-*optimal* if it gives output of size at most $\lambda|S|H_k(S)+o(|S|H_k(S))$ [21]. They showed many negative results for λ-optimality of LZ77 and LZ78: the LZ78 is not λ-optimal for any k, and that LZ77 is not λ-optimal for $k = 1$ (but is 8-optimal for $k = 0$). Still, we know very little about λ-optimality of compression algorithms and our results seem to only give partial negative answer for large enough k.

2 Strings and their parsings

A string is a sequence of elements, called *letters*, from a finite set, called *alphabet*, and it is denoted as $w = w_1 w_2 \cdots w_k$, where each w_i is a letter, its length $|w|$ is k; alphabet's size is denoted by σ, the alphabet is usually not named explicitly as it is clear from the context or not needed, Γ is used when some name is needed. For any two strings w, w' the ww' denotes their concatenation. By $w[i\,..\,j]$ we denote $w_i w_{i+1} \cdots w_j$, this is a *substring* of w; ϵ denotes the empty string. For a pair of strings v, w the $|w|_v$ denotes the number of different (possibly overlapping) substrings of w equal to v; if $v = \epsilon$ then we set $|w|_\epsilon = |w|$.

A *parsing* of a string S is any representation $S = y_1 y_2 \cdots y_c$, where each y_i is nonempty and is called a *phrase*. We denote a parsing as $Y_S = y_1, \ldots, y_c$ and treat it as a string of length c over the alphabet $\{y_1, \ldots, y_c\}$; in particular $|Y_S| = c$ is its size.

Given a string w its *k-order empirical entropy* is

$$H_k(w) = -\frac{1}{|w|} \sum_{\substack{v:\ |v|=k \\ a:\ \text{letter}}} |w|_{va} \log \left(\frac{|w|_{va}}{|w|_v} \right) \ ,$$

with the convention that the summand is 0 whenever $|w|_{va} = 0$. We are mostly interested in the H_k entropy of the input string S and in the $H_0(Y_S)$ for parsing Y_S of S. The former is a natural measure of the input, to which we shall compare the size of considered algorithms, and the latter is a space lower bound for those algorithms, see Definition 5.

3 Natural parsers

We define the class of *natural parsers*. The general idea is that phrases generated by natural parsers are either short or occur at least twice in the string. Those are natural heuristics and so natural parsers include many practically used algorithms.

▶ **Definition 5.** *An algorithm is a* natural-parser *if given a string S over an alphabet of size σ it produces its parsing Y_S such that for each phrase $y = wa$, $|a| = 1$ of Y_S either $|S|_w > 1$, or $y \leq \log_\sigma |S|$; moreover, it encodes Y_S using at least $|Y_S|H_0(Y_S)$ bits.*

The condition that for a phrase $y = wa$ we require $|S|_w > 1$ instead of $|S|_y > 1$ was added so that natural parsers include also LZ78 and non self-referencing LZ77. Note that phrases of length 1 occurring once are allowed, as for them $w = \epsilon$ and $|S|_\epsilon = |S| > 1$.

Natural parsers include: Re-Pair, algorithms producing irreducible grammars, LZ78, LZ77 (for some natural encodings) and compressed text representations that partition the text into short (i.e. $\Theta(\log_\sigma |S|)$) blocks and encode the blocks using zeroth-order entropy [16, 13]. We now argue that all of the above algorithms are natural parsers.

LZ77 and LZ78. The LZ77 [34] and LZ78 [35] are probably the most known dictionary compressors. Each of them processes the input from left to right and builds a parsing in the process. In the LZ77 if, at some point, we processed some prefix P (of length j) of S the next phrase will be the longest possible substring starting at $S[j+1]$ which occurs in S at position at most j, plus one letter. We distinguish between *self-referencing* LZ77, where the previous occurrence of string of a factor may overlap the factor, and *non-self-referencing*, where it cannot (in other words, it occurs in P). Observe that both definition satisfy the first condition of natural parsers. When it comes to the entropy condition, the original encoding of LZ77 stored for each phrase its starting position in P using fixed length code, so it consumed at least $\log |S|$ bits per phrase, in total this is at least the entropy of the parsing (recall that we have $H_0(S) \leq \log |S|$ for every S). Other encodings are also possible, they are more of a practical than theoretical improvement: for example Kosaraju and Manzini [21] stored the position using $\log |P|$ bits, which can be shown to be equivalent to the original one minus lower order term of $o(|S| \log \sigma)$. Sometimes we store the offset, i.e. the difference between the starting position of phrase in P and j, though on average none of this encodings give better bound than $\log |Y_S|$ bits per phrase. On top of that there are also some LZ77-based data structures, like LZ-indices [26, 19, 22], they also use at least $\log |Y_S|$ bits per phrase.

In the case of LZ78, we build a dictionary (at the beginning it is empty), when we processed some prefix of length j the next phrase is wa where w is the longest string which starts in $S[j]$ and is in the dictionary. After each step we update our dictionary and add newly created phrase to it. Thus the description of phrase consists of index of string in a dictionary and a letter. Even though this descriptions can be encoded differently, usually the encodings (including the original one [35]) assign different (prefix-free) bit-codes to different phrases, thus by optimality of Huffman coding $|Y_S|H_0(Y_S)$ is the lower bound on output size. Note also that no two phrases have the same description, so it is sufficient (and necessary) to assign each phrase a fixed code of $\log |Y_S|$ bits, in fact some LZ78-based methods work this way [31]. Note though, that in some variants with limited dictionary size this may not be true.

Re-Pair. Re-Pair [23] is a grammar compressor, which builds a grammar generating the input string in the following way: we start with input string S and iteratively replace occurrences of the most frequent digram AB in S with a new symbol X, adding a rule $X \to AB$ to grammar. We iterate this procedure as long as there is a digram occurring at least twice. At the end we are left with a string which naturally induces a parsing of S, the original encoding of Re-Pair uses zeroth-order entropy to encode the parsing. Note that due to the fact that we only replace digrams which occur twice, each phrase in the parsing has at least two occurrences in S.

Irreducible Grammars. There is a large family of grammar compressors generating *irreducible grammars*. For example this family includes Greedy [4, 5], Sequential [33], Sequitur [29] and LongestMatch [20]. The conditions for a grammar to be irreducible are that:

- each nonterminal that occurs on right-hand side of grammars productions must occur at least twice
- no pair of nonterminals occurs twice on the right-hand side of the grammars productions
- no two nonterminals expands to the same string.

As an example $\{S \to AABB; A \to ab; B \to cd\}$ is an irreducible grammar, but neither $\{S \to AAB; A \to ab; B \to cd\}$ nor $\{S \to AAAA; A \to ab\}$ are.

The parsing induced by the right-hand side of starting symbol already satisfies the condition of a natural parser. There are many methods of encoding such grammars. The simplest involves entropy-coding of grammars right-hand sides, other assignes prefix-free codes to nonterminals [33, 4, 29]. A more sophisticated method was proposed by Kieffer and Yang [20], they showed that we can obtain the parsing from irreducible grammars by subsequently substituting one occurrence of each nonterminal till there are none left (at the end we are left with one string consisting of nonterminals) and then apply entropy coder. Such methods still are natural parsers, as each nonterminal of an original grammar still occurs in the obtained string (by the property that each nonterminal occurs twice) and thus each phrase of a parsing has at least two occurrences in the input string.

Other examples. There are compressed text representations which partition the string into short (i.e. of length at most $\log_\sigma |S|$) blocks and encode the blocks using zeroth-order entropy coder [16, 13, 12]; clearly they are also natural parsers.

4 Lower bound on parsing-based methods

In this section we construct a family of strings for which the bounds from Theorem 2 and Theorem 1 are tight for natural parsers. Furthermore, for those strings the $H_0, \ldots, H_{k-1}$ are by a constant factor c larger than $H_k, \ldots, H_l$, for adequate $l \in \Theta(\log_\sigma |S|)$. For ease of presentation we first show the construction for $c = 2$ and next we generalize to arbitrarily large c. Note that $H_0 = cH_k$ means that the mean of entropies cannot be contained in lower order term such as $o(|S| \log \sigma)$.

Our construction extends the one of de Bruijn strings, which, for a given alphabet Γ and order k, contain exactly once each string $w \in \Gamma^k$ as a substring; de Bruijn strings were used for proving lower bounds on compressibility before [14]; yet, contrary to previous results, our lower bounds are not information-theoretic.

▶ **Theorem 6.** *For every $k > 0$, $l \geq 0$, $p \geq 1$ there exists a string S over alphabet of size $\sigma = 4^p$ of length $\sigma^{k+\frac{l+1}{2}}$ such that:*

1. $\log \sigma - \mathcal{O}\left(\frac{i \log |S|}{|S|}\right) \leq H_i(S) \leq \log \sigma$ *for $i < k$;*
2. $\frac{\log \sigma}{2} - \mathcal{O}\left(\frac{i \log |S|}{|S|}\right) \leq H_i(S) \leq \frac{\log \sigma}{2}$ *for $k \leq i \leq k + l$;*
3. *no string of length $k + l + 1$ occurs more than once in S.*

For $l = 0$ the promised family are constructed from de Bruijn strings by appropriate letter merges. For de Bruijn strings the frequency of each substring depends (almost) only on its length, thus the bounds for the entropy of strings constructed in this way are easy to show. For larger l we make an inductive (on l) construction, similar in spirit to construction of de Bruijn strings: we construct a graph with edges labelled with letters and the desired string corresponds to an Eulerian cycle in this graph; to be more precise, the $(l+1)$st graph is exactly the line graph of the lth one. We guarantee that the frequency of strings depends only on their lengths, the exact condition is more involved than in case of de Bruijn strings.

The promised family of strings is much easier to define if we think of them as *cyclic strings*, meaning that after reading the last letter we can continue to read from the beginning. To distinguish strings from cyclic strings we denote by $S^{\circlearrowleft}$ the cyclic variant of S. Note that for a cyclic string $S^{\circlearrowleft}$ we are interested only in the occurrences of substrings in it (as defined below) thus the technical details of how many times we read S cyclically or when do we stop, are unimportant.

A string w occurs in $S^{\circlearrowleft}$ if w occurs in S, or $w = w_1w_2$ and w_1 is a suffix of S and w_2 a prefix; we still require that $|w| \leq |S|$. The starting positions of an occurrence of w in $S^{\circlearrowleft}$ is defined naturally, two occurrences are different if they start at different positions. Denote by $|S|_w^{\circlearrowleft}$ the number of different occurrences of w in $S^{\circlearrowleft}$. Using this notation we define cyclic k-order entropy as:

$$H_k^{\circlearrowleft}(w) = -\frac{1}{|w|} \sum_{\substack{v:\ |v|=k \\ a:\ \text{letter}}} |w|_{va}^{\circlearrowleft} \log\left(\frac{|w|_{va}^{\circlearrowleft}}{|w|_v^{\circlearrowleft}}\right) \ .$$

The difference between cyclic and standard k-th order entropy is that it takes into the account also the first k letters of w. It is easy to show that it differs from $|w|H_k(w)$ in a small amount.

▶ **Lemma 7.** *For any string S and for any k we have:*

$$|S|H_k(S) + k\log|S| + \mathcal{O}(k) \geq |S|H_k^{\circlearrowleft}(S) \geq |S|H_k(S) \ .$$

We now give the main construction, using the cyclic occurrences the estimation of Theorem 6 simplify as follows:

▶ **Lemma 8.** *For every $k > 0$, $l \geq 0$, $p \geq 1$ there exists a string S of length $\sigma^{k+\frac{l+1}{2}}$ over an alphabet Γ' of size $\sigma = 4^p$ such that:*

(dB1) *For every $w \in (\Gamma')^i, i < k$ we have $|S|_w^{\circlearrowleft} = \sigma^{k-i+(l+1)/2}$,*

(dB2) *For every $w \in (\Gamma')^i, k \leq i \leq k+l+1$ we have either $|S|_w^{\circlearrowleft} = \sigma^{(k+l+1-i)/2}$ or $|S|_w^{\circlearrowleft} = 0$,*

(dB3) *No string of length $k+l+1$ occurs cyclically more than once in S.*

Proof. Fix k, by S_l we will denote the string that satisfies the conditions (dB1–dB3) for l.

Let us first construct S_0. Consider cyclic de Bruijn sequence $B^{\circlearrowleft} = a_1a_2\cdots a_n$ of order $2k+1$ over an alphabet Γ of size $\sqrt{\sigma}$ (this is well defined, as $\sigma = 4^p$). Then $|B| = (\sqrt{\sigma})^{2k+1} = \sigma^{k+\frac{1}{2}}$. Consider two parsings of $B^{\circlearrowleft}$ into pairs of letters:

$$Y_B^1 = |a_1a_2|a_3a_4|\cdots|a_{n-3}a_{n-2}|a_{n-1}a_n| \quad Y_B^2 = |a_2a_3|a_4a_5|\cdots|a_{n-2}a_{n-1}|a_na_1|$$

Now replace each pair a_i, a_j with a new symbol $b_{i,j}$, such that $b_{i,j} \neq b_{i',j'}$ if and only if $(a_i, a_j) \neq (a_{i'}, a_{j'})$. The size of the new alphabet Γ' is $\sigma = 4^p$. Consider the corresponding strings B_1' and B_2', treated in the following as cyclic strings:

$$B_1' = b_{1,2}b_{3,4}\cdots b_{n-3,n-2}b_{n-1,n} \quad B_2' = b_{2,3}b_{4,5}\cdots b_{n-2,n-1}b_{n,1}$$

We can choose B such that it begins with a_i^{2k+1}, for some a_i. Then both strings B_1' and B_2' begin with $b_{i,i}^k$. Take $S_0 = B_1'B_2'$. Then, as the starting k-letters of both of them are the same, for each v of length at most $k+1$ it holds that

$$|B_1'|_v^{\circlearrowleft} + |B_2'|_v^{\circlearrowleft} = |S_0|_v^{\circlearrowleft} \ .$$

We now calculate $|S_0|_w^{\circlearrowleft}$ for each possible w. For each k-letter string w' over Γ' the $|B_1'|_w^{\circlearrowleft} + |B_2'|_w^{\circlearrowleft}$ is $\sqrt{\sigma}$: w' is obtained from a fixed $2k$-letter string $w \in (\Gamma)^{2k}$ and such a string occurs cyclically $\sqrt{\sigma}$-times in $B^{\circlearrowleft}$, as there are $\sqrt{\sigma}$ ways to extend w to a $(2k+1)$-letter string and each such a string occurs cyclically exactly once in $B^{\circlearrowleft}$ and each cyclic occurrence of w in $B^{\circlearrowleft}$ yields one cyclic occurrence of w' in exactly one of $B_1'^{\circlearrowleft}$ and $B_2'^{\circlearrowleft}$. Moreover, as each string v of length $2k+1$ has exactly one occurrence in B, the letters after different cyclic occurrence of $w' \in (\Gamma')^k$ in $B_1'^{\circlearrowleft}$ or $B_2'^{\circlearrowleft}$ are pairwise different. Hence, each string of

length at least $k+1$ over Γ' has at most one occurrence in S_0. For a string w of length $i<k$ observe that each of its σ^{k-i} extensions to a k-letter string occurs cyclically exactly $\sqrt{\sigma}$ times in $S_0^{\circlearrowleft}$, thus w occurs exactly $\sigma^{k-i+1/2}$. Thus S_0 satisfies conditions (dB1–dB3).

We now move to the general case of $l>0$. We cannot simply define S_l as a power of S_0, as then (dB3) is violated. Instead, we proceed similarly to the standard construction of de Bruijn strings: we will build a graph with vertices labelled with different strings of length $k+l+1$, define edges between strings that can be obtained by shifting by one letter to the right and show that this graph has a Hamiltonian cycle.

Define a family of directed graphs $G_0, G_1, \ldots$, where $G_i = (V_i, E_i)$. The nodes in V_i are labelled with (some) strings of length $k+1+i$ over Γ' (which is of size σ) and $E_i = \{(u,u') : u[2\ldots|u|] = u'[1\ldots|u'|-1]\}$. We label the edge from av to vb with avb. In case of G_0 its vertices V_0 are all cyclic substrings of $S_0^{\circlearrowleft}$ of length $k+1$. Recall that given a directed graph G its *line graph* $L(G)$ has edges of G as nodes and there is an edge (e,f) in $L(G)$ if and only if the end of e is the beginning of f. Define $G_{i+1} = L(G_i)$, observe that edges of G_i have labels that are strings of length $k+i+2$, those labels are reused as labels of nodes in G_{i+1}.

Let us state some basic properties of the defined graph: firstly, G_0 has in-degree and out-degree equal to $\sqrt{\sigma}$ (so it is $\sqrt{\sigma}$-regular): Given a node with a $k+1$-letter label w all its outgoing labels correspond to occurrences of the k letter suffix of w. And by (dB1) each k letter string has $\sqrt{\sigma}$ cyclic occurrences in $|S_0|^{\circlearrowleft}$ and each $k+1$ letter string has at most 1. So there are $\sqrt{\sigma}$ outgoing edges, each leading to a different node. Similar argument applies to the incoming edges. It is a folklore knowledge (and easy to show) that if G is d-regular then so is $L(G)$, moreover, if G is connected then so is $L(G)$; clearly G_0 is connected, as S_0 corresponds to a Hamiltonian path in it. Thus, all G_i's are Eulerian. It is well-known and easy to see that an Eulerian cycle in G corresponds to a Hamiltonian cycle in $L(G)$, thus each G_i has a Hamiltonian cycle.

We define the string $S_i^{\circlearrowleft}$ as the string read when traversing a Hamiltonian path in G_i (note that there may be many such paths: choose one arbitrarily): we begin with an arbitrary vertex u_0 in G_i, write its label and when we traverse the edge avb (so from av to vb) then we append b to the string. By easy induction we can show that when we are at a node labelled with u then the current string has u as a suffix. In particular, after traversing the whole path begins and ends with u_0. By identifying those two copies of u_0 we obtain a cyclic string. Note that a string w of length $k+i+1$ occurs at position p if and only if p-th vertex on the path is labelled with w. Concerning the length $|S_i|$, this is exactly $|V_i| = |E_{i-1}| = \sqrt{\sigma}|V_{i-1}|$, as each G_i is $\sqrt{\sigma}$ regular. Since $|V_0| = \sigma^{k+\frac{1}{2}}$, we conclude that $|V_i| = \sigma^{k+\frac{i+1}{2}}$. We also show that each occurrence of a string w of length $k+i$ in S_i is followed by a different letter, in particular this implies that a string w' of length $k+i+1$ has at most one occurrence in S_i, i.e. (dB3). We know that this is true for G_0, we proceed by induction. Consider all nodes labelled with wa for some letter a in G_{i+1}, where $|w| = k+i+1$. They all correspond to edges in G_i labelled with the same strings. Those edges originate from nodes labelled with w and as $|w| = k+i+1$, by induction assumption there is exactly one such node. Now, if there were two edges outgoing edge from w labelled with wa then they would lead to two vertices labelled with the same label w' (where $w' = w[2\,..\,|w|]a$), which is not the case by the induction assumption. Hence, wa has at most one occurrence.

It is left to show that S_l satisfies (dB1)–(dB2). We proceed by induction on l: first we show that for any string w of length at most $k+l+1$ it holds that $|S_{l+1}|_w^{\circlearrowleft} = |S_l|_w^{\circlearrowleft} \cdot \sqrt{\sigma}$: observe that $|S_{l+1}|_w^{\circlearrowleft}$ is the number of nodes in G_{l+1} that have w as their prefix. This is exactly the number of edge-labels in G_l that have w as their prefix. But those labels are of

length $k+l+2 > |w|$, thus edge $e = (u, u')$ has w as the prefix of its label if and only if this labels is also a prefix of label of u. As G_l is $\sqrt{\sigma}$-regular, each u is counted $\sqrt{\sigma}$ times (once for each of the $\sqrt{\sigma}$ outgoing edges) and so we obtain the claim.

It is left to consider the case when $|w| = k+l+2$, but strings of this lengths are labels of vertices of G_{l+1} and if w is label of some vertex of G_{l+1} then clearly $|S_{l+1}|_w = 1$. ◀

It is worth noting that the construction suggests that for a fixed k (and de Bruijn string) there are exponentially many (in l) strings satisfying conditions (dB1–dB3). Moreover the construction suggests that there are exponentially many (in k) such strings: it seems that for each de Bruijn string the constructed strings is different, and there are exponentially many de Bruijn strings of order k. Proving this does not seem easy and we leave it for future work.

▶ **Example 9.** For $\sigma = 4, k = 2, l = 0$: $S = aababcbbadccdbddaacadaccbdbbcddc$.
Observe that each k-letter substring occurs cyclically $\sqrt{\sigma}$ times, the letters after those occurrences are pairwise different. $H_0^{\circlearrowleft}(S) = H_1^{\circlearrowleft}(S) = \log\sigma$ and $H_2^{\circlearrowleft}(S) = \frac{\log\sigma}{2}$.

For $\sigma = 4, k = 1, l = 1$ the string is $S = abbbdacdcacabdcd$ and $H_0^{\circlearrowleft}(S) = \log\sigma$, $H_1^{\circlearrowleft}(S) = H_2^{\circlearrowleft}(S) = \frac{\log\sigma}{2}$.

Proof of Theorem 6. Take the string S from Lemma 8 for a given k, l, p. By definition of cyclic entropy $H_i^{\circlearrowleft}(S) = \log\sigma$ for $i < k$ and $H_i^{\circlearrowleft}(S) = \frac{\log\sigma}{2}$ for $k \le i \le k+l$, moreover no string of length $k+l+1$ occurs cyclically more than once in S. By Lemma 7, string S satisfies the conditions stated in the Theorem. ◀

In the following we estimate, how bad a natural parser performs on a string from Theorem 6. It is easy to see that for such a string for parameters k, l it cannot make a phrase longer than $k+l+1$, as by (dB3) they have at most one occurrence, so we first lower-bound the entropy of such parses.

▶ **Lemma 10.** *Let S be a string from Theorem 6 for parameters k and l, and let $z = k+l+1$. Then for every parsing $Y_S = y_1 y_2 \dots y_{|Y_S|}$ of S such that $|y_i| \le z$ we have:*

$$|Y_S| H_0(Y_S) \ge \frac{|S|(z+k)}{2z} \log\sigma - |Y_S| \log \frac{|S|}{|Y_S|} \quad .$$

Proof. Let $m = |Y_S|$ and $n = |S|$. For a string w let l_w be the number of occurrences of w in Y_S. Clearly $l_w \le |S|_w^{\circlearrowleft}$ and by construction for any w such that $|S|_w^{\circlearrowleft} > 0$:

$$|S|_w^{\circlearrowleft} = \begin{cases} \frac{n}{\sigma^{|w|}} & \text{for } |w| \le k \\ \frac{n}{\sigma^{(|w|+k)/2}} & \text{for } k < |w| \le z \end{cases} \quad . \tag{4}$$

thus

- $l_w \le \frac{n}{\sigma^{|w|}}$, for $|w| \le k$;
- $l_w \le \frac{n}{\sigma^{(|w|+k)/2}}$, for $k < |w| \le z$.

Define:

$$m_1 = \sum_{w \in Y_S, |w| \le k} l_w \qquad m_2 = \sum_{w \in Y_S, |w| > k} l_w$$

$$n_1 = \sum_{w \in Y_S, |w| \le k} |w| l_w \qquad n_2 = \sum_{w \in Y_S, |w| > k} |w| l_w$$

Note that as each phrase in Y_s has length at most z, we have that

$$m_2 z \ge n_2 \tag{5}$$

Then

$$\begin{aligned}
|Y_S|H_0(Y_S) &= \sum_{w\in Y_S} l_w \log\frac{m}{l_w} \\
&= \sum_{|w|\le k} l_w \log\frac{m}{l_w} + \sum_{|w|>k} l_w \log\frac{m}{l_w} \\
&\ge \sum_{|w|\le k} l_w \log\frac{m\sigma^{|w|}}{n} + \sum_{|w|>k} l_w \log\frac{m\sigma^{(|w|+k)/2}}{n} \\
&= \sum_{w} l_w \log\frac{m}{n} + \sum_{|w|\le k} l_w \log\sigma^{|w|} + \sum_{|w|>k} l_w \log\sigma^{(|w|+k)/2} && \text{from (4)} \\
&= m\log\frac{m}{n} + \sum_{|w|\le k} l_w|w|\log\sigma + \sum_{|w|>k}\frac{l_w|w|}{2}\log\sigma + \sum_{|w|>k}\frac{l_w k}{2}\log\sigma \\
&= m\log\frac{m}{n} + n_1\log\sigma + \frac{n_2}{2}\log\sigma + \frac{m_2 k}{2}\log\sigma \\
&\ge m\log\frac{m}{n} + n_1\log\sigma + \frac{n_2}{2}\log\sigma + \frac{n_2 k}{2z}\log\sigma && \text{from (5)} \\
&\ge \frac{n}{2}\log\sigma + \frac{nk}{2z}\log\sigma + m\log\frac{m}{n} \\
&= \frac{n(z+k)}{2z}\log\sigma - m\log\frac{n}{m}\ . \qquad \blacktriangleleft
\end{aligned}$$

Natural parsers on strings defined in Theorem 6 cannot do much better than the mean of entropies, which gives general bounds on algorithms inducing natural parsers.

▶ **Theorem 11.** *Let* A *be a natural parser. Let* $k \ge 0$ *be an integer function of* $|S|$ *and* σ *such that for every* σ *for infinitely many* $|S|$ *it holds that* $k_{|S|,\sigma} \le \log_\sigma |S| - \frac{1}{2}$*, where* $k_{|S|,\sigma}$ *denotes the value of* k *for* $|S|$ *and* σ*. Then for any natural* $p > 0$ *there exists an infinite family of strings* $\{S_n\}_{n=1}^{\infty} \subseteq \Gamma^*$*, where* $|\Gamma| = 4^p$*, such that the bit-size of output* $\mathsf{A}(S)$ *of* A *on* $S \in \{S_n\}_{n=1}^{\infty}$ *is at least:*

$$\mathsf{A}(S) \ge |S|H_k(S) + \frac{\rho|S|\log\sigma}{2} - \lambda|S| \ge (1+\rho)\,|S|H_k(S) - \lambda|S|\ ,$$

where $\rho = \frac{k}{2\log_\sigma |S| - k}$ *and* $\lambda < 0.54$*. If the size of parsing induced by* A *is* $o(|S|)$ *then:*

$$\mathsf{A}(S) \ge |S|H_k(S) + \frac{\rho|S|\log\sigma}{2} - o(|S|) \ge (1+\rho)\,|S|H_k(S) - o(|S|)\ .$$

Proof. Denote $n = |S|$ and $m = |Y_S|$. The proof is a simple application of Lemma 10. Fix alphabet Γ of size $\sigma = 4^p$. Take k such that $k_{|S|,\sigma} \le \log_\sigma |S| - \frac{1}{2}$, then $l = 2\log_\sigma |S| - 2k_{|S|,\sigma} - 1$ is non-negative; note that due to assumptions we can take arbitrarily large $|S|$. Then $k_{|S|,\sigma} + \frac{l+1}{2} = \log_\sigma |S|$. So it is possible to construct a string S (of length $n = |S|$) from Theorem 6, for parameters $k_{|S|,\sigma}, l, p$.

Let $Y_S = y_1 y_2 \cdots y_{|Y_S|}$ be a parsing of S induced by A. As A is a natural parser, we have that $|y_i| \le k + \frac{l+1}{2}$. We use Lemma 10 to lower bound the output of the algorithm:

$$\begin{aligned}
\mathsf{A}(S) &\geq |Y_S|H_0(Y_S) \\
&\geq \frac{|S|(2k+l+1)}{2(k+l+1)}\log\sigma - m\log\frac{n}{m} \\
&\geq |S|H_k(S) + \frac{|S|k}{2(k+l+1)}\log\sigma - m\log\frac{n}{m} && \text{, as } \frac{\log\sigma}{2} \geq H_k(S) \\
&\geq |S|H_k(S) + \frac{\rho|S|\log\sigma}{2} - m\log\frac{n}{m} \\
&\geq (1+\rho)|S|H_k(S) - m\log\frac{n}{m} \ .
\end{aligned}$$

The expression $m\log\frac{n}{m}$ is minimized when $m = n/e$, so we can bound it by $n\frac{\log e}{e} < 0.54n$, and by $o(n)$ if $m = o(n)$. Plugging those values to the above equation yields the claim. ◀

There are several consequences of Theorem 11 for natural parsers (the proofs of the Corollaries are in the Appendix). First, for $k \leq \log_\sigma |S| - \frac{1}{2}$ they cannot achieve better redundancy than $\mathcal{O}(|S|k\log\sigma/\log_\sigma|S|)$ bits. If $o(\log_\sigma|S|)$ is the best bound on k, then the redundancy of $o(|S|\log\sigma)$ is necessary.

▶ **Corollary 12.** *Let* A *be a natural parser. Then for large enough* σ *there exists an infinite family of strings* $\{S_n\}_{n\in\mathbb{N}}$ *over a* σ*-size alphabet such that for each* $S \in \{S_n\}_{n\in\mathbb{N}}$*, the size of the output generated by* A *on* S *is at least*

$$\mathsf{A}(S) \geq |S|H_k(S) + \Omega\left(\frac{|S|k\log\sigma}{\log_\sigma|S|}\right) \ ,$$

where $k \leq \log_\sigma|S| - \frac{1}{2}$ *can be any function of* $(|S|,\sigma)$*.*

Theorem 2 is tight in the sense that we cannot make the constant at the mean of entropies smaller than 1, even if we allow phrases of different lengths (not too large, though).

▶ **Corollary 13.** *Let* k *be an integer function of* $(|S|,\sigma)$ *such that* $k \leq \log_\sigma|S| - \frac{1}{2}$*. Then for large enough* σ *there exists an infinite family of strings* $\{S_n\}_{n\in\mathbb{N}}$ *over a* σ*-size alphabet such that for each* $S \in \{S_n\}_{n=1}^{\infty}$

$$2H_k(S) + \mathcal{O}\left(\frac{\log|S|}{|S|}\right) \geq H_0(S) \geq 2H_k(S)$$

and no parsing Y_S *with phrases of length at most* $j = 2\log_\sigma|S| - k$ *achieves*

$$|Y_S|H_0(Y_S) \leq (1-\epsilon)\frac{|S|}{j}\sum_{i=0}^{j-1}H_i(S) + o(|S|\log\sigma) \ ,$$

for $\epsilon > 0$*.*

Finally, we show that extending the bounds to $k = \alpha\log_\sigma n$ for a constant $0 < \alpha < 1$ implies that $|S|H_k(S)$ (without a constant coefficient) is not achievable. This gives partial (negative) answer to the question, whether we can prove optimality results for Re-Pair for $k = \alpha\log_\sigma|S|$ [27]. For a statement in full generality, we need to extend the construction from Theorem 6. For the constructed strings the ratio of $H_0(S)$ and $H_k(S)$ can be arbitrary large (at the cost of increasing l).

▶ **Corollary 14.** *Let $0 < \alpha < 1$ be a rational constant and* A *a natural parser. For large enough σ there exists an infinite family of strings $\{S_n\}_{n=1}^{\infty}$ over σ-size alphabet such that if* A *achieves $\beta|S|H_k(S) + o(|S|\log\sigma)$ bits on each $S \in \{S_n\}_{n=1}^{\infty}$ for $k = \alpha\log_\sigma|S|$, then $\beta \geq \frac{1}{1-\alpha}$.*

Substituting $k = \alpha\log_\sigma n$ to Corollary 12 already shows that an output of a natural parser is at least $\frac{2}{2-\alpha}|S|H_k(S)$ (see (7) in the Appendix for calculations). To show a bound with coefficient $\frac{1}{1-\alpha}$ we generalize the construction from Lemma 8.

Lemma 8 shows that there exist strings for which $H_l(S) = H_k(S) = \frac{1}{2}H_{k-1}(S) = \frac{1}{2}H_0(S)$, for every k, l, such that $k < \log_\sigma|S| \leq l$. When $k = \alpha\log_\sigma|S|$ (for a constant $0 < \alpha < 1$) this guarantees that the mean of l first entropies is larger by a constant factor than $H_k(S)$. The intuition is that if we could construct the strings such that $|S|H_k(S) = \frac{1}{r}|S|H_0(S)$, for arbitrarily large r, we would get that the mean of entropies can be arbitrarily large with respect to $H_k(S)$. This can be done, we show corresponding properties.

▶ **Lemma 15** (cf. Lemma 8). *For every $k > 0$, $l \geq 0$, $p \geq 1, r \geq 2$ there exists a string S over alphabet Γ' of size $\sigma = (2^r)^p$ of length $\sigma^{k+\frac{l+1}{r}}$ such that:*
(dB1') *For any $w \in (\Gamma')^i, i < k$ we have $|S|_w^{\circlearrowleft} = \sigma^{k-i+(l+1)/r}$,*
(dB2') *For any $w \in (\Gamma')^i, k \leq i \leq k+l+1$ we have either $|S|_w^{\circlearrowleft} = \sigma^{(k+l+1-i)/r}$ or $|S|_w^{\circlearrowleft} = 0$,*
(dB3') *No string of length $k+l+1$ occurs cyclically more than once in S.*

For the construction from Lemma 15, an appropriate variant of Lemma 10 holds.

▶ **Lemma 16** (cf. Lemma 10). *Let S be a string from Lemma 15 for parameters k, l, p and r, let $z = k+l+1$. Then for every parsing $Y_S = y_1y_2\cdots y_{|Y_S|}$ of S such that $|y_i| \leq z$ we have:*

$$|Y_S|H_0(Y_S) \geq \frac{|S|(z+(r-1)k)}{r\cdot z}\log\sigma - |Y_S|\log\frac{|S|}{|Y_S|} \ .$$

Now we can state the generalized version of Theorem 6:

▶ **Theorem 17** (cf. Theorem 6). *For every $k > 0$, $l \geq 0$, $p \geq 1$, $r \geq 2$ there exists a string S over alphabet of size $\sigma = (2^r)^p$ of length $\sigma^{k+\frac{l+1}{r}}$ such that:*
1. $\log\sigma - \mathcal{O}\left(\frac{i\log|S|}{|S|}\right) \leq H_i(S) \leq \log\sigma$ *for $i < k$;*
2. $\frac{\log\sigma}{r} - \mathcal{O}\left(\frac{i\log|S|}{|S|}\right) \leq H_i(S) \leq \frac{\log\sigma}{r}$ *for $k \leq i \leq k+l$;*
3. *no string of length $k+l+1$ occurs more than once in S.*

5 Conclusions and open problems

Conclusions. We have shown space lower bounds for a large class of parsing-based compression and parsing methods: they yield output greater than $|S|H_k(S)$ by at least $\Omega(|S|k\log\sigma/\log_\sigma n)$ additional bits, thus even for fixed k and σ this value grows with $|S|$. Moreover we have shown that if $k = \alpha\log_\sigma|S|$ then parsing-based methods produce output of size at least $1/(1-\alpha)|S|H_k(S)$. These bounds hold assuming that we encode the parsing using zeroth-order entropy or similar method. Those bounds are strictly higher than upper bounds for methods based on BWT or PPM.

Open problems. There are parsing based compressed text representations [17, 12], which allow for fast random access to letters and substring retrieval, achieving $|S|H_k(S)+\mathcal{O}(\sigma^k\log|S|) = |S|H_k(S) + o(|S|)$ for $k = \alpha\log_\sigma|S|$, where $\alpha < \frac{1}{8}$; the difference is that they encode the parsing using first order entropy coder. Still, they do not allow random access in constant time nor short (i.e. $\Theta(\log_\sigma|S|)$) substring retrieval in constant time; both operations are

facilitated when zeroth-entropy coder is used [16, 13, 12]. More precisely, structures based on first-order entropy coders achieve $\mathcal{O}(\log |S|/ \log\log |S|)$ time for such operations. Can we estimate time-space tradeoffs?

Kosaraju and Manzini [21] considered the notion of λ-optimality: an algorithm is λ *optimal* if it achieves $\lambda|S|H_k(S) + o(|S|H_k(S))$ bits for some constant λ. They showed [21] that LZ77 and the slight modification of LZ78 are λ optimal for $k = 0$, and that neither LZ77 nor LZ78 are for $k > 0$. They left an open question, whether there is an parsing-based algorithm which is λ-optimal for $k > 0$. Corollary 14 implies that no natural parser is λ-optimal for $k = \alpha \log_\sigma n$ for any constant α, which partially answers this question. Still, other cases, for instance: of constant k, remain open. This is interesting because our constructed strings have high-entropy, and previously low entropy strings where used in the context of λ-optimality [21]. A natural question is, whether the dichotomy between natural parsers and PPM methods holds also for low entropy strings?

Is any grammar compressor λ-optimal, even for $k = 0$? On one hand, there are examples of small entropy strings on which most grammar compressors perform badly [6, 18], still this does not apply to all of them, e.g. Greedy is an exception.

References

1. Test results for data from Canterbury Corpus. `http://corpus.canterbury.ac.nz/details/cantrbry/RatioByRatio.html`. Accessed: 2019-01-17.
2. Test results for data from Pizza & Chilli corpus. `http://pizzachili.dcc.uchile.cl/texts.html`. Accessed: 2019-01-17.
3. Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.
4. Alberto Apostolico and Stefano Lonardi. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference, 1998. DCC'98. Proceedings*, pages 119–128. IEEE, 1998.
5. Alberto Apostolico and Stefano Lonardi. Compression of Biological Sequences by Greedy Off-Line Textual Substitution. In *Proceedings of the Conference on Data Compression*, DCC '00, pages 143–152, Washington, DC, USA, 2000. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=789087.789789`.
6. Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005. `doi:10.1109/TIT.2005.850116`.
7. Francisco Claude and Gonzalo Navarro. Self-Indexed Grammar-Based Compression. *Fundam. Inform.*, 111(3):313–337, 2011. `doi:10.3233/FI-2011-565`.
8. Francisco Claude and Gonzalo Navarro. Improved Grammar-Based Compressed Indexes. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, pages 180–192, 2012. `doi:10.1007/978-3-642-34109-0_19`.
9. J. Cleary and I. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984. `doi:10.1109/TCOM.1984.1096090`.
10. Łukasz Dębowski. Is Natural Language a Perigraphic Process? The Theorem about Facts and Words Revisited. *Entropy*, 20(2):85, 2018. `doi:10.3390/e20020085`.
11. Paolo Ferragina and Giovanni Manzini. Compression Boosting in Optimal Linear Time Using the Burrows-Wheeler Transform. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 655–663, Philadelphia, PA, USA, 2004.

Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=982792.982892.

12 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2):20, 2007. doi:10.1145/1240233.1240243.

13 Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007. doi:10.1016/j.tcs.2006.12.012.

14 Travis Gagie. Large alphabets and incompressibility. *Inf. Process. Lett.*, 99(6):246–251, 2006. doi:10.1016/j.ipl.2006.04.008.

15 Michał Gańczorz. Entropy bounds for grammar compression. *CoRR*, abs/1804.08547, 2018. arXiv:1804.08547.

16 Rodrigo González and Gonzalo Navarro. Statistical Encoding of Succinct Data Structures. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2006. doi:10.1007/11780441_27.

17 Roberto Grossi, Rajeev Raman, Srinivasa Rao Satti, and Rossano Venturini. Dynamic Compressed Strings with Random Access. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, pages 504–515. Springer, 2013. doi:10.1007/978-3-642-39206-1_43.

18 Danny Hucke, Artur Jeż, and Markus Lohrey. Approximation ratio of RePair. *CoRR*, abs/1703.06061, 2017.

19 Juha Kärkkäinen and Erkki Sutinen. Lempel-Ziv Index for q-Grams. *Algorithmica*, 21(1):137–154, 1998. doi:10.1007/PL00009205.

20 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000. doi:10.1109/18.841160.

21 S. Rao Kosaraju and Giovanni Manzini. Compression of Low Entropy Strings with Lempel-Ziv Algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999. doi:10.1137/S0097539797331105.

22 Sebastian Kreft and Gonzalo Navarro. Self-indexing Based on LZ77. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, pages 41–54, 2011. doi:10.1007/978-3-642-21458-5_6.

23 N. Jesper Larsson and Alistair Moffat. Offline Dictionary-Based Compression. In *Data Compression Conference*, pages 296–305. IEEE Computer Society, 1999. doi:10.1109/DCC.1999.755679.

24 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3):32:1–32:38, 2008. doi:10.1145/1367064.1367072.

25 Gonzalo Navarro. Indexing Text Using the Ziv-Lempel Trie. *J. of Discrete Algorithms*, 2(1):87–114, March 2004. doi:10.1016/S1570-8667(03)00066-2.

26 Gonzalo Navarro and Veli Mäkinen. Compressed Full-text Indexes. *ACM Comput. Surv.*, 39(1), April 2007. doi:10.1145/1216370.1216372.

27 Gonzalo Navarro and Luís M. S. Russo. Re-Pair Achieves High-Order Entropy. In *DCC*, page 537. IEEE Computer Society, 2008. doi:10.1109/DCC.2008.79.

28 Craig G Nevill-Manning and Ian H Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

29 Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997. doi:10.1613/jair.374.

30 C. Ochoa and G. Navarro. RePair and All Irreducible Grammars are Upper Bounded by High-Order Empirical Entropy. *IEEE Transactions on Information Theory*, 2019. to appear.

31 Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1230–1239. Society for Industrial and Applied Mathematics, 2006.

32 Michal Vasinek and Jan Platos. Prediction and Evaluation of Zero Order Entropy Changes in Grammar-Based Codes. *Entropy*, 19(5):223, 2017. doi:10.3390/e19050223.

33 En-Hui Yang and John C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform - Part one: Without context models. *IEEE Trans. Information Theory*, 46(3):755–777, 2000. doi:10.1109/18.841161.

34 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.

35 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

A Additional material for Section 4

Construction of Generalized de Bruijn strings

Proof of Lemma 7. Fix some string v, $|v| = k$. Consider the difference

$$\sum_{a\text{: letter}} |w|_{va}^{\circlearrowleft} \log\left(\frac{|w|_{va}^{\circlearrowleft}}{|w|_{v}^{\circlearrowleft}}\right) - \sum_{a\text{: letter}} |w|_{va} \log\left(\frac{|w|_{va}}{|w|_{v}}\right) \ , \tag{6}$$

our goal is to estimate it when summed over all v of length k.

Define S_1, S_2 as the strings of letters that follow cyclic occurrences (standard occurrences) of v in w, formally for each letter a they should satisfy

$$|S_1|_a = |w|_{va}^{\circlearrowleft} \quad |S_2|_a = |w|_{va} \ ,$$

note that this implies that

$$|S_1| = |w|_{v}^{\circlearrowleft} \quad |S_2| = |w|_{v} \ .$$

Then left and right summands from (6) are equal to, respectively:

$$|S_1|H_0(S_1) = \sum_{a\text{: letter}} |w|_{va}^{\circlearrowleft} \log\left(\frac{|w|_{va}^{\circlearrowleft}}{|w|_{v}^{\circlearrowleft}}\right) \qquad |S_2|H_0(S_2) = \sum_{a\text{: letter}} |w|_{va} \log\left(\frac{|w|_{va}}{|w|_{v}}\right)$$

We first show that (6) is positive, which yields the second inequality of the Lemma.

Clearly $|w|_{va}^{\circlearrowleft} \geq |w|_{va}$, and so we can obtain S_2 from S_1 by removing and permuting letters, which cannot increase the entropy. Hence $|S_1|H_0(S_1) \geq |S_2|H_0(S_2)$, which yields that (6) is positive and so the second inequality of the lemma follows.

To upper bound the difference in (6) observe that S_1 is obtained by adding symbols to S_2 and the addition of one letter to a string of length at most $|S| - 1$ increases the entropy by at most $\log|S| + \beta$, for some constant β. Moreover, there are at most k such additions, when summing over all possible k-length contexts v. Thus the first inequality holds. ◀

Proof of Corollary 12. For any function k and any $\sigma = 4^p, p > 0$ by Theorem 11 we can build an infinite family of strings $\{S_n\}_{n=1}^{\infty}$ such that output generated by any natural parsing method on $S \in \{S_n\}_{n=1}^{\infty}$ is lower bounded by:

$$\mathsf{A}(S) \geq |S|H_k(S) + \frac{|S|k(\log\sigma - \mathcal{O}(1))}{2(2\log_\sigma|S| - k)} \ . \tag{7}$$

As $k \leq \log_\sigma|S| - \frac{1}{2}$ and the inequality holds for any $\sigma = 4^p$, we can choose big enough σ so that the second summand is $\Omega(\frac{|S|k\log\sigma}{\log_\sigma|S|})$. ◀

Proof of Corollary 13. We use the same construction as in the case of Corollary 12, i.e. we build the string for parameter k. By Lemma 7, the mean of the first j entropies is at least:

$$\frac{1}{j}\left(k\log_\sigma|S| + (j-k)\frac{\log_\sigma|S|}{2} - \mathcal{O}\left(\frac{j\log|S|}{|S|}\right)\right) \geq H_k(S) + \frac{k\log\sigma}{2j} - \mathcal{O}\left(\frac{j\log|S|}{|S|}\right) \quad . \quad (8)$$

On the other hand, by (7) we get that the output is lower bounded by (as $2\log_\sigma|S| - k = j$):

$$\mathsf{A}(S) \geq |S|H_k(S) + \frac{|S|k(\log\sigma - \mathcal{O}(1))}{2j} \quad . \quad (9)$$

As we can choose $\sigma = 4^p$ arbitrarily we have that for each ϵ there exists such σ and s_0 so that for all constructed strings with $|S| \geq s_0$ the difference (8) will be larger than (9) multiplied by $(1-\epsilon)$. ◀

Generalization of strings from Lemma 8, Proof of Corollary 14

Sketch of the Proof of Lemma 15. We comment on how to modify the construction and proof of Lemma 8; observe that, in essence we replace a constant 2 in Lemma 8 with $r \geq 2$. As a first step, Lemma 8 takes de Bruijn string B of order $2k+1$ over binary alphabet and parse it into phrases of length two in two ways and combine those two strings together. We proceed similarly, but we take de Bruijn string B of order $rk+1$ and instead of parsing B into phrases of length two, we parse it into phrases of length r in r ways and combine all of the r strings. For example for $r = 3$ we have:

$$Y_B^1 = |a_1a_2a_3|a_4a_5a_6|\cdots|a_{n-5}a_{n-4}a_{n-3}|a_{n-2}a_{n-1}a_n|$$
$$Y_B^2 = |a_2a_3a_4|a_5a_6a_7|\cdots|a_{n-4}a_{n-3}a_{n-2}|a_{n-1}a_na_1|$$
$$Y_B^3 = |a_3a_4a_5|a_6a_7a_8|\cdots|a_{n-3}a_{n-2}a_{n-1}|a_na_1a_2|$$

As in proof of Lemma 8, we can choose a shift of B such that $a_1 = a_2 = \cdots = a_5$, so that all those strings begin with the same triple, so after the merging of letters: with the same letter. In this way, the (short enough) cyclic occurrences in the concatenation are the same as cyclic occurrences in the separate strings. Moreover we can use the same argument with constructing the graph so we get the Lemma for arbitrary l.

Concerning the degree of the graph, recall that in the the proof of Lemma 8 each k-letter string over the alphabet $(4^p) = \sigma$ had $(2^p) = \sqrt{\sigma}$ occurrences, all followed by different letters, thus the degree of the graph G_0 (and so each G_i) was $\sqrt{\sigma}$. Now, since we merge r letters into one and the B was a $rk+1$ de Bruijn string, each k-letter string over the alphabet $(2^{rp}) = \sigma$ has $(2^p) = \sqrt[r]{\sigma}$ occurrences, thus the degree of the graph is $\sqrt[r]{\sigma}$. In essence this justifies the replacement of $1/2$ by $1/r$ in all the exponents of σ. Thus we get that that $|S|H_k^{\circlearrowleft}(S) = \frac{1}{r}|S|H_0^{\circlearrowleft}(S)$, and from the Lemma 7 the same (roughly) holds between $|S|H_k(S)$ and $|S|H_0(S)$. Thus we get the generalized version of Theorem 6, which, intuitively, says that we can construct strings whose mean of entropies can be arbitrarily large with respect to $H_k(S)$. ◀

Sketch of the Proof of Lemma 16. We skip the calculations, as they are almost the same as in the proof of Lemma 10, the only difference is that we use different bounds for $|S|_w^{\circlearrowleft}$ (and so for l_w), i.e. for w such that $|S|_w^{\circlearrowleft} > 0$ we have (by Lemma 15):

$$|S|_w^{\circlearrowleft} = \begin{cases} \frac{n}{\sigma^{|w|}} & \text{for } |w| \leq k \\ \frac{n}{\sigma^{|w|/r+(r-1)k/r}} & \text{for } k < |w| \leq z \end{cases} \quad . \quad \blacktriangleleft$$

Proof of Theorem 17. The proof is similar to the one of Theorem 6: we take the string from Lemma 16 and apply Lemma 7 to change from cyclic entropy to entropy. ◀

Proof of Corollary 14. We show that Lemma 16 implies Corollary 14. We construct the family of strings as following: fix p and r and let $\sigma = (2^r)^p$, as in Lemma 15. Take l such that $k = \frac{\alpha}{(1-\alpha)r}(l+1)$ is a natural number. Construct the string S from Lemma 15, its length is $\sigma^{k+\frac{l+1}{r}}$. As r and α are fixed and k is an increasing function of l, for sufficiently large l we have $\alpha \log_\sigma |S| \leq \log_\sigma |S| - \frac{1}{r}$, as this is equivalent to $\frac{1}{r} \leq (1-\alpha)(k + \frac{l+1}{r})$. This means we can construct the family of strings from Lemma 15 for parameter k. By easy calculation

$$\begin{aligned}
\alpha \log_\sigma |S| &= \alpha \left(k + \frac{l+1}{r} \right) \\
&= \alpha \left(\frac{\alpha}{1-\alpha} \cdot \frac{l+1}{r} + \frac{l+1}{r} \right) \\
&= \alpha \cdot \frac{\alpha + (1-\alpha)}{1-\alpha} \cdot \frac{l+1}{r} \\
&= \frac{\alpha}{1-\alpha} \cdot \frac{l+1}{r} \\
&= k \ ,
\end{aligned}$$

as desired. Thus $\frac{k}{\alpha} = \log_\sigma |S| = k + (l+1)/r$. Define $z = k + l + 1$ as in Lemma 16; again by easy calculations $z = k + kr/\alpha - kr = kr/\alpha - k(r-1)$.

To prove that A cannot perform better than $\frac{1}{1-\alpha}|S|H_k(S) + o(|S| \log \sigma)$ we lower bound the ratio of A's output, i.e. $|Y_S|H_0(Y_S)$, and $|S|H_k(S)$ by $\frac{1}{1-\alpha}$. We estimate the former by Lemma 16 and use the same estimation on $|Y_S| \log \frac{|S|}{|Y_S|}$ as in the proof of Theorem 11, i.e. $|Y_S| \log \frac{|S|}{|Y_S|} \leq \lambda |S|$, $\lambda = 0.54$:

$$\begin{aligned}
|Y_S|H_0(Y_S) &\geq \frac{|S|(z + (r-1)k)}{r \cdot z} \log \sigma - |Y_S| \log \frac{|S|}{|Y_S|} \\
&\geq \frac{|S|(z + (r-1)k)}{r \cdot z} \log \sigma - \lambda |S| \ . \qquad (10)
\end{aligned}$$

On the other hand, by Theorem 17 we have that:

$$|S|H_k(S) \leq \frac{\log_\sigma |S|}{r} \ . \qquad (11)$$

Combining these two we obtain:

$$\begin{aligned}
\frac{|Y_S|H_0(Y_S)}{|S|H_k(S)} &\geq \frac{|Y_S|H_0(Y_S)}{(|S| \log \sigma)/r} && \text{by (11)} \\
&\geq \frac{z + (r-1)k}{z} - \frac{\lambda |S| r}{|S| \log \sigma} && \text{by (10)} \\
&\geq \frac{kr/\alpha}{kr/\alpha - k(r-1)} - \frac{\lambda |S| r}{|S| rp} && \text{as } z = kr/\alpha - k(r-1),\ \log \sigma = rp \\
&= \frac{r/\alpha}{r/\alpha - (r-1)} - \lambda/p \\
&= \frac{1}{1 - \alpha \cdot \frac{(r-1)}{r}} - \lambda/p \ . \qquad (12)
\end{aligned}$$

Now, if A achieves $\beta|S|H_k(S)+f(|S|\log\sigma)$ bits, for some $f(m) \in o(m)$, and as $|Y_S|H_0(Y_S)$ lower bounds the output size for natural parsers, we have:

$$\begin{aligned}\frac{|Y_S|H_0(Y_S)}{|S|H_k(S)} &\leq \beta + \frac{f(|S|\log\sigma)}{|S|H_k(S)} \\ &\leq \beta + \frac{f(|S|\log\sigma)}{\frac{|S|\log\sigma}{r} - \mathcal{O}(k\log|S|)} && \text{by Theorem 17} \\ &= \beta + \frac{rf(|S|\log\sigma)}{|S|\log\sigma - \mathcal{O}(rk\log|S|)} && (13)\end{aligned}$$

Combining (12) and (13) yields a lower bound on β:

$$\beta \geq \frac{1}{1-\alpha\cdot\frac{(r-1)}{r}} - \lambda/p - \frac{rf(|S|\log\sigma)}{|S|\log\sigma - \mathcal{O}(rk\log|S|)} \qquad (14)$$

Now, fix r and set $p = r$ in (14), this in particular makes σ fixed as well. Consider the last term in (14), i.e. $\frac{rf(|S|\log\sigma)}{|S|\log\sigma-\mathcal{O}(rk\log|S|)}$. When $|S| \to \infty$, we have $rk\log|S| = o(|S|\log\sigma)$ and so $f(|S|\log\sigma)/|S|\log\sigma$ is arbitrarily small. Thus this term vanishes when $|S| \to \infty$ and so β has to be at least

$$\beta \geq \frac{1}{1-\alpha\cdot\frac{(r-1)}{r}} - \lambda/r \ .$$

This holds for any r, so also for the limit with $r \to \infty$, and so

$$\beta \geq \frac{1}{1-\alpha} \ ,$$

as claimed. ◀

A New Class of Searchable and Provably Highly Compressible String Transformations

Raffaele Giancarlo
University of Palermo, Dipartimento di Matematica e Informatica, Italy
raffaele.giancarlo@unipa.it

Giovanni Manzini
University of Eastern Piedmont, Alessandria, Italy
IIT-CNR, Pisa, Italy
giovanni.manzini@uniupo.it

Giovanna Rosone
University of Pisa, Dipartimento di Informatica, Italy
giovanna.rosone@unipi.it

Marinella Sciortino
University of Palermo, Dipartimento di Matematica e Informatica, Italy
marinella.sciortino@unipa.it

Abstract

The Burrows-Wheeler Transform is a string transformation that plays a fundamental role for the design of self-indexing compressed data structures. Over the years, researchers have successfully extended this transformation outside the domains of strings. However, efforts to find non-trivial alternatives of the original, now 25 years old, Burrows-Wheeler string transformation have met limited success. In this paper we bring new lymph to this area by introducing a whole new family of transformations that have all the "myriad virtues" of the BWT: they can be computed and inverted in linear time, they produce provably highly compressible strings, and they support linear time pattern search directly on the transformed string. This new family is a special case of a more general class of transformations based on *context adaptive alphabet orderings*, a concept introduced here. This more general class includes also the Alternating BWT, another invertible string transforms recently introduced in connection with a generalization of Lyndon words.

2012 ACM Subject Classification Theory of computation → Data compression; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Data Indexing and Compression, Burrows-Wheeler Transformation, Combinatorics on Words

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.12

Funding GR and SM are partially supported by MIUR-SIR project CMACBioSeq "Combinatorial methods for analysis and compression of biological sequences" grant n. RBSI146R5L; RG and GM are partially supported by INdAM-GNCS project 2018 "Innovative methods for the solution of medical and biological big data" and MIUR-PRIN project "Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond" grant n. 2017WR7SHH.

1 Introduction

The Burrows Wheeler Transform [2] (BWT) is a string transformation that had a revolutionary impact in the design of succinct or compressed data structures. Originally proposed as a tool for text compression, shortly after its introduction [9] it has been shown that, in addition to making easier to represent a string in space close to its entropy, it also makes easier to search for pattern occurrences in the original string. After this discovery, data transformations inspired by the BWT have been proposed for compactly representing and search other

combinatorial objects such as: trees, graphs, finite automata, and even string alignments. See [11] for an attempt to unify some of these results and [25] for an in-depth treatment of the field of compact data structures.

Going back to the original Burrows-Wheeler string transformation, we can summarize its salient features as follows: **1**) it can be computed and inverted in linear time, **2**) it produces strings which are provably compressible in terms of the high order entropy of the input, **3**) it supports pattern search directly on the transformed string in time proportional to the pattern length. It is the *combination* of these three properties that makes the BWT a fundamental tool for the design of compressed self-indices. In Section 2 we review these properties and also the many attempts to modify the original design. However, we recall that, despite more than twenty years of intense scrutiny, the only non trivial known BWT variant that fully satisfies properties **1**–**3** is the *Alternating BWT* (ABWT). The ABWT has been introduced in [13] in the field of combinatorics of words and its basic algorithmic properties have been described in [15].

In this paper we introduce a new *whole family* of transformations that satisfy properties **1**–**3** and can therefore replace the BWT in the construction of compressed self-indices with the same time efficiency of the original BWT and the potential of achieving better compression. We show that our family, supporting linear time computation, inversion, and search, is a special case of a much larger class of transformations that also satisfy properties **1**–**3** except that, in the general case, inversion and pattern search may take quadratic time. Our larger class includes as special cases also the BWT and the ABWT and therefore it constitutes a natural candidate for the study of additional properties shared by all known BWT variants.

More in detail, in Section 3 we describe a class of string transformations based on *context adaptive alphabet orderings*. The main feature of the above class of transformations is that, in the rotation sorting phase, we use alphabet orderings that depend on the context (i.e., the longest common prefix of the rotations being compared). In Section 4 we consider the subclass of transformations based on *local orderings*. In this subclass, the alphabet orderings only depend on a constant portion of the context. We prove that local ordering transformations can be inverted in linear time, and that pattern search in the transformed string takes time proportional to the pattern length. Thus, these transformations have the same properties **1**–**3** that were so far prerogative of the BWT and ABWT.

Having now at our disposal a wide class of string transformations with the same remarkable properties of the BWT, it is natural to use them to improve BWT-based data structures by selecting the one more suitable for the task. In this paper we initiate this study by considering the problem of selecting the BWT variant that minimizes the number of runs in the transformed string. The motivation is that data centers often store highly repetitive collections, such as genome databases, source code repositories, and versioned text collections. For such highly repetitive collections there is theoretical and practical evidence that the entropy underestimates the compressibility of the collection and much better compression ratios are obtained exploiting runs of equal symbols in the BWT [4, 12, 18, 19, 21, 22, 23]. In Section 5 we show that, for constant size alphabet, for the most general class of transformations considered in this paper, the BWT variant that minimizes the number of runs can be found in linear time using a dynamic programming algorithm.

2 Notation and background

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet of size σ with $c_1 < c_2 < \cdots < c_\sigma$, where $<$ denotes the standard lexicographic order. We denote by Σ^* the set of strings over Σ. Given a string $x = x_1 x_2 \cdots x_n \in \Sigma^*$ we denote by $|x|$ its length n. We use ϵ to denote the empty string.

A *factor* of x is written as $x[i,j] = x_i \cdots x_j$ with $1 \leq i \leq j \leq n$. A factor of type $x[1,j]$ is called a *prefix*, while a factor of type $x[i,n]$ is called a *suffix*. The i-th symbol in x is denoted by $x[i]$. Two strings $x, y \in \Sigma^*$ are called *conjugate*, if $x = uv$ and $y = vu$, where $u, v \in \Sigma^*$. We also say that x is a *cyclic rotation* of y. A string x is *primitive* if all its cyclic rotations are distinct. Given a string x and $c \in \Sigma$, we write $\mathsf{rank}_c(x,i)$ to denote the number of occurrences of c in $x[1,i]$, and $\mathsf{select}_c(x,j)$ to denote the position of the j-th c in x.

Given a primitive string s, we consider the matrix of all its cyclic rotations sorted in lexicographic order. Note that the rotations are all distinct by the primitivity of s. The last column of the matrix is called the Burrows-Wheeler Transform of the string s and it is denoted by $BWT(s)$ (see Figure 1 (left)). The BWT can be computed in $\mathcal{O}(|s|)$ time using any algorithm for Suffix Array construction [16, 17]. It is shown in [2] that $BWT(s)$ is always a permutation of s, and that there exists a linear time procedure to recover s given $BWT(s)$ and the position I of s in the rotations matrix (it is $I = 2$ in Figure 1 (left)).

The BWT has been introduced as a data compression tool: it was empirically observed that $BWT(s)$ usually contains long runs of equal symbols. This notion was later mathematically formalized in terms of the empirical entropy of the input string [8, 24]. For $k \geq 0$, the k-th order empirical entropy of a string x, denoted as $H_k(x)$, is a lower bound to the compression ratio of any algorithm that encodes each symbol of x using a codeword that only depends on the k symbols preceding it in x. The simplest compressors, such as Huffman coding, in which the code of a symbol does not depend on the previous symbols, typically achieve a (modest) compression bounded in terms of the zeroth-order entropy H_0. This class of compressors are referred to as *memoryless* compressors.

It is proven in [8, Theorem 5.4] that the informal statement "the output of the BWT is highly compressible" can be formally restated saying that $BWT(s)$ can be compressed up to $H_k(s)$, for any $k > 0$, using any tool able to compress up to the zeroth-order entropy. In other words, after applying the BWT we can achieve high order compression using a simple (and fast) memoryless compressor. This property is often referred to as the "boosting" property of the BWT. Another remarkable property of the BWT is that it can be used to build compressed indices. It is shown in [10] how to compute the number of occurrences of a pattern x in s in $\mathcal{O}(t_R|x|)$ time, where t_R is the cost of executing a rank query over $BWT(s)$. This result has spurred a great interest in data structures representing compactly a string x and efficiently supporting the queries rank, select, and access (return $x[i]$ given i, which is a nontrivial operation when x is represented in compressed form) and there are now many alternative solutions with different trade-offs. In this paper we assume a RAM model with word size w and an alphabet of size $\sigma = w^{\mathcal{O}(1)}$. Under this assumption we make use of the following result (Theorem 7 in [1]).

▶ **Theorem 1.** *Let s denote a string over an alphabet of size $\sigma = w^{\mathcal{O}(1)}$. We can represent s in $|s|H_0(s) + o(|s|)$ bits and support constant time* rank, select, *and* access *queries.*

The properties of the BWT of being *compressible* and *searchable* combine nicely to give us *indexing capabilities* in *compressed space*. Indeed, combining a zero order representation supporting rank, select, and access queries with the boosting property of the BWT, we obtain a full text self-index for s that uses space bounded by $|s|H_k(s) + o(|s|)$ bits; see [10, 20, 25, 26] for further details on these results and on the field of compressed data structures and algorithms that originated from this area of research.

2.1 Known BWT variants

We observed that the salient features of the Burrows-Wheeler transformation can be summarized as follows: **1**) it can be computed and inverted in linear time, **2**) it produces strings which are provably compressible in terms of the high order entropy of the input, **3**) it supports

	F								L
	$\downarrow$								$\downarrow$
	a	a	a	b	a	c	a	a	b
$s \rightarrow$	a	a	b	a	a	a	b	a	c
	a	a	b	a	c	a	a	b	a
	a	b	a	a	a	b	a	c	a
	a	b	a	c	a	a	b	a	a
	a	c	a	a	b	a	a	a	b
	b	a	a	a	b	a	c	a	a
	b	a	c	a	a	b	a	a	a
	c	a	a	b	a	a	a	b	a

	F								L
	$\downarrow$								$\downarrow$
	a	c	a	a	b	a	a	a	b
	a	b	a	c	a	a	b	a	a
	a	b	a	a	a	b	a	c	a
	a	a	a	b	a	c	a	a	b
$s \rightarrow$	a	a	b	a	a	a	b	a	c
	a	a	b	a	c	a	a	b	a
	b	a	a	a	b	a	c	a	a
	b	a	c	a	a	b	a	a	a
	c	a	a	b	a	a	a	b	a

Figure 1 The original BWT matrix for the string $s = aabaaabac$ (left), and the ABWT matrix of cyclic rotations sorted using the alternating lexicographic order (right). In both matrices the horizontal arrow marks the position of the original string s, and the last column L is the output of the transformation.

linear time pattern search directly on the transformed string. The *combination* of these three properties makes the BWT a fundamental tool for the design of compressed self-indices. Over the years, many variants of the original BWT have been proposed; in the following we review them, in roughly chronological order, emphasizing to what extent they share the features **1**–**3** mentioned above.

The original BWT is defined by sorting in lexicographic order all the cyclic rotations of the input string. In [28] Schindler proposes a *bounded context* transformation that differs from the BWT in the fact that the rotations are lexicographically sorted considering only the first ℓ symbols of each rotation. Recent studies [6, 27] have shown that this variant satisfies properties **1**–**3**, with the limitation that the compression ratio can reach at maximum the ℓ-th order entropy and that it supports searches of patterns of length at most ℓ. Chapin and Tate [3] have experimented with computing the BWT using a different alphabet order. This simple variant still satisfies properties **1**–**3**, but it clearly does not bring any new theoretical insight. More recently, some authors have proposed variants in which the lexicographic order is replaced by a different order relation. The interested reader can find relevant work in a recent review [7]; it turns out that these variants satisfy property **1** in part but nothing is known with respect to properties **2** and **3**.

To the best of our knowledge, the only non trivial BWT variant that fully satisfies properties **1**–**3** is the Alternating BWT (ABWT). This transformation has been derived in [13] starting from a result in combinatorics of words [5] characterizing the BWT as the inverse of a known bijection between words and multisets of primitive necklaces [14]. The ABWT is defined as the BWT except that when sorting rotation instead of the standard lexicographic order we use a different lexicographic order, called the *alternating* lexicographic order. In the alternating lexicographic order, the first character of each rotation is sorted according to the standard order of Σ (i.e., $a < b < c$). However, if two rotations start with the same character we compare their second characters using the reverse ordering (i.e., $c < b < a$) and so on alternating the standard and reverse orderings in odd and even positions. Figure 1 (right) shows how the rotations of an input string are sorted using the alternating ordering and the resulting ABWT.

The algorithmic properties of the BWT and ABWT are compared in [15]. It is shown that they can be both computed and inverted in linear time and that their main difference is in the definition of the LF-map, i.e. the correspondence between the characters in the first

$$
\begin{array}{cccccccccc}
 & F & & & & & & & & L \\
 & \downarrow & & & & & & & & \downarrow \\
 & b & a & a & a & b & a & c & a & a \\
 & b & a & c & a & a & b & a & a & a \\
 & a & c & a & a & b & a & a & a & b \\
s \rightarrow & a & a & b & a & a & a & b & a & c \\
 & a & a & b & a & c & a & a & b & a \\
 & a & a & a & b & a & c & a & a & b \\
 & a & b & a & a & a & b & a & c & a \\
 & a & b & a & c & a & a & b & a & a \\
 & c & a & a & b & a & a & a & b & a
\end{array}
$$

Figure 2 The generalized BWT matrix for the string $s = aabaaabac$ computed using the orderings $\pi_\epsilon = (b, a, c)$, $\pi_a = (c, a, b)$, $\pi_{aa} = (c, b, a)$, and $\pi_x = (a, b, c)$ for every other substring x. The horizontal arrow marks the position of the original string s; the last column L is the output of the transformation.

and last column of the sorted rotations matrix. In the original BWT the i-th occurrence of a character c in the first column F corresponds to the i-th occurrence of c in the last column L. Instead, in the ABWT the i-th occurrence of c from the *top* in F corresponds to the i-th occurrence of c from the *bottom* in L. Since this modified LF-map can be still computed efficiently using rank operations, the ABWT can replace the BWT for the construction of self-indices.

3 BWTs based on Context Adaptive Alphabet Orderings

In this section we introduce a class of string transformations that generalize the BWT in a very natural way. Given a primitive string s, as in the original BWT definition, we consider the matrix containing all its cyclic rotations. In the original BWT the matrix rows are sorted according to the standard lexicographic order. We generalize this concept by sorting the rows using an ordering that *depends on their common context*, i.e., their common prefix. Formally, for each string x that prefixes two or more rows, we assume that an ordering π_x is defined on the symbols of Σ. When comparing two rows which are both prefixed by x, their relative rank is determined by the ordering π_x. Once the matrix rows have been ordered with this procedure, the output of the transformation is the last column of the matrix as in the original BWT. Thus, these BWT variants are based on *context adaptive alphabet orderings*. For simplicity in the following we call them *context adaptive BWTs*.

An example is shown in Figure 2: the ordering associated to the empty string ϵ is $\pi_\epsilon = (b, a, c)$ so, among the rows that have no common prefix, first we have those starting with b, then those starting with a, and finally the one starting with c. Since $\pi_a = (c, a, b)$, among the rows which have a as their common prefix, first we have the ones starting with c, then the ones starting with a, followed by the ones starting with b. The complete ordering of the rows is established in a similar way on the basis of the orderings π_x.

We denote by $M_*(s)$ the matrix obtained using this generalized sorting procedure, and by $L = BWT_*(s)$ the last column of $M_*(s)$. Clearly L depends on s and the ordering used for each common prefix. Since we can arbitrarily choose an alphabet ordering for any substring x of s, and there are $\sigma!$ orderings to choose from, our definition includes a very large number of string transformations. This class of transformations has been mentioned in [8, Sect. 5.2]

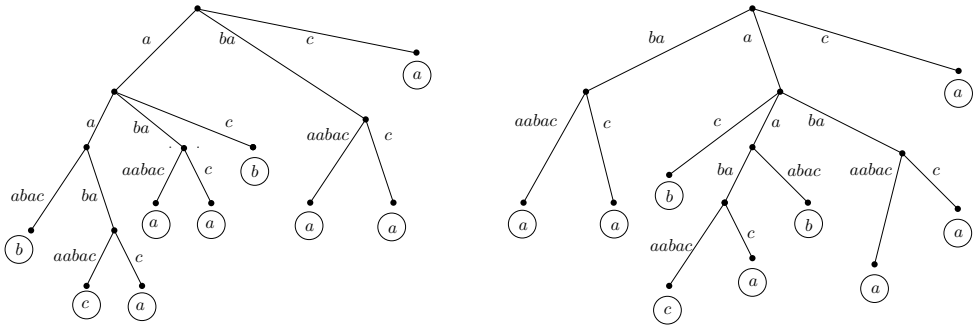


Figure 3 Standard suffix tree for $s = aabaaabac$ with the symbol c used as a string terminator (left), and suffix tree with edges reordered using the same orderings of Figure 2 (right). To each leaf it is associated the symbol preceding in s the suffix spelled by that leaf. Note that reading left to right the symbols associated to each leaf gives $BWT(s)$ (left) and $BWT_*(s)$ (right).

under the name of *string permutations realized by a Suffix Tree* (the definition in [8] is slightly more general; for example it includes the bounded context BWT, which is not included in our class). Indeed, if the input string s has a unique end-of-string terminator, one can easily see that these transformations can be obtained assigning an ordering to the children of each node of the suffix tree of s

Although in [8] the authors could not prove the invertibility of context adaptive transformations, which we do in Section 3.2, they observed that their relationship with the suffix tree has two important consequences: 1) they can be computed in $\mathcal{O}(n \log \sigma)$ time with a proper suffix tree visit (see Figure 3), and 2) they provably produce *highly compressible* strings, i.e., they have the "boosting" property of transforming a zeroth order compressor into a k-th order compressor.

To see that the generalized BWTs can be computed in $\mathcal{O}(n \log \sigma)$ time consider first the simpler case in which the string s has a unique end-of-string terminator. To build $L = BWT_*(s)$ we first build the suffix tree for s. Then, we visit the suffix tree in depth first order except that when we reach a node u (including the root), we sort its outgoing edges according to their first characters using the permutation associated to the string u_x labeling the path from the root to u. During such visit, each time we reach a leaf we write the symbol associated to it: the resulting string is exactly $L = BWT_*(s)$. The above argument also shows that the number of permutations required to define a generalized BWT on a fixed string s is at most $|s|$, i.e. the number of internal suffix tree nodes. If s doesn't have a unique terminator, the argument is analogous except that we replace the suffix tree with the compressed trie containing all the cyclic rotations of s. To see that generalized BWTs have the boosting property we observe that the proof for the BWT (Theorem 5.4 in [8]) is based on structural properties of the suffix tree, and can be repeated verbatim for the generalized BWTs.

Summing up, context adaptive transformations generalize the BWT in two important aspects: efficient (linear time in n) computation and compressibility. In [8] the only known instances of *reversible* suffix tree induced transformations were the original BWT and the bounded context BWT. In the following, we prove that *all* context adaptive BWTs defined above are invertible. Interestingly, to prove invertibility we first establish another important property of these transformations, namely that they can be used to count the number of occurrences of a pattern in s, which is another fundamental property of the original BWT.

We conclude this section observing that both the BWT and ABWT belong to the class we have just defined. To get the BWT we trivially define π_x to be the standard Σ ordering for every x, to get the ABWT we define π_x to be the standard Σ ordering for every x with $|x|$ even, and the reverse ordering for Σ for every x with $|x|$ odd. Indeed in the full paper we will show that the complete class of transformations studied in [15] is a subclass of context adaptive transformations.

3.1 Counting occurrences of patterns in Context Adaptive BWTs

Let $L = BWT_*(s)$ denote a context adaptive BWT. In the following we assume that L is enriched with data structures supporting constant time rank queries as in Theorem 1. In this section we show that given L and the set of alphabet permutations used to build $M_*(s)$ then, for each string x, we can determine in $\mathcal{O}(\sigma|x|^2)$ time the set of $M_*(s)$ rows prefixed by x. We preliminary observe that by construction this set of rows, if non-empty, form a contiguous range inside $M_*(s)$. This observation justifies the following definitions.

▶ **Definition 2.** *Given a string x, we denote by $R[x] = [b_x, \ell_x]$ the range of rows of $M_*(s)$ prefixed by x. More precisely, if $R[x] = [b_x, \ell_x]$, then row i is prefixed by x if and only if it is $b_x \leq i < b_x + \ell_x$. If no rows are prefixed x we set $R[x] = [0, 0]$. Note that ℓ_x is the number of occurrences of x in the circular string s.*

For technical reasons, given x, we are also interested in the set of rows prefixed by the strings xc as c varies in Σ. Clearly, these sets of rows are consecutive in $M_*(s)$ and their union coincides with $R[x]$.

▶ **Definition 3.** *Given a string x, we denote by $R^*[x]$ the set of $\sigma+1$ integers $[b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$ such that b_x is the lower extreme of $R[x]$ and, for $i = 1, \ldots, \sigma$, ℓ_i is the number of rows of $M_*(s)$ prefixed by xc_i.*

Since $R[x]$ is the union of the ranges $R[xc]$ for $c \in \Sigma$, we have that if $R^*[x] = [b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$, then $R[x] = [b_x, \sum_i \ell_i]$. Note also that the ordering of the ranges $R[xc]$ within $R[x]$ is determined by the permutation π_x. As observed in Section 2, we can assume that L supports constant time rank queries. This implies that in constant time we are also able to count the number of occurrences of a symbol c inside a substring $L[i, j]$.

▶ **Lemma 4.** *Given $R^*[x]$ and the permutation π_x, the set of values $R[xc_i]$ for all $c_i \in \Sigma$ can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. If $R^*[x] = [b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$ then $R[xc_i] = [b, \ell]$ with

$$b = b_x + \sum_j \ell_j, \qquad \ell = \ell_i \tag{1}$$

where the summation in (1) is done over all $j \in \{1, 2, \ldots, \sigma\}$ such that c_j is smaller than c_i according to the permutation π_x. ◀

▶ **Lemma 5.** *Let $x = x_1 x_2 \cdots x_m$ be any length-m string with $m > 1$. Then, given $R^*[x_1 \cdots x_{m-1}]$ and $R^*[x_2 \cdots x_m]$, the set of values $R^*[x_1 \cdots x_m]$ can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. By Lemma 4, given $R^*[x_1 \cdots x_{m-1}]$ and x_m, we can compute $R[x_1 \cdots x_m] = [b_x, \ell_x]$. In order to compute $R^*[x_1 \cdots x_m]$, we additionally need the number of rows prefixed by $x_1 x_2 \cdots x_m c$, for any $c \in \Sigma$. These numbers can be obtained by first computing the ranges

$R[x_2 \cdots x_m c]$ using again Lemma 4, and then counting the number of rows prefixed by $x_1 x_2 \cdots x_m c$, counting the number of x_1 in the portions of L corresponding to each range $R[x_2 \cdots x_m c]$. The counting takes $\mathcal{O}(\sigma)$ time since we are assuming L supports constant time rank as in Theorem 1. ◀

▶ **Theorem 6.** *Suppose we are given $BWT_*(s)$ with constant time rank support, and the set of permutations used to compute the matrix $M_*(s)$. Then, given any string $x = x_1 x_2 \cdots x_p$, the range of rows $R[x]$ prefixed by x can be computed in $\mathcal{O}(\sigma p^2)$ time and $\mathcal{O}(\sigma p)$ space.*

Proof. We need to compute $R[x_1 x_2 \cdots x_p]$. To this end we consider the following scheme, inspired by the Newton finite difference formula:

$$
\begin{array}{lllll}
R^*[x_1] & R^*[x_1x_2] & R^*[x_1x_2x_3] & \cdots & R^*[x_1x_2\cdots x_{p-1}] \quad R^*[x_1x_2\cdots x_p] \\
R^*[x_2] & R^*[x_2x_3] & R^*[x_2x_3x_4] & \cdots & R^*[x_2\cdots x_p] \\
R^*[x_3] & R^*[x_3x_4] & \cdots & & \\
\vdots & & & & \\
R^*[x_p] & & & &
\end{array}
$$

Using Lemma 5 we can compute $R^*[x_i \cdots x_j]$ given $R^*[x_i \cdots x_{j-1}]$ and $R^*[x_{i+1} \cdots x_j]$. Thus, from two consecutive entries in the same column we can compute one entry in the following column. To compute $R[x_1 x_2 \cdots x_p]$ we can for example perform the computation bottom-up, proceeding row by row. In this case we are essentially computing the ranges corresponding to x_p, $x_{p-1}x_p$, $x_{p-2}x_{p-1}x_p$ and so on, in a sort of backward search. However, we can also perform the computation top down, diagonal by diagonal, and in this case we are computing the ranges corresponding to x_1, x_1x_2, and so on up to $x_1 \cdots x_p$. In both cases, the information one need to store from one iteration to the next is $\mathcal{O}(p)$ $R^*[\cdot]$ values, which take $\mathcal{O}(\sigma p)$ words. By Lemma 5, the computation of each value takes $\mathcal{O}(\sigma)$ time so the overall complexity is $\mathcal{O}(\sigma p^2)$ time. ◀

3.2 Inverting Context Adaptive BWTs

We now show that the machinery we set up for counting occurrences can be used to retrieve s given $BWT_*(s)$, thus to invert any context adaptive BWT.

▶ **Lemma 7.** *Given $R^*[x] = [b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$ and a row index i with $b_x \leq i < b_x + \sum_{j=1}^{\sigma} \ell_j$, the $(|x|+1)$-st character of row i can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. Let $\rho_1, \ldots, \rho_\sigma$ denote the alphabet symbol reordered according to the permutation π_x, and let $\ell'_1, \ldots, \ell'_\sigma$ denote the values $\ell_1, \ldots, \ell_\sigma$ reordered according to the same permutation. Since $i \in R[x]$, row i is prefixed by x. Since the rows prefixed by x are sorted in their $(|x|+1)$-st position according to π_x, the $(|x|+1)$-st symbol of row i is the symbol ρ_j such that

$$b_x + \sum_{1 \leq h < j} \ell'_h \;\leq\; i \;<\; b_x + \sum_{1 \leq h \leq j} \ell'_h$$ ◀

▶ **Theorem 8.** *Given $BWT_*(s)$ with constant time rank support, the permutations π_x used to build the matrix $M_*(s)$, and the row index i containing s in $M_*(s)$, the original string s can be recovered in $\mathcal{O}(\sigma|s|^2)$ time and $\mathcal{O}(\sigma|s|)$ working space.*

Proof. Let $s = s_1 s_2 \cdots s_n$. From $BWT_*(s)$, in $\mathcal{O}(n)$ time we retrieve the number of occurrences of each character in s and hence the ranges $R[c_1]$, $R[c_2]$, ..., $R[c_\sigma]$. From those and the row index i, we retrieve s's first character s_1. Next, counting the number of occurrences of s_1 in the ranges of $BWT_*(s)$ corresponding to $R[c_1]$, $R[c_2]$, ..., $R[c_\sigma]$, we compute $R^*[s_1]$.

Finally, we show by induction that, for $m = 1, \ldots, n-1$, given $R^*[s_1 s_2 \cdots s_m]$, we can retrieve s_{m+1} and $R^*[s_1 s_2 \cdots s_{m+1}]$ in $\mathcal{O}(m\sigma)$ time. By Lemma 7, from $R^*[s_1 s_2 \cdots s_m]$ and i we retrieve s_{m+1}. Next, assuming we maintained the ranges $R^*[s_j \cdots s_m]$, for $j = 1, \ldots, m$ we can compute $R^*[s_j \cdots s_{m+1}]$ adding one diagonal to the scheme shown in the proof of Theorem 6. By Lemma 5, the overall cost is $\mathcal{O}(\sigma|s|^2)$ as claimed. ◀

4 BWTs based on local orderings

In our definition of context adaptive transformation, the alphabet ordering π_x associated to x can depend on the whole string x; in this sense the context has full memory. In this section we consider transformations in which the context has a bounded memory, in that it only depends on the last k symbols of x, where k is fixed. In the following we refer to these string transformations as *BWTs based on local orderings*.

We start by analyzing the case $k = 1$. For such local ordering transformations the matrix $M_*(s)$ depends on only $\sigma + 1$ alphabet orderings: one for each symbol plus the one used to sort the first column of $M_*(s)$. The following lemma establishes an important property of local ordering transformations.

▶ **Lemma 9.** *If $M_*(s)$ is based on a local ordering, then for any pair of characters x_1, x_2 there is an order preserving bijection between the set of rows starting with x_1x_2 and the set of rows starting with x_2 and ending with x_1.*

Proof. Note that both sets of rows contain a number of elements equal to the number of occurrences of x_1x_2 in the circular string s. In the following, we write $s[i \cdots]$ to denote the cyclic rotation of s starting with $s[i]$. Assume that rotations $s[i \cdots]$ and $s[j \cdots]$ both start with x_2 and end with x_1 and let h denote the first column in which the two rotations differ. Rotation $s[i \cdots]$ precedes $s[j \cdots]$ in $M_*(s)$ if and only if $s[i+h]$ is smaller than $s[j+h]$ according to the alphabet ordering associated to symbol $s[i+h-1] = s[j+h-1]$. The two rotations $s[i-1 \cdots]$ and $s[j-1 \cdots]$ both start with x_1x_2 and their relative position also depends on the relative ranks of $s[i+h]$ and $s[j+h]$ according to the alphabet ordering associated to symbol $s[i+h-1] = s[j+h-1]$. Hence the relative order of $s[i-1 \cdots]$ and $s[j-1 \cdots]$ is the same as the one of $s[i \cdots]$ and $s[j \cdots]$. ◀

Armed with the above lemma, we now show that for local ordering transformations we can establish much stronger results than the one provided in Section 3.1.

▶ **Lemma 10.** *Suppose $BWT_*(s)$ supports constant time rank queries. Let $x = x_1x_2 \cdots x_m$ be any length-m string with $m > 1$. Then, given $R[x_1x_2]$, $R[x_2]$ and $R[x_2 \cdots x_m]$, the value $R[x_1 \cdots x_m]$ can be computed in $\mathcal{O}(1)$ time.*

Proof. By Lemma 9 there is an order preserving bijection between the rows in $R[x_1x_2]$ and those in $R[x_2]$ ending with x_1. In this bijection, the rows in $R[x_1 \cdots x_m]$ correspond to those in $R[x_2 \cdots x_m]$ ending with x_1. Hence, if, among the rows starting with x_2 and ending with x_1, those prefixed by $x_2 \cdots x_m$ are in positions $r, r+1, \ldots, r+h$, then, among the rows starting with x_1x_2, those prefixed by $x_1x_2 \cdots x_m$ are in positions $r, r+1, \ldots, r+h$. ◀

▶ **Theorem 11.** *Suppose $BWT_*(s)$ is based on a local ordering and supports constant time rank queries. After a $\mathcal{O}(\sigma^2)$ time preprocessing, given any string $x = x_1x_2\cdots x_p$, the range of rows prefixed by x can be computed in $\mathcal{O}(p)$ time and $\mathcal{O}(p)$ space.*

Proof. We reason as in the proof of Theorem 6, except that because of Lemma 10 we can work with $R[\cdot]$ instead of $R^*[\cdot]$ and we only need to compute the first two columns and the diagonal. In the preprocessing step, we compute $R[c_i]$ and $R[c_ic_j]$ for any pair $(c_i, c_j) \in \Sigma^2$. During the search phase, we compute each diagonal entry in constant time. ◀

Another immediate consequence of Lemma 9 is that we can efficiently "move back in the text" as in the original BWT. Note this operation is the base for BWT inversion and for snippet extraction and locate operations on FM-indices [10].

▶ **Lemma 12.** *Suppose $BWT_*(s)$ is based on a local ordering and supports constant time rank and access queries. Then, after a $\mathcal{O}(\sigma^2)$ time preprocessing, given a row index i we can compute in $\mathcal{O}(1)$ time the index of the row obtained from the i-th row with a circular right shift by one position.*

Proof. Compute the first and last symbol of row i and then apply Lemma 9. ◀

▶ **Corollary 13.** *If $BWT_*(s)$ is based on a local ordering and supports constant time rank and access queries, $BWT_*(s)$ can be inverted in $\mathcal{O}(\sigma^2 + |s|)$ time and $\mathcal{O}(\sigma^2)$ working space.*

In the full paper we will show that bounded context adaptive BWTs can be generalized to the case in which the ordering π_x depends only on the last $k > 1$ symbols of x. Search and inversion can still be performed in linear time with the only difference that preprocessing now takes $\mathcal{O}(\sigma^{k+1})$ time and space.

5 Run minimization problem

In this section we consider the following problem: given a string s and a class of BWT variants, find the variant that minimizes the number of runs in the transformed string. As we mentioned in the introduction this problem is relevant for the compression of highly repetitive collections.

We consider the general class of context adaptive BWTs described in Section 3. In this class we can select an alphabet ordering π_x independently for every substring x. However, it is easy to see that the only orderings that influence the output of the transform are those associated to strings corresponding to the internal nodes of the suffix tree of s. Given a suffix tree node v we denote by $bw(v)$ the multiset of symbols associated to the leaves in the subtree rooted at v. We say that a string z_v is a *feasible* arrangement of $bw(v)$ if we can reorder the nodes in the subtree rooted at v so that z_v is obtained reading left to right the symbols in the reordered subtree. For example, in the suffix tree of Figure 3 (left), if v is the internal node with upward path aa it is $bw(v) = \{a, b, c\}$ and bac, bca, acb, cab are feasible arrangements of $bw(v)$, while abc and cba are *not* feasible arrangements. If τ is the suffix tree root, using the above notation our problem becomes that of finding the feasible arrangement of $bw(\tau)$ with the minimal number of runs. For constant alphabets the following theorem, proven in the Appendix, shows that the optimal arrangement can be found in linear time using dynamic programming.

▶ **Theorem 14.** *Given a string s over a constant size alphabet, the context adaptive transformation BWT_* minimizing the number of runs in $BWT_*(s)$ can be found in $\mathcal{O}(|s|)$ time.*

Proof. Let *Opt* denote the minimal number of runs. We show how to compute *Opt* with a dynamic programming algorithm; the computation of the alphabet orderings giving *Opt* is done using standard techniques. For each suffix tree node v and pairs of symbols c_i, c_j let $\rho(v, c_i, c_j)$ denote the minimal number of runs among all feasible arrangements of $bw(v)$ starting with c_i and ending with c_j. Clearly, if τ is the suffix tree root, then $Opt = \min_{i,j} \rho(\tau, c_i, c_j)$.

For each leaf ℓ it is $\rho(\ell, c_i, c_j) = 1$ if $c_i = c_j = bw(\ell)$ and $\rho(\ell, c_i, c_j) = \infty$ otherwise. We need to show how to compute, for each internal node v, the σ^2 values $\rho(v, c_i, c_j)$ for c_i, c_j in Σ, given the, up to σ^3 values, $\rho(w_k, c_\ell, c_m)$, $k = 1, \ldots, h$, where $w_1, \ldots, w_h$ are the children of v. To this end, we show that for each ordering π of $w_1, \ldots, w_h$ we can compute in constant time the minimal number of runs among all the feasible arrangements of $bw(v)$ starting with c_i and ending with c_j and with the additional constraint that v's children are ordered according to π.

To simplify the notation assume $w_1, \ldots, w_h$ have been already reordered according to π. For $k = 1, \ldots, h$ let $M_\pi[k, c_\ell, c_m]$ denote the minimal number of runs among all strings x such that $x = y_1 \cdots y_k$ where y_t, for $t = 1, \ldots, k$, is a feasible arrangement of $bw(w_t)$, and with the additional constraints that y_1 starts with c_ℓ and y_k ends with c_m. We have

$$M_\pi[1, c_\ell, c_m] = \rho(w_1, c_\ell, c_m)$$

and for $k = 2, \ldots, h$

$$M_\pi[k, c_\ell, c_m] = \min_{i,j} \left(M_\pi[k-1, c_\ell, c_i] + \rho(w_k, c_j, c_m) - \delta_{ij} \right) \tag{2}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. Essentially, (2) states that to find the minimal number of runs for $w_1, \ldots, w_k$ we consider all possible ways to combine an optimal solution for $w_1, \ldots, w_{k-1}$ followed by a feasible arrangement of $bw(w_k)$. The δ_{ij} term comes from the fact that the number of runs in the concatenation of two strings is equal to the sum of the runs in each string, minus one if the last symbol of the first string is equal to the first symbol of the second string.

Once we have the values $M_\pi[h, c_i, c_j]$, the desired values $\rho(v, c_i, c_j)$ are obtained taking the minimum over all possible alphabet ordering π. ◀

Note that, both the assumptions on the alphabet size and the constant-time rank operations could be relaxed without affecting the correctness of the results provided in this paper, accordingly the running time increases. For instance, in Theorem 14, the algorithm runs in $\mathcal{O}(|s|\sigma^2)$ time, for any alphabet.

Clearly the above theorem does not immediately yield a practical compressor, since the cost of specifying the alphabet ordering at each node is likely to outweigh the advantage of minimizing the number of runs. However we notice that: 1) the optimal transformation for a string will reasonably produce good results on similar strings so we can compute and store the ordering once and use it many times, 2) since Theorem 14 holds for the most general class, it provides a lower bound for the more interesting and practical BWTs based on local orderings and the ABWT.

References

1 D. Belazzougui and G. Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM T. Algorithms*, 11(4):31:1–31:21, 2015.

2 M. Burrows and D. J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DIGITAL System Research Center, 1994.

3 B. Chapin and S. Tate. Higher Compression from the Burrows-Wheeler Transform by Modified Sorting. In *DCC*, page 532. IEEE Computer Society, 1998. Full version available from `https://www.uncg.edu/cmp/faculty/srtate/papers/bwtsort.pdf`.

4 A. Cox, M. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

5 M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows-Wheeler transformation. *Theor. Comput. Sci.*, 332:567–572, 2005.

6 J. S. Culpepper, M. Petri, and S. J. Puglisi. Revisiting bounded context block-sorting transformations. *Software Pract. Exper.*, 42(8):1037–1054, 2012.

7 J. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, and É. Prieur-Gaston. A survey of string orderings and their application to the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 2017.

8 P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.

9 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS 2000*, pages 390–398. IEEE Computer Society, 2000.

10 P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52:552–581, 2005.

11 T. Gagie, G. Manzini, and J. Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017.

12 T. Gagie, G. Navarro, and N. Prezza. Optimal-Time Text Indexing in BWT-runs Bounded Space. In *SODA*, pages 1459–1477. SIAM, 2018.

13 I. M. Gessel, A. Restivo, and C. Reutenauer. A bijection between words and multisets of necklaces. *Eur. J. Combin.*, 33(7):1537–1546, 2012.

14 I. M. Gessel and C. Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory A*, 64(2):189–215, 1993.

15 R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, and M. Sciortino. Block Sorting-Based Transformations on Words: Beyond the Magic BWT. In *DLT*, volume 11088 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2018.

16 R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387:236–248, 2007.

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

18 D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *STOC*, pages 827–840. ACM, 2018.

19 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

20 V. Mäkinen, D. Belazzougui, F. Cunial, and A. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015. ISBN 978-1-107-07853-6.

21 V. Mäkinen, G. Navarro, J. Sirén, and Niko Välimäki. Storage and Retrieval of Highly Repetitive Sequence Collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

22 S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Burrows-Wheeler Transform and Run-Length Enconding. In *Combinatorics on Words - 11th International Conference, WORDS 2017. Proceedings*, volume 10432 of *LNCS*, pages 228–239. Springer, 2017.

23 S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, and L. Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017.

24 G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

25 G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0.

26 G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Comput. Surv.*, 39(1), 2007.

27 M. Petri, G. Navarro, J. S. Culpepper, and S. J. Puglisi. Backwards Search in Context Bound Text Transformations. In *CCP*, pages 82–91. IEEE Computer Society, 2011.

28 M. Schindler. A Fast Block-Sorting algorithm for Lossless Data Compression. In *DCC*, page 469. IEEE Computer Society, 1997.

Compressed Multiple Pattern Matching

Dmitry Kosolobov
University of Helsinki, Helsinki, Finland
dkosolobov@mail.ru

Nikita Sivukhin
Ural Federal University, Ekaterinburg, Russia
sivukhin.nikita@yandex.ru

Abstract

Given d strings over the alphabet $\{0, 1, \ldots, \sigma-1\}$, the classical Aho–Corasick data structure allows us to find all occ occurrences of the strings in any text T in $O(|T| + occ)$ time using $O(m \log m)$ bits of space, where m is the number of edges in the trie containing the strings. Fix any constant $\varepsilon \in (0, 2)$. We describe a compressed solution for the problem that, provided $\sigma \leq m^\delta$ for a constant $\delta < 1$, works in $O(|T| \frac{1}{\varepsilon} \log \frac{1}{\varepsilon} + occ)$ time, which is $O(|T| + occ)$ since ε is constant, and occupies $mH_k + 1.443m + \varepsilon m + O(d \log \frac{m}{d})$ bits of space, for all $0 \leq k \leq \max\{0, \alpha \log_\sigma m - 2\}$ simultaneously, where $\alpha \in (0, 1)$ is an arbitrary constant and H_k is the kth-order empirical entropy of the trie. Hence, we reduce the $3.443m$ term in the space bounds of previously best succinct solutions to $(1.443 + \varepsilon)m$, thus solving an open problem posed by Belazzougui. Further, we notice that $L = \log \binom{\sigma(m+1)}{m} - O(\log(\sigma m))$ is a worst-case space lower bound for any solution of the problem and, for $d = o(m)$ and constant ε, our approach allows to achieve $L + \varepsilon m$ bits of space, which gives an evidence that, for $d = o(m)$, the space of our data structure is theoretically optimal up to the εm additive term and it is hardly possible to eliminate the term $1.443m$. In addition, we refine the space analysis of previous works by proposing a more appropriate definition for H_k. We also simplify the construction for practice adapting the fixed block compression boosting technique, then implement our data structure, and conduct a number of experiments showing that it is comparable to the state of the art in terms of time and is superior in space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases multiple pattern matching, compressed space, Aho–Corasick automaton

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.13

Related Version `https://arxiv.org/abs/1811.01248`

Funding Supported by the Russian Science Foundation (RSF), project 18-71-00002.

1 Introduction

Searching for multiple patterns in text is a fundamental stringology problem that has numerous applications, including bioinformatics [20], search engines [36], intrusion detection systems [27, 32], shortest superstring approximation [2], and others. The classical solution for the multiple pattern matching is the Aho–Corasick data structure [1], which, however, does not always fulfil space requirements of many such applications due to the rapid growth of the amounts of data in modern systems. To address this issue, several space-efficient multiple pattern matching data structures were developed in the last decade. In this paper we improve the space consumption in a state-of-the-art solution for the problem, simplify the compression method used in this solution by adapting the known fixed block compression boosting technique, and give an evidence that the achieved space is, in a sense, close to optimal; in addition, we refine the theoretical space analysis of a previous best result, and implement our construction and conduct a number of experiments showing that it is comparable to the existing practical data structures in terms of time and is superior in space. Before discussing our contribution in details, let us briefly survey known results in this topic.

The Aho–Corasick solution builds a trie of all patterns augmented with additional structures overall occupying $O(m \log m)$ bits (hereafter, log denote logarithm with base 2), where m is the number of edges in the trie, and allows us to find all *occ* occurrences of the patterns in any text T in $O(|T| + occ)$ time. The $O(m \log m)$-bit space overhead imposed by this solution might be unacceptably high if one is processing large sets of patterns; for this case, several succinct and compressed data structures were developed in the last decade [3, 5, 11, 21, 22, 25, 35]. In this paper we are especially interested in the two closely related results from [3] and [21], which provide currently the best time and space bounds for the multiple pattern matching problem. In [3] Belazzougui designed a compact representation of the Aho–Corasick scheme that works in the same $O(|T| + occ)$ time but stores the trie with all additional structures in only $m \log \sigma + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits, where σ is the alphabet size and d is the number of patterns; in addition, he showed that the space can be improved to $mH_0 + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits with no slowdown provided $\sigma \leq m^\delta$ for a constant $\delta < 1$, where H_0 is the zeroth-order empirical entropy of the trie. In [21] Hon et al. further lowered the space to $mH_k + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits (again, assuming $\sigma \leq m^\delta$) by simply applying the compression boosting technique [12], where H_k is the kth-order empirical entropy of the trie (see clarifications below) and k is any fixed integer such that $0 \leq k \leq \alpha \log_\sigma m - 1$, for arbitrary constant $\alpha \in (0, 1)$. This topic is rich with other related results, which, for instance, support dynamic modifications of the patterns [5, 10, 11], try to process T in real-time [25], allow randomization [7, 18], consider hardware implementations [9], etc. In this paper we focus on the basic functionality as in [3] and [21].

Belazzougui posed the following open problem [3]: can we reduce the constant 3.443 in the space of his (and Hon et al.'s) result without any significant slowdown? We solve this problem affirmatively describing a data structure that, provided $\sigma \leq m^\delta$ for a constant $\delta < 1$, occupies $mH_k + 1.443m + \varepsilon m + O(d \log \frac{m}{d})$ bits of space, for an arbitrarily chosen constant $\varepsilon \in (0, 2)$, and answers pattern matching queries on any text T in $O(|T|\varepsilon^{-1} \log \varepsilon^{-1} + occ)$ time, which is $O(|T| + occ)$ since ε is constant. Then, we notice that $L = \log \binom{\sigma(m+1)}{m} - O(\log(\sigma m))$ is a worst-case space lower bound for any multiple pattern matching data structure and, for $d = o(m)$ and constant ε, our construction allows to achieve $L + \varepsilon m$ bits of space; observe that, for $\sigma = \omega(1)$, we have $L = m \log \sigma + m \log e + o(m) \approx m \log \sigma + 1.443m + o(m)$ (see [3]), which gives an evidence that, for $d = o(m)$, our space bound is optimal up to the εm additive term and it is hardly possible to remove the term $1.443m$. In addition, we argue that the definition of H_k borrowed by Hon et al. [21] from [13], denoted H_k^* in our paper, is not satisfactory: in particular, H_k^* can be greater than $\log \sigma$, which contradicts the idea of the empirical entropy (H_k^* was devised in [13] for a slightly different problem); we propose a more appropriate definition for H_k, which is not worse since $H_k \leq H_k^*$, and refine the analysis of Hon et al. showing that their data structure indeed occupies $mH_k + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits even according to our definition of H_k. Further, our solution, unlike [21], does not require to fix k and the space bound holds for all k such that $0 \leq k \leq \max\{0, \alpha \log_\sigma m - 2\}$ simultaneously; this is achieved by adapting the fixed block compression boosting technique [17, 24] to our construction, which is also better for practice than the compression boosting used in [21]. Finally, we implement our data structure and conduct a number of experiments showing that it is comparable to the state of the art in terms of time and is superior in space.

The paper is organized as follows. In the following section we introduce some basic notions and define the kth-order empirical entropy of tries. In Section 3 we survey the solution of Belazzougui [3]. In Section 4 we discuss compression boosting techniques and investigate the flaws of the space analysis of Hon et al. [21]. Section 5 describes our main data structure and considers space optimality. Appendix contains implementation details and experiments.

2 Preliminaries

Throughout the paper, we mainly consider strings drawn from the alphabet $\{0, 1, \ldots, \sigma-1\}$ of size σ (not necessarily constant). For a string $s = c_0c_1 \cdots c_{n-1}$, denote by $|s|$ its length n. The *reverse* $c_{n-1} \cdots c_1c_0$ of s is denoted s^r. We write $s[i]$ for the letter c_i of s and $s[i..j]$ for the *substring* $c_ic_{i+1} \cdots c_j$, assuming $s[i..j]$ is the empty string if $i > j$. We say that a string p *occurs* in s at position i if $s[i..i+|p|-1] = p$. A string p is called a *prefix* (resp., *suffix*) of s if p occurs in s at position 0 (resp., $|s| - |p|$). For integer segments, we use the following notation: $[i..j] = \{i, i+1, \ldots, j\}$, $(i..j] = [i..j] \setminus \{i\}$, $[i..j) = [i..j] \setminus \{j\}$. The set of all strings of lengths k over an alphabet A is denoted A^k; we use this notation for the set $[0..\sigma]^k$. For letter c, the only string of the set $\{c\}^k$ is denoted c^k.

The trie containing a set of strings S is the minimal in the number of vertices rooted tree with edges labeled by letters so that each $s \in S$ can be spelled out on the path from the root to a vertex. For vertex v, denote the string written on the path from the root to v by $\mathsf{str}(v)$.

The *zeroth-order empirical entropy* (see [26, 29]) of a string t of length n is defined as $H_0(t) = \sum_{c \in [0..\sigma)} \frac{n_c}{n} \log \frac{n}{n_c}$, where n_c is the number of letters c in t and $\frac{n_c}{n} \log \frac{n}{n_c} = 0$ whenever $n_c = 0$. For a string w of length k, let t_w be a string formed by concatenating all letters immediately following occurrences of w in the string $\$^k t$, where $\$ = \sigma$ is a new special letter introduced for technical convenience; e.g., $t_{ab} = aac$ and $t_{\$a} = b$ for $t = abababc$. The *kth-order empirical entropy* of t is defined as $H_k(t) = \sum_{w \in [0..\sigma]^k} \frac{|t_w|}{n} H_0(t_w)$ (see [26, 28, 29]). It is well known that $\log \sigma \geq H_0 \geq H_1 \geq \cdots$ and H_k makes sense as a measure of string compression only for $k < \log_\sigma n$ (see [15] for a deep discussion). For the sake of completeness, let us show that $H_k \geq H_{k+1}$; this proof then can be easily adapted for the empirical entropy of tries below. Curiously, to our knowledge, all sources refer to this simple and intuitive fact as "obvious" but do not give a proof; even the original paper [29], the survey [28], and the book [30].

▶ **Lemma 1** (see [24, Lemma 3]). *For any strings $t_1, t_2, \ldots, t_\ell$ and the string $t = t_1t_2 \cdots t_\ell$, we have $|t|H_0(t) \geq \sum_{i=1}^{\ell} |t_i|H_0(t_i)$.*

Since, without loss of generality, one can assume that $t_w = t_{a_0w}t_{a_1w} \cdots t_{a_\sigma w}$, where $a_0, \ldots, a_\sigma$ are all letters of $[0..\sigma]$, Lemma 1 directly implies the inequality $|t|H_k(t) = \sum_{w \in [0..\sigma]^k} |t_w|H_0(t_w) \geq \sum_{w \in [0..\sigma]^{k+1}} |t_w|H_0(t_w) = |t|H_{k+1}(t)$ and, hence, $H_k \geq H_{k+1}$.

By analogy, one can define the empirical entropy for tries (see also [13]). Let $\mathcal{T}$ be a trie with n edges over the alphabet $[0..\sigma)$. For a string w of length k, denote by $\mathcal{T}_w$ a string formed by concatenating in an arbitrary order the letters on the edges (u, v) of $\mathcal{T}$ (here u is the parent of v) such that w is a suffix of $\$^k\mathsf{str}(u)$; e.g., $\mathcal{T}_{\$^k}$ consists of all letters labeling the edges incident to the root. Then, the *kth-order empirical entropy* of $\mathcal{T}$ is defined as $H_k(\mathcal{T}) = \sum_{w \in [0..\sigma]^k} \frac{|\mathcal{T}_w|}{n} H_0(\mathcal{T}_w)$. (Note that $\sum_{w \in [0..\sigma]^k} |\mathcal{T}_w| = n$.) Analogously to the case of strings, one can show that $\log \sigma \geq H_0(\mathcal{T}) \geq H_1(\mathcal{T}) \geq \cdots$ and $H_k(\mathcal{T})$ makes sense as a measure of compression only for $k < \log_\sigma n$. For the definition of the kth-order empirical entropy of tries as given by Hon et al. [21], see Section 4.

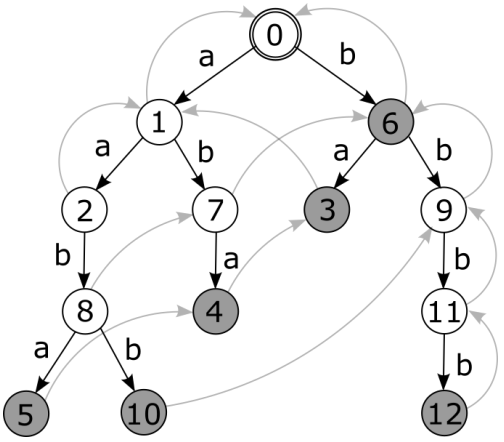
3 Basic Algorithm

Given a dictionary $\mathcal{D}$ of d patterns, the multiple pattern matching problem is to preprocess $\mathcal{D}$ in order to efficiently find all occurrences of the patterns in an arbitrary given text. In this section we describe for this problem the classical Aho–Corasick solution [1] and its space-efficient version developed by Belazzougui [3].

The main component of the Aho–Corasick data structure is the trie $\mathcal{T}$ containing $\mathcal{D}$. Each vertex v of the trie is augmented with the following structures (see Figure 1a):

- a flag $\mathsf{mark}(v)$ that indicates whether $\mathsf{str}(v) \in \mathcal{D}$: it is so iff $\mathsf{mark}(v) = 1$;
- a hash table $\mathsf{next}(v, \cdot)$ that, for each letter c, either maps c to a vertex $u = \mathsf{next}(v, c)$ such that $\mathsf{str}(u) = \mathsf{str}(v)c$, or returns nil if there is no such u;
- a link $\mathsf{failure}(v)$ to a vertex such that $\mathsf{str}(\mathsf{failure}(v))$ is the longest proper suffix of $\mathsf{str}(v)$ that can be spelled out on a root-vertex path ($\mathsf{failure}(v)$ is undefined if v is the root);
- a link $\mathsf{report}(v)$ to a vertex such that $\mathsf{str}(\mathsf{report}(v))$ is the longest proper suffix of $\mathsf{str}(v)$ that belongs to $\mathcal{D}$, or $\mathsf{report}(v) = root$ if there is no such suffix.

It is well known that the described data structure allows us to find all occ occurrences of the patterns from $\mathcal{D}$ in text T in $O(|T| + occ)$ time: we read T from left to right maintaining a "current" vertex v in $\mathcal{T}$ (initially, v is the root) and, when a new letter $T[i]$ arrives, we put $v = \mathsf{next}(u, T[i])$ for the first u in the series $v, \mathsf{failure}(v), \mathsf{failure}(\mathsf{failure}(v)), \ldots$ for which $\mathsf{next}(u, T[i])$ is not nil, or put $v = root$ if there is no such u, then we report all patterns ending at position i using the report links and the flag $\mathsf{mark}(v)$ (see details in, e.g., [8]).



(a) The trie $\mathcal{T}$ and failure links; each vertex v is numbered by $\mathsf{num}(v)$ and is shaded iff $\mathsf{mark}(v) = 1$.

$\mathsf{num}(v)$	$\mathsf{str}(v)$	B_a	B_b
0		1	1
1	a	1	1
2	aa	0	1
3	ba	0	0
4	aba	0	0
5	$aaba$	0	0
6	b	1	1
7	ab	1	0
8	aab	1	1
9	bb	0	1
10	$aabb$	0	0
11	bbb	0	1
12	$bbbb$	0	0

(b) Encoding of transitions next in the trie $\mathcal{T}$ using the bit arrays B_a and B_b.

Figure 1 The trie $\mathcal{T}$ with $\mathcal{D} = \{aaba, aabb, aba, b, ba, bbbb\}$.

It is easy to see that the data structure occupies $O(m \log m)$ bits of space, where m is the number of edges in $\mathcal{T}$. Let us describe now how Belazzougui could reduce the space consumption of this solution with no slowdown (see [3] for a more detailed explanation).

First, he assigned to each vertex v of $\mathcal{T}$ a unique number $\mathsf{num}(v) \in [0..m]$ so that, for any two vertices u and v, we have $\mathsf{num}(u) < \mathsf{num}(v)$ iff $\mathsf{str}(u)^r < \mathsf{str}(v)^r$ lexicographically (see Figure 1a). As it was pointed out by Hon et al. [21], this subtle numbering scheme corresponds to the numbering of vertices in the so-called XBW of $\mathcal{T}$ (see [13]), a generalization of the classical Burrows–Wheeler transform (BWT) [4] for tries. It turns out that the numbering allows us to organize fast navigation in the trie in small space, simulating the tables next in a manner that resembles the so-called FM-indexes [14] based on the BWT.

For each letter $c \in [0..\sigma)$, define a bit array $\mathsf{B}_c[0..m]$ such that, for any vertex v, we have $\mathsf{B}_c[\mathsf{num}(v)] = 1$ iff $\mathsf{next}(v, c) \neq \mathsf{nil}$ (see Figure 1b). Let $\mathsf{rank}(i, \mathsf{B}_c)$ be an operation on B_c, called *partial rank*, that returns nil if $\mathsf{B}_c[i] \neq 1$, and returns the number of ones in $\mathsf{B}_c[0..i]$ otherwise. By standard arguments, one can show that $\mathsf{num}(\mathsf{next}(v, c)) = \mathsf{rank}(\mathsf{num}(v), \mathsf{B}_c) + e_{<c}$, provided $\mathsf{next}(v, c) \neq \mathsf{nil}$, where $e_{<c}$ is the number of edges in the trie with labels smaller than c

(see [3, 13]). Let us concatenate $\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_{\sigma-1}$, thus obtaining a new bit array B of length $(m+1)\sigma$. Since $e_{<c}$ is equal to the number of ones in the arrays $\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_{c-1}$, we obtain $\mathsf{num}(\mathsf{next}(v, c)) = \mathsf{rank}(c(m+1) + \mathsf{num}(v), \mathsf{B})$ and $\mathsf{rank}(c(m+1) + \mathsf{num}(v), \mathsf{B}) = \mathsf{nil}$ iff $\mathsf{next}(v, c) = \mathsf{nil}$. In order to support rank in $O(1)$ time, we equip B with the following data structure, which, in addition, supports the operation $\mathsf{select}(i, \mathsf{B})$ that returns the position of the ith one in B (select is used below to compute the parent of v by number $\mathsf{num}(v)$; see [3]).

▶ **Lemma 2** (see [33]). *Every bit array of length n with m ones has an encoding that occupies $\log \binom{n}{m} + o(m)$ bits and supports select and partial rank queries in $O(1)$ time.*

Since B contains exactly m ones, the space occupied by B encoded as in Lemma 2 is $\log \binom{(m+1)\sigma}{m} + o(m)$, which in [3] was estimated by $m \log \sigma + 1.443m + o(m)$ (here $1.443 \approx \log e$). To encode mark, Belazzougui constructs a bit array of length $m+1$ containing exactly d ones at positions $\mathsf{num}(v)$ for all d vertices v with $\mathsf{mark}(v) = 1$, and stores the array as in Lemma 2 (for this array, we need only access queries, which can be simulated by rank), thus occupying $\log \binom{m+1}{d} + o(d) \le O(d \log \frac{m}{d})$ bits of space. It remains to encode failure and report links.

Belazzougui noticed that the failure links form a tree on the vertices of $\mathcal{T}$ in which, for vertex v, $\mathsf{failure}(v)$ returns the parent of v; more importantly, the numbering num corresponds to the order of vertices in a depth first traversal of this tree. This allows us to represent the tree of failure links in $2m + o(m)$ bits using the following lemma.

▶ **Lemma 3** (see [31]). *Every tree with m edges can be encoded in $2m + o(m)$ bits with the support of the operation $\mathsf{parent}(v)$, which returns the parent of vertex v, in constant time, provided all the vertices are represented by their numbers in a depth first traversal.*

An analogous observation is also true for the report links and the tree induced by them has at most d internal vertices, which allows to spend only $O(d \log \frac{m}{d}) + o(m)$ bits for it.

▶ **Lemma 4** (see [23]). *Every tree with m edges and d internal vertices can be encoded in $d \log \frac{m}{d} + O(d) + o(m)$ bits with the support of the operation $\mathsf{parent}(v)$ in constant time, provided all the vertices are represented by their numbers in a depth first traversal.*

Thus, the total space consumed by the succinct versions of the structures mark, next, $\mathsf{failure}$, and report is $m \log \sigma + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits.

4 Compression Boosting

In [3] it was noticed that if each array $\mathsf{B}_0, \mathsf{B}_1, \ldots, \mathsf{B}_{\sigma-1}$ is encoded separately using Lemma 2, then they altogether occupy $\sum_{0 \le c < \sigma} (\log \binom{m+1}{n_c} + o(n_c))$ bits, where n_c is the number of labels c in $\mathcal{T}$, which is upper bounded by $\sum_{0 \le c < \sigma} (n_c \log \frac{m}{n_c} + 1.443 n_c) + o(m) = mH_0(\mathcal{T}) + 1.443m + o(m)$ (here we apply the inequalities $\log \binom{m+1}{n} \le n \log \frac{m+1}{n} + n \log e$ and $\log e < 1.443$, and estimate $n_c \log \frac{m+1}{n_c}$ as $n_c \log \frac{m}{n_c} + o(n_c)$; see [3]). Using such separated array encodings and some auxiliary data structures, one can reduce the space of the whole data structure to $mH_0(\mathcal{T}) + 3.443m + o(m) + O(d \log \frac{m}{d})$ bits, provided $\sigma \le m^\delta$ for some constant $\delta < 1$. Hon et al. [21] further developed this idea, applying the compression boosting technique [12].

Compression boosting. Choose an integer k such that $0 \le k \le \alpha \log_\sigma m - 1$, where $\alpha \in (0, 1)$ is an arbitrary fixed constant. For $i \in [0..m]$, denote by $\mathsf{istr}(i)$ the string $\mathsf{str}(v)$ such that v is the vertex of $\mathcal{T}$ with $\mathsf{num}(v) = i$. For technical reasons, we introduce a special letter $\$ = \sigma$. Hon et al. [21] partition the set $[0..m]$ into segments in which the strings $\mathsf{istr}(i)$, for i from the same segment, have a common suffix of length k; then the subarrays of B_c corresponding

to these segments are encoded separately using Lemma 2. More precisely, let $[\ell_\rho..r_\rho]$, for $\rho \in [0..\sigma]^k$, be the set of all $i \in [0..m]$ such that ρ is a suffix of $\$^k\mathsf{istr}(i)$ (the definition of num implies that this set forms a segment); each array $\mathsf{B}_c[0..m]$, for $c \in [0..\sigma)$, is partitioned into the subarrays $\mathsf{B}_c[\ell_\rho..r_\rho]$, where $\rho \in [0..\sigma]^k$ (empty segments $[\ell_\rho..r_\rho]$ are excluded), and each such subarray is encoded separately using Lemma 2. Then, all the encoded subarrays in total occupy $\sum_{0\le c<\sigma}\sum_{\rho\in[0..\sigma]^k}(\log\binom{m_\rho}{n_{c,\rho}} + o(n_{c,\rho}))$ bits, where $m_\rho = r_\rho - \ell_\rho + 1$ and $n_{c,\rho}$ is the number of ones in the subarray $\mathsf{B}_c[\ell_\rho..r_\rho]$. Hon et al. upper bound this sum by $mH_k^*(\mathcal{T}) + 1.443m + o(m)$, where $H_k^*(\mathcal{T}) = \frac{m+\ell}{m+1}H_k(\mathcal{T}^*)$ is their definition of the kth-order empirical entropy (see [21]; for simplicity, we use our notation H_k to define H_k^*) in which ℓ is the number of leaves in $\mathcal{T}$ and $\mathcal{T}^*$ is the trie obtained from $\mathcal{T}$ by attaching to each leaf an outgoing edge labeled with \$. Note that $H_k^*(\mathcal{T}) \ge H_k(\mathcal{T}^*)$.

We believe that the definition of H_k^* by Hon et al. is not satisfactory. The problem is that the inequality $H_k^*(\mathcal{T}) \le \log\sigma$ (and even $H_k(\mathcal{T}^*) \le \log\sigma$), which seems to be natural for any proper definition of the empirical entropy, does not necessarily hold; as a corollary, according to the analysis of Hon et al., the encoding of next occupying $m\log\sigma + 1.443m$ bits in the Belazzougui's data structure might "grow" after compression up to $mH_k^*(\mathcal{T}) + 1.443m$ bits (however, there is no growth in reality, just the upper bound of Hon et al. is too rough). For instance, one can observe such behavior in the trie $\mathcal{T}$ of all strings of length h over the alphabet $\{0,1\}$: while it is straightforward that $H_1(\mathcal{T}) = \log\sigma = 1$, it can be shown that $H_1(\mathcal{T}^*) \approx \log 3$ since, for $c \in \{0,1\}$, the string $\mathcal{T}_c^*$ consisting of all labels in $\mathcal{T}^*$ with "context" c contains roughly $m/4$ of each of the letters 0, 1, and \$, where $m = 2^{h+1} - 2$ is the number of edges in $\mathcal{T}$ (we omit further details as they are straightforward).

For brevity, let us denote the summations $\sum_{\rho\in[0..\sigma]^k}$ and $\sum_{0\le c<\sigma}$ in this paragraph by $\sum_\rho$ and $\sum_c$, respectively. For the sake of completeness, we show that the compression boosting technique allows to achieve the kth-order empirical entropy $H_k(\mathcal{T})$, which, unlike H_k^*, satisfies the inequality $\log\sigma \ge H_k(\mathcal{T})$, i.e., we are to prove that $\sum_c\sum_\rho(\log\binom{m_\rho}{n_{c,\rho}} + o(n_{c,\rho})) \le mH_k(\mathcal{T}) + 1.443m + o(m)$ (but we do not discuss additional structures of Hon et al. required for navigation; see [21]). First, $\sum_c\sum_\rho(\log\binom{m_\rho}{n_{c,\rho}} + o(n_{c,\rho}))$ is upper bounded by $\sum_\rho\sum_c(n_{c,\rho}\log\frac{m_\rho}{n_{c,\rho}} + 1.443n_{c,\rho}) + o(m) = \sum_\rho\sum_c n_{c,\rho}\log\frac{m_\rho}{n_{c,\rho}} + 1.443m + o(m)$. Denote $n_\rho = \sum_c n_{c,\rho}$. Note that, by definition, we have $mH_k(\mathcal{T}) = \sum_\rho\sum_c n_{c,\rho}\log\frac{n_\rho}{n_{c,\rho}}$. Since the function $\log x$ is concave, we have $\log(x+d) \le \log x + d(\log x)' = \log x + d\log e/x$, for any $x > 0$ and any real d such that $x + d > 0$. Hence, we deduce $\sum_\rho\sum_c n_{c,\rho}\log\frac{m_\rho}{n_{c,\rho}} = \sum_\rho\sum_c n_{c,\rho}\log(\frac{n_\rho}{n_{c,\rho}} + \frac{m_\rho - n_\rho}{n_{c,\rho}}) \le \sum_\rho\sum_c n_{c,\rho}(\log\frac{n_\rho}{n_{c,\rho}} + \frac{m_\rho-n_\rho}{n_\rho}\log e) = \sum_\rho\sum_c n_{c,\rho}\log\frac{n_\rho}{n_{c,\rho}} + \sum_\rho(m_\rho - n_\rho)\log e = mH_k(\mathcal{T}) + \log e$; the equality $\sum_\rho(m_\rho - n_\rho)\log e = \log e$ holds since $\sum_\rho m_\rho = m + 1$ and $\sum_\rho n_\rho = m$. Finally, we hide the constant $\log e$ under $o(m)$ and obtain the kth-order entropy compression: $\sum_c\sum_\rho(\log\binom{m_\rho}{n_{c,\rho}} + o(n_{c,\rho})) \le mH_k(\mathcal{T}) + 1.443m + o(m)$.

Fixed block compression boosting. The described partition lacks uniformity and requires relatively complex auxiliary data structures in order to support navigation and queries. Hon et al. [21] indeed organize such an infrastructure using $o(m)$ bits, provided $\sigma \le m^\delta$ for a constant $\delta < 1$ (the condition $0 \le k \le \alpha\log_\sigma m - 1$ plays its role in this part). But it turns out that we can considerably simplify their whole construction using the fixed block boosting by Kärkkäinen and Puglisi [24]. The approach relies on the following lemma.

▶ **Lemma 5** (see [24, Lemma 4]). *Let $s = s_1 s_2 \cdots s_\ell$ be an arbitrary partition of a string s into ℓ substrings. Let $s = s'_1 s'_2 \cdots s'_t$ be a different partition of s into t substrings each of which has length at most b. Then, we have $\sum_{i=1}^t |s'_i|H_0(s'_i) \le \sum_{i=1}^\ell |s_i|H_0(s_i) + (\ell - 1)b$.*

We encode the bit array B, which represents the transitions next, as follows.

▶ **Lemma 6.** *Provided $\sigma \le m^\delta$ for a constant $\delta < 1$, B has an encoding that supports select and partial rank in $O(1)$ time and occupies $mH_k(\mathcal{T}) + 1.443m + o(m)$ bits simultaneously for all $k \in [0..\max\{0, \alpha \log_\sigma m - 2\}]$, where $\alpha \in (0,1)$ is an arbitrary fixed constant.*

Proof. We first discuss a fixed block encoding of the arrays B_c and prove, by means of Lemma 5, that it achieves the kth-order entropy compression. Then, we describe auxiliary structures that occupy only $o(m)$ bits and are used for queries and navigation in the blocks.

We partition each array $\mathsf{B}_c[0..m]$ into $t = \lceil \frac{m+1}{b} \rceil$ blocks of length $b = \sigma\lceil \log^2 m \rceil$ (the last block can be shorter), encode each block using Lemma 2, and concatenate the encodings. Thus, we consume $\sum_{i=1}^{t} \sum_{0 \le c < \sigma} (\log \binom{b_i}{n_{c,i}} + o(n_{c,i}))$ bits, where $n_{c,i}$ is the number of ones in the block $\mathsf{B}_c[(i-1)b.. \min\{ib-1, m\}]$ and b_i is the length of the ith block (so that $b_i = b$, for $i \in [0..t)$, and $b_t \le b$). Since $\sum_{i=1}^{t} \log \binom{b_i}{n_{c,i}} \le \log \binom{m+1}{n_c}$ for $n_c = \sum_{i=1}^{t} n_{c,i}$, the result trivially holds for $k = 0$. The sum is upper bounded by $\sum_{i=1}^{t} \sum_{0 \le c < \sigma} n_{c,i} \log \frac{b_i}{n_{c,i}} + 1.443m + o(m)$. Let us prove that $\sum_{i=1}^{t} \sum_{0 \le c < \sigma} n_{c,i} \log \frac{b_i}{n_{c,i}} \le mH_k(\mathcal{T}) + o(m)$ for all $k \in (0..\alpha \log_\sigma m - 2]$.

Fix $k \in (0..\alpha \log_\sigma m - 2]$. Denote $n_i = \sum_{0 \le c < \sigma} n_{c,i}$. Using the inequality $\log(x + d) \le \log x + d \log e / x$, we deduce the following upper bound:

$$\sum_{\substack{1 \le i \le t \\ 0 \le c < \sigma}} n_{c,i} \log \frac{b_i}{n_{c,i}} = \sum_{\substack{1 \le i \le t \\ 0 \le c < \sigma}} n_{c,i} \log\left(\frac{n_i}{n_{c,i}} + \frac{b_i - n_i}{n_{c,i}}\right) \le \sum_{\substack{1 \le i \le t \\ 0 \le c < \sigma}} n_{c,i} \log \frac{n_i}{n_{c,i}} + \sum_{i=1}^{t} (b_i - n_i) \log e.$$

First, we have $\sum_{i=1}^{t} (b_i - n_i) \log e = \log e$ since $\sum_{i=1}^{t} b_i = m + 1$ and $\sum_{i=1}^{t} n_i = m$. Second, $\sum_{i=1}^{t} \sum_{0 \le c < \sigma} n_{c,i} \log \frac{n_i}{n_{c,i}} = \sum_{i=1}^{t} |s'_i| H_0(s'_i)$, where s'_i is a string of length n_i formed by concatenating the letters on the edges (u, v) such that $\mathsf{num}(u)$ is inside the ith block, i.e., $\mathsf{num}(u) \in [(i-1)b \,..\, (i-1)b + b_i)$. For $\rho \in [0..\sigma]^k$, let $[\ell_\rho..r_\rho]$ be the set of all i such that ρ is a suffix of $\$^k\mathsf{istr}(i)$, and let s_ρ be a string formed by concatenating the letters on the edges (u, v) such that $\mathsf{num}(u) \in [\ell_\rho..r_\rho]$. By definition, $\sum_{\rho \in [0..\sigma]^k} |s_\rho| H_0(s_\rho) = mH_k(\mathcal{T})$. Note that at most $\ell = \sum_{i=0}^{k} \sigma^i = \frac{\sigma}{\sigma-1}(\sigma^k - \frac{1}{\sigma})$ strings s_ρ are nonempty and $\ell \le 2\sigma^k$ since $\sigma \ge 2$. Let $\rho_1, \rho_2, \ldots, \rho_{(\sigma+1)^k}$ be an ordering of all strings $\rho \in [0..\sigma]^k$ such that $\ell_{\rho_1} \le \cdots \le \ell_{\rho_{(\sigma+1)^k}}$. The definitions of s'_i and s_ρ imply that the letters in s'_i and s_ρ can be arranged so that $s'_1 s'_2 \cdots s'_t = s_{\rho_1} s_{\rho_2} \cdots s_{\rho_{(\sigma+1)^k}}$. Therefore, by Lemma 5, we obtain the next inequality:

$$\sum_{i=1}^{t} |s'_i| H_0(s'_i) \le \sum_{\rho \in [0..\sigma]^k} |s_\rho| H_0(s_\rho) + (\ell - 1) \max_{i \in [1..t]} n_i \le mH_k(\mathcal{T}) + 2\sigma^{k+1} b.$$

As $k \le \alpha \log_\sigma m - 2$, we have $\sigma^{k+1} b = \sigma^{k+2} \lceil \log^2 m \rceil \le m^\alpha \lceil \log^2 m \rceil = o(m)$.

It remains to describe the auxiliary data structures that help to answer select and partial rank queries on the (now virtual) array B. First, we store σt pointers to the data structures encoding the blocks $\mathsf{B}_c[(i-1)b.. \min\{ib-1, m\}]$, for $c \in [0..\sigma)$ and $i \in [1..t]$. For rank, we create an array of length σt that stores the number of ones in the subarrays $\mathsf{B}[0..(m+1)c + ib - 1]$, for $i \in [0..t)$ and $c \in [0..\sigma)$. All this takes $O(\sigma t \log m) = O(\frac{m}{\log^2 m} \log m) = o(m)$ bits. For select, we create a bit array S formed by concatenating unary encodings for the number of ones in the blocks: e.g., if the first four blocks (of all σt blocks) contain, resp., 3, 2, 0, and 2 ones, then $S = 11101100110 \cdots$; S is encoded using Lemma 2 and, thus, occupies $\log \binom{m+\sigma t}{m} + o(m) = \log \binom{m+\sigma t}{\sigma t} + o(m) \le O(\sigma t \log m) + o(m) = o(m)$ bits. Using these structures, one can straightforwardly perform select and partial rank on B in $O(1)$ time. ◀

5 Main Data Structure

The encoding of failure links imposes a $2m$-bit overhead, which, for small alphabet or highly compressible data, might be comparable to the space required for other structures. In this section, we describe a different encoding that uses only $\varepsilon m + o(m)$ bits, for any $\varepsilon \in (0, 2)$.

The key idea is to store the failure links only for some trie vertices. We call a subset W of the vertices of a tree a *t-dense* subset if each vertex $v \notin W$ has an ancestor $p \in W$ located at a distance less than t edges from v. (Note that the definition implies that W must contain the root.) Now we can formulate the main lemma.

▶ **Lemma 7.** *Suppose that* $\mathsf{failure}(v)$ *can be calculated in* $O(1)$ *time only for* $v \in W$*, where* W *is a vertex set that is* t*-dense in the tree* $\mathcal{T}$*; then there is a modification of the Aho–Corasick algorithm that uses the links* $\mathsf{failure}(v)$ *only for* $v \in W$ *and processes any text* T *in* $O(t|T| + occ)$ *time, where* occ *is the number of occurrences of the patterns in* T*.*

Proof. Our algorithm essentially simulates the Aho–Corasick solution but in the case when the usual algorithm calculates $\mathsf{failure}(v)$ for $v \notin W$, the new one instead finds the nearest ancestor $p \in W$ of v, "backtracks" the input string T accordingly, then computes $\mathsf{failure}(p)$, and continues the execution from this point (if p is the root and, thus, $\mathsf{failure}(p)$ is undefined, we simply skip one letter and continue). The pseudocode is as follows (the omitted code reporting patterns in line 2 simply traverses report links and checks whether $\mathsf{mark}(v) = 1$):

1: **function** $auto(v,\ i,\ i_{max})$
2: **if** $i > i_{max}$ **then** $i_{max} \leftarrow i$ and report all patterns ending at position $i - 1$;
3: **if** $i = |T|$ **then return** v;
4: **if** $\mathsf{next}(v, T[i]) \neq$ **nil then return** $auto(\mathsf{next}(v, T[i]), i + 1, i_{max})$;
5: **for** $(p \leftarrow v;\ p \notin W;\ i \leftarrow i - 1)$ **do** ▷ C-style loop with three parameters
6: $(p, c) \leftarrow \mathsf{parent\_edge}(v)$; ▷ p and c are such that $v = \mathsf{next}(p, c)$
7: **if** p is *root* **then return** $auto(root, i + 1, i_{max})$;
8: **return** $auto(\mathsf{failure}(p), i, i_{max})$;

The execution starts with $auto(root, 0, 0)$. The function $\mathsf{parent\_edge}(v)$ returns the parent p of v and the letter c such that $v = \mathsf{next}(p, c)$ (note that we only use p); in [3] it was shown that $\mathsf{num}(p) = x \bmod (m + 1)$ and $c = \lfloor x/(m + 1) \rfloor$, where $x = \mathsf{select}(\mathsf{num}(v), \mathsf{B})$ and B is the bit array that encodes the next transitions (see above). To prove the correctness, let us show by induction on the length of T that $auto(root, 0, 0)$ returns the vertex v_T such that $\mathsf{str}(v_T)$ is the longest suffix of T that can be spelled out on a root-vertex path in $\mathcal{T}$.

The base $|T| = 0$ is trivial. Suppose that the claim holds for all lengths smaller than $|T|$. We are to show that $auto(root, 0, 0) = v_T$. By the inductive hypothesis, when $auto(v, i, i_{max})$ is called with $i = |T|-1$ for the first time, the string $\mathsf{str}(v)$ is the longest suffix of $T[0..|T|-2]$ that can be read on a root-vertex path of $\mathcal{T}$. Therefore, if $\mathsf{next}(v, T[|T|-1]) \neq$ **nil**, $\mathsf{next}(v, T[|T|-1])$ obviously is equal to v_T and we return it in line 4. Now suppose that $\mathsf{next}(v, T[|T|-1]) =$ **nil**. In lines 5–6 we find the nearest ancestor p of v belonging to W or put $p = v$ if $v \in W$, and backtrack accordingly to $T[0..i]$, for $i \in [0..|T|)$, such that $\mathsf{str}(p)$ is a suffix of $T[0..i-1]$. Observe the following claims: (i) for any vertex w such that $\mathsf{str}(w)$ is a suffix of $T[0..i-1]$, the result of $auto(w, i, i_{max})$ is the same as the result of $auto(root, 0, 0)$ with $T := T[i-|\mathsf{str}(w)|\ ..\ |T|-1]$; (ii) $\mathsf{str}(v_T)$ is either the empty string or a proper suffix of $\mathsf{str}(v)$ concatenated with $T[|T|-1]$. When p is not the root, the claim (ii) and the fact that $\mathsf{str}(\mathsf{failure}(p))$ is the longest proper suffix of $\mathsf{str}(p)$ present in the trie imply that $\mathsf{str}(v_T)$ is a suffix of $T[i-|\mathsf{str}(\mathsf{failure}(p))|\ ..\ |T|-1]$. Then, by the claim (i) and the inductive hypothesis, the recursion $auto(\mathsf{failure}(p), i, i_{max})$ in line 8 returns v_T. When p is the root, (i) and (ii) analogously imply that the call to $auto(root, i + 1, i_{max})$ in line 7 returns v_T.

Let us analyze the time complexity. The algorithm maintains two indices: i and $k = i - |\mathsf{str}(v)|$. Each call to $auto(v, i, i_{max})$ with $i \neq |T|$ either increases i in line 4 or increases k in lines 7 or 8. Since W is a t-dense subset, the loop 5–6 can decrease i by at most t before increasing k. Therefore, i can be decreased by at most $t|T|$ in total and, hence, the running time of the whole algorithm is $O(t|T|)$ plus $O(occ)$ time to report pattern occurrences.

The presented solution explicitly stores T (or at least its last m letters) in order to support "backtracking" during the calculations. We, however, cannot afford to allocate the $m \log \sigma$ bits for T and desire to fit the additional space within an $o(m)$ bound. To this end, we maintain only a substring $T[i..i']$ such that $i' - i \leq 2\sqrt{m}$. While decreasing i in the loop 5–6, we grow this substring to the left using the letters c returned by $\mathsf{parent\_edge}(v)$. Once $i' - i$ becomes larger than $2\sqrt{m}$, we simply decrement i'. Once i becomes larger than i' and $i' < i_{max}$, we must somehow restore the letters $T[i], T[i+1], \ldots$ Denote by P the set of all positions $j \in [0..i_{max}]$ such that j is a multiple of $\lceil\sqrt{m}\rceil$. For each $j \in P \cup \{i_{max}\}$, we store a vertex v_j that was the vertex v in the function $auto$ when we reached the position j for the first time (so that $\mathsf{str}(v_j)$ is a suffix of $T[0..j-1]$). Since the loop 5–6 cannot make i smaller than $i - |\mathsf{str}(v)|$, one can easily show that we always have $i \geq j - |\mathsf{str}(v_j)|$ for each $j \in P \cup \{i_{max}\}$. Thus, once $i > i'$, we compute in $O(1)$ time the position $j = \min\{\{i_{max}\} \cup \{j \in P \colon j \geq i + \sqrt{m}\}\}$, then put $i' = j - 1$, and restore the string $T[i..i']$ in $O(\sqrt{m})$ time iteratively applying the function $\mathsf{parent\_edge}$ to the vertex v_j.

Since $i_{max} - i$ cannot exceed m, it suffices to store v_j only for the $\lceil\sqrt{m}\rceil + 1$ largest positions from P. One can maintain these v_j in a straightforward way using a deque on circular array of length $O(\sqrt{m})$, so that access to arbitrary v_j takes $O(1)$ time. Therefore, the additional space used is $O(\sqrt{m} \log m) = o(m)$ bits. By standard arguments, one can show that the time $O(\sqrt{m})$ required to restore $T[i..j-1]$ is amortized among at least $\sqrt{m}$ increments of i that were performed to make $i > i'$. Thus, the total running time is $O(t|T|)$ as in the version that stores T explicitly. ◀

To perform $\mathsf{failure}(v)$, for $v \in W$, and to check whether $v \in W$, we use the next lemma.

▶ **Lemma 8.** *Let W be a subset of vertices of a rooted tree $\mathcal{F}$ with m edges. There is an encoding of $\mathcal{F}$ that occupies $2|W| \log \frac{m+1}{|W|} + O(|W|) + o(m)$ bits and, for any vertex v, allows to determine whether $v \in W$ in $O(1)$ time and to compute the parent of v if $v \in W$ in $O(1)$ time, provided all the vertices are represented by their numbers in a depth first traversal of $\mathcal{F}$.*

Proof. Let $\mathsf{num}(v)$ be a vertex numbering that corresponds to a depth first traversal of $\mathcal{F}$ (the numbers are from the range $[0..m]$). To check whether $v \in W$, we create a bit array A of length $m + 1$ such that, for each vertex v, we have $A[\mathsf{num}(v)] = 1$ iff $v \in W$. The array A is encoded in $\log \binom{m+1}{|W|} + o(|W|) = |W| \log \frac{m+1}{|W|} + O(|W|)$ bits in the data structure of Lemma 2 that supports access to $A[i]$ in $O(1)$ time using partial rank queries.

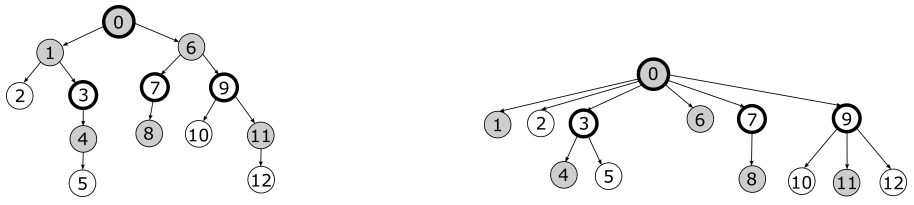


Figure 2 The transformation of Lemma 8 for the tree of failure links from Figure 1: the left tree is original, the right tree is transformed; vertices from W are gray, important vertices are bold.

Let us now describe a transformation of the tree $\mathcal{F}$ that preserves the function $\mathsf{parent}(v)$ for $v \in W$ (see Figure 2). We call a vertex v *important* if either v is the root or v has a child from W. In the transformed tree, for each vertex u, the parent of u is the nearest

important ancestor of u in $\mathcal{F}$. Obviously, there are at most $|W|+1$ internal vertices in thus defined tree. Further, it is easy to see that the vertex numbering num corresponds to a depth first traversal of the transformed tree. Therefore, we can encode the new tree in $|W|\log\frac{m+1}{|W|}+O(|W|)+o(m)$ bits using the data structure of Lemma 4; for $v\in W$, the query $\mathsf{parent}(v)$ on this structure returns the number num of the parent of v in the tree $\mathcal{F}$. ◀

Combining Lemmas 7 and 8, we prove the main theorem.

▶ **Theorem 9.** *Let $\varepsilon\in(0,2)$ be an arbitrary constant and let $\mathcal{D}$ be a set of d patterns over the alphabet $[0..\sigma)$ such that $\sigma\le m^{\delta}$, for some constant $\delta<1$, where m is the number of edges in the trie $\mathcal{T}$ containing $\mathcal{D}$. Then, there is a data structure that allows to find all occ occurrences of the patterns in any text T in $O(|T|+occ)$ time and occupies $mH_k(\mathcal{T})+1.443m+\varepsilon m+O(d\log\frac{m}{d})$ bits simultaneously for all $k\in[0..\max\{0,\alpha\log_\sigma m-2\}]$, where $\alpha\in(0,1)$ is an arbitrary constant.*

Proof. Let us construct a small t-dense subset for the tree $\mathcal{T}$. Observe that the set W_i that consists of the root and all vertices of $\mathcal{T}$ with height $h\equiv i \pmod t$ is a t-dense subset. Obviously, there exists $j\in[0..t)$ such that $|W_j|\le\lceil\frac{m+1}{t}\rceil$. We apply Lemma 8 to the tree $\mathcal{F}$ of failure links and the subset W_j, provided all vertices are represented by the numbering num defined in Section 3 (in [3] it was shown that this numbering corresponds to a depth first traversal of $\mathcal{F}$). By Lemma 7, this allows us to solve the dictionary matching problem in $O(t|T|+occ)$ time with only $2\lceil\frac{m+1}{t}\rceil\log t+O(\frac{m}{t})+o(m)$ bits used for the failure links, which is upper bounded by $m\frac{c\log t}{t}$, for an appropriate constant $c>0$. We add to the failure links the data structures for mark, next, and report described in Sections 3 and 4 (Lemma 6), which consume $mH_k(\mathcal{T})+1.443m+o(m)+O(d\log\frac{m}{d})$ bits (simultaneously for all $k\in[0..\max\{0,\alpha\log_\sigma m-2\}]$), and choose t in such a way that $\frac{c\log t}{t}\le\varepsilon/2$, so that the failure links take only $\varepsilon m/2$ bits. Solving the equation, we obtain $t=\Theta(\varepsilon^{-1}\log\varepsilon^{-1})$. Since ε is constant, the additive terms $o(m)$ in space can be upper bounded by $\varepsilon m/2$ and t in $O(t|T|+occ)$ can be hidden under the big-O, so that the total space is $mH_k(\mathcal{T})+1.443m+\varepsilon m+O(d\log\frac{m}{d})$ bits and the processing time is $O(|T|+occ)$. ◀

Note that, as it follows from the proof, the big-O notation in Theorem 9 hides a slowdown of the processing time to $O(|T|\varepsilon^{-1}\log\varepsilon^{-1}+occ)$, which is the price of the space improvement.

Let us now show that our solution is, in a sense, close to space optimal when $d=o(m)$. It is easy to see that any data structure solving the multiple pattern matching problem implicitly encodes the trie $\mathcal{T}$ containing the dictionary $\mathcal{D}$ of patterns: if the data structure is a black box, then we can enumerate all possible strings of length $\le m$ over the alphabet $[0..\sigma)$ and check which of them are recognized by the black box, thus finding all the patterns from $\mathcal{D}$. It is known that the number of tries with m edges over an alphabet of size σ is at least $\frac{1}{\sigma(m+1)+1}\binom{\sigma(m+1)}{m+1}$ (see [19, eq. 7.66] and [6, Thm 2.4]). Note that $\binom{\sigma(m+1)}{m+1}\ge\binom{\sigma(m+1)}{m}$, for $\sigma\ge2$. Therefore, $\log(\frac{1}{\sigma(m+1)+1}\binom{\sigma(m+1)}{m+1})\ge\log\binom{\sigma(m+1)}{m}-O(\log(\sigma m))$ is a lower bound for the worst-case space consumption of any solution for the multiple pattern matching.

▶ **Theorem 10.** *For any constant $\varepsilon\in(0,2)$ and any set $\mathcal{D}$ of d patterns over the alphabet $[0..\sigma)$ such that $d=o(m)$, where m is the number of edges in the trie containing $\mathcal{D}$, there is a data structure that allows to find all occ occurrences of the patterns in text T in $O(|T|+occ)$ time and occupies $L+\varepsilon m$ bits of space, where $L=\log\binom{\sigma(m+1)}{m}-O(\log(\sigma m))$ is a lower bound on the worst-case space consumption for any such data structure.*

Proof. Since $d = o(m)$, we have $d = \frac{m}{f(m)}$, where $f(m) \xrightarrow{m\to\infty} +\infty$, and therefore $d \log \frac{m}{d} = \frac{m}{f(m)} \log f(m) = o(m)$. Thus, applying Theorem 9 for $\frac{\varepsilon}{2}$, we obtain a data structure occupying $mH_k + 1.443m + \frac{\varepsilon}{2}m + o(m)$ bits. Further, applying the simple encoding from Lemma 2 for the bit array B representing the transitions next, we obtain a solution occupying $\log \binom{\sigma(m+1)}{m} + \frac{\varepsilon}{2}m + o(m)$ bits. Since ε is constant, we have $o(m) \le \frac{\varepsilon}{2}m$; hence, the result follows. ◀

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. Jarno Alanko and Tuukka Norri. Greedy shortest common superstring approximation in compact space. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 1–13. Springer, 2017. doi:10.1007/978-3-319-67428-5_1.
3. Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Proc. CPM*, volume 6129 of *LNCS*, pages 88–100. Springer, 2010. doi:10.1007/978-3-642-13509-5_9.
4. Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
5. Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Proc. SODA*, pages 13–22. SIAM, 2005.
6. David Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.
7. Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In *Proc. ESA*, volume 9294 of *LNCS*, pages 361–372. Springer, 2015. doi:10.1007/978-3-662-48350-3_31.
8. Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific Publishing Co. Pte. Ltd., 2002.
9. Vassilis Dimopoulos, Ioannis Papaefstathiou, and Dionisios Pnevmatikatos. A memory-efficient reconfigurable Aho–Corasick FSM implementation for intrusion detection systems. In *Proc. IC-SAMOS*, pages 186–193. IEEE, 2007. doi:10.1109/ICSAMOS.2007.4285750.
10. Guy Feigenblat, Ely Porat, and Ariel Shiftan. Linear time succinct indexable dictionary construction with applications. In *Proc. DCC, 2016*, pages 13–22. IEEE, 2016. doi:10.1109/DCC.2016.70.
11. Guy Feigenblat, Ely Porat, and Ariel Shiftan. A grouping approach for succinct dynamic dictionary matching. *Algorithmica*, 77(1):134–150, 2017. doi:10.1007/s00453-015-0056-0.
12. Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005. doi:10.1145/1082036.1082043.
13. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS*, pages 184–193. IEEE, 2005. doi:10.1109/SFCS.2005.69.
14. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398. IEEE, 2000. doi:10.1109/SFCS.2000.892127.
15. Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006. doi:10.1016/j.ipl.2006.04.008.
16. Simon Gog. SDSL - succinct data structure library, 2015. URL: https://github.com/simongog/sdsl-lite.
17. Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Faster, minuter. In *Proc. DCC*, pages 53–62. IEEE, 2016. doi:10.1109/DCC.2016.94.
18. Shay Golan and Ely Porat. Real-time streaming multi-pattern search for constant alphabet. In *Proc. ESA*, volume 87 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.41.
19. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Massachusetts: Addison-Wesley, 1989.

20 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997. doi:10.1017/CBO9780511574931.

21 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey S. Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, 475:113–119, 2013. doi:10.1016/j.tcs.2012.10.050.

22 Wing-Kai Hon, Tak-Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey S. Vitter. Compressed index for dictionary matching. In *Proc. DCC*, pages 23–32. IEEE, 2008. doi:10.1109/DCC.2008.62.

23 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *Journal of Computer and System Sciences*, 78(2):619–631, 2012. doi:10.1016/j.jcss.2011.09.002.

24 Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. SPIRE*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011. doi:10.1007/978-3-642-24583-1_18.

25 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In *Proc. CPM*, volume 54 of *LIPIcs*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.6.

26 S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999. doi:10.1137/S0097539797331105.

27 Hung-Jen Liao, Chun-Hung R. Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013. doi:10.1016/j.jnca.2012.09.004.

28 Veli Mäkinen and Gonzalo Navarro. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

29 Giovanni Manzini. An analysis of the Burrows–Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.

30 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016. doi:10.1017/CBO9781316588284.

31 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16, 2014. doi:10.1145/2601073.

32 Derek Pao, Xing Wang, Xiaoran Wang, Cong Cao, and Yuesheng Zhu. String searching engine for virus scanning. *IEEE Transactions on Computers*, 60(11):1596–1609, 2011. doi:10.1109/TC.2010.250.

33 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

34 Dina Sokol and Marcus Shoshana. Engineering small space dictionary matching. *arXiv preprint*, 2013. arXiv:1301.6428.

35 Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *Proc. SPIRE*, pages 39–50. Springer, 2009. doi:10.1007/978-3-642-03784-9_5.

36 Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992. doi:10.1145/135239.135244.

A Implementation and Experiments

We implemented the data structure described in this paper in C++ and compared its runtime and memory consumption with the Belazzougui's solution [3] and a naive algorithm. We could not find implementations of the Belazzougui's data structure and implemented it too. In [34] we found a different data structure based on compressed suffix trees but it showed a very poor performance, so we decided to exclude it from the tests.

The experiments were performed on a machine equipped with six 1.8 GHz Intel Xeon E5-2650L v3 CPUs with 30 MiB L3 cache and 16 GiB of RAM. The OS was Ubuntu 16.04.3 LTS, 64bit running kernel 4.4.0. All programs were compiled using `g++` v5.4.0 with `-O3 -march=x86-64` options. The source codes of all tested algorithms are available at `https://bitbucket.org/umqra/multiple-pattern-matching`. At the same URL one can find the 6 texts and 7 dictionaries on which the experiments were run.

The texts are as follows (see also Table 1):

- `program`: a sample binary file generated by a simple algorithm (see the URL above);
- `chr1.dna`: Human first chromosome genome sequence in FASTA format[1];
- `wiki.en/wiki.ja/wiki.ru/wiki.zh`: the first 200Mb of the dump of all English/Japanese/Russian/Chinese wikipedia articles[2].

The dictionaries are as follows (see also Table 1):

- `dna.dict`: reads generated for `chr1.dna` by `wgsim` simulator[3] with read length ~100;
- `urls.dict`: URLs from the .su/.nu zones[4] and the Alexa database of popular URLs[5];
- `virus.dict`: virus signatures from the `main.cvd` file of the ClamAV database[6];
- `ttl.en.dict/ttl.ja.dict/ttl.ru.dict/ttl.zh.dict`: the titles (in lower case) of some English/Japanese/Russian/Chinese wikipedia articles with length at least 3 letters[7] (the dictionaries were truncated to reduce temporary space used in the index construction).

Our implementations use the `SDSL` library by Gog [16]. We tested the following algorithms:

- `blz`: the original Belazzougui's compressed data structure [3] in which the bitvectors $\mathsf{B}_0, \ldots, \mathsf{B}_{\sigma-1}$ were implemented using `sd_vector` from the `SDSL`;
- `cblz`: our algorithm with fixed block compression boosting;
- `cblz8`: the same as `cblz` but with failure links sparsified using an 8-dense vertex subset;
- `smp`: a simple $O(nm)$-time algorithm that uses only two components of the `blz` data structure: the bitvectors $\mathsf{B}_0, \ldots, \mathsf{B}_{\sigma-1}$ and an array of length $m+1$ encoding the mark flags; they all are implemented using `sd_vector` from the `SDSL`.

For each algorithm, we ran 7 tests: in each test we search the patterns from a dictionary of Table 1 in the corresponding text from the same table row (note that `wiki.en` is used in two dictionaries). The results are present in Figure 3. The Aho–Corasick data structure required too much memory in our experiments, so we do not include it.

All dictionaries can be split into two unequal groups: dictionaries with long patterns (in our case it is only `dna.dict`) and short patterns (all other dictionaries). On `dna.dict` the fastest algorithms are `blz` and `cblz`; both `cblz8` and `smp` are about two times slower than them. Not surprisingly, `smp` is the fastest algorithm on short patterns and the algorithms `blz`, `cblz`, and `cblz8` are 2–3 times slower than `smp`. Because of some implementation details (we use a specially tailored version of `sd_vector`), `cblz` is faster than the simpler algorithm `blz` in our tests. We were unable to explain why `cblz8` often works faster than the simpler `cblz` algorithm. To sum up, the three `blz` algorithms have acceptable running times, but it really makes sense to use them only on relatively long dictionary patterns.

[1] `ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/seq/hs_alt_CHM1_1.1_chr1.fa.gz`
[2] `https://dumps.wikimedia.org/`
[3] `https://github.com/lh3/wgsim`
[4] `https://zonedata.iis.se/`
[5] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`
[6] `https://www.clamav.net/downloads`
[7] `https://dumps.wikimedia.org/`

Table 1 Statistics of the test dictionaries of patterns (number of patterns, average pattern length, m, σ) and texts in which the patterns were searched (length, number of found occurrences).

Dictionary	patterns	avg. len.	m	σ	Text	length	found occ.
dna.dict	1,999,911	100.0	178,323,409	5	chr1.dna	228,503,292	157,393
virus.dict	4,059,198	16.0	56,430,521	256	program	104,857,600	1,347,031
urls.dict	3,825,132	16.7	39,385,319	40	wiki.en	209,040,222	3,461,097
ttl.en.dict	3,875,263	18.3	32,255,913	2881	wiki.en	209,040,222	137,071,073
ttl.ja.dict	1,691,693	8.4	7,182,594	7329	wiki.ja	109,018,155	15,196,791
ttl.ru.dict	4,145,276	22.5	39,908,335	1350	wiki.ru	131,898,987	65,428,137
ttl.zh.dict	1,649,209	7.3	6,164,034	12113	wiki.zh	126,021,996	18,925,337

The central chart shows that in most cases the dictionaries are very well compressed (in comparison with $\frac{\log\sigma}{8}$ bytes per letter; see the shaded columns). But, as one can conclude from the low compression ratio of the corresponding trie on the right chart, the dictionary compression is mostly due to the assembling of the patterns in the trie (for instance, while the total length of the patterns $\{a^i b\}_{i=0}^{k-1}$ is $k(k+1)/2$, their trie contains only $m = 2k$ edges). In the right chart, one can observe that the effect of compression boosting in `cblz` is hidden under the overhead imposed by additional structures, and, because of this, `cblz` occupies about the same space as `blz`. The space optimized version `cblz8` lowers the overhead and the resulting data structure occupies about the same space as `smp`; this, surely, is possible only because `cblz8` is H_k-compressed while `smp` is only H_0-compressed.

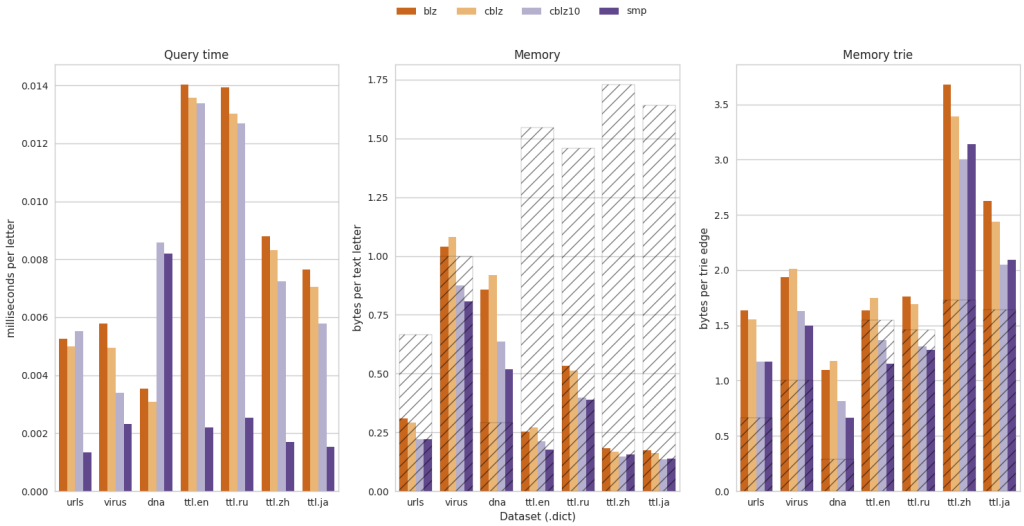


Figure 3 Performance and space consumption; shaded columns correspond to $\frac{\log\sigma}{8}$.

Hamming Distance Completeness

Karim Labib
Google Zürich, Switzerland[1]
karimlabib@google.com

Przemysław Uznański
Institute of Computer Science, University of Wrocław, Poland[1]
puznanski@cs.uni.wroc.pl

Daniel Wolleb-Graf
Department of Computer Science, ETH Zürich, Switzerland
daniel.graf@inf.ethz.ch

Abstract

We show, given a binary integer function $\diamond$ that is piecewise polynomial, that $(+, \diamond)$ vector products are equivalent under one-to-polylog reductions to the computation of the Hamming distance. Examples include the dominance and ℓ_{2p+1} distances for constant p. Our results imply equivalence (up to polylog factors) between the complexity of computing All Pairs Hamming Distance, All Pairs ℓ_{2p+1} Distance and Dominance Matrix Product, and equivalence between Hamming Distance Pattern Matching, ℓ_{2p+1} Pattern Matching and Less-Than Pattern Matching. The resulting algorithms for ℓ_{2p+1} Pattern Matching and All Pairs ℓ_{2p+1}, for $2p+1 = 3, 5, 7, \ldots$ are likely to be optimal, given lack of progress in improving upper bounds for Hamming distance in the past 30 years. While reductions between selected pairs of products were presented in the past, our work is the first to generalize them to a general class of functions, showing that a wide class of "intermediate" complexity problems are in fact equivalent.

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

Keywords and phrases fine grained complexity, approximate pattern matching, matrix products

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.14

Related Version A full version of the paper is available at [15], `https://arxiv.org/abs/1711.03887`.

1 Introduction

In the last few decades, many classical algorithmic problems received new attention when formulated as algebraic problems. In pattern matching, instead of looking for occurrences of a pattern as a substring of a text, we can define a similarity score between two strings and ask for this score between the pattern $\mathbf{P}$ of length m and every m-substring of the text $\mathbf{T}$ of length $n \geq m$. For example, scores of Hamming distance or L_1 distance between numerical strings generalize the classical pattern matching: the total score for a given alignment is zero iff the pattern occurs exactly there in the text. One can go in this framework even further, and consider a function that is not a metric, e.g. LessThanPatternMatching which outputs the number of coordinates for which the pattern element is no larger than the corresponding text element. However, all those problems share a common additive structure, where for an input pattern $\mathbf{P}$ and text $\mathbf{T}$, the score vector $\mathbf{O}$ is such that $\mathbf{O}[i] = \sum_j \mathbf{P}[j] \diamond \mathbf{T}[i+j]$ for some binary function $\diamond$.

Just as those pattern matching generalizations are based on *convolution*, there is a family of problems based on *matrix multiplication*, varying in flavour according to the vector product used. There, we are given two matrices A and B representing n vectors of dimension

[1] Most of the work was done while the authors were affiliated with ETH Zürich.

Table 1 Summary of different score functions and the corresponding problems. $\mathbf{1}[\varphi]$ is 1 iff φ and 0 otherwise.

Name	Score function	Pattern Matching problem	Matrix Product problem
Hamming	$\mathbf{1}[x \neq y]$	$\mathbf{O}[i] = \lvert\{j : \mathbf{P}[j] \neq \mathbf{T}[i+j]\}\rvert$	$O[i][j] = \lvert\{k : \mathbf{A}_i[k] \neq \mathbf{B}_j[k]\}\rvert$
Dominance	$\mathbf{1}[x \leq y]$	$\mathbf{O}[i] = \lvert\{j : \mathbf{P}[j] \leq \mathbf{T}[i+j]\}\rvert$	$O[i][j] = \lvert\{k : \mathbf{A}_i[k] \leq \mathbf{B}_j[k]\}\rvert$
Threshold	$\mathbf{1}[\lvert x-y\rvert \geq \delta]$	$\mathbf{O}[i] = \lvert\{j : \lvert\mathbf{P}[j] - \mathbf{T}[i+j]\rvert > \delta\}\rvert$	$O[i][j] = \lvert\{k : \lvert\mathbf{A}_i[k] - \mathbf{B}_j[k]\rvert > \delta\}\rvert$
ℓ_1 distance	$\lvert x-y\rvert$	$\mathbf{O}[i] = \sum_j \lvert\mathbf{P}[j] - \mathbf{T}[i+j]\rvert$	$O[i][j] = \sum_{k=1}^{n} \lvert\mathbf{A}_i[k] - \mathbf{B}_j[k]\rvert$
ℓ_2 distance	$(x-y)^2$	$\mathbf{O}[i] = \sum_j (\mathbf{P}[j] - \mathbf{T}[i+j])^2$	$O[i][j] = \sum_{k=1}^{n} (\mathbf{A}_i[k] - \mathbf{B}_j[k])^2$

d, and the output is the matrix $O[i][j] = \sum_k A[i][k] \diamond B[k][j]$. This is equivalent to the computation of all pairwise $(+, \diamond)$-vector products for two vector families, the so called MatrixProduct problems.

In both of those worlds, the complexity is spanned between *easy* and *hard* cases. An easy case is observed for e.g. $(+, \times)$ products, which have an upper bound of $\mathcal{O}(n \log n)$ for convolution (the classical Discrete Fast Fourier Transform algorithm) or $\mathcal{O}(n^\omega)$ (where $\omega < 2.373$ c.f. Le Gall [20]) for matrix multiplication. A hard case is considered to be respectively either near quadratic time or near cubic time problems. In the world of $(+, \diamond)$ vector products, we have not observed problems of the hard type, and instead they are either easy, or admit some intermediate complexity.[†] For many pattern matching generalizations there are independently achieved algorithms of identical complexity $\mathcal{O}(n\sqrt{m \log m})$. Similarly, for many MatrixProduct problems, the best algorithms are of complexity $\mathcal{O}(n^{(\omega+3)/2})$ or similar. Why are so many different problems of essentially the same complexity?

Our contribution

We show that there is a shared source of hardness to those problems. That is, we show that for a wide class of $(+, \diamond)$ products, the corresponding problems are of (almost) equivalent hardness. This class includes not only products like Hamming distance or Dominance, but in fact any piecewise polynomial function of two variables (for appropriate definition of piecewise polynomiality, c.f. Definition 3) excluding certain degenerate forms (e.g. polynomials). Thus, we show that we should not expect the problems based on $(+, \diamond)$ products to be significantly harder to compute than e.g. ones based on Hamming distance (given reasonable restrictions on $\diamond$). The reduction applies both to the Pattern Matching setting and to the Matrix Product setting alike. We refer to Table 1 for a summary of considered problems, to Table 2 for a summary of existing upper bounds, and to Figure 1 for a summary of the old and new reductions.

Yuster [32] improved the exponent of DominanceMatrixProduct (when $d = n$) from $(3+\omega)/2 \leq 2.6865$ to $\rho \leq 2.6834$, where ρ satisfies $\rho = \omega(1, 4-\rho, 1)$, and $\omega(a, b, c)$ is the exponent of fast multiplication of rectangular matrices $n^a \times n^b$ with $n^b \times n^c$. By our reduction, this improvement applies to all other MatrixProduct- problems considered here. Similarly, the tradeoff achieved for one problem (e.g. HammingDistanceMatrixProduct) between vectors of dimension d and the exponent (c.f. [25] and [14]) applies by our results to all the other MatrixProduct problems considered here. Looking at the sparsity of the input, the tradeoff between the number of relevant entries in the input and the runtime (c.f. Vassilevska [26], Vassilevska et. al. [28] and Duan and Pettie [10]) applies to all of the mentioned problems. (See Section 2 for precise upper bounds.)

[†] To observe good candidates for *hard* problems, we have to go beyond $(+, \diamond)$ products, and consider either $(\min, +)$ convolution (c.f. [9, 19]) or $(\min, +)$ matrix product (c.f. [30]).

Table 2 Overview of the known results and of how we abbreviate the corresponding problem names.

Name	Pattern Matching problem			Matrix Product problem		
Hamming	HAMPM	$\mathcal{O}(n\sqrt{m \log m})$	[1]	HAMPROD	$\mathcal{O}(n^{(\omega+3)/2})$	[25]
Dominance	LESSTHANPM	$\mathcal{O}(n\sqrt{m \log m})$	[2]	DOMPROD	$\mathcal{O}(n^{\rho})$	[32]
Threshold	THRPM	$\mathcal{O}(n\sqrt{m \log m})$	[5]	THRPROD	$\mathcal{O}(n^{(\omega+3)/2})$	[17]
ℓ_1 distance	L_1PM	$\mathcal{O}(n\sqrt{m \log m})$	[8, 3]	L_1PROD	$\mathcal{O}(n^{(\omega+3)/2})$	[17]
ℓ_2 distance	L_2PM	$\mathcal{O}(n \log m)$	[23]	L_2PROD	$\mathcal{O}(n^{\omega})$	[17]

We thus observe that there is a shared barrier in a broad class of problems and one is unlikely to improve upon existing upper bounds without some significant breakthrough. For both pattern matching problems and geometric problems we consider here, existing runtimes come from a tradeoff between the number of buckets and the size of these buckets. Without a novel technique, this runtime is unlikely to be improved. Similarly, any lower bound proof for one of the listed problems would immediately apply to every other problem.

Further applications

We provide the following further applications of our reductions.

- Since our reductions preserve structural properties of inputs like *size of Run Length Encoding*, in [13] they were used to establish equivalence between running time of HAMMINGDISTANCEPATTERNMATCHING and L_1PATTERNMATCHING on instances with bounded Run Length Encoding.
- In Censor-Hillel et al. [6] authors analyze the complexity of sparse matrix multiplication under the restricted bandwidth all-to-all communication model (the so called CONGESTED CLIQUE model). We note that their analysis immediately implies bounds to computation of Hamming Distance Matrix Product (and thus other matrix products as well) by presented, in the full version of this paper [15], bi-directional reductions to and from sparse matrix multiplication.
- Consider the problem of Image Template Matching, where one is given as an input a two-dimensional text T and a pattern P of dimensions $n \times n$ and $m \times m$ respectively. The goal is to compute the dissimilarity score between P and all $m \times m$-subsquares of T. Atallah in [4] gives an $\widetilde{\mathcal{O}}(n^2 m)$ running time algorithm for the L_1 version of this problem (so called *Sum of the Absolute Value* difference measure). We note, that by our reductions, an equivalence between the L_1, Hamming, Dominance and Threshold versions of this problem is established.

2 Related work

We now list different pattern matching problems that differ in their underlying score functions. The **HammingDistancePatternMatching**‡ was studied by Abrahamson [1], **LessThanPatternMatching** was introduced by Amir and Farach [2], **L_1PatternMatching** was studied first by Lipsky [21] and later in conference publications by Clifford, Clifford and Iliopoulous [8] and Amir, Lipsky, Porat and Umanski [3] (although the 2-dimensional version

‡ Also known as Pattern Matching with Mismatches.

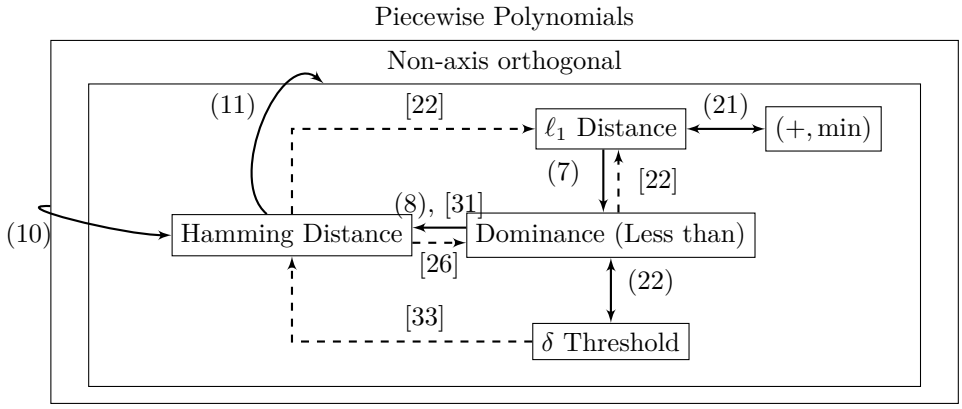


Figure 1 Existing (dashed line) and new (solid line) reductions between problems, together with problem classes.

of this problem was studied earlier by Atallah [4]) and **ThresholdPatternMatching** was studied by Atallah and Duket [5]. For all those problems, the currently best known algorithms run in time $\mathcal{O}(n\sqrt{m \log m})$ using similar techniques: high/low frequency, bucketing and convolution.

For $\mathbf{L_2}$**PatternMatching**, since $\mathbf{O}[i] = \sum_j \mathbf{P}[j]^2 + \sum_j \mathbf{T}[i+j]^2 - 2\sum_j \mathbf{P}[j]\mathbf{T}[i+j]$, the dominating term in the computation arises from computing a single convolution in time $\mathcal{O}(n \log m)$ via Fast Fourier Transform (FFT) as observed by Lipsky and Porat [23]. This approach extends to L_{2p}PM, with running time $\mathcal{O}(p^2 n \log m)$.

On the side of reductions, only little was known. Lipsky and Porat [22] showed that both HamPM and LessThanPM reduce to L_1PM showing that the latter problem is no easier than the former problems. The question of whether e.g. HamPM could be substantially easier than L_1PM remained open. The first non-trivial reduction (although not stated as a lower-bound type result) was provided by Zhang and Atallah [33], where they showed that ThrPM with threshold δ reduces to $\mathcal{O}(\log \delta)$ instances of HamPM (see Figure 1).

In computational geometry, a classical problem is to process a set of n points given in d-dimensional space. One can consider e.g. the metric space and ask for a pair of closest or farthest points. Some of those problems in low-dimensions (i.e. $d = \operatorname{poly} \log n$) exhibit a structure that allows for solutions almost linear in n for some metrics (see Williams [29]). However, in high-dimensional data, the situation is usually dire, as the so called *curse of dimensionality* kicks in (c.f. [18] and [16]) and for processing such spaces usually the fastest known approach is to compute all pairwise distances [17].

Those problems come in two flavours, AllPairs- where the input is a single matrix A and the goal is to compute all corresponding pairwise vector products between the rows of A, and MatrixProduct, where the input are two matrices A and B, and the goal is to compute products between rows of A and columns of B. Those two formulations are in fact equivalent by the folklore reduction.

DominanceMatrixProduct was introduced by Matoušek [24], where he provided a solution working in time $\mathcal{O}(n^{(\omega+3)/2}) \subseteq \mathcal{O}(n^{2.687})$. Vassilevska [26] and Vassilevska et al. [28] considered the dominance product on sparse inputs where we denote by m_1 and m_2 the number of entries in A and B, respectively that contribute to the score. They obtain a bound

of $\mathcal{O}(\min(n^\omega + \sqrt{m_1 m_2} \cdot n^{\frac{\omega-1}{2}}, n^2 + (m_1 m_2)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{2-\alpha\omega}{\omega-\alpha-1}}))$.[§] Duan and Pettie [10] simplified this analysis. For $d \ll n$, Vassilevska and Williams [27] and [26] gave an algorithm with a time bound of $\mathcal{O}(n^{\frac{2\omega-\omega\alpha-2}{\omega-\alpha-1}} d^{\frac{2\omega-4}{\omega-\alpha-1}} + n^{2+o(1)})$. Yuster [32] improved the bound of the case $d = n$ to $\mathcal{O}(n^\rho)$, where ρ is a solution to $\rho = \omega(1, 4-\rho, 1)$. The bound $\rho \leq 2.6834$ is provided. Recently, Gold and Sharir [14] presented an updated analysis of the time vs. dimension tradeoff using newer bounds on rectangular matrix multiplication. For $d = n$, this gives a running time of $\mathcal{O}(n^{2.6598})$.

AllPairsL$_1$Distance was considered by Indyk et al. [17], with an $\mathcal{O}(n^{(\omega+3)/2})$ algorithm for the case when $d = n$. Although not stated as such, one algorithm presented in [17] can be adapted to computing **ThresholdMatrixProducts** in time $\mathcal{O}(n^{(3+\omega)/2})$. [26] introduced **(+, max)-MatrixProduct** where the score matrix is $O[i][j] = \sum_{k=1}^{n} \max(\mathbf{A}_i[k], \mathbf{B}_j[k])$ and presented a bucketing solution with running time $\mathcal{O}(n^{(\omega+3)/2})$. The algorithm follows in spirit the L$_1$Prod algorithm from [17] with a tweaked score contribution. **ClosestHammingPair** was considered by Min et al. [25], where as an intermediate step the **AllPairsHamming-Distance** is computed (without actually naming the problem). Inspired by the reduction from Hamming distance to L_1 in [22], they utilized the L$_1$Prod algorithm from [17]. This resulted in a $\mathcal{O}(n^{(\omega+3)/2})$ time algorithm when $d = n$. They also utilized rectangular matrix multiplication bounds to provide a tradeoff in the complexity when $d \ll n$. Writing their upper bound in a general form, the complexity is $\mathcal{O}(n^{1+\omega(1,s,1)/2}\sqrt{d})$ where $d = n^s$. Given the improved bounds for rectangular matrix multiplication by Le Gall [11] and subsequently by Le Gall and Urrutia [12], the bounds from [25] are stronger. **AllPairsL$_2$Distance** as observed by Indyk et al. [17] reduces to a single matrix multiplication and thus admits a running time of $\mathcal{O}(n^\omega)$. Similarly, L$_{2p}$Prod admits a running time of $\mathcal{O}(p^2 n^\omega)$.

We observe that L_2 is an "easy" score function. For every other score function mentioned, all solutions presented use a bucketing or a high/low frequency technique to decompose the problem into ones solvable by convolution (for Pattern Matching problems) or matrix multiplication (for All Pairs problems). We refer to Tables 1 and 2 for a summary. There are several related problems that use the aforementioned problems as subroutines. **Weighted Pattern Matching** in the most general setting asks for $\mathbf{O}[i] = \sum_j w(P[j], T[i+j])$ for some weight function w. In [23] Lipsky and Porat presented a simple $\mathcal{O}(|\Sigma| n \log m)$ algorithm. **Pattern Matching with Wildcards** admits a simple determinsitic $\mathcal{O}(n \log m)$ solution via weighted L$_2$PM, as shown by Clifford and Clifford [7]. **Closest L_∞ Pair** was considered by Indyk et al. [17] where the presented algorithm uses binary search on top of ThrProd (implicitly, the intermediate problem they solve is not named). The total running time is $\mathcal{O}(n^{(\omega+3)/2} \log D)$, where D is the diameter of the input point set.

3 Preliminaries

The problems discussed previously have at their core the computation of a $(+, \diamond)$ vector product, that is $\sum_i x_i \diamond y_i$ for some binary function $\diamond$. Formally, for vectors $\mathbf{A}, \mathbf{B}$ and matrices $\mathcal{A}, \mathcal{B}$, we denote the $(+, \diamond)$ vector product as $\textsc{vprod}(\diamond, \mathbf{A}, \mathbf{B}) \stackrel{\text{def}}{=} \sum_i \mathbf{A}[i] \diamond \mathbf{B}[i]$, the $(+, \diamond)$ convolution as $\textsc{conv}(\diamond, \mathbf{A}, \mathbf{B}) = \mathbf{C}$ such that $\mathbf{C}[k] = \sum_{i+j=k} \mathbf{A}[i] \diamond \mathbf{B}[j]$ and the $(+, \diamond)$ matrix product as $\textsc{mprod}(\diamond, \mathcal{A}, \mathcal{B}) = \mathcal{C}$ such that $\mathcal{C}[i, j] = \sum_k \mathcal{A}[i, k] \diamond \mathcal{B}[k, j]$.

We define $\mathbf{1}[\varphi]$ to be 1 iff φ and 0 otherwise. Since $\mathsf{Ham}(x, y) \stackrel{\text{def}}{=} \mathbf{1}[x \neq y]$, then $\textsc{vprod}(\mathsf{Ham}, \mathbf{X}, \mathbf{Y})$ is the Hamming Distance between $\mathbf{X}$ and $\mathbf{Y}$, HamPM is essentially $\textsc{conv}(\mathsf{Ham}, \mathbf{X}, \mathbf{Y}^R)$, and HamProd between vectors $\{X_1, \ldots, X_n\}$ and $\{Y_1, \ldots, Y_n\}$ is $\textsc{mprod}\left(\mathsf{Ham}, \begin{bmatrix} X_1 & \cdots & X_n \end{bmatrix}^T, \begin{bmatrix} Y_1 & \cdots & Y_n \end{bmatrix}\right)$.

[§] $\alpha = \sup\{0 \leq r \leq 1 : \omega(1, r, 1) = 2 + o(1)\} \geq 0.31389$.

We now shift our attention to the relations between the binary functions.

▶ **Definition 1.** *We say that $\diamond$ reduces preserving linearity to instances of $\square_1, \ldots, \square_K$ (for some positive integer K), if there are functions $f_1, \ldots, f_K$ and $g_1, \ldots, g_K$ and coefficients $\alpha_1, \ldots, \alpha_K$, such that for any x, y:*¶

$$x \diamond y = \sum_i \alpha_i \cdot \Big(f_i(x) \ \square_i \ g_i(y) \Big).$$

A one-to-many reduction from $\diamond$ to $\square$ is also a one-to-many reduction from $(+, \diamond)$ vector product/convolution/matrix multiplication to $(+, \square)$ vector product/convolution/matrix multiplication. Indeed, given Definition 1, we have for any vectors $\mathbf{A}, \mathbf{B}$ and matrices $\mathcal{A}, \mathcal{B}$: $\textsc{vprod}(\diamond, \mathbf{A}, \mathbf{B}) = \sum_i \alpha_i \cdot \textsc{vprod}(\square_i, f_i(\mathbf{A}), g_i(\mathbf{B}))$, $\textsc{conv}(\diamond, \mathbf{A}, \mathbf{B}) = \sum_i \alpha_i \cdot \textsc{conv}(\square_i, f_i(\mathbf{A}), g_i(\mathbf{B}))$ and $\textsc{mprod}(\diamond, \mathcal{A}, \mathcal{B}) = \sum_i \alpha_i \cdot \textsc{mprod}(\square_i, f_i(\mathcal{A}), g_i(\mathcal{B}))$, where $f(\mathbf{A})$ and $f(\mathcal{A})$ denotes a coordinate-wise application of f to vector $\mathbf{A}$ and matrix $\mathcal{A}$, respectively.

4 Main results

▶ Remark 2. We assume that all input values and coefficients are integers bounded in absolute value by M. All reductions presented use standard arithmetic operations and require thus $\operatorname{poly}\log M$ computation time.

▶ **Definition 3.** *For integers A, B, C and polynomial $P(x, y)$ we say that the function $P(x, y) \cdot \mathbf{1}[Ax + By + C > 0]$ is* halfplane polynomial. *We call a sum of halfplane polynomial functions a* piecewise polynomial. *We say that a function is* axis-orthogonal piecewise polynomial, *if it is piecewise polynomial and for every i, $A_i = 0$ or $B_i = 0$.*

Observe that $\mathsf{Ham}(x, y) = \mathbf{1}[x > y] + \mathbf{1}[x < y]$, $\max(x, y) = x \cdot \mathbf{1}[x \geq y] + y \cdot \mathbf{1}[x < y]$, $|x - y|^{2p+1} = (x - y)^{2p+1} \cdot \mathbf{1}[x > y] + (y - x)^{2p+1} \cdot \mathbf{1}[x < y]$, and $\mathsf{Thr}_\delta(x, y) \stackrel{\text{def}}{=} \mathbf{1}[|x - y| \geq \delta] = \mathbf{1}[x \leq y - \delta] + \mathbf{1}[x \geq y + \delta]$.

▶ **Theorem 4.** *Let $\diamond$ be a piecewise polynomial of constant degree and $\operatorname{poly}\log n$ number of summands.*

- *If $\diamond$ is axis orthogonal, then $\diamond$ is "easy": $(+, \diamond)$ convolution takes $\widetilde{O}(n)$ time, $(+, \diamond)$ matrix multiplication takes $\widetilde{O}(n^\omega)$ time.*
- *Otherwise, $\diamond$ is* Hamming distance complete*: under one-to-polylog reductions, on inputs bounded in absolute value by $\operatorname{poly}(n)$, $(+, \diamond)$ product is equivalent to Hamming distance, $(+, \diamond)$ convolution is equivalent to* HamPM *and $(+, \diamond)$ matrix multiplication is equivalent to* HamProd.

Theorem 4 "hard" case follows from two technical results presented in Section 6, Theorem 10 and Theorem 11. The "easy" case is resolved by Lemma 13.

▶ **Corollary 5.** *The following problems are equivalent under one-to-polylog reductions:* HamPM, LessThanPM, L_{2p+1}PM *for a constant integer $p \geq 0$,* ThrPM *and $(+, \max)$-*Convolution.

▶ **Corollary 6.** *The following problems are equivalent under one-to-polylog reductions:* HamProd, DomProd, L_{2p+1}Prod *for a constant integer $p \geq 0$,* ThrProd *and $(+, \max)$-*MatrixProduct.

¶ For the sake of simplicity, we are omitting in the definition the post-processing function necessary e.g. $(\,\cdot\,)^{1/p}$ for L_p norms.

5 Warm-up

We start by showing a reduction from ℓ_1 distance to $\mathcal{O}(\log^2 M)$ instances of Hamming distance, with an intermediate step of $\mathcal{O}(\log M)$ instances of dominance. Note that the reduction from dominance to $\mathcal{O}(\log M)$ Hamming distances follows from adapting reductions from [31] to our setting, and reduction from ℓ_1 to $\mathcal{O}(\log M)$ dominance relations follows as a natural adaptation of the same technique. However, since they serve as a nice overview of techniques used in our main result, and already have a nontrivial consequence (e.g. collapsing hardness of L_1PM and HamPM), we present them separately.

Scaling

Observe that for many "natural" functions $\diamond$ and integers x, y, $x \diamond y$ is approximated by $\lfloor x/2 \rfloor \diamond \lfloor y/2 \rfloor$ (up to some fixed multiplicative factor). This allows us to unwind $x \diamond y$ into a weighted sum of $\mathcal{O}(\log(\max(|x|, |y|))$ corrective terms. For example, if for some constant C, integers $x, y \geq 0$ and some corrective function ξ: $x \diamond y = C \cdot (\lfloor x/2 \rfloor \diamond \lfloor y/2 \rfloor) + \xi(x, y)$ then naturally $x \diamond y = 0 \diamond 0 + \sum_{i \geq 0} C^i \cdot \xi(\lfloor x/2^i \rfloor, \lfloor y/2^i \rfloor)$.

Sparsity

We consider a generalized version of the input with special "ignore" marks $\star$ as possible elements. Those elements of the input never contribute to the final score of the $(+, \diamond)$ product. Formally, we operate on $\mathbb{Z} + \{\star\}$, with special arithmetic rules (unless stated otherwise):

- for any *single argument* function: $f(\star) = \star$,
- for any *double argument* function: $g(\star, \star) = g(\star, y) = g(x, \star) = 0$.[‖]

The goal of this formalism is twofold. The first one is to handle sparse inputs formally (i.e. vectors with $\mathcal{O}(n^{1-\varepsilon})$ relevant entries). The second one is that such "ignore" marks coupled with filtering (defined below) allow us to split the input based on properties of its values. We note that these "ignore" marks do not increase the computational complexity of Hamming distance (see Lemma 9 in the Appendix).

Filtering

We define the following functions:

$$\texttt{even}(x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \text{ is even} \\ \star & \text{otherwise} \end{cases} \qquad \texttt{odd}(x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \text{ is odd} \\ \star & \text{otherwise} \end{cases}$$

Those functions, when applied to a vector or a matrix, allows us to filter values according to parity, e.g. for $\mathbf{A} = [1, 2, 3, 4]$ one gets $\texttt{even}(\mathbf{A}) = [\star, 2, \star, 4]$.

We now give two reductions that illustrate the usefulness of these techniques. Both reductions are illustrated in Appendix C (Figures 2 and 3). In the following theorems, recall that M is the largest possible integer input.

▶ **Theorem 7.** *The L_1 distance reduces to $\mathcal{O}(\log M)$ instances of dominance.*

[‖] We have to keep in mind that whether a function is a single or double argument is context dependent: e.g. writing: $\mathbf{1}[x \neq y] = 1 - \mathbf{1}[x = y]$, we have to treat 1 as a function of x and y as well.

Proof. Since L_1 distance is shift-invariant, i.e. $|(x+\Delta)-(y+\Delta)| = |x-y|$ for any Δ, we can assume that $0 \le x, y < M$ for some $M = \text{poly}(n)$. Observe that for $x, y \ge 0$, $|x-y| = 2 \cdot \Big|\lfloor x/2 \rfloor - \lfloor y/2 \rfloor\Big| + \eta(x,y)$, where, denoting $\mathsf{Dom}(x,y) \stackrel{\text{def}}{=} \mathbf{1}[x \le y]$,

$$\begin{aligned}\eta(x,y) &= \mathbf{1}[(x \text{ is odd}) \wedge (y \text{ is even}) \wedge (x \ge y)] - \mathbf{1}[(x \text{ is even}) \wedge (y \text{ is odd}) \wedge (x \ge y)] \\ &\quad + \mathbf{1}[(y \text{ is odd}) \wedge (x \text{ is even}) \wedge (y \ge x)] - \mathbf{1}[(y \text{ is even}) \wedge (x \text{ is odd}) \wedge (y \ge x)] \\ &= \mathsf{Dom}(\mathtt{odd}(-x), \mathtt{even}(-y)) - \mathsf{Dom}(\mathtt{even}(-x), \mathtt{odd}(-y)) \\ &\quad + \mathsf{Dom}(\mathtt{even}(x), \mathtt{odd}(y)) - \mathsf{Dom}(\mathtt{odd}(x), \mathtt{even}(y)).\end{aligned}$$

By unwinding, we get $|x-y| = \sum_{i=0}^{\log M} 2^i \cdot \eta(\lfloor x/2^i \rfloor, \lfloor y/2^i \rfloor)$ which completes the reduction. ◀

▶ **Theorem 8.** *Dominance reduces to $\mathcal{O}(\log M)$ instances of Hamming distance and multiplication.*

Proof. Since dominance is shift-invariant, w.l.o.g. we assume that $0 \le x, y < M$ for some $M = \text{poly}(n)$. Observe the following recurrence relation, for $x, y \ge 0$:

$$\begin{aligned}\mathsf{Dom}(x,y) &= \mathsf{Dom}(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor) - \mathbf{1}[(x \text{ is odd}) \wedge (x = y+1)] \\ &= \mathsf{Dom}(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor) - \mathbf{1}[x \text{ is odd}] + \mathbf{1}[x \text{ is odd}] \cdot \mathsf{Ham}(x, y+1)\end{aligned}$$

By unwinding, we get:

$$\mathsf{Dom}(x,y) = 1 - \sum_{i=0}^{\log M} \mathbf{1}\big[\lfloor x/2^i \rfloor \text{ is odd}\big] + \sum_{i=0}^{\log M} \mathbf{1}\big[\lfloor x/2^i \rfloor \text{ is odd}\big] \cdot \mathsf{Ham}(\lfloor x/2^i \rfloor, \lfloor y/2^i \rfloor + 1).$$

Using filtering notation, this becomes

$$\mathsf{Dom}(x,y) \quad = \quad \underbrace{1 - \sum_{i=0}^{\log M} \mathbf{1}\big[\lfloor x/2^i \rfloor \text{ is odd}\big]}_{(*)} + \underbrace{\sum_{i=0}^{\log M} \mathsf{Ham}(\mathtt{odd}(\lfloor x/2^i \rfloor), \lfloor y/2^i \rfloor + 1)}_{(**)}$$

Now observe, that $(*)$ is purely a function of x. If x is guaranteed to be an integer, then evaluating it as part of an operator (i.e. inside convolution or matrix-multiplication) is trivial. As y is never mapped to $\star$ in $(**)$, treating $(*)$ as a single argument function suffices.

The second term $(**)$ uses our filtering function and the convention that Ham evaluates to 0 if at least one of its inputs is $\star$. Thus $(**)$ is a sum of $\mathcal{O}(\log n)$ Hamming distances on inputs from $\mathbb{Z} \cup \{\star\}$. By Lemma 9, each of those reduces to two instances of Hamming distance on inputs from $\mathbb{Z}$. ◀

▶ **Lemma 9.** *Hamming distance in $\mathbb{N} + \{\star\}$ reduces preserving linearity to two instances of Hamming distance in $\mathbb{N}$.*

▶ Remark. In general, we have to take into account that both $x, y \in \mathbb{Z} \cup \{\star\}$. Thus, we have to treat term $(*)$ as a function of *both* x and y, that is evaluating to 0 if $x = \star$ or $y = \star$. In general, $(*)$ reduces to evaluating, after the reduction step, some polynomial $Q(x', y') = f(x')$ (where y' might be $\star$) with $f(x') = 1 - \sum_{i=0}^{\log M} \mathbf{1}\big[\lfloor x'/2^i \rfloor \text{ is odd}\big]$. By Lemma 13, $f(x')$ can be resolved in the time of a regular convolution or matrix multiplication and thus the computation time for $(*)$ is dominated by $(**)$, that is HamPM and HamProd, respectively.

6 Hamming distance completeness

The goal of this section is proving Theorem 4. We achieve this by showing two separate reductions, one from *all piecewise polynomial functions* to Hamming distance and one from Hamming distance to all non axis-orthogonal piecewise polynomials.

▶ **Theorem 10.** *If $\diamond$ is a piecewise polynomial of degree d with c summands then it reduces to $\mathcal{O}(c \cdot d^2 \cdot \log^{d+1} M)$ instances of Hamming distance. The reduction works even if we allow "ignore" symbols.*

▶ **Theorem 11.** *If $\diamond$ is a piecewise polynomial of degree d but is not axis-orthogonal piecewise polynomial, then Hamming distance reduces to $\mathcal{O}(d^2)$ instances of $\diamond$ and multiplication.*

We devote the rest of this section to the proofs of both Theorem 10 and Theorem 11.

To prove Theorem 10, we consider every summand separately. We show that summands with "simple" conditions (that depend on only one argument) are no harder than simple multiplication. Every other summand with conditional term $\mathbf{1}[A_i x + B_i y + C_i > 0]$ reduces under linear transformations of its arguments to $\mathbf{1}[x < y]$. It is thus enough to consider terms of the form $x^a y^b \mathbf{1}[x < y]$. We decompose such terms recursively into a sum of: terms with smaller values ($x/2, y/2$ instead of x, y), terms of smaller degree, and terms with a conditional term of a simpler form of $\mathbf{1}[x = y]$. Exhaustively applying this decomposition leaves us with a polylog number of terms of the form $w(x) \cdot \mathbf{1}[x = y]$, with which we deal separately (those decompose into a logarithmic number of regular Hamming distances).

▶ **Lemma 12.** *For an integer weight function w, the character weighted matches, that is $w(x) \cdot \mathbf{1}[x = y]$, reduce to $\mathcal{O}(\log M)$ instances of Hamming distance and multiplication.*

▶ **Lemma 13.** *An axis-orthogonal piecewise polynomial $\diamond$ of c summands of degree d reduces to $\mathcal{O}(d^2 c)$ multiplications.*

▶ **Lemma 14.** *Given integers $a, b \geq 0$, the binary function $x^a y^b \cdot \mathbf{1}[x < y]$ reduces to $\mathcal{O}(\log^{a+b+1} M)$ instances of Hamming distance and multiplication.*

Proof. Denote $\mathrm{MDom}_{a,b}(x, y) = x^a y^b \cdot \mathbf{1}[x < y]$, $\mathrm{MEq}_a(x, y) = x^a \cdot \mathbf{1}[x = y]$. First, we argue that w.l.o.g. $x, y \geq 0$. Indeed, observe that $\mathrm{MDom}_{a,b}(x+\Delta, y+\Delta) = (x+\Delta)^a (y+\Delta)^b \cdot \mathbf{1}[x < y]$, thus for large enough Δ, the computation of $\mathrm{MDom}_{a,b}$ on inputs of arbitrary sign reduces to at most $(a+1)(b+1)$ instances of MDom on non-negative inputs. Thus we assume that $0 \leq x, y \leq M$ for some $M = \mathrm{poly}(n)$.

We proceed with the following decomposition, where $u = \lfloor \frac{x}{2} \rfloor$ and $v = \lfloor \frac{y}{2} \rfloor$.

$$\begin{aligned} \mathrm{MDom}_{a,b}(x, y) &= (2u)^a (2v)^b \cdot \mathbf{1}[u < v] && (*) \\ &+ (2u)^a (2v)^b \cdot \Big(\mathbf{1}[x < y] - \mathbf{1}[u < v] \Big) && (**) \\ &+ \big(x^a y^b - (2u)^a (2v)^b \big) \cdot \mathbf{1}[x < y] && (***) \end{aligned}$$

Intuitively, we are representing the term $\mathrm{MDom}_{a,b}(x, y)$ with simpler terms: (*) represents rounding down x, y to even numbers $2u, 2v$, (**) is the corrective term for the indicator part and (***) is the corrective term for the monomial part. Simplifying those terms separately, we have

$$\begin{aligned} (*) &= 2^{a+b} \cdot \mathrm{MDom}_{a,b}(u, v), \\ (**) &= x^a (y-1)^b \cdot \mathbf{1}[\mathtt{even}(x) = \mathtt{odd}(y) - 1] = \mathrm{MEq}_{a+b}(\mathtt{even}(x), \mathtt{odd}(y) - 1), \\ (***) &= P_{a,b}(x, y) \cdot \mathbf{1}[\mathtt{odd}(x) < \mathtt{even}(y)] + Q_{a,b}(x, y) \\ &\quad \cdot \mathbf{1}[\mathtt{even}(x) < \mathtt{odd}(y)] + R_{a,b}(x, y) \cdot \mathbf{1}[\mathtt{odd}(x) < \mathtt{odd}(y)], \end{aligned}$$

where $P_{a,b}(x,y) = (x^a y^b - (x-1)^a y^b)$, $Q_{a,b}(x,y) = (x^a y^b - x^a (y-1)^b)$ and $R_{a,b}(x,y) = (x^a y^b - (x-1)^a (y-1)^b)$.

All in all, our recursion decomposes $\mathrm{MDom}_{a,b}(x,y)$ into several terms – either with the inputs reduced by a factor of 2, the test for dominance replaced with a test for equality, or to monomials of smaller degree (observe that each of $P_{a,b}(x,y)$, $Q_{a,b}(x,y)$ and $R_{a,b}(x,y)$ is of degree at most $a+b-1$). Let $T(a,b,m)$ denote the number of instances of Hamming distance that a single instance of $\mathrm{MDom}_{a,b}$, with inputs bounded in value by 2^m, is reduced to. Since by Lemma 12, MEq_{a+b} reduces to $\mathcal{O}(m \cdot (a+b))$ instances of Hamming distance, there is

$$T(a,b,m) \leq \mathcal{O}(m \cdot (a+b)) + T(a,b,m-1) + \sum_{\substack{0 \leq i \leq a \\ 0 \leq j \leq b \\ (i,j) \neq (a,b)}} 3T(i,j,m),$$

which is satisfied (for some constant C) by $T(a,b,m) \leq C \cdot m \cdot (a+b) \cdot \binom{a+b+m}{a,b,m} \cdot 4^a \cdot 4^b$. For fixed values a, b this is $\mathcal{O}(\log^{a+b+1} M)$. ◀

Proof of Theorem 10. Consider an arbitrary piecewise-polynomial binary function $\diamond$. Consider its summand $P(x,y) \cdot \mathbf{1}[Ax + By + C > 0]$. If $A = 0$ or $B = 0$ then it reduces to a binary function of degenerated form $P(x,y) \cdot \mathbf{1}[Ax + C > 0]$ which in turn reduces to $\mathcal{O}(d^2)$ multiplications by Lemma 13.

Otherwise, if $A \neq 0$ and $B \neq 0$, then there is a one-to-one linear input reduction, $u = -Ax$ and $v = By + C$, that reduces from $(-Ax)^i (By + C)^j \cdot \mathbf{1}[Ax + By + C > 0]$ to $u^i v^j \cdot \mathbf{1}[u < v]$. Note that any polynomial of degree a and b over x and y is a linear combination of $(-Ax)^i (By + C)^j$ for $0 \leq i \leq a$ and $0 \leq j \leq b$.

By applying those reductions to each summand and applying Lemma 14 to each monomial of the summand, we reach the claimed bound. ◀

To prove Theorem 11, we need the following technical Lemma:

▶ **Lemma 15.** *Consider a family of distinct lines $\Lambda = \{\lambda_i\}_{i=1}^{|\Lambda|}$, $\lambda_i = \{x, y : A_i x + B_i y + C_i = 0\}$ for integers A_i, B_i, C_i such that $|A_i|, |B_i|, |C_i| \leq M$. If there is at least one $\lambda \in \Lambda$ that is not axis-orthogonal, then there exists $\lambda_i \in \Lambda$ and $\alpha, \beta, \gamma, \delta$ such that:*

- *for any line λ_j that is not parallel to λ_i, the set $\{(\alpha x + \gamma, \beta y + \delta) : x, y \in [0 \dots N]\}$ lies* on the same side *of λ_j,*
- *for any line λ_j that is parallel to λ_i, the sets $\{(\alpha x + \gamma, \beta y + \delta) : x > y\}$ and $\{(\alpha x + \gamma, \beta y + \delta) : x < y\}$ are separated by λ_j.*

Moreover, $|\alpha|, |\beta|, |\gamma|, |\delta| \leq \mathrm{poly}(M, N)$.

Proof. Pick λ_i that is not axis-orthogonal, that is $A_i, B_i \neq 0$.

Let us denote the grid $\mathcal{G} = \{(\alpha x + \gamma, \beta y + \delta) : x, y \in [0 \dots N]\}$. To guarantee that main diagonal of $\mathcal{G}$ lies on λ_i, we need to have $\alpha = B_i \cdot k$ and $\beta = -A_i \cdot k$ for some nonzero integer k, and select values of γ, δ accordingly so that $(\gamma, \delta) \in \lambda_i$.

For non-parallel λ_i, λ_j, the coordinates of intersection point are:

$$x_{i,j} = -\begin{vmatrix} C_i & B_i \\ C_j & B_j \end{vmatrix} / \begin{vmatrix} A_i & B_i \\ A_j & B_j \end{vmatrix} \qquad y_{i,j} = -\begin{vmatrix} A_i & C_i \\ A_j & C_j \end{vmatrix} / \begin{vmatrix} A_i & B_i \\ A_j & B_j \end{vmatrix}.$$

To guarantee that whole $\mathcal{G}$ lies on the same side of λ_j, it is enough to make sure that all 4 corners are on the same side. However, we observe that iff e.g. corners (γ, δ) and $(\alpha N + \gamma, \delta)$ are separated by λ_j, it means that for some $r \in [0,1]$ lines λ_j and $A_i(x - r\alpha N) + B_i y + C_i = 0$

(that is λ_i shifted in x by $+r\alpha N$) intersect on point with $x = \delta$. To satisfy the first condition of the lemma, it is enough if every point of the convex closure of $\mathcal{G}$ has x coordinate with absolute value at least $2M^2 + |\alpha|MN$, since that is larger than any possible intersection point as described above (condition (a)). Similarly for y coordinate it should be at least $2M^2 + |\beta|MN$ (condition (b)).

Take λ_j parallel to λ_i, that is they differ only on value of C. We first make sure that all such λ_j fall between lines $\{(\alpha t + \gamma, \beta(t+1) + \delta) : t \in \mathbb{R}\}$ and λ_i or λ_i and $\{(\alpha(t+1) + \gamma, \beta t + \delta) : t \in \mathbb{R}\}$ (those lines are λ_i "shifted" one step up or down in the grid), by making sure α and β are large enough in absolute value. Indeed, it is enough to have $|\alpha A_i| = |\beta B_i| > 2M$ being largest possible difference between two values of C. It is enough to select $k = 3M$, and $\alpha = 3MB_i, \beta = -3MA_i$.

We then select γ and δ as smallest in absolute value points of λ_i such that conditions (a) and (b) are satisfied. ◀

Proof of Theorem 11. Let us take the binary function $x \diamond y = \sum_i P_i(x,y) \cdot \mathbf{1}[A_i x + B_i y + C_i > 0]$ as in the theorem statement, assuming it is of the simplest form (no redundant terms and minimal number of summands possible). We construct a reduction from Hamming distance to $\diamond$ by a series of intermediate operators.

Let d be the highest degree of any $P_1, P_2, \ldots$. Consider all the lines being borders of regions, that is $\lambda_i = \{(u,v) : A_i u + B_i v + C = 0\}$ (as elements of the continuous Euclidean plane).

We now apply Lemma 15, with $N = 3dM + 2d$. Consider $F(x,y) \stackrel{\text{def}}{=} (\alpha x + \gamma) \diamond (\beta y + \delta)$. Limited to $x, y \in [0 \ldots N]$, $F(x,y)$ is piecewise linear of a much simpler form:

$$F(x,y) = Q_>(x,y) \cdot \mathbf{1}[x > y] + Q_=(x,y) \cdot \mathbf{1}[x = y] + Q_<(x,y) \cdot \mathbf{1}[x < y]$$

for $Q_>, Q_=, Q_<$ being polynomials of degree at most d, and $Q_< \not\equiv Q_>$. Let D_x, D_y be the operators of discrete differentiation, that is $D_x F(x,y) \stackrel{\text{def}}{=} F(x+1,y) - F(x,y)$, $D_y F(x,y) \stackrel{\text{def}}{=} F(x,y+1) - F(x,y)$. There are integers $0 \le a, b \le d$ such that $D_x^a D_y^b (Q_<(x,y) - Q_>(x,y)) \equiv c$ for some constant $c \neq 0$. Thus if we consider the function:

$$G(x,y) \stackrel{\text{def}}{=} \frac{1}{c} \cdot D_x^a D_y^b (F(x,y) - Q_>(x,y)),$$

it has the following properties on $x, y \in [0 \ldots N - d]$: for $y - x > d$: $G(x,y) = 1$, and for $y - x < -d$: $G(x,y) = 0$. We observe that for $x, y \in [0 \ldots M]$, there is $\mathsf{Dom}(x,y) = G(3d \cdot x, 3d \cdot y + d)$. All in all, Ham reduces to $\mathcal{O}(d^2)$ instances of $\diamond$ and a single evaluation of a fixed polynomial $Q_>(x,y)$, which reduces to $\mathcal{O}(d^2)$ multiplications. ◀

7 Conclusion

There are several immediate applications of Theorem 10 and Theorem 11. The first one is that the improvement to DomProd from [32] translates to other MatrixProduct problems:

▶ **Corollary 16.** DomProd, L_1Prod, L_{2p+1}Prod, ThrProd, HamProd *and* $(+, \min)$-MatrixProduct *are solvable in time* $\widetilde{\mathcal{O}}(n^\rho)$*, where* $\rho \le 2.6834$ *is a solution to* $\rho = \omega(1, 4 - \rho, 1)$*.*

Observe that the reductions we presented map $\star$ to $\star$. Thus, e.g. by [26],[28] and [10], we immediately get that all considered MatrixProduct problems are of the same complexity

even on sparse inputs, up to a poly $\log n$ multiplicative term and additive term of time it takes to compute (classical) sparse matrix product of relevant matrices (which is a simpler problem).

▶ **Corollary 17.** *Consider sparse inputs where we denote by m_1 and m_2 the number of entries in A and B that contribute to the score, where A and B are matrices of n vectors of dimension n.* DomProd*,* L_1Prod*,* L_{2p+1}Prod*,* ThrProd*,* HamProd *and* $(+, \min)$-MatrixProduct *are solvable in time* $\widetilde{\mathcal{O}}(\min(n^\omega + \sqrt{m_1 m_2} \cdot n^{\frac{\omega-1}{2}}, n^2 + (m_1 m_2)^{\frac{\omega-2}{\omega-\alpha-1}} n^{\frac{2-\alpha\omega}{\omega-\alpha-1}}))$.

Since our reductions preserve the dimension of the problems, any tradeoff between $d \ll n$ and the running time translates to all other problems as well, with a poly $\log n$ multiplicative term and a $\widetilde{\mathcal{O}}(n^\omega)$ additive term. One can improve the running time of the algorithm presented in [25] using the trick of batch-processing via rectangular matrix multiplication in [32], as done for Dominance Product in [14], to obtain the following time complexity:

▶ **Corollary 18.** *For n vectors of dimension $d = n^s$ for $0 \leq s \leq 1$,* DomProd*,* L_1Prod*,* L_{2p+1}Prod*,* ThrProd*,* HamProd *and* $(+, \min)$-MatrixProduct *are solvable in time* $\widetilde{\mathcal{O}}(n^{\rho(s)})$ *where* $\rho(s) = \inf\{x : 2 \leq x \leq 3 \text{ and } \omega(1, 2+2s-x, 1) \geq x\}$*. In particular, for* $d = \mathcal{O}(n^{\alpha/2}) \supseteq \mathcal{O}(n^{0.156945})$ *all those problems are solvable in time* $\widetilde{\mathcal{O}}(n^2)$*.*

Similarly, one can look into the relation between sparsity and running time for pattern matching problems. Here, we obtain the following result:

▶ **Theorem 19.** *For a text of length n and a pattern of length m, $n \geq m$, with s_t and s_p relevant entries, respectively, the running time of* HamPM*,* LessThanPM*,* ThrPM *and* L_{2p+1}PM *is* $\widetilde{\mathcal{O}}(\sqrt{n s_t s_p} + n)$*.*

We present the following application of the scaling/filtering framework: weighted mismatches. We distinguish between *position weighted mismatches* and *character weighted mismatches*. In the pattern matching setting, the former asks for $\mathbf{O}[i] = \sum_{j:\mathbf{P}[j] \neq \mathbf{T}[i+j]} w(j)$, whereas the latter asks for $\mathbf{O}[i] = \sum_{j:\mathbf{P}[j] \neq \mathbf{T}[i+j]} w(\mathbf{P}[j])$, for some given weight function $w : \mathbb{Z} \to \mathbb{Z}$. We see that character weighted mismatches are expressible by a function $w(x) \cdot \mathbf{1}[x \neq y]$ and get by Lemma 12 that Hamming Distance Pattern Matching with Character Weights is no harder than HamPM (up to a $\log n$ factor). For position weights, we present the following:

▶ **Theorem 20.** *Hamming Distance Pattern Matching with Position Weights reduces to $\mathcal{O}(\log n)$ instances of* HamPM*.*

While it is no surprise that for example the technique of [32] can be applied to other MatrixProduct problems, it is a nice side effect of our reduction that it can be applied "automatically" without looking deeper into the structure of any of the different MatrixProduct problems involved. The reductions presented signify that regardless of whether we are looking for improved upper bounds, or new lower bounds, it is enough to concentrate on a single score function from the whole class of equivalent functions. In our opinion, Hamming distance is the "cleanest" score function, since it is the simplest – it assumes no arithmetic underlying structure of the alphabet (unlike e.g. L_1 distance) and not even an ordering of the alphabet.

References

1 Karl R. Abrahamson. Generalized String Matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

2 Amihood Amir and Martin Farach. Efficient matching of nonrectangular shapes. *Annals of Mathematics and Artificial Intelligence*, 4(3):211–224, September 1991. doi:10.1007/BF01531057.

3 Amihood Amir, Ohad Lipsky, Ely Porat, and Julia Umanski. Approximate Matching in the L_1 Metric. In *CPM*, pages 91–103, 2005. doi:10.1007/11496656_9.

4 Mikhail J. Atallah. Faster image template matching in the sum of the absolute value of differences measure. *IEEE Trans. Image Processing*, 10(4):659–663, 2001. doi:10.1109/83.913600.

5 Mikhail J. Atallah and Timothy W. Duket. Pattern matching in the Hamming distance with thresholds. *Inf. Process. Lett.*, 111(14):674–677, 2011. doi:10.1016/j.ipl.2011.04.004.

6 Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse Matrix Multiplication and Triangle Listing in the Congested Clique Model. In *OPODIS 2018*, pages 4:1–4:17, 2018. doi:10.4230/LIPIcs.OPODIS.2018.4.

7 Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi:10.1016/j.ipl.2006.08.002.

8 Peter Clifford, Raphaël Clifford, and Costas S. Iliopoulos. Faster Algorithms for δ,γ-Matching and Related Problems. In *CPM*, pages 68–78, 2005. doi:10.1007/11496656_7.

9 Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. On Problems Equivalent to (min, +)-Convolution. In *ICALP*, pages 22:1–22:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.22.

10 Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *SODA*, pages 384–391, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496813.

11 François Le Gall. Faster Algorithms for Rectangular Matrix Multiplication. In *FOCS*, pages 514–523, 2012. doi:10.1109/FOCS.2012.80.

12 Francois Le Gall and Florent Urrutia. Improved Rectangular Matrix Multiplication using Powers of the Coppersmith-Winograd Tensor. In *SODA*, pages 1029–1046, 2018. doi:10.1137/1.9781611975031.67.

13 Paweł Gawrychowski and Przemysław Uznański. Towards Unified Approximate Pattern Matching for Hamming and L_1 Distance. In *ICALP 2018*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.

14 Omer Gold and Micha Sharir. Dominance Product and High-Dimensional Closest Pair under L_∞. In *ISAAC*, pages 39:1–39:12, 2017. doi:10.4230/LIPIcs.ISAAC.2017.39.

15 Daniel Graf, Karim Labib, and Przemysław Uznański. Hamming distance completeness and sparse matrix multiplication. *CoRR*, abs/1711.03887, 2017. arXiv:1711.03887.

16 Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. *Theory of Computing*, 8(1):321–350, 2012. doi:10.4086/toc.2012.v008a014.

17 Piotr Indyk, Moshe Lewenstein, Ohad Lipsky, and Ely Porat. Closest Pair Problems in Very High Dimensions. In *ICALP*, pages 782–792, 2004. doi:10.1007/978-3-540-27836-8_66.

18 Eamonn J. Keogh and Abdullah Mueen. Curse of Dimensionality. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning and Data Mining*, pages 314–315. Springer, 2017. doi:10.1007/978-1-4899-7687-1_192.

19 Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. On the Fine-Grained Complexity of One-Dimensional Dynamic Programming. In *ICALP 2017*, pages 21:1–21:15, 2017. doi:10.4230/LIPIcs.ICALP.2017.21.

20 François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *ISSAC*, pages 296–303, 2014. doi:10.1145/2608628.2608664.

21 Ohad Lipsky. Eficient Distance Computations. Master's thesis, Bar-Ilan University. Department of Mathematics and Computer Science., 2003.

22 Ohad Lipsky and Ely Porat. L_1 pattern matching lower bound. *Inf. Process. Lett.*, 105(4):141–143, 2008. doi:10.1016/j.ipl.2007.08.011.

23 Ohad Lipsky and Ely Porat. Approximate Pattern Matching with the L_1, L_2 and L_∞ Metrics. *Algorithmica*, 60(2):335–348, 2011. doi:10.1007/s00453-009-9345-9.

24 Jiří Matoušek. Computing Dominances in E^n (Short Communication). *Inf. Process. Lett.*, 38(5):277–278, June 1991. doi:10.1016/0020-0190(91)90071-O.

25 Kerui Min, Ming-Yang Kao, and Hong Zhu. The Closest Pair Problem under the Hamming Metric. In *COCOON*, pages 205–214, 2009. doi:10.1007/978-3-642-02882-3_21.

26 Virginia Vassilevska. *Efficient algorithms for path problems in weighted graphs*. PhD thesis, Carnegie Mellon University, 2008.

27 Virginia Vassilevska and Ryan Williams. Finding a Maximum Weight Triangle in $n^{3-\delta}$ Time, with Applications. In *STOC*, pages 225–231, 2006. doi:10.1145/1132516.1132550.

28 Virginia Vassilevska, Ryan Williams, and Raphael Yuster. All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time. *Theory of Computing*, 5(1):173–189, 2009. doi:10.4086/toc.2009.v005a009.

29 Ryan Williams. On the Difference Between Closest, Furthest, and Orthogonal Pairs: Nearly-Linear vs Barely-Subquadratic Complexity. In *SODA*, pages 1207–1215, 2018. doi:10.1137/1.9781611975031.78.

30 Virginia Vassilevska Williams and Ryan Williams. Subcubic Equivalences between Path, Matrix and Triangle Problems. In *FOCS 2010*, pages 645–654, 2010. doi:10.1109/FOCS.2010.67.

31 Virginia Vassilevska Williams and Ryan Williams. Finding, Minimizing, and Counting Weighted Subgraphs. *SIAM J. Comput.*, 42(3):831–854, 2013. doi:10.1137/09076619X.

32 Raphael Yuster. Efficient algorithms on sets of permutations, dominance, and real-weighted APSP. In *SODA*, pages 950–957, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496873.

33 Peng Zhang and Mikhail J. Atallah. On approximate pattern matching with thresholds. *Inf. Process. Lett.*, 123:21–26, 2017. doi:10.1016/j.ipl.2017.03.001.

A Omitted proofs from Section 6

▶ **Lemma 12.** *For an integer weight function w, the character weighted matches, that is $w(x) \cdot \mathbf{1}[x = y]$, reduce to $\mathcal{O}(\log M)$ instances of Hamming distance and multiplication.*

Proof. Let M be the upper bound on all values of w in the considered domain of inputs. Given two integers x, y, we observe the following equality:

$$w(x) \cdot [x = y] \quad = \quad \sum_{i=0}^{\log M} 2^i \cdot \mathbf{1}[w_i(x) = w_i(y)]$$

where the filtering function w_i is defined based on w:

$$w_i(x) = \begin{cases} x & i\text{-th bit of } w(x) \text{ is } 1 \\ \star & \text{otherwise.} \end{cases}$$

Observing that $\mathbf{1}[x = y] = 1 - \mathsf{Ham}(x, y)$ finishes the proof. ◀

▶ **Lemma 13.** *An axis-orthogonal piecewise polynomial $\diamond$ of c summands of degree d reduces to $\mathcal{O}(d^2 c)$ multiplications.*

Proof. Given an axis orthogonal piecewise polynomial $F(x, y) = \sum_{i=1}^{c} P_i(x, y) \cdot \mathbf{1}[A_i x + B_i y + C_i > 0]$ of degree d. Consider summand $P_i(x, y)\mathbf{1}[A_i x + C_i > 0]$ (w.l.o.g. we assume that $B_i = 0$ and $A_i \neq 0$). Consider a monomial of $P_i(x, y)$, e.g. $x^a y^b$. Define $x' = x^a$ iff $A_i x + C_i > 0$ and $x' = \star$ otherwise, and $y' = y^b$. Then $x^a y^b \cdot \mathbf{1}[A_i x + C_i > 0] = x' y'$. ◀

Axis orthogonal piecewise polynomial $\diamond$ are no harder than multiplication in e.g. vector convolution or matrix multiplication. By Theorem 13 it reduces to multiplication in $\mathbb{Z} \cup \{\star\}$, which in turn reduces to multiplication in $\mathbb{Z}$. Indeed, it is enough to consider a map $\mathbb{Z} \cup \{\star\} \to \mathbb{Z}$ that is identity on $\mathbb{Z}$ and maps $\star \to 0$.

▶ **Theorem 19.** *For a text of length n and a pattern of length m, $n \geq m$, with s_t and s_p relevant entries, respectively, the running time of* HamPM, LessThanPM, ThrPM *and* L_{2p+1}PM *is* $\widetilde{\mathcal{O}}(\sqrt{ns_ts_p} + n)$.

Proof. Consider LessThanPM. The proof follows the non-sparse case. W.l.o.g. all the $2s_p$ actual entries are distinct integers (if it is not so, they can be made so using small $\varepsilon > 0$ shifts and then re-arranged back into integers preserving order). The s_p relevant entries of the pattern are sorted and partitioned into k buckets $B_1, \ldots, B_k$ so that B_1 gets s_p/k smallest elements, B_2 following s_p/k smallest elements, etc. We get inter-bucket contribution for bucket B_i from convolution of P_i^R with T_i, where P_i, T_i are binary strings such that $P_i[j] = 1$ iff $P[j] \in B_1 \cup \ldots \cup B_i$ and $T_i[j] = 1$ iff $T[j] \in B_i$. This in a total takes $\mathcal{O}(kn \log m)$ time for all k buckets. Intra-bucket contributions are captured in a brute force manner in $\mathcal{O}(s_ts_p/k)$ where each relevant text element is compared with at most s_p/k elements in its corresponding bucket. Choosing k to be $\max(1, \sqrt{(s_ts_p)/(n \log m)})$ gives the time bound of $\widetilde{\mathcal{O}}(n + \sqrt{ns_ts_p})$. ◀

▶ **Theorem 20.** *Hamming Distance Pattern Matching with Position Weights reduces to* $\mathcal{O}(\log n)$ *instances of* HamPM.

Proof. We solve $\mathcal{O}(\log n)$ instances of HamPM with filtering involved. This is done by constructing different pattern strings where $\mathbf{P}_i$ is defined as follows:

$$\mathbf{P}_i[j] = \begin{cases} P[j] & i\text{-th bit of } w(j) \text{ is } 1 \\ \star & \text{otherwise.} \end{cases}$$

Let $\mathbf{O}_i$ be the result vector of HamPM between text $\mathbf{T}$ and pattern $\mathbf{P}_i$. The final result vector, $\mathbf{O}$, for the Hamming distance pattern matching with position weights can be computed such that $\mathbf{O}[k] = \sum_{i=0}^{\lceil \log W \rceil} 2^i \cdot \mathbf{O}_i[k]$ where W is the maximum position weight. Given our assumption that $W = \mathrm{poly}(n)$, the result follows. ◀

What remains is to show that one can get rid of $\star$ when e.g. computing Hamming distance. We show this in the pattern matching setting for simplicity. However this can be easily extended for matrix multiplication problems as well.

▶ **Lemma 9.** *Hamming distance in $\mathbb{N} + \{\star\}$ reduces preserving linearity to two instances of Hamming distance in $\mathbb{N}$.*

Proof. Let $x, y \in \mathbb{N} + \{\star\}$. To compute $\mathsf{Ham}(x, y)$, we first use mapping that puts $\star$ into separate integer, and then apply correction that fixes distances between $\star$.

For the first instance:

$$f(t) = \begin{cases} 0 & \text{if } t = \star \\ t + 1 & \text{otherwise} \end{cases}$$

As for the second instance:

$$g(t) = \begin{cases} 0 & \text{if } t = \star \\ 1 & \text{otherwise} \end{cases}$$

Observe that $\mathsf{Ham}(x, y) = \mathsf{Ham}(f(x), f(y)) - \mathsf{Ham}(g(x), g(y))$. ◀

B Supplementary reductions

▶ **Theorem 21.** *L_1 distance reduces to* min *and multiplications.* min *reduces to L_1 and multiplications.*

Proof. $\min(x, y) = x/2 + y/2 - |x - y|/2$ and $|x - y| = x + y - \min(x, y)$. ◀

▶ **Lemma 22.** *Dominance and δ-threshold are equivalent.*

Proof. Since both dominance and threshold are shift-invariant, we assume $0 \leq x, y \leq M$ for some M bounded by $\text{poly}(n)$. Dominance reduces to one instance of threshold as $\mathsf{Dom}(x, y) = \mathsf{Thr}_\delta(x + \delta, y)$ for any $\delta > M$. Threshold reduces to two instances of dominance as $\mathsf{Thr}_\delta(x, y) = \mathsf{Dom}(y + \delta, x) + \mathsf{Dom}(x + \delta, y)$ for $\delta > 0$. ◀

Remark: Thus, the result from [33] is implied by combining Theorem 8 with Lemma 22.

▶ **Lemma 23** ([26])**.** *Hamming distance reduces to 2 instances of dominance.*

Proof. $\mathsf{Ham}(x, y) = \mathsf{Dom}(x + 1, y) + \mathsf{Dom}(-x + 1, -y)$. ◀

▶ **Lemma 24** ([22])**.** *Dominance reduces to 2 instances of L_1, Hamming distance reduces to 3 instances of L_1.*

Proof. $\mathsf{Dom}(x, y) = |x - (y + 1)|/2 - |x - y|/2 + 1/2$ and $\mathsf{Ham}(x, y) = 1 + |x - y| - |x - (y + 1)|/2 - |(x + 1) - y|/2$ ◀

C Example reductions

Figures 2 and 3 illustrate our reductions from Theorems 7 and 8, respectively.

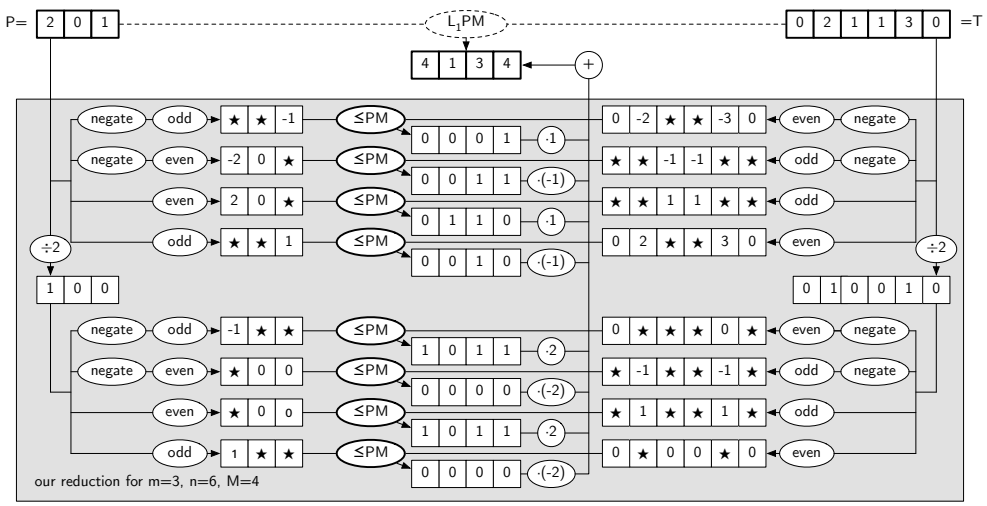


Figure 2 Our reduction from L1PatternMatching to LessThanPatternMatching in Theorem 7 instantiated for a pattern P of length $m = 3$ and a text T of length $n = 6$ over an alphabet of integers $\{0, 1, 2, 3\}$, so $M = 2^2 = 4$.

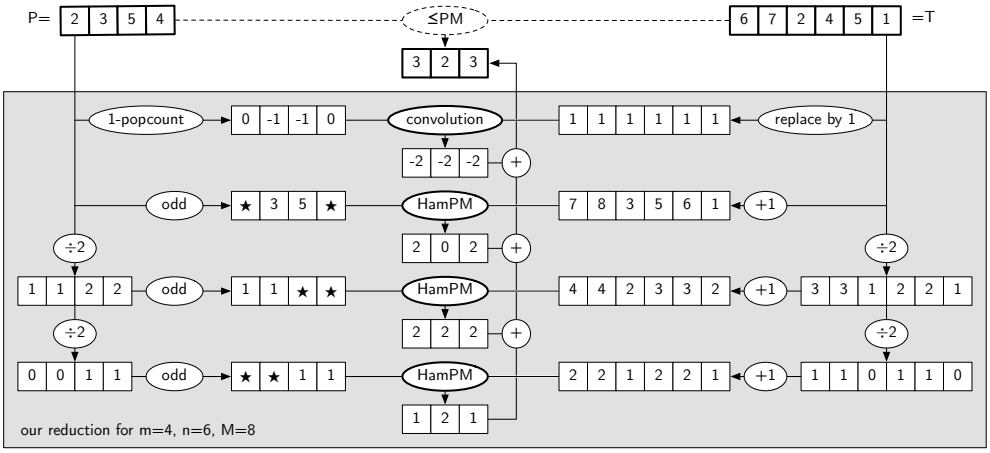


Figure 3 Our reduction from LessThanPatternMatching to HammingDistancePatternMatching in Theorem 8 instantiated for a pattern P of length $m = 4$ and a text T of length $n = 6$ over an alphabet of integers $\{0, 1, 2, 3, 4, 5, 6, 7\}$, so $M = 2^3 = 8$.

Approximating Approximate Pattern Matching

Jan Studený
Department of Computer Science, ETH Zürich, Switzerland
studenyj@student.ethz.ch

Przemysław Uznański
Institute of Computer Science, University of Wrocław, Poland[1]
puznanski@cs.uni.wroc.pl

Abstract

Given a text T of length n and a pattern P of length m, the approximate pattern matching problem asks for computation of a particular *distance* function between P and every m-substring of T. We consider a $(1 \pm \varepsilon)$ multiplicative approximation variant of this problem, for ℓ_p distance function. In this paper, we describe two $(1+\varepsilon)$-approximate algorithms with a runtime of $\widetilde{\mathcal{O}}(\frac{n}{\varepsilon})$ for all (constant) non-negative values of p. For constant $p \geq 1$ we show a deterministic $(1+\varepsilon)$-approximation algorithm. Previously, such run time was known only for the case of ℓ_1 distance, by Gawrychowski and Uznański [ICALP 2018] and only with a randomized algorithm. For constant $0 \leq p \leq 1$ we show a randomized algorithm for the ℓ_p, thereby providing a smooth tradeoff between algorithms of Kopelowitz and Porat [FOCS 2015, SOSA 2018] for Hamming distance (case of $p = 0$) and of Gawrychowski and Uznański for ℓ_1 distance.

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

Keywords and phrases Approximate Pattern Matching, ℓ_p Distance, ℓ_1 Distance, Hamming Distance, Approximation Algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.15

Related Version A full version of the paper is available at [24], `http://arxiv.org/abs/1810.01676`.

1 Introduction

Pattern matching is one of the core problems in text processing algorithms. Given a text T of length n and a pattern P of length m, $m \leq n$, both over an alphabet Σ, one searches for occurrences of P in T as a substring. A generalization of a pattern matching is to find substrings of T that are similar to P, where we consider a particular string distance and ask for all m-substrings of T where the distance to P does not exceed a given threshold, or simply report the distance from P to every m-substring of T. Typical distance functions considered are Hamming distance, ℓ_1 distance, or in general ℓ_p distances for some constant p, assuming input is over a numerical, e.g. integer, alphabet.

For reporting all Hamming distances, Abrahamson [1] described an algorithm with the complexity of $\mathcal{O}(n\sqrt{m \log m})$. Using a similar approach, the same complexity was obtained in [18] and later in conference works [3, 5] for reporting all ℓ_1 distances. It is a major open problem whether near-linear time algorithm, or even $\mathcal{O}(n^{3/2-\varepsilon})$ time algorithm, is possible for such problems. A conditional lower bound [6] was shown, via a reduction from matrix multiplication. This means that existence of combinatorial algorithm with runtime $\mathcal{O}(n^{3/2-\varepsilon})$ solving the problem for Hamming distances implies combinatorial algorithms for boolean matrix multiplication with $\mathcal{O}(n^{3-\delta})$ runtime, which existence is unlikely. If one is uncomfortable with poorly defined notion of *combinatorial* algorithms, one can apply the reduction to obtain a lowerbound of $\Omega(n^{\omega/2})$ for Hamming distances pattern matching,

[1] Most of the work was done while the author was affiliated with ETH Zürich.

where $2 \leq \omega < 2.373$ is a matrix multiplication exponent.[2] Later, the complexity of pattern matching under Hamming distance and under ℓ_1 distance was proven to be identical (up to polylogarithmic terms) [11, 19].

The mentioned hardness results serve as a motivation for considering relaxation of the problems, with $(1+\varepsilon)$ multiplicative approximation being the obvious candidate. For Hamming distance, Karloff [14] was the first to propose an efficient approximation algorithm with a run time of $\mathcal{O}(\frac{n}{\varepsilon^2}\log^3 m)$. The $\frac{1}{\varepsilon^2}$ dependency was believed to be inherent, as is the case for e.g. space complexity of sketching of Hamming distance, cf. [25, 12, 4]. However, for approximate pattern matching that was refuted by Kopelowitz and Porat [15, 16], by providing randomized algorithms with complexity $\mathcal{O}(\frac{n}{\varepsilon}\log n \log m \log\frac{1}{\varepsilon}\log|\Sigma|)$ and $\mathcal{O}(\frac{n}{\varepsilon}\log n\log m)$ respectively. Moving to ℓ_1 distance, Lipsky and Porat [20] gave a deterministic algorithm with a run time of $\mathcal{O}(\frac{n}{\varepsilon^2}\log m\log U)$, while later Gawrychowski and Uznański [10] have improved the complexity to a (randomized) $\mathcal{O}(\frac{n}{\varepsilon}\log^2 n\log m\log U)$, where U is the maximal integer value on the input. Additionally, we refer the reader to the line of work on other relaxations on exact the distance reporting [2, 7, 10, 3].

A folklore result (c.f. [20]) states that the randomized algorithm with a run time of $\widetilde{\mathcal{O}}(\frac{n}{\varepsilon^2})$ is in fact possible for any ℓ_p distance, $0 < p \leq 2$, with use of p-stable distributions and convolution.[3] Such distributions exist only when $p \leq 2$, which puts a limit on this approach. See [22] for wider discussion on p-stable distributions. Porat and Efremenko [23] have shown how to approximate general distance functions between pattern and text in time $\mathcal{O}(\frac{n}{\varepsilon^2}\log^2 m\log^3|\Sigma|\log B_d)$, where B_d is upperbound on distance between two characters in Σ. Their solution does not immediately translates to ℓ_p distances, since it allows only for score functions of form $\sum_j d(t_{i+j}, p_j)$ where d is arbitrary metric over Σ. Authors state that their techniques generalize to computation of ℓ_2 distances, but the dependency ε^{-2} in their approach is unavoidable. [20] observe that ℓ_2 pattern matching can be in fact computed in $\mathcal{O}(n\log m)$ time, by reducing it to a single convolution computation. This case and analogously case of $p = 4, 6, \ldots$ are the only ones where fast and exact algorithm is known.

We want to point that for ℓ_∞ pattern matching there is an approximation algorithm of complexity $\mathcal{O}(\frac{n}{\varepsilon}\log m\log U)$ by Lipsky and Porat [20]. Moving past pattern matching, we want to point that in a closely related problem of computing $(\min,+)$-convolution there exists $\mathcal{O}(\frac{n}{\varepsilon}\log\frac{n}{\varepsilon}\log U)$ time algorithm computing $(1+\varepsilon)$ approximation, cf. Mucha et al. [21].

Two questions follow naturally. First, is there a $\widetilde{\mathcal{O}}(\frac{n}{\mathrm{poly}(\varepsilon)})$ algorithm for ℓ_p norms pattern matching when $p > 2$? Second, is there anything special to $p = 0$ and $p = 1$ cases that allows for faster algorithms, or can we extend their complexities to other ℓ_p norms? To motivate further those questions, observe that in the regime of maintaining ℓ_p sketches in the *turnstile* streaming model (sequence of updates to vector coordinates), one needs small space of $\Theta(\log n)$ bits when $p \leq 2$ (cf. [13]), while when $p > 2$ one needs large space of $\Theta(n^{1-2/p}\log n)$ bits (cf. [9, 17]) meaning there is a sharp transition in problem complexity at $p = 2$. Similar phenomenon of transition at $p = 2$ is observed for p-stable distributions, and one could expect such transition to happen in the pattern matching regime as well.

In this work we show that for any *constant* $p \geq 0$ there is an algorithm of complexity $\widetilde{\mathcal{O}}(\frac{n}{\varepsilon})$, replicating the phenomenon of linear dependency on ε^{-1} from Hamming distance and ℓ_1 distance to all ℓ_p norms. Additionally this provides evidence that no transition at $p = 2$ happens, and so far to our understanding cases of $p > 2$ and $p < 2$ are of similar hardness.

---

[2] Although the issue is that we do not even know whether $\omega > 2$ or not.

[3] We use $\widetilde{\mathcal{O}}$ notation to hide factors polylogarithmic in $n, m, |\Sigma|, U$ and ε^{-1}.

1.1 Definitions and preliminaries

Model. In most general setting, our inputs are strings taken from arbitrary alphabet Σ. We use this notation only when structure of alphabet is irrelevant for the problem (e.g. Hamming distances). However, when considering ℓ_p distances we focus our attention over an integer alphabet $[U] \stackrel{\text{def}}{=} \{0, 1, ..., U-1\}$ for some U. One can usually assume that $U = \text{poly}(n)$, and then $\log U$ term can be safely hidden in the $\widetilde{\mathcal{O}}$ notation, however we provide the dependency explicitly in Theorem statements. Even without such assumption, we can assume standard word RAM model, in which arithmetic operations on words of size $\log U$ take constant time. Otherwise the complexities have an additional $\log U$ factor. We also denote $u = \log U$. While we restrict input integer values, we allow intermediate computation and output to consist of floating point numbers having u bits of precision.

Distance between strings. Let $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_n$ be two strings. For any $p > 0$, we define their ℓ_p distance as

$$\ell_p(X, Y) = \left(\sum_i |x_i - y_i|^p \right)^{1/p}.$$

Particularly, ℓ_1 distance is known as *Manhattan distance*, and ℓ_2 distance is known as *Euclidean distance*. Observe that the p-th power of ℓ_p distance has particularly simpler form of $\ell_p(X, Y)^p = \sum_i |x_i - y_i|^p$.

The *Hamming distance* between two strings is defined as

$$\text{Ham}(X, Y) = |\{i : x_i \neq y_i\}|.$$

Adopting the convention that $0^0 = 0$ and $x^0 = 1$ for $x \neq 0$, we observe that $(\ell_p)^p$ approaches Hamming distance as $p \to 0$. Thus Hamming distance is usually denoted as ℓ_0 (although $(\ell_0)^0$ is more precise notation).

Text-to-pattern distance. For text $T = t_1 t_2 \dots t_n$ and pattern $P = p_1 p_2 \dots p_m$, the text-to-pattern distance is defined as an array S such that, for every i, $S[i] = d(T[i+1 \,..\, i+m], P)$ for particular distance function d. Thus, for ℓ_p distance $S[i] = \left(\sum_{j=1}^m |t_{i+j} - p_j|^p \right)^{1/p}$, while for Hamming distance $S[i] = |\{j \in \{1, \dots, m\} : t_{i+j} \neq p_j\}|$. Then $(1+\varepsilon)$-approximate distance is defined as an array S_ε such that, for every i, $(1-\varepsilon) \cdot S[i] \leq S_\varepsilon[i] \leq (1+\varepsilon) \cdot S[i]$.

Rounding and arithmetic operations. For any value x, we denote by $x^{(i)} = \lfloor x/2^i \rfloor \cdot 2^i$ the value with i y bits rounded. However, with a little stretch of notation, we do not limit value of i to be positive. We denote by $\|r\|_c$ the norm *modulo* c, that is $\|r\|_c = \min(r \bmod c, c - (r \bmod c))$.

1.2 Our results

In this paper we answer favorably both questions by providing relevant algorithms. First, we show how to extend the deterministic ℓ_1 distances algorithm into ℓ_p distances, when $p \geq 1$.

▶ **Theorem 1.** *For any $p \geq 1$ there is a* deterministic *algorithm computing $(1+\varepsilon)$ approximation to pattern matching under ℓ_p distances in time $\mathcal{O}(\frac{n}{\varepsilon} \log m \log U)$ (assuming $\varepsilon \leq 1/p$).*

We then move to the case of ℓ_p distances when $p < 1$. We show that it is possible to construct a randomized algorithm with the desired complexity.

▶ **Theorem 2.** *For $0 < p < 1$, there is a* randomized *algorithm computing $(1+\varepsilon)$ approximation to pattern matching under ℓ_p distances in time $\mathcal{O}(p^{-1}\varepsilon^{-1} n \log m \log^2 U \log n)$. The algorithm is correct with high probability.*[4]

Finally, combining with existing ℓ_0 algorithm from [16] we obtain as a corollary that for *constant* $p \geq 0$ approximation of pattern matching under ℓ_p distances can be computed in $\widetilde{\mathcal{O}}(\frac{n}{\varepsilon})$ time.

2 Approximation of ℓ_p distances

We start by showing how convolution finds its use in counting versions of pattern matching, either exact or approximation algorithms. Consider the case of pattern matching under ℓ_2 distances. Observe that we are looking for S such that $S[i]^2 = \sum_{j-k=i}(t_j - p_k)^2 = \sum_j t_j^2 + \sum_k p_k^2 - 2\sum_{j-k=i} t_j p_k$. The last term is just a convolution of vectors in disguise and is equivalent to computing convolution of T and reverse ordered P. Such approach can be applied to solving exact pattern matching via convolution (observing that ℓ_2 distance is 0 iff there is an exact match).

We follow with a technique for computing exact text-to-pattern distance, for arbitrary distance functions, introduced by [20], which is a generalization of a technique used in [8]. We provide a short proof for completeness.

▶ **Theorem 3** ([20])**.** *Text-to-pattern distance where strings are over* arbitrary *alphabet Σ can be computed* exactly *in time $\mathcal{O}(|\Sigma| \cdot n \log m)$.*

Proof. For every letter $c \in \Sigma$, construct a new text T^c by setting $T^c[i] = 1$ if $t_i = c$ and $T^c[i] = 0$ otherwise. A new pattern P^c is constructed by setting $P^c[i] = d(c, p_i)$. Since $d(t_{i+j}, p_j) = \sum_{c\in\Sigma} T^c[i+j] \cdot P^c[j]$, it is enough to invoke $|\Sigma|$ times convolution. ◀

Theorem 3 allows us to compute text-to-pattern distance exactly, but the time complexity $\mathcal{O}(|\Sigma| n \log m)$ is prohibitive for large alphabets (when $|\Sigma| = poly(n)$). However, it is enough to reduce the size of alphabet used in the problem (at the cost of reduced precision) to reach desired time complexity. While this might be hard, we proceed as follows: we decompose our weight function into a sum of components, each of which is approximated by a corresponding function on a reduced alphabet.

We say that a function d is *effectively over smaller alphabet* Σ' if it is represented as $d(x, y) = d'(\iota_1(x), \iota_2(y))$ for some $\iota_1, \iota_2 : \Sigma \to \Sigma'$ and d'. It follows from Theorem 3 that text-to-pattern under distance d can be computed in time $\widetilde{\mathcal{O}}(|\Sigma'|n)$ (ignoring the cost of computing ι_1 and ι_2).

Decomposition. Let $D(x, y) = |x - y|^p$ be a function corresponding to $(\ell_p)^p$ distance, that is $\ell_p(X, Y)^p = \sum_i D(x_i, y_i)$. Our goal is to decompose $D(x, y) = \sum_i \alpha_i(x, y)$ into small (polylogarithmic) number of functions, such that each $\alpha_i(x, y)$ is approximated by $\beta_i(x, y)$ that is effectively over alphabet of $\mathcal{O}(\frac{1}{\varepsilon})$ size (up to polylogarithmic factors). Now we can use Theorem 3 to compute contribution of each β_i. We then have that $G(x, y) = \sum_i \beta_i(x, y)$ approximates F, and text-to-pattern distance under G can be computed in the desired $\widetilde{\mathcal{O}}(\frac{n}{\varepsilon})$ time. We present such decomposition, useful immediately in case of $p \geq 1$ and as we see in section 2.2 with a little bit of effort as well in case when $0 < p \leq 1$.

[4] Probability at least $1 - 1/n^c$ for arbitrarily large constant c.

Useful estimations. We use following estimations in our proofs. For $p \geq 1$

$$(1-\varepsilon)^p \geq 1 - p\varepsilon, \qquad \text{for } 0 \leq \varepsilon \leq 1, \tag{1}$$
$$(1+\varepsilon)^p \geq 1 + p\varepsilon, \qquad \text{for } 0 \leq \varepsilon, \tag{2}$$
$$(1-\varepsilon)^p \leq 1 - p\varepsilon(1-1/e), \qquad \text{for } 0 \leq \varepsilon \leq 1/p, \tag{3}$$
$$a^p - (a-b)^p \leq pa^{p-1}b, \qquad \text{for } a \geq b \geq 0. \quad \text{(follows from 2)} \tag{4}$$

For $0 \leq p \leq 1$

$$(1-\varepsilon)^p \leq 1 - p\varepsilon, \qquad \text{for } 0 \leq \varepsilon \leq 1, \tag{5}$$
$$(1-\varepsilon)^p \geq 1 - 2p\varepsilon \ln 2, \qquad \text{for } 0 \leq \varepsilon \leq 1/2, \tag{6}$$
$$(1+\varepsilon)^p \geq 1 + p\varepsilon \ln 2, \qquad \text{for } 0 \leq \varepsilon \leq 1, \tag{7}$$
$$a^p - (a-b)^p \leq 2pa^{p-1}b \ln 2, \qquad \text{for } a \geq 2b \geq 0. \quad \text{(follows from 6)} \tag{8}$$

2.1 Algorithm for $p \geq 1$

In this section we prove Theorem 1. We start by constructing a family of functions F_i, which are better refinements of F as i decreases.

First step. Let us denote

$$F_i(x,y) = \Big(\max(0, |x-y| - 2^i)\Big)^p \qquad \text{and} \qquad f_i = F_i - F_{i+1}.$$

Observe that $F_u = 0$ (for $0 \leq x, y \leq U$). Moreover, there is a telescoping sum $F_i = \sum_{j=i}^{u} f_j$. To better see the the telescopic sum, consider case $p = 1$. We then represent $F_{-u}(x,y) = \sum_{i=-u}^{u} f_i(x,y) = (-2^{-u} + 2^{-u+1}) + (-2^{-u+1} + 2^{-u+2}) + \ldots + (-2^{t-1} + 2^t) + (|x-y| - 2^t) + 0 + \ldots + 0$. Such decomposition (for $p = 1$) was first considered, to our knowledge, in [20].

Second step. Instead of using x and y for evaluation of F_i, we evaluate F_i using x and y with all bits younger than i-th one set to zero. Formally, define $x^{(i)} = \lfloor x/2^i \rfloor \cdot 2^i$, $y^{(i)} = \lfloor y/2^i \rfloor \cdot 2^i$. Now we denote

$$G_i(x,y) = F_i(x^{(i)}, y^{(i)})$$

Similarly as for f_i, define $g_i = G_i - G_{i+1}$. Using the same reasoning, we have $G_u = 0$. For integers $i \leq 0$ the functions F_i and G_i are the same (as we are not rounding) and therefore $F_{-u} = G_{-u} = \sum_{i=-u}^{u} g_i$. Intuitively, g_i captures contribution of i-th bit of input to the output value (assuming all older bits are set and known, and all younger bits are unknown).

Third step. Let η be a value to be fixed later, depending on ε and p. Assume w.l.o.g. that η is such that $1/\eta$ is an integer. We now define $\widehat{g}_i$ as a refinement of g_i, by replacing $|x^{(i)} - y^{(i)}|$ with $\|x^{(i)} - y^{(i)}\|_{B_i}$ and $|x^{(i+1)} - y^{(i+1)}|$ with $\|x^{(i+1)} - y^{(i+1)}\|_{B_i}$, where $B_i = 2^i/\eta$, that is doing all the computation *modulo* B_i. To be precise, define

$$\overrightarrow{G}_i(x,y) = \Big(\max(0, \|x^{(i)} - y^{(i)}\|_{B_i} - 2^i)\Big)^p$$
$$\overleftarrow{G}_{i+1}(x,y) = \Big(\max(0, \|x^{(i+1)} - y^{(i+1)}\|_{B_i} - (2^{i+1})\Big)^p$$

and then $\widehat{g}_i = \overrightarrow{G}_i - \overleftarrow{G}_{i+1}$. Additionally, we denote for short $\widehat{G}_i = \sum_{j=i}^{u} \widehat{g}_j$.

Intuitively, $\widehat{g}_i$ approximates g_i in the scenario of limited knowledge – it estimates contribution of i-th bit of input to the output, assuming knowledge of bits $i+1$ to $i+\log \eta^{-1}$ of input. We are now ready to provide an approximation algorithm to $(\ell_p)^p$ text-to-pattern distances.

▶ **Algorithm 4.**
Input:
- *T is the text,*
- *P is the pattern,*
- *η controls the precision of the approximation.*

Steps:
1. *For each $i \in \{-u, \ldots, u\}$ compute array S_i being the text-to-pattern distance between T and P using $\widehat{g}_i$ distance function (parametrized by η) using Theorem 3.*
2. *Output array* $S_\varepsilon[i] = \left(\sum_{j=-u}^{u} S_j[i] \right)^{1/p}$.

To get the $(1+\varepsilon)$ approximation we run the Algorithm 4 with $\eta = \frac{\varepsilon}{128}$.

Now, we need to show the running time and correctness of the result. Firstly, to prove the correctness, we divide summands $\widehat{g}_i$ into three groups and reason about them separately. As computing F_{-u}, G_{-u}(by summing f_i's and g_i's respectively) yields $(1+\varepsilon)$ multiplicative error, we will show that the difference between computing g_i and $\widehat{g}_i$ brings only an additional $(1+\varepsilon)$ multiplicative error.

▶ **Lemma 5.** *For i such that $|x-y| \leq 2^i$ both $g_i(x,y) = 0$ and $\widehat{g}_i(x,y) = 0$.*

Proof. As both $g_i, \widehat{g}_i$ are symmetric functions, we can w.l.o.g. assume $x \geq y$. $\forall j \geq i$:

$$\left| x^{(j)} - y^{(j)} \right| = 2^j \left(\left\lfloor \frac{x}{2^j} \right\rfloor - \left\lfloor \frac{y}{2^j} \right\rfloor \right) \leq 2^j \left(\left\lfloor \frac{x}{2^j} \right\rfloor - \left\lfloor \frac{x-2^i}{2^j} \right\rfloor \right) \leq 2^j.$$

Therefore $G_j = 0$ from which $g_i(x,y) = 0$ follows. And because $\|x^{(j)} - y^{(j)}\|_{B_j} \leq |x^{(j)} - y^{(j)}|$ we have $\widehat{g}_i(x,y) = 0$ as well. ◀

▶ **Lemma 6.** *For i such that $|x-y| > 2^i \geq 4\eta|x-y|$ we have $g_i(x,y) = \widehat{g}_i(x,y)$.*

Proof. For $g_i(x,y) = \widehat{g}_i(x,y)$ to hold, it is enough to show that both *norms* $|\cdot|$ and $\|\cdot\|_{B_i}$ are the same for $x^{(i)} - y^{(i)}$ and $x^{(i+1)} - y^{(i+1)}$. This happens if the absolute values of the respective inputs are smaller than $B_i/2$. Let us bound both $|x^{(i)} - y^{(i)}|$ and $|x^{(i+1)} - y^{(i+1)}|$:

$$\max(|x^{(i)} - y^{(i)}|, |x^{(i+1)} - y^{(i+1)}|) \leq |x-y| + 2^{i+1} \leq 2^{i+1}(1 + \frac{1}{8\eta}).$$

We can w.l.o.g. assume $\eta \leq 1/8$ in order to make $\frac{1}{8\eta}$ a dominant term in the parentheses and reach:

$$\max(|x^{(i)} - y^{(i)}|, |x^{(i+1)} - y^{(i+1)}|) \leq 2^{i+1}(1 + \frac{1}{8\eta}) \leq \frac{2^i}{2\eta} = \frac{B_i}{2}.$$

Therefore $\|x^{(i)} - y^{(i)}\|_{B_i} = |x^{(i)} - y^{(i)}|$ as well as $\|x^{(i+1)} - y^{(i+1)}\|_{B_i} = |x^{(i+1)} - y^{(i+1)}|$ which completes the proof. ◀

▶ **Lemma 7.** *If $p \geq 1$ then for i such that $4\eta|x-y| > 2^i$ we have $|g_i(x,y)| \leq 2p2^i \cdot |x-y|^{p-1}$.*

Proof. For the sake of the proof, we will w.l.o.g. assume $\eta \leq 1/8$. Denote $A = |x^{(i)} - y^{(i)}|$, $B = |x^{(i+1)} - y^{(i+1)}|$, $A' = \max(0, A - 2^i)$ and $B' = \max(0, B - 2^{i+1})$. Observe that $|x-y|-2^i \leq A \leq |x-y|+2^i$ thus $|x-y|-2\cdot 2^i \leq A' \leq |x-y|$, and similarly $|x-y|-2\cdot 2^{i+1} \leq B' \leq |x-y|$ so $|A'-B'| \leq 2 \cdot 2^i$. Assume w.l.o.g. that $A' \geq B'$. We bound

$$\begin{aligned} |g_i(x,y)| &= (A')^p - (A' - (A' - B'))^p \\ &\leq p(A' - B')(A')^{p-1} && \text{(by (4))} \\ &\leq 2p2^i \cdot |x-y|^{p-1} \end{aligned}$$

◀

▶ **Lemma 8.** *If $p \geq 1$ then for i such that $4\eta|x-y| > 2^i$ we have $|\widehat{g}_i(x,y)| \leq 2p2^i \cdot |x-y|^{p-1}$.*

Proof. Follows by the same proof strategy as in proof of Lemma 7, replacing $|\cdot|$ with $\|\cdot\|_{B_i}$. ◀

▶ **Theorem 9.** *$\widehat{G}_{-u} = \sum_{i \geq -u} \widehat{g}_i$ approximates F_{-u} up to an additive $32 \cdot p \cdot \eta \cdot |x-y|^p$ term.*

Proof. We bound the difference between two terms:

$$\begin{aligned} |F_{-u}(x,y) - \sum_{i=-u}^{u} \widehat{g}_i(x,y)| &\leq \sum_{i=-u}^{\log_2(4\eta|x-y|)} (|\widehat{g}_i(x,y)| + |g_i(x,y)|) \\ &\leq 2 \cdot \left(\sum_{i=-\infty}^{\log_2(4\eta|x-y|)} 2^i \right) \cdot 2 \cdot p \cdot |x-y|^{p-1} \\ &\leq 32 \cdot \eta|x-y| \cdot p \cdot |x-y|^{p-1} \end{aligned}$$

where the bound follows from Lemma 5, 6, 7 and 8. ◀

We now show that F_{-u} is a close approximation of D (recall $D(x,y) = |x-y|^p$).

▶ **Lemma 10.** *For integers x, y there is $D(x,y) \cdot (1 - (2\ln 2)p/U) \leq F_{-u}(x,y) \leq D(x,y)$.*

Proof. For $x = y$ the lemma trivially holds, so for the rest of the proof we will assume $x \neq y$. As x, y are integers only, their smallest non-zero distance is 1. As $-u < 0$ the $|x-y| - 2^{-u} > 0$ and we bound $|x-y| \cdot (1 - 1/U) \leq \max(0, |x-y| - 2^{-u}) \leq |x-y|$. By (1) (when $p \geq 1$) or (6) (when $p \leq 1$) the claim follows. ◀

By combining Theorem 9 with the Lemma 10 above we conclude that additive error of Algorithm 4 at each position is $(32p \cdot \eta + \frac{p}{U}) \cdot |x-y|^p = p(\varepsilon/4 + 1/U) \cdot |x-y|^p \leq p\varepsilon|x-y|^p$ (since w.l.o.g. $\varepsilon \geq 4/U$), thus the relative error is $(1 + p\varepsilon/2)$.

Observe that each $\widehat{g}_i$ is effectively a function over the alphabet of size $B_i/2^i = 1/\eta$. Thus, the complexity of computing text-to-pattern distance using $\widehat{g}_i$ distance is $\mathcal{O}(\eta^{-1} n \log m)$, and iterating over at most $2u$ summands makes the total time $\mathcal{O}(\varepsilon^{-1} n \log m \log U)$.

Finally, since $p \geq 1$ and w.l.o.g. $\varepsilon \leq 1/p$, by (2) and (3) $(1 + p\varepsilon/2)$ approximation of ℓ_p^p distances is enough to guarantee $(1+\varepsilon)$ approximation of ℓ_p distances.

2.2 Algorithm for $0 < p \leq 1$

In this section we prove Theorem 2. We note that the algorithm presented in the previous section does not work, since in the proof of Lemma 7 and 8 we used the convexity of function $|t|^p$, which is no longer the case when $p < 1$.

However, we observe that Lemma 5 and 6 hold even when $0 < p \leq 1$. To combat the situation where adversarial input makes the estimates in Lemma 7 and 8 to grow too large, we use a very weak version of hashing. Specifically, we pick at random a linear function

$\sigma(t) = r \cdot t$, where $r \in [1, 9)$ is a random independent variable. Such function applied to the input makes its bit sequences appear more "random" while preserving the inner structure of the problem.

Consider a following approach:

▶ **Algorithm 11.**
1. *Fix* $\eta = \frac{\varepsilon \cdot p}{15555 \log U \ln 2}$.
2. *Pick* $r \in [1, 9)$ *uniformly at random.*
3. *Compute* $T' = r \cdot T$ *and* $P' = r \cdot P$.
4. *Use Algorithm 4 to compute* S'*, text-to-pattern distance between* T' *and* P' *using* $\widehat{G}_{-u}$ *distance function.*
5. *Output* $S'' = S' \cdot r^{-1}$.

Now we analyze the expected error made by estimation from Algorithm 11. We denote the expected additive error of estimation of $(\ell_p)^p$ distances as

$$\mathrm{err}(x, y) \stackrel{\text{def}}{=} \mathbb{E}_{r \in [1,9)} \Big[[\Big(\frac{1}{r} \Big)^p \Big| \widehat{G}_{-u}(rx, ry) - |rx - ry|^p \Big| \Big].$$

▶ **Theorem 12.** *The procedure of Algorithm 11 has the* expected *additive error* $err(x, y) \leq \frac{\varepsilon p}{3 \ln 2} |x - y|^p$.

Proof. Assume that $x \neq y$, as otherwise the bound trivially follows. We bound the absolute error as follow, denoting $k = \log(8\eta|x - y|)$.

$$\begin{aligned}
\mathrm{err}(x, y) &\leq \mathbb{E}_{r \in [1,9)} \Big[\Big(\frac{1}{r} \Big)^p \Big| \widehat{G}_{-u}(rx, ry) - F_{-u}(rx, ry) \Big| \Big] \\
&\quad + \mathbb{E}_{r \in [1,9)} \Big[\, |F_{-u}(rx, ry) - D(rx, ry)| \, \Big] \\
&\leq \mathbb{E}_{r \in [1,9)} \Big[\Big| \sum_{i=-u}^{u} (\widehat{g}_i(rx, ry) - g_i(rx, ry)) \Big| \Big] \\
&\quad + \mathbb{E}_{r \in [1,9)} \Big[\Big(\frac{1}{r} \Big)^p 2(\ln 2) \frac{p}{U} D(rx, ry) \Big] && ((1/r)^p \leq 1) \\
&\leq \sum_{i=-u}^{k} \mathbb{E}_{r \in [1,9)} \Big[\, |\widehat{g}_i(rx, ry)| \, \Big] + \mathbb{E}_{r \in [1,9)} \Big[\Big| \sum_{i=-u}^{k} g_i(rx, ry) \Big| \Big] \\
&\quad + 2(\ln 2) \frac{p}{U} |x - y|^p && \text{(Lemma 5, 6)}
\end{aligned}$$

Now, we bound the first two summands separately in following lemmas.

▶ **Lemma 13.** $|\sum_{i=-u}^{k} g_i(rx, ry)|$ *is upper bounded by* $32(\ln 2)\eta|x - y|^p$.

Proof. Since w.l.o.g. $\eta \leq 1/32$ thus $2^{k+1} \leq 1/2 \cdot r|x - y|$):

$$\begin{aligned}
\Big| \sum_{i=-u}^{k} g_i(rx, ry) \Big| &\leq \Big| \sum_{i=-\infty}^{k} g_i(rx, ry) \Big| \\
&\leq |G_{k+1}(rx, ry) - D(rx, ry)| \\
&\leq ((r|x - y|)^p - (r|x - y| - 2^{k+1})^p) \\
&\leq r^p |x - y|^p \cdot 2p(\ln 2) \frac{2^{k+1}}{r|x - y|} && \text{(by (8))} \\
&\leq 32(\ln 2)\eta|x - y|^p. && (\, r^{p-1} \leq 1) \qquad \blacktriangleleft
\end{aligned}$$

▶ **Lemma 14.** *For* $i \le k = \log(8\eta|x-y|)$ *we have* $\mathbb{E}_{r\in[1,9)}\Big[\,|\widehat{g}_i(rx,ry)|\,\Big] \le (1152 + 192(\ln 2)))\eta|x-y|^p$.

(Due to the space constraints, the proof of this Lemma is deferred to the appendix.)

By combining bounds from Lemma 13, and Lemma 14 we get:

$$\begin{aligned}
\mathrm{err}(x,y) &\le \sum_{i=-u}^{k} \mathbb{E}_{r\in[1,9)}\Big[\,|\widehat{g}_i(rx,ry)|\,\Big] + \mathbb{E}_{r\in[1,9)}\Big[\,\Big|\sum_{i=-u}^{k} g_i(rx,ry)\Big|\,\Big] + 2(\ln 2)\frac{p}{U}|x-y|^p \\
&\le 2(\ln 2)\frac{p}{U}|x-y|^p + (32(\ln 2)\eta|x-y|^p + \sum_{i=-u}^{u}(1152 + 192(\ln 2)))\eta|x-y|^p \\
&\le 2(\ln 2)\frac{p}{U}|x-y|^p + (32(\ln 2)\eta|x-y|^p + 2\log U(1152 + 192(\ln 2)))\eta|x-y|^p \\
&\le 2(\ln 2)\frac{p}{U}|x-y|^p + (32(\ln 2) + 2\log U(1152 + 192(\ln 2))))\eta|x-y|^p \\
&\le 2(\ln 2)\frac{p}{U}|x-y|^p + 2593\log U\eta|x-y|^p \\
&\le \frac{\varepsilon p}{6\ln 2}|x-y|^p + \frac{\varepsilon p}{6\ln 2}|x-y|^p \qquad \text{w.l.o.g. } \varepsilon \ge \frac{12(\ln 2)^2}{U} \\
&\le \frac{\varepsilon p}{3\ln 2}|x-y|^p
\end{aligned}$$

◀

To finish the proof of Theorem 2 we observe, that for any position i of output, Algorithm 11 outputs $S''[i]$ such that $\mathbb{E}_r[|(S''[i])^p - (S[i])^p|] \le \frac{p\varepsilon}{3\ln 2}\cdot(S[i])^p$. By Markov's inequality it means that with probability $2/3$ the relative error of $(\ell_p)^p$ approximation is at most $\frac{p}{\ln 2}\cdot\varepsilon$. Thus, by (5) and (7) relative error of ℓ_p approximation is ε with probability at least $2/3$. Now a standard amplification procedure follows: invoke Algorithm 11 independently t times and take the median value from $S''_{(1)}[i], \ldots S''_{(t)}[i]$ as the final estimate $S_\varepsilon[i]$. Taking $t = \Theta(\log n)$ to be large enough makes the final estimate good with high probability, and by the union bound whole S_ε is a good estimate of S. The complexity of the whole procedure is thus $\mathcal{O}(\log n \cdot \log U \cdot \eta^{-1} \cdot n \log m) = \mathcal{O}(p^{-1}\varepsilon^{-1} n \log m \log^2 U \log n)$.

3 Hamming distances

As a final note we comment on a particularly simple form that Algorithm 11 takes for Hamming distances (limit case of $p = 0$).

$$\widehat{g}_i(x,y) = \begin{cases} 1 & \text{if } \|x^{(i)} - y^{(i)}\|_{B_i} = 1 \\ 0 & \text{otherwise,} \end{cases}$$

with Algorithm being simply: pick at random $r \in [1,9]$, apply it multiplicatively to the input, compute text-to-pattern distance using $\sum_i \widehat{g}_i$ function.

Taking a limit of $p \to 0$ in proof of Theorem 2, we reach that bound from Lemma 14 becomes

$$\mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx,ry)|\Big] \le 24\eta$$

and since all other terms in error estimate have multiplicative term p in front, we reach

$$\mathrm{err}(x,y) \le 2\log U \cdot \mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx,ry)|\Big] \le 48\eta\log U.$$

We thus observe that expected relative error in estimation of Hamming distance is: $\mathbb{E}[S''[i] - S[i]] \leq 48\eta \log U \cdot S[i]$. With probability at least 2/3 the relative error is at most $144\eta \log U$. Setting $\eta = \frac{\varepsilon}{144 \log U}$ and repeating the randomized procedure $\Theta(\log n)$ with taking median for concentration completes the algorithm. The total runtime is, by a standard trick of reducing alphabet size to $2m$, $\mathcal{O}(\frac{n}{\varepsilon} \log^2 m \log n)$, and while it compares unfavorably to algorithm from [16] (in terms of runtime), it gives another insight on why $\widetilde{\mathcal{O}}(n/\varepsilon)$ time algorithm is possible for Hamming distance version of pattern matching.

References

1 Karl R. Abrahamson. Generalized String Matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

2 Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi:10.1016/S0196-6774(03)00097-X.

3 Amihood Amir, Ohad Lipsky, Ely Porat, and Julia Umanski. Approximate Matching in the L_1 Metric. In *CPM*, pages 91–103, 2005. doi:10.1007/11496656_9.

4 Amit Chakrabarti and Oded Regev. An Optimal Lower Bound on the Communication Complexity of Gap-Hamming-Distance. *SIAM J. Comput.*, 41(5):1299–1317, 2012. doi:10.1137/120861072.

5 Peter Clifford, Raphaël Clifford, and Costas S. Iliopoulos. Faster Algorithms for δ,γ-Matching and Related Problems. In *CPM*, pages 68–78, 2005. doi:10.1007/11496656_7.

6 Raphaël Clifford. Matrix multiplication and pattern matching under Hamming norm. http://www.cs.bris.ac.uk/Research/Algorithms/events/BAD09/BAD09/Talks/BAD09-Hammingnotes.pdf, 2009. Retrieved March 2017.

7 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana A. Starikovskaya. The k-mismatch problem revisited. In *SODA*, pages 2039–2052. SIAM, 2016.

8 M. J. Fischer and M. S. Paterson. STRING-MATCHING AND OTHER PRODUCTS. Technical report, Massachusetts Institute of Technology, 1974.

9 Sumit Ganguly. Taylor Polynomial Estimator for Estimating Frequency Moments. In *ICALP*, pages 542–553, 2015. doi:10.1007/978-3-662-47672-7_44.

10 Paweł Gawrychowski and Przemysław Uznański. Towards Unified Approximate Pattern Matching for Hamming and L_1 Distance. In *ICALP*, pages 62:1–62:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.62.

11 Daniel Graf, Karim Labib, and Przemysław Uznański. Brief Announcement: Hamming Distance Completeness and Sparse Matrix Multiplication. In *ICALP*, pages 109:1–109:4, 2018. doi:10.4230/LIPIcs.ICALP.2018.109.

12 T. S. Jayram, Ravi Kumar, and D. Sivakumar. The One-Way Communication Complexity of Hamming Distance. *Theory of Computing*, 4(1):129–135, 2008. doi:10.4086/toc.2008.v004a006.

13 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. On the Exact Space Complexity of Sketching and Streaming Small Norms. In *SODA*, pages 1161–1178, 2010. doi:10.1137/1.9781611973075.93.

14 Howard J. Karloff. Fast Algorithms for Approximately Counting Mismatches. *Inf. Process. Lett.*, 48(2):53–60, 1993. doi:10.1016/0020-0190(93)90177-B.

15 Tsvi Kopelowitz and Ely Porat. Breaking the Variance: Approximating the Hamming Distance in $1/\epsilon$ Time Per Alignment. In *FOCS*, pages 601–613, 2015. doi:10.1109/FOCS.2015.43.

16 Tsvi Kopelowitz and Ely Porat. A Simple Algorithm for Approximating the Text-To-Pattern Hamming Distance. In *SOSA*, pages 10:1–10:5, 2018. doi:10.4230/OASIcs.SOSA.2018.10.

17 Yi Li and David P. Woodruff. A Tight Lower Bound for High Frequency Moment Estimation with Small Error. In *APPROX-RANDOM*, pages 623–638, 2013. doi:10.1007/978-3-642-40328-6_43.

18 Ohad Lipsky. Eficient Distance Computations. Master's thesis, Bar-Ilan University. Department of Mathematics and Computer Science., 2003.

19 Ohad Lipsky and Ely Porat. L_1 pattern matching lower bound. *Inf. Process. Lett.*, 105(4):141–143, 2008. doi:10.1016/j.ipl.2007.08.011.

20 Ohad Lipsky and Ely Porat. Approximate Pattern Matching with the L_1, L_2 and L_∞ Metrics. *Algorithmica*, 60(2):335–348, 2011. doi:10.1007/s00453-009-9345-9.

21 Marcin Mucha, Karol Węgrzycki, and Michał Włodarczyk. A Subquadratic Approximation Scheme for Partition. In *SODA*, pages 70–88, 2019. doi:10.1137/1.9781611975482.5.

22 John Nolan. *Stable distributions: models for heavy-tailed data*. Birkhauser New York, 2003.

23 Ely Porat and Klim Efremenko. Approximating general metric distances between a pattern and a text. In *SODA*, pages 419–427, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347128.

24 Jan Studený and Przemysław Uznański. Approximating Approximate Pattern Matching. *CoRR*, abs/1810.01676, 2018. arXiv:1810.01676.

25 David P. Woodruff. Optimal space lower bounds for all frequency moments. In *SODA*, pages 167–175, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982817.

A Omitted proofs

▶ **Lemma 14.** *For* $i \le k = \log(8\eta|x-y|)$ *we have* $\mathbb{E}_{r\in[1,9)}\left[\,|\widehat{g}_i(rx,ry)|\,\right] \le (1152 + 192(\ln 2)))\eta|x-y|^p$.

Proof. First, we define symbols A, B, A', B' to be parts of the $\widehat{g}_i$.

$$A = \|(rx)^{(i)} - (ry)^{(i)}\|_{B_i}$$
$$B = \|(rx)^{(i+1)} - (ry)^{(i+1)}\|_{B_i}$$
$$A' = \max(0, A - 2^i)$$
$$B' = \max(0, B - 2^{i+1})$$

Repeating reasoning from proof of Lemma 7, we get

$$|A' - B'| \le 2\cdot 2^i \tag{9}$$
$$\|rx-ry\|_{B_i} - 2\cdot 2^i \le A' \le \|rx-ry\|_{B_i} \tag{10}$$
$$\|rx-ry\|_{B_i} - 2\cdot 2^{i+1} \le B' \le \|rx-ry\|_{B_i} \tag{11}$$

We also bound $B_i = 2^i/\eta \le 2^k/\eta = 8|x-y|$. Now let's bound the $|\widehat{g}_i(rx,ry)|$. A simple bound that comes from the definition of $\widehat{g}_i$ gives us:

$$|\widehat{g}_i(rx,ry)| = |A'^p - B'^p| \le \max(A'^p, B'^p) \le \|rx-ry\|^p_{B_i}. \qquad \text{(Use of 10,11)} \tag{12}$$

Unfortunately, this bound is not tight enough for larger values of $\|rx-ry\|_{B_i}$, so for $\|rx-ry\|_{B_i} \ge 6\cdot 2^i$, we prove stronger bound:

$$\begin{aligned}
|\widehat{g}_i(rx,ry)| &= |(A')^p - (B')^p| \\
&= \max(A',B')^p - \min(A',B')^p \\
&= \max(A',B')^p - (\max(A',B') - |A'-B'|)^p \\
&= \max(A',B')^p \left(1 - \left(1 - \frac{|A'-B'|}{\max(A',B')}\right)^p\right) \\
&\le \|rx-ry\|^p_{B_i} \cdot (1 - (1 - \frac{2\cdot 2^i}{\|rx-ry\|_{B_i} - 2\cdot 2^i})^p) \\
&\le \|rx-ry\|^p_{B_i} \cdot (1 - (1 - \frac{3\cdot 2^i}{\|rx-ry\|_{B_i}})^p) \\
&\le 6p(\ln 2)\|rx-ry\|^{p-1}_{B_i}\cdot 2^i && (\|rx-ry\|_{B_i} \ge 6\cdot 2^i, \text{ by (6)})
\end{aligned}$$

The *norm* function $\|x\|_{B_i} = \min(x \bmod B_i, B_i - (x \bmod B_i))$ is in fact a triangle wave function varying between 0 and $B_i/2$ with periodicity of B_i. So if the input is a random variable that follows uniform distribution at interval that is larger than its period (in our case B_i), the output has piece-wise uniform distribution, and its probability density function can be bounded by two times the probability density function of the uniform distribution for the whole domain. Formally, if $X = U(a, b)$ with $b - a \geq B_i$ then for $Y = \|X\|_{B_i}$ its probability density function $f_Y(y)$ is:

$$f_Y(y) \leq \frac{2}{B_i/2} \text{ for } 0 \leq y \leq B_i/2 \tag{13}$$

As the input in the expression $\|rx - ry\|_{B_i}$ to the *norm* function is uniformly distributed between $a = |x-y|$ and $b = 9|x-y|$ and $B_i \leq 8|x-y|$, we can use 13 to bound the probability density function of the $Z = \|rx - ry\|_{B_i}$ by $f_Z(y) \leq \frac{2}{B_i/2}$.

Now when we have the approximate probability density function (namely its upper bound) we can condition on the value of $\|rx - ry\|_{B_i}$ to be able to use the bounds for small and large values of $\|rx - ry\|_{B_i}$.

$$\begin{aligned}
&\mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)|\Big] = \\
&\quad = \mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)| \;\Big|\; \|rx - ry\|_{B_i} \leq 6\eta B_i\Big] \Pr_{r\in[1,9)}\Big[\|rx - ry\|_{B_i} \leq 6\eta B_i\Big] + \\
&\qquad + \mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)| \;\Big|\; \|rx - ry\|_{B_i} > 6\eta B_i\Big] \Pr_{r\in[1,9)}\Big[\|rx - ry\|_{B_i} > 6\eta B_i\Big]
\end{aligned}$$

We bound those two summands separately. Now, bound on the first part:

$$\begin{aligned}
&\mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)| \;\Big|\; \|rx - ry\|_{B_i} \leq 6\eta B_i\Big] \Pr_{r\in[1,9)}\Big[\|rx - ry\|_{B_i} \leq 6\eta B_i\Big] \leq \\
&\quad \leq \mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)| \;\Big|\; \|rx - ry\|_{B_i} \leq 6\eta B_i\Big] 24\eta \\
&\quad \leq \mathbb{E}_{r\in[1,9)}\Big[\|rx - ry\|^p_{B_i} \;\Big|\; \|rx - ry\|_{B_i} \leq 6\eta B_i\Big] 24\eta \qquad \text{(by 12)} \\
&\quad \leq (6\eta B_i)^p 24\eta \\
&\quad \leq 24\eta(6 \cdot 2^i)^p \\
&\quad \leq 24 \cdot 6\eta(8\eta|x - y|)^p \\
&\quad \leq 1152\eta|x - y|^p
\end{aligned}$$

And on the second part:

$$\begin{aligned}
&\mathbb{E}_{r\in[1,9)}\Big[|\widehat{g}_i(rx, ry)|\Big|\|rx - ry\|_{B_i} > 6\eta B_i\Big] \Pr_{r\in[1,9)}\Big[\|rx - ry\|_{B_i} > 6\eta B_i\Big] \leq \\
&\quad \leq \mathbb{E}_{r\in[1,9)}\Big[6p(\ln 2)\|rx - ry\|^{p-1}_{B_i} \cdot 2^i \Big| \|rx - ry\|_{B_i} > 6\eta B_i\Big] \cdot \\
&\qquad \cdot \Pr_{r\in[1,9)}\Big[\|rx - ry\|_{B_i} > 6\eta B_i\Big]
\end{aligned}$$

$$\ldots \leq \int_1^9 6p(\ln 2)\|rx-ry\|_{B_i}^{p-1} \cdot 2^i \frac{1}{8} \cdot \mathbf{1}[\|rx-ry\|_{B_i} > 6\eta B_i]\, dr$$

(1/8 is the density of r.v. r, $\mathbf{1}[\cdot]$ is the indicator function)

$$\leq 6p(\ln 2) \cdot 2^i \int_0^{B_i/2} z^{p-1} \frac{2}{B_i/2} \mathbf{1}[z > 6\eta B_i]\, dz$$

(changed to r.v. $z = \|rx-ry\|_{B_i}$)

$$\leq p(\ln 2)\frac{24}{B_i} \cdot 2^i \int_0^{B_i/2} z^{p-1} dz$$
$$\leq (\ln 2)\frac{24}{B_i} \cdot 2^i \left(\frac{B_i}{2}\right)^p$$
$$\leq 24(\ln 2) B_i^{p-1} \cdot 2^i$$
$$\leq 24(\ln 2) 2^{ip} \eta^{-p+1}$$
$$\leq 24(\ln 2)(8\eta|x-y|)^p \eta^{-p+1}$$
$$\leq 192(\ln 2)\eta|x-y|^p$$

So finally, we reach:

$$\begin{aligned}
&\mathbb{E}_{r\in[1,9)}\left[|\widehat{g}_i(rx,ry)|\right] = \\
&\quad = \mathbb{E}_{r\in[1,9)}\left[|\widehat{g}_i(rx,ry)| \middle| \|rx-ry\|_{B_i} \leq 6\eta B_i\right] \Pr_{r\in[1,9)}\left[\|rx-ry\|_{B_i} \leq 6\eta B_i\right] + \\
&\qquad + \mathbb{E}_{r\in[1,9)}\left[|\widehat{g}_i(rx,ry)| \middle| \|rx-ry\|_{B_i} > 6\eta B_i\right] \Pr_{r\in[1,9)}\left[\|rx-ry\|_{B_i} > 6\eta B_i\right] \\
&\quad \leq 1152\eta|x-y|^p + 192(\ln 2)\eta|x-y|^p \\
&\quad \leq (1152 + 192(\ln 2)))\eta|x-y|^p
\end{aligned}$$

◀

Cartesian Tree Matching and Indexing

Sung Gwan Park
Seoul National University, Korea
sgpark@theory.snu.ac.kr

Amihood Amir
Bar-Ilan University, Israel
amir@esc.biu.ac.il

Gad M. Landau
University of Haifa, Israel
New York University, USA
landau@univ.haifa.ac.il

Kunsoo Park[1]
Seoul National University, Korea
kpark@theory.snu.ac.kr

Abstract

We introduce a new metric of match, called *Cartesian tree matching*, which means that two strings match if they have the same Cartesian trees. Based on Cartesian tree matching, we define single pattern matching for a text of length n and a pattern of length m, and multiple pattern matching for a text of length n and k patterns of total length m. We present an $O(n+m)$ time algorithm for single pattern matching, and an $O((n+m)\log k)$ deterministic time or $O(n+m)$ randomized time algorithm for multiple pattern matching. We also define an index data structure called Cartesian suffix tree, and present an $O(n)$ randomized time algorithm to build the Cartesian suffix tree. Our efficient algorithms for Cartesian tree matching use a representation of the Cartesian tree, called the *parent-distance representation*.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Cartesian tree matching, Pattern matching, Indexing, Parent-distance representation

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.16

Acknowledgements S.G. Park and K. Park were supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). A. Amir and G.M. Landau were partially supported by the Israel Science Foundation grant 571/14, and Grant No. 2014028 from the United States-Israel Binational Science Foundation (BSF).

1 Introduction

String matching is one of fundamental problems in computer science, and it can be applied to many practical problems. In many applications string matching has variants derived from exact matching (which can be collectively called *generalized matching*), such as order-preserving matching [19, 20, 22], parameterized matching [4, 7, 8], jumbled matching [9], overlap matching [3], pattern matching with swaps [2], and so on. These problems are characterized by the way of defining a *match*, which depends on the application domains of the problems. In financial markets, for example, people want to find some patterns in

[1] Corresponding author

the time series data of stock prices. In this case, they would like to know more about some pattern of price fluctuations than exact prices themselves [15]. Therefore, we need a definition of match which is appropriate to handle such cases.

The Cartesian tree [27] is a tree data structure that represents an array, only focusing on the results of comparisons between numeric values in the array. In this paper we introduce a new metric of match, called *Cartesian tree matching*, which means that two strings match if they have the same Cartesian trees. If we model the time series stock prices as a numerical string, we can find a desired pattern from the data by solving a Cartesian tree matching problem. For example, let's assume that the pattern we want to find looks like the picture on the left of Figure 1, which is a common pattern called the head-and-shoulder [15] (in fact there are two versions of the head-and-shoulder: one is the picture in Figure 1 and the other is the picture reversed). The picture on the right of Figure 1 is the Cartesian tree corresponding to the pattern on the left. Cartesian tree matching finds every position of the text which has the same Cartesian tree as the picture on the right of Figure 1.

Even though order-preserving matching [19, 20, 22] can also be applied to finding patterns in time series data, Cartesian tree matching may be more appropriate than order-preserving matching in finding patterns. For instance, let's assume that we are looking for the pattern in Figure 1 in time series stock prices. An important characteristic of the pattern is that the price hit the bottom (head), and it has two shoulders before and after the head. But the relative order between the two shoulders (i.e., which one is higher) does not matter. If we model this pattern into order-preserving matching, then order-preserving matching imposes a relative order between two shoulders $S[2]$ and $S[6]$. Moreover, it imposes an unnecessary order between two valleys $S[3]$ and $S[5]$. Hence, order preserving matching may not be able to find such a pattern in time series data. In contrast, the pattern in Figure 1 can be represented by one Cartesian tree, and therefore Cartesian tree matching is a more appropriate metric in such cases.

In this paper we define string matching problems based on Cartesian tree matching: single pattern matching for a text of length n and a pattern of length m, and multiple pattern matching for a text of length n and k patterns of total length m, and we present efficient algorithms for them. We also define an index data structure called Cartesian suffix tree as in the cases of parameterized matching and order-preserving matching [8, 13], and present an efficient algorithm to build the Cartesian suffix tree. To obtain efficient algorithms for Cartesian tree matching, we define a representation of the Cartesian tree, called the *parent-distance representation*.

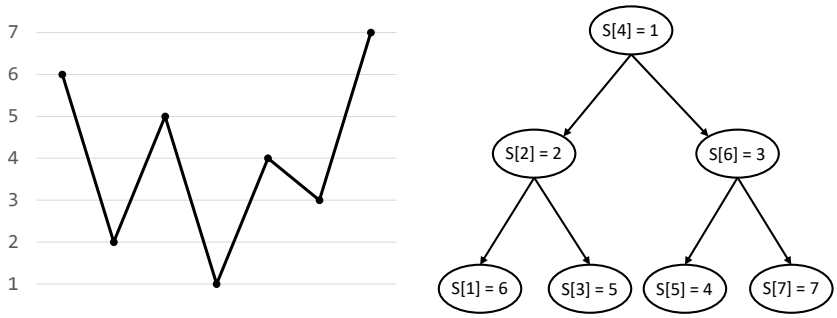


Figure 1 Example pattern $S = (6, 2, 5, 1, 4, 3, 7)$ and its corresponding Cartesian tree.

In Section 2 we give basic definitions for Cartesian tree matching. In Section 3 we propose an $O(n+m)$ time algorithm for single pattern matching. In Section 4 we present an $O((n+m)\log k)$ deterministic time or $O(n+m)$ randomized time algorithm for multiple pattern matching. In Section 5 we define the Cartesian suffix tree, and present an $O(n)$ randomized time algorithm to build the Cartesian suffix tree of a string of length n.

2 Problem Definition

2.1 Basic notations

A *string* is a sequence of characters in an alphabet Σ, which is a set of integers. We assume that the comparison between any two characters can be done in $O(1)$ time. For a string S, $S[i]$ represents the i-th character of S, and $S[i..j]$ represents a substring of S starting from i and ending at j.

2.2 Cartesian tree matching

A string S can be associated with its corresponding Cartesian tree $CT(S)$ according to the following rules [27]:

- If S is an empty string, $CT(S)$ is an empty tree.
- If $S[1..n]$ is not empty and $S[i]$ is the minimum value among S, $CT(S)$ is the tree with $S[i]$ as the root, $CT(S[1..i-1])$ as the left subtree, and $CT(S[i+1..n])$ as the right subtree. If there are two or more minimum values, we choose the leftmost one as the root.

Since each character in string S corresponds to a node in Cartesian tree $CT(S)$, we can treat each character as a node in the Cartesian tree.

Cartesian tree matching is the problem to find all the matches in the text which have the same Cartesian tree as a given pattern. Formally, we define it as follows:

▶ **Definition 1** (Cartesian tree matching). *Given two strings text $T[1..n]$ and pattern $P[1..m]$, find every $1 \le i \le n-m+1$ such that $CT(T[i..i+m-1]) = CT(P[1..m])$.*

For example, let's consider a sample text $T = (41, 36, 15, 8, 41, 23, 28, 16, 26, 22, 56, 29, 12, 61)$. If we find the pattern in Figure 1, which is $P = (6, 2, 5, 1, 4, 3, 7)$, we can find a match at position 5 of the text, i.e., $CT(T[5..11]) = CT(P[1..7])$. Note that the matched text is not a match in order-preserving matching [20, 22] because the relative order between $T[6] = 23$ and $T[10] = 22$ is different from that between $P[2] = 2$ and $P[6] = 3$, but it is a match in Cartesian tree matching.

3 Single Pattern Matching in $O(n+m)$ Time

3.1 Parent-distance representation

In order to solve Cartesian tree matching without building every possible Cartesian tree, we propose an efficient representation to store the information about Cartesian trees, called the *parent-distance representation*.

▶ **Definition 2** (Parent-distance representation). *Given a string $S[1..n]$, the* parent-distance representation *of S is an integer string $PD(S)[1..n]$, which is defined as follows:*

$$PD(S)[i] = \begin{cases} i - \max_{1 \le j < i}\{j : S[j] \le S[i]\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

For example, the parent-distance representation of string $S = (2, 5, 4, 2, 2, 1)$ is $PD(S) = (0, 1, 2, 3, 1, 0)$. Note that $S[j]$ in Definition 2 represents the parent of $S[i]$ in Cartesian tree $CT(S[1..i])$. Furthermore, if there is no such j, $S[i]$ is the root of Cartesian tree $CT(S[1..i])$.

Theorem 3 shows that the parent-distance representation has a one-to-one mapping to the Cartesian tree, so it can substitute the Cartesian tree without any loss of information.

▶ **Theorem 3.** *Two strings S_1 and S_2 have the same Cartesian trees if and only if S_1 and S_2 have the same parent-distance representations.*

Proof. If two strings have different lengths, they have different Cartesian trees and different parent-distance representations, so the theorem holds. Therefore, we can only consider the case where S_1 and S_2 have the same length. Let n be the length of S_1 and S_2. We prove the theorem by an induction on n.

If $n = 1$, S_1 and S_2 will always have the same Cartesian trees with only one node. Furthermore, they will have the same parent-distance representation (0). Therefore, the theorem holds when $n = 1$.

Let's assume that the theorem holds when $n = k$, and show that it holds when $n = k + 1$.

($\Longrightarrow$) Assume that $S_1[1..k + 1]$ and $S_2[1..k + 1]$ have the same Cartesian trees (i.e., $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$). There are two cases.

- If $S_1[k + 1]$ and $S_2[k + 1]$ are not roots of the Cartesian trees, let $S_1[j]$ be the parent of $S_1[k + 1]$, and $S_2[l]$ the parent of $S_2[k + 1]$. Since $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$, we have $j = l$. If we remove $S_1[k+1]$ from Cartesian tree $CT(S_1[1..k+1])$, we obtain the tree $CT(S_1[1..k])$, where the left subtree of $S_1[k + 1]$ is attached to its parent $S_1[j]$. If we remove $S_2[k + 1]$ from $CT(S_2[1..k + 1])$, we obtain $CT(S_2[1..k])$ in the same way. Since $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$, we get $CT(S_1[1..k]) = CT(S_2[1..k])$, and therefore $PD(S_1)[1..k] = PD(S_2)[1..k]$ by induction hypothesis. Since $PD(S_1)[k + 1] = k + 1 - j$ and $PD(S_2)[k + 1] = k + 1 - l$ (and $j = l$), we have $PD(S_1) = PD(S_2)$.
- If $S_1[k+1]$ and $S_2[k+1]$ are roots, we remove $S_1[k+1]$ and $S_2[k+1]$ to get $CT(S_1[1..k])$ and $CT(S_2[1..k])$. Since $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$, we have $CT(S_1[1..k]) = CT(S_2[1..k])$, and therefore $PD(S_1)[1..k] = PD(S_2)[1..k]$ by induction hypothesis. Since $PD(S_1)[k + 1] = PD(S_2)[k + 1] = 0$ in this case, we get $PD(S_1) = PD(S_2)$.

($\Longleftarrow$) Assume that $S_1[1..k + 1]$ and $S_2[1..k + 1]$ have the same parent-distance representations (i.e., $PD(S_1)[1..k + 1] = PD(S_2)[1..k + 1]$). Since $PD(S_1)[1..k] = PD(S_2)[1..k]$, we have $CT(S_1[1..k]) = CT(S_2[1..k])$ by induction hypothesis. From $CT(S_1[1..k])$, we can derive $CT(S_1[1..k + 1])$ as follows. If $PD(S_1)[k + 1] > 0$, let x be $S_1[k + 1 - PD(S_1)[k + 1]]$. We insert $S_1[k + 1]$ into $CT(S_1[1..k])$ so that the parent of $S_1[k + 1]$ is x and the original right subtree of x becomes the left subtree of $S_1[k+1]$. If $PD(S_1)[k+1] = 0$, $S_1[k+1]$ is the root of $CT(S_1[1..k + 1])$ and $CT(S_1[1..k])$ becomes the left subtree of $S_1[k + 1]$. We derive $CT(S_2[1..k + 1])$ from $CT(S_2[1..k])$ in the same way. Since $CT(S_1[1..k]) = CT(S_2[1..k])$ and $PD(S_1)[k + 1] = PD(S_2)[k + 1]$, we can conclude that $CT(S_1[1..k + 1]) = CT(S_2[1..k + 1])$.

Therefore, we have proved that there is a one-to-one mapping between Cartesian trees and parent-distance representations. ◀

3.2 Computing parent-distance representation

Given a string $S[1..n]$, we can compute the parent-distance representation in linear time using a stack, as in [13, 14]. The main idea is that if two characters $S[i]$ and $S[j]$ for $i < j$ satisfy $S[i] > S[j]$, $S[i]$ cannot be the parent of $S[k]$ for any $k > j$. Therefore, we will only store $S[i]$ which does not have such $S[j]$ while scanning from left to right. If we store such $S[i]$ only,

Algorithm 1 Computing parent-distance representation of a string.

```
1:  procedure PARENT-DIST-REP(S[1..n])
2:      ST ← an empty stack
3:      for i ← 1 to n do
4:          while ST is not empty do
5:              (value, index) ← ST.top
6:              if value ≤ S[i] then
7:                  break
8:              ST.pop
9:          if ST is empty then
10:             PD(S)[i] ← 0
11:         else
12:             PD(S)[i] ← i − index
13:         ST.push((S[i], i))
14:     return PD(S)
```

they form a non-decreasing subsequence of S. When we consider a new value, therefore, we can pop values that are larger than the new value, find its parent, and push the new value and its index into the stack. Algorithm 1 describes the algorithm to compute $PD(S)$.

Furthermore, given the parent-distance representation of string S, we can compute the parent-distance representation of any substring $S[i..j]$ easily. To compute $PD(S[i..j])[k]$, we need only check whether the parent of $S[i+k-1]$ is within $S[i..j]$ or not (i.e., the parent is outside if $PD(S)[i+k-1] \geq k$).

$$PD(S[i..j])[k] = \begin{cases} 0 & \text{if } PD(S)[i+k-1] \geq k \\ PD(S)[i+k-1] & \text{otherwise.} \end{cases} \tag{1}$$

For example, the parent-distance representation of string $S = (2, 7, 5, 6, 4, 3, 1)$ is $PD(S) = (0, 1, 2, 1, 4, 5, 0)$. For $PD(S[2..7])$, we can use the above equation and compute the value at each position in constant time, getting $PD(S[2..7]) = (0, 0, 1, 0, 0, 0)$.

3.3 Failure function

We can define a failure function similar to the one used in the KMP algorithm [21].

▶ **Definition 4** (Failure function). *The* failure function π *of string* P *is an integer string such that:*

$$\pi[q] = \begin{cases} \max\{k : CT(P[1..k]) = CT(P[q-k+1..q]) \text{ for } 1 \leq k < q\} & \text{if } q > 1 \\ 0 & \text{if } q = 1 \end{cases}$$

That is, $\pi[q]$ is the largest k such that the prefix and the suffix of $P[1..q]$ of length k have the same Cartesian trees. For example, assuming that $P = (5, 7, 4, 6, 1, 3, 2)$, the corresponding failure function is $\pi = (0, 1, 1, 2, 3, 4, 1)$. We can see that $CT(P[1..4]) = CT(P[3..6])$ from $\pi[6] = 4$. We will present an algorithm to compute the failure function of a given string in Section 3.5.

Algorithm 2 Text search of Cartesian tree matching.

```
1: procedure CARTESIAN-TREE-MATCH(T[1..n], P[1..m])
2:     PD(P) ← PARENT-DIST-REP(P)
3:     π ← FAILURE-FUNC(P)
4:     len ← 0
5:     DQ ← an empty deque
6:     for i ← 1 to n do
7:         Pop elements (value, index) from back of DQ such that value > T[i]
8:         while len ≠ 0 do
9:             if PD(T[i − len..i])[len + 1] = PD(P)[len + 1] then
10:                break
11:            else
12:                len ← π[len]
13:                Delete elements (value, index) from front of DQ such that index < i − len
14:        len ← len + 1
15:        DQ.push_back((T[i], i))
16:        if len = m then
17:            print "Match occurred at i − m + 1"
18:            len ← π[len]
19:            Delete elements (value, index) from front of DQ such that index ≤ i − len
```

3.4 Text search

As in the original KMP text search algorithm, we can use the failure function in order to achieve linear time text search: scan the text from left to right, and use the failure function every time we find a mismatch between the text and the pattern. We apply this idea to Cartesian tree matching.

In order to perform a text search using $O(m)$ space, we compute the parent-distance representation of the text *online* as we read the text, so that we don't need to store the parent-distance representation of the whole text, which would cost $O(n)$ space. Furthermore, among the text characters which are matched with the pattern, we only have to store elements that form a non-decreasing subsequence by using a *deque* (instead of a stack in Section 3.2) in order to delete elements in front. Using this idea, we can keep the size of the deque to be always smaller than or equal to m. Therefore, we can perform the text search using $O(m)$ space. Algorithm 2 shows the text search algorithm of Cartesian tree matching. In line 9 we need to compute $x = PD(T[i-len..i])[len+1]$. If the deque is empty, then $x = 0$. Otherwise, let $(value, index)$ be the element at the back of the deque. Then $x = i - index$. This computation takes constant time. Just before line 14, we do not compare $PD(T[i])$ and $PD(P)[1]$ when $len = 0$, because they always match. Therefore, we can safely perform line 14.

3.5 Computing failure function

We compute the failure function π in a way similar to the text search, as in the KMP algorithm. However, we can compute the parent-distance representation of the pattern in $O(m)$ time before we compute the failure function. Hence we don't need a deque and the computation is slightly simpler than text search. Algorithm 3 shows the procedure to compute the failure function.

Algorithm 3 Computing failure function in Cartesian tree matching.

```
1:  procedure FAILURE-FUNC(P[1..m])
2:      PD(P) ← PARENT-DIST-REP(P)
3:      len ← 0
4:      π[1] ← 0
5:      for i ← 2 to m do
6:          while len ≠ 0 do
7:              if PD(P[i − len..i])[len + 1] = PD(P[1..len + 1])[len + 1] then
8:                  break
9:              else
10:                 len ← π[len]
11:         len ← len + 1
12:         π[i] ← len
```

3.6 Correctness and time complexity

Since our algorithm for Cartesian tree matching including text search and the computation of the failure function follow the KMP algorithm, it is easy to see that our algorithm correctly finds all occurrences (in the sense of Cartesian tree matching) of the pattern in the text. Since our algorithm checks one character of the parent-distance representation in constant time, it takes $O(n)$ time for text search and $O(m)$ time to compute the failure function, as in KMP algorithm. Therefore, our algorithm requires $O(m + n)$ time for Cartesian tree matching using $O(m)$ space.

3.7 Cartesian tree signature

There is an alternative representation of Cartesian trees, called *Cartesian tree signature* [14]. The Cartesian tree signature of $S[1..n]$ is an array $L[1..n]$ such that $L[i]$ equals the number of the elements popped from the stack in the i-th iteration of Algorithm 1. Furthermore, the Cartesian tree signature can be represented as a bit string $1^{L[1]}01^{L[2]}0 \cdots 1^{L[n]}0$ of length less than $2n$, which is a succinct representation of a Cartesian tree. For example, the Cartesian tree signature of string $S = (2, 7, 5, 6, 4, 3, 1)$ is $L = (0, 0, 1, 0, 2, 1, 2)$, and its corresponding bit string is 0010011010110.

We can use this representation to perform Cartesian tree matching. While we compute the Cartesian tree signature, we store one more array $D[1..n]$, which is defined as follows: If $S[i]$ is never popped out from the stack, $D[i] = 0$. Otherwise, let $S[j]$ be the value which popped $S[i]$ out from the stack, and $D[i] = j - i$. For string $S = (2, 7, 5, 6, 4, 3, 1)$, we have $D = (6, 1, 2, 1, 1, 1, 0)$.

Using array D, we can delete one character at the front of string $S[1..n]$ in constant time. In order to get Cartesian tree signature L' and its corresponding D' for $S[2..n]$, we do the following: If $D[1] > 0$, we decrease $L[D[1] + 1]$ by one and erase $L[1]$ from L. If $D[1] = 0$, we just erase $L[1]$. After that, we delete $D[1]$ from D to get D'. For example, if we want to delete one character at the front of $S = (2, 7, 5, 6, 4, 3, 1)$, we decrease $L[D[1] + 1] = L[7]$ by one, and delete $L[1]$ and $D[1]$. This results in $L' = (0, 1, 0, 2, 1, 1)$ and $D' = (1, 2, 1, 1, 1, 0)$. These arrays are the correct Cartesian tree signature and its corresponding array D of $S[2..7] = (7, 5, 6, 4, 3, 1)$. In this way, we can perform Algorithm 2 using the Cartesian tree signature. Computing the failure function can also be done in a similar way.

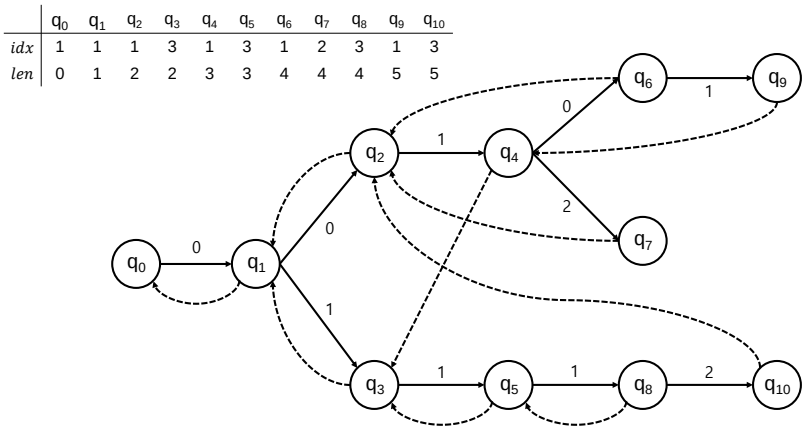


Figure 2 Aho-Corasick automaton for $P_1 = (4, 2, 3, 1, 5), P_2 = (3, 1, 4, 2), P_3 = (1, 2, 3, 5, 4)$.

Note that the Cartesian tree signature can represent a Cartesian tree using less space than the parent-distance representation, but it needs an auxiliary array D to perform string matching, which uses the same space as the parent-distance representation. For Cartesian tree matching, therefore, it uses more space than Algorithm 2.

4 Multiple Pattern Matching in $O((n + m) \log k)$ Time

In this section we extend Cartesian tree matching to the case of multiple patterns. Definition 5 gives the formal definition of multiple pattern matching.

▶ **Definition 5** (Multiple pattern Cartesian tree matching). *Given a text $T[1..n]$ and patterns $P_1[1..m_1], P_2[1..m_2], ..., P_k[1..m_k]$, where $m = m_1 + m_2 + \cdots + m_k$,* multiple pattern Cartesian tree matching *is to find every position in the text which matches at least one pattern, i.e., it has the same Cartesian tree as that of at least one pattern.*

We modify the Aho-Corasick algorithm [1] using the parent-distance representation defined in Section 3.1 to do multiple pattern matching in $O((n + m) \log k)$ time.

4.1 Constructing the Aho-Corasick automaton

Instead of using the patterns themselves in the Aho-Corasick automaton, we use their parent-distance representations to make an automaton. Each node in the automaton corresponds to the prefix of the parent-distance representation of some pattern. We maintain two integers idx and len for every node such that the node corresponds to the parent-distance representation of the pattern prefix $P_{idx}[1..len]$. If there are more than one possible indexes, we store the smallest one. Each node also has a state transition function $trans(x)$, which gets an integer x as an input and returns the next node, or report that there is no such node. We can construct the trie and the state transition function for every node in $O(m \log k)$ time, assuming that we use a balanced binary search tree to implement the transition function. Figure 2 shows an Aho-Corasick automaton for three patterns $P_1 = (4, 2, 3, 1, 5), P_2 = (3, 1, 4, 2), P_3 = (1, 2, 3, 5, 4)$, where we use the parent-distance representations of the patterns, $PD(P_1) = (0, 0, 1, 0, 1), PD(P_2) = (0, 0, 1, 2), PD(P_3) = (0, 1, 1, 1, 2)$ to construct the automaton.

Algorithm 4 Computing failure function in multiple pattern matching.

```
1:  procedure MULTIPLE-FAILURE-FUNC(P_1, P_2, ..., P_k)
2:      for i ← 1 to k do
3:          PD(P_i) ← PARENT-DIST-REP(P_i)
4:      TR ← Build trie with PD(P_i)'s
5:      for node ← breadth-first traversal of the trie do
6:          len ← len[node]
7:          idx ← idx[node]
8:          π[node] ← TR.root
9:          ptr ← parent of node in the trie
10:         while ptr ≠ TR.root do
11:             ptr ← π[ptr]
12:             plen ← len[ptr]
13:             x ← PD(P_idx[len − plen..len])[plen + 1]
14:             if ptr.trans(x) exists then
15:                 π[node] ← ptr.trans(x)
16:                 break
```

The failure function π of the Aho-Corasick automaton is defined as follows: Let q_i be a node in the automaton, and s_i be the substring that node q_i represents in the trie. Let s_j be the longest proper suffix of s_i which matches (in the sense of Cartesian tree matching) prefix s_k of some pattern P_k. The failure function of q_i is defined as node q_k (i.e., $\pi[q_i] = q_k$). The dotted lines in Figure 2 shows the failure function of each node. For example, node q_7 represents $P_2[1..4]$, and its failure function q_2 represents $P_2[1..2]$. We can see that $P_2[1..2]$ matches $P_2[3..4]$ (i.e., $PD(P_2[1..2]) = PD(P_2[3..4]) = (0, 0)$), which is the longest proper suffix of $P_2[1..4]$ that matches a prefix of some pattern. Note that the parent-distance representation of s_k may not be the suffix of the parent-distance representation of s_i. For example, q_7 has the parent-distance representation $(0, 0, 1, 2)$, but its failure function q_2 has the parent-distance representation $(0, 0)$ which is not a suffix of $(0, 0, 1, 2)$.

Algorithm 4 computes the failure function of the trie. As in the original Aho-Corasick algorithm, we traverse the trie with breadth-first order (except the root) and compute the failure function. The main difference between Algorithm 4 and the Aho-Corasick algorithm is at line 13, where we decide the next character to match. According to the definition of the trie, *node* corresponds to the parent-distance representation of $P_{idx}[1..len]$, and so the parent of *node* corresponds to the parent-distance representation of $P_{idx}[1..len - 1]$. In the while loop from line 10 to 16, *ptr* corresponds to the parent-distance representation of some suffix of $P_{idx}[1..len - 1]$, because *ptr* is a node that can be reached from the parent of *node* following the failure links. Since *ptr* corresponds to some string of length *plen*, we can conclude that *ptr* represents $P_{idx}[len - plen..len - 1]$. We want to check whether $P_{idx}[len - plen..len]$ matches some node in the trie, so we should check whether *ptr* has the transition using $x = PD(P_{idx}[len - plen..len])[plen + 1]$. If *ptr* has the transition $ptr.trans(x)$, it corresponds to $P_{idx}[len - plen..len]$, and we can conclude that $\pi[node] = ptr.trans(x)$. If *ptr* doesn't have such a transition, there is no node that represents $P_{idx}[len - plen..len]$, and thus we have to continue the loop.

For example, suppose that we compute the failure function of q_7 in Figure 2. From $idx[q_7] = 2$ and $len[q_7] = 4$, we know that q_7 represents $P_2[1..4]$, and so q_4, which is the parent of q_7, represents $P_2[1..3]$. We begin the while loop starting from $ptr = \pi[q_4] = q_3$.

Since $len[q_3] = 2$, we know that q_3, which represents $P_3[1..2]$, matches $P_2[2..3]$. In order to check whether $P_2[2..4]$ matches some node in the trie, we compute $x = PD(P_2[2..4])[3] = 2$ and check whether $q_3.trans(x)$ exists. Since there is no such transition, we continue the while loop with $ptr = \pi[q_3] = q_1$. We know that q_1, which represents $P_1[1..1]$, matches $P_2[3..3]$ from $len[q_1] = 1$. In order to check whether $P_2[3..4]$ matches some node, we compute $x = PD(P_2[3..4])[2] = 0$ and check whether $q_1.trans(x)$ exists. Since there is such a transition, we conclude that $\pi[q_7] = q_1.trans(0) = q_2$. Note that x may change during the while loop, which is not the case in the Aho-Corasick algorithm.

While computing the failure function, we can also compute the output function in the same way as the Aho-Corasick algorithm. The output function of node q_i is the set of patterns which match some suffix of s_i. This function is used to output all possible matches at the node.

4.2 Multiple pattern matching

Using the automaton defined above, we can solve multiple pattern Cartesian tree matching in $O(n \log k)$ time. The text search algorithm is essentially the same as that of the Aho-Corasick algorithm, following the trie and using the failure links in case of any mismatches. As in the single pattern case, we compute the parent-distance representation of the text online in the same way as Algorithm 2 (using a deque) to ensure $O(m)$ space. The time complexity of our multiple pattern Cartesian tree matching is $O((n + m) \log k)$ using $O(m)$ space, where the $\log k$ factor is included due to the binary search tree in each node. Since there can be at most k outgoing edges from each node, we can perform an operation in the binary search tree in $O(\log k)$ time. Combined with the time-complexity analysis of the Aho-Corasick algorithm, this shows that our algorithm has the time complexity of $O((n + m) \log k)$. We can reduce the time complexity further to randomized $O(n+m)$ time by using a hash instead of a binary search tree [12].

5 Cartesian Suffix Tree in Randomized $O(n)$ Time

In this section we apply the notion of Cartesian tree matching to the suffix tree as in the cases of parameterized matching and order-preserving matching [8, 13]. We first define the Cartesian suffix tree, and show that it can be built in randomized $O(n)$ time or worst-case $O(n \log n)$ time using the result from Cole and Hariharan [12].

5.1 Defining Cartesian suffix tree

The Cartesian suffix tree is an index data structure that allows us to find an occurrence of a given pattern $P[1..m]$ in randomized $O(m)$ time or worst-case $O(m \log n)$ time, where n is the length of the text string. In order to store the information of Cartesian suffix trees efficiently, we again use the parent-distance representation from Section 3.1. Definition 6 gives the formal definition of the Cartesian suffix tree.

▶ **Definition 6** (Cartesian suffix tree). *Given a string $T[1..n]$, the* Cartesian suffix tree *of T is a compacted trie built with $PD(T[i..n]) \cdot (-1)$ for every $1 \leq i \leq n$ (where the special character -1 is concatenated to the end of $PD(T[i..n])$) and string (-1).*

Note that we append a special character -1 to the end of each parent-distance representation to ensure that no string is a prefix of another string.

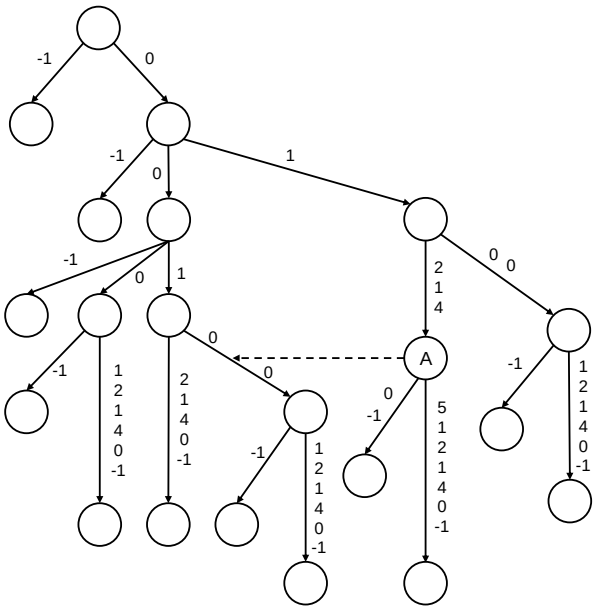


Figure 3 Cartesian suffix tree of $S = (2, 7, 5, 6, 4, 3, 11, 9, 10, 8, 1)$.

Figure 3 shows an example Cartesian suffix tree of $T = (2, 7, 5, 6, 4, 3, 11, 9, 10, 8, 1)$. Each edge actually stores the suffix number, start position, and end position instead of the parent-distance representation itself. For example, node A corresponds to substring $T[1..5]$ or $T[6..10]$, whose parent-distance representation is $PD(T[1..5]) = PD(T[6..10]) = (0, 1, 2, 1, 4)$. Hence, the edge that goes into node A stores the suffix number 1 or 6, start and end positions 3 and 5.

5.2 Constructing Cartesian suffix tree

There are several algorithms efficiently constructing the suffix tree, such as McCreight's algorithm [24] and Ukkonen's algorithm [26]. However, the *distinct right context property* [16, 8] should hold in order to apply these algorithms, which means that the suffix link of every internal node should point to an explicit node. The Cartesian suffix tree does not have the distinct right context property. In Figure 3, the internal node marked with A does not satisfy this property because $PD(T[2..6]) = PD(T[7..11]) = (0, 0, 1, 0, 0)$ and thus there is no explicit node corresponding to parent-distance representation $(0, 0, 1, 0)$.

In order to handle this issue, we use an algorithm due to Cole and Hariharan [12]. This algorithm can construct a compacted trie for a *quasi-suffix collection*, which satisfies the following properties:

1. A quasi-suffix collection is a set of n strings $s_1, s_2, ..., s_n$, where the length of s_i is $n+1-i$.
2. For any two different strings s_i and s_j, s_i should not be a prefix of s_j.
3. For any i and j, if s_i and s_j have a common prefix of length l, s_{i+1} and s_{j+1} should have a common prefix of length at least $l - 1$.

A collection of parent-distance representations for the Cartesian suffix tree satisfies all of the above properties. The first two properties are trivial. Furthermore, if $s_i = PD(T[i..n]) \cdot (-1)$ and $s_j = PD(T[j..n]) \cdot (-1)$ have a common prefix of length l, i.e., $PD(T[i..i + l - 1]) = PD(T[j..j + l - 1])$, we can show that $PD(T[i + 1..i + l - 1]) =$

$PD(T[j+1..j+l-1])$ by Equation 1. Therefore, $s_{i+1} = PD(T[i+1..n]) \cdot (-1)$ and $s_{j+1} = PD(T[j+1..n]) \cdot (-1)$ have a common prefix of length $l-1$ or more, showing the third property holds.

One more property we need to perform Cole and Hariharan's algorithm is a *character oracle*, which returns the i-th character of s_j in constant time. We can do this in constant time using Equation 1, once the parent-distance representation of T is computed.

Since we have all properties needed to perform Cole and Hariharan's algorithm, we can construct a Cartesian suffix tree in randomized $O(n)$ time using $O(n)$ space [12]. In the worst case, it can be built in $O(n \log n)$ time by using a binary search tree instead of a hash table to store the children of each node in the suffix tree, because the alphabet size $|\Sigma|$ is $O(n)$. We can also modify our algorithm to construct a Cartesian suffix tree online, using the idea in [23, 25].

6 Conclusion

We have defined Cartesian tree matching and the parent-distance representation of a Cartesian tree. We developed a linear time algorithm for single pattern matching and an $O((n+m)\log k)$ deterministic time or $O(n+m)$ randomized time algorithm for multiple pattern matching. Finally, we defined an index data structure called Cartesian suffix tree, and showed that it can be constructed in $O(n)$ randomized time. We believe that the notion of Cartesian tree matching, which is a new metric on string matching and indexing over numeric strings, can be used in many applications.

There have been many works on approximate generalized matching. For example, there are results for approximate order-preserving matching [11], approximate jumble matching [10], approximate swapped matching [5], and approximate parameterized matching [6, 18]. There are also results on computing the period of a generalized string, such as computing the period in the order-preserving model [17]. Since Cartesian tree matching is first introduced in this paper, many problems including approximate matching and computing the period in the Cartesian tree matching model are future research topics.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 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, 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.

4 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.

5 Amihood Amir, Moshe Lewenstein, and Ely Porat. Approximate swapped matching. *Inf. Process. Lett.*, 83(1):33–39, 2002. doi:10.1016/S0020-0190(01)00302-7.

6 Alberto Apostolico, Péter L. Erdös, and Moshe Lewenstein. Parameterized matching with mismatches. *J. Discrete Algorithms*, 5(1):135–140, 2007. doi:10.1016/j.jda.2006.03.014.

7 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993. doi:10.1145/167088.167115.

8 Brenda S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997. doi:10.1137/S0097539793246707.

9 Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Algorithms for Jumbled Pattern Matching in Strings. *Int. J. Found. Comput. Sci.*, 23(2):357–374, 2012. doi:10.1142/S0129054112400175.

10 Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On Approximate Jumbled Pattern Matching in Strings. *Theory Comput. Syst.*, 50(1):35–51, 2012. doi:10.1007/s00224-011-9344-5.

11 Tamanna Chhabra, Emanuele Giaquinta, and Jorma Tarhio. Filtration Algorithms for Approximate Order-Preserving Matching. In *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, pages 177–187, 2015. doi:10.1007/978-3-319-23826-5_18.

12 Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 407–415, 2000. doi:10.1145/335305.335352.

13 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.

14 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.

15 Tak-Chung Fu, Korris Fu-Lai Chung, Robert Wing Pong Luk, and Chak-man Ng. Stock time series pattern matching: Template-based vs. rule-based approaches. *Eng. Appl. of AI*, 20(3):347–364, 2007. doi:10.1016/j.engappai.2006.07.003.

16 Raffaele Giancarlo. A Generalization of the Suffix Tree to Square Matrices, with Applications. *SIAM J. Comput.*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.

17 Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny M. Shur, and Tomasz Walen. String Periods in the Order-Preserving Model. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 38:1–38:16, 2018. doi:10.4230/LIPIcs.STACS.2018.38.

18 Carmit Hazay, Moshe Lewenstein, and Dina Sokol. Approximate Parameterized Matching. In *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 414–425, 2004. doi:10.1007/978-3-540-30140-0_38.

19 Jinil Kim, Amihood Amir, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. On Representations of Ternary Order Relations in Numeric Strings. *Mathematics in Computer Science*, 11(2):127–136, 2017. doi:10.1007/s11786-016-0282-0.

20 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

21 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.

22 Marcin Kubica, Tomasz Kulczynski, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013. doi:10.1016/j.ipl.2013.03.015.

23 Taehyung Lee, Joong Chae Na, and Kunsoo Park. On-line construction of parameterized suffix trees for large alphabets. *Inf. Process. Lett.*, 111(5):201–207, 2011. doi:10.1016/j.ipl.2010.11.017.

24 Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

25 Joong Chae Na, Raffaele Giancarlo, and Kunsoo Park. On-Line Construction of Two-Dimensional Suffix Trees in $O(n^2 \log n)$ Time. *Algorithmica*, 48(2):173–186, 2007. doi:10.1007/s00453-007-0063-x.

26 Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

27 Jean Vuillemin. A Unifying Look at Data Structures. *Commun. ACM*, 23(4):229–239, 1980. doi:10.1145/358841.358852.

Indexing the Bijective BWT

Hideo Bannai
Department of Informatics, Kyushu University, Fukuoka, Japan
bannai@inf.kyushu-u.ac.jp

Juha Kärkkäinen
Helsinki Institute of Information Technology (HIIT), Finland
juha.karkkainen@cs.helsinki.fi

Dominik Köppl
Department of Informatics, Kyushu University, Japan Society for Promotion of Science (JSPS)
dominik.koeppl@inf.kyushu-u.ac.jp

Marcin Piątkowski
Nicolaus Copernicus University, Toruń, Poland
marcin.piatkowski@mat.umk.pl

Abstract

The Burrows-Wheeler transform (BWT) is a permutation whose applications are prevalent in data compression and text indexing. The *bijective BWT* is a bijective variant of it that has not yet been studied for text indexing applications. We fill this gap by proposing a self-index built on the bijective BWT. The self-index applies the backward search technique of the FM-index to find a pattern P with $\mathcal{O}(|P| \lg |P|)$ backward search steps.

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

Keywords and phrases Burrows-Wheeler Transform, Lyndon words, Text Indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.17

Funding This research received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941.
Dominik Köppl: JSPS KAKENHI Grant Number JP18F18120.

1 Introduction

The Burrows-Wheeler transform (BWT) [6] is a transformation permuting all symbols of a given string T. It is obtained by sorting all cyclic rotations (conjugates) of T with respect to the lexicographical order and writing the last character of the i-th sorted cyclic rotation in a linear manner from $i = 1$ to $i = |T|$. The BWT tends to group identical characters together. All cyclic rotations of a given string share the same BWT. However, there are strings that are not the BWT of any string (e.g., `bccaab` cannot be reversed). A variant, called the bijective BWT [15], is a *bijective* transformation. It is based on the Lyndon factorization [7] of the input string. In this variant, the output consists of the last symbols of the lexicographically sorted cyclic rotations of all Lyndon factors of the input. Since the Lyndon factorization of a string is uniquely defined, the bijective BWT induces a bijection between strings of a given length n and multisets of Lyndon words of total length n.

In the following, we call the BWT *traditional* to ease the distinguishability of both transformations. It is well known that the traditional BWT has many applications in data compression [1] and text indexing [9, 10, 11]. For the latter, the algorithms for pattern searching are based on the backward search [20]: given a pattern P and the traditional BWT

of T, the occurrences of P in a text T can be computed with $\mathcal{O}(|P|)$ backward search steps. In this light, one may ask whether it is possible to build similar index data structures by exchanging the traditional BWT with the bijective BWT.

In this article, we answer affirmatively the above question: We show that searching a pattern P on the bijective BWT can be conducted with $\mathcal{O}(|P|\hat{p})$ backward search steps, where $\hat{p}$ is the number of distinct factors in the Lyndon factorization of the longest pre-Lyndon suffix of P, where $\hat{p}$ is known to be in $\mathcal{O}(\lg|P|)$ [13]. Thus, we can reduce the number of backward search steps to $\mathcal{O}(|P|\lg|P|)$.

Our results are based on combinatoric properties of Lyndon words and the bijective BWT. They may have applications in distributed implementations of the BWT index [14] or in practical database systems storing dynamic yet compressed data [4, 5].

2 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. We write $[\mathsf{b}(I)..\mathsf{e}(I)] = I$ for an interval I of natural numbers.

2.1 Strings

Let Σ denote a finite alphabet. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. Given an integer j with $1 \leq j \leq |T|$, we access the j-th character of T with $T[j]$. Concatenating a string $T \in \Sigma^*$ k times is abbreviated by T^k. A *bit vector* is a string on the binary alphabet $\{0, 1\}$.

When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X, Y and Z are called a *prefix*, *substring* and *suffix* of T, respectively; A prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. For two integers i, j with $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. If $i > j$, then $T[i..j]$ is the empty string. In particular, the suffix starting at position j of T is called the *j-th suffix* of T, and denoted with $T[j..]$. An occurrence of a substring S in T is treated as a sub-interval of $[1..|T|]$ such that $S = T[\mathsf{b}(S)..\mathsf{e}(S)]$.

The *longest common prefix (LCP)* of two strings S and T is the longest string that is a prefix of both S and T. The length of the LCP of two strings S and T is given by the function $\mathsf{lcp}(S, T)$ returning an integer ℓ such that $T[1..\ell] = S[1..\ell]$ and either (a) $T[\ell+1] \neq S[\ell+1]$ or (b) $\ell = \min(|T|, |S|)$ holds.

Lexicographic Order. We denote the *lexicographic order* with $\prec$. Given two string S and T, then $S \prec T$ if S is a prefix of T or there exists an integer ℓ with $1 \leq \ell \leq \min(|S|, |T|)$ such that $S[1..\ell-1] = T[1..\ell-1]$ and $S[\ell] < T[\ell]$. We write $S \prec_\omega T$ if the infinite concatenation $S^\omega := SSS\cdots$ is lexicographically smaller than $T^\omega := TTT\cdots$. For instance, $\mathtt{ab} \prec \mathtt{aba}$ but $\mathtt{aba} \prec_\omega \mathtt{ab}$.

Support Data Structures. Given a string $T \in \Sigma^*$, a character $\mathsf{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathsf{rank}_\mathsf{c}(j)$ counts the occurrences of c in $T[1..j]$, and the *select* query $T.\mathsf{select}_\mathsf{c}(j)$ gives the position of the j-th c in T. We stipulate that $\mathsf{rank}_\mathsf{c}(0) = \mathsf{select}_\mathsf{c}(0) = 0$.

2.2 Lyndon Words

Given a string $T = T[1..n]$, its i-th *conjugate* $\mathsf{conj}_i(T)$ is defined as $T[i+1..n]T[1..i]$ for an integer i with $0 \leq i \leq n-1$. We say that T and every of its conjugates belongs to the *conjugate class* $\mathsf{conj}(T) := \{\mathsf{conj}_0(T), \ldots, \mathsf{conj}_{n-1}(T)\}$. If a conjugate class contains *exactly*

one conjugate that is lexicographically smaller than all other conjugates, then this conjugate is called a *Lyndon word* [16]. Equivalently, a string T is said to be a Lyndon word if and only if $T \prec S$ for every proper suffix S of T. A consequence is that a Lyndon word is border-free, i.e., there is no Lyndon word $T = SUS$ with $S \in \Sigma^+$ and $U \in \Sigma^*$. A *pre-Lyndon* word is a string that is a prefix of a Lyndon word.

The *Lyndon factorization* [7] of $T \in \Sigma^+$ is the factorization of T into a sequence of lexicographically non-increasing Lyndon words $T_1 \cdots T_t$, where (a) each $T_x \in \Sigma^+$ is a Lyndon word, and (b) $T_x \geq T_{x+1}$ for each $1 \leq x < t$.

▶ **Lemma 1** ([8, Algo. 2.1]). *The Lyndon-factorization of a string can be computed in linear time.*

Each Lyndon word T_x is called a *Lyndon factor*. We denote the multiset of T's Lyndon factors by $\mathsf{LynF}(T) := \{T_1, \ldots, T_t\}$. There is a bijection between $\mathsf{LynF}(T)$ and T in such a sense that $\mathsf{LynF}(T)$ uniquely defines T. That is because we can restore T by

1. sorting the Lyndon factors of $\mathsf{LynF}(T)$ in lexicographically descending order, and
2. subsequently concatenating them.

The last factor T_t is special, as it has the following property:

▶ **Lemma 2** ([8, Prop. 1.9]). *The last Lyndon factor of a string T is the smallest suffix of T.*

We borrow from [13, Sect. 2.2] the notation $\mathrm{lfs}_T(j) := T_j \cdots T_t$ for the suffix of T starting with the j-th Lyndon factor. For what follows, we fix a string $T[1..n]$ over an alphabet Σ with size σ. We use the string $T := \mathtt{acababdababcababbab}$ as our running example. Its Lyndon factorization is $\mathsf{LynF}(T) = \{\mathtt{ac}, \mathtt{ababd}, \mathtt{ababc}, \mathtt{ababb}, \mathtt{ab}\}$. The suffix $\mathrm{lfs}_T(4)$ is $T_4T_5 = \mathtt{ababbab}$.

2.3 Bijective Burrows-Wheeler transform

We denote the bijective BWT of T by BBWT, where $\mathsf{BBWT}[i]$ is the last character of the i-th string in the list storing the conjugates of all Lyndon factors $T_1, \ldots, T_t$ of T sorted with respect to $\prec_\omega$. Figure 1 shows BBWT for our running example.

Gill and Scott [12] and Mantaci et al. [19] postulated that BBWT can be built in linear time. However, we could only verify an algorithm running in $\mathcal{O}(n \lg n / \lg\lg n)$ time in the word RAM model with $\sigma = n^{\mathcal{O}(1)}$, provided by Bonomo et al. [3]. This algorithm processes all Lyndon factors in their lexicographical order, starting with the largest one. Since the algorithm processes all Lyndon factors in text order, we can use it to build BBWT *online*, i.e., we can build BBWT up to the Lyndon factor of a string T that starts with the longest pre-Lyndon suffix of the currently read T. That is because all previous Lyndon factors cannot change when appending characters to T.

▶ **Theorem 3.** *The* BBWT *can be constructed* online *in $\mathcal{O}(n \lg n / \lg\lg n)$ time for a string of length n.*

This is an interesting property, since the only known online technique [23, 22] for computing the traditional BWT needs the text to be given in reversed order (starting with the last character).

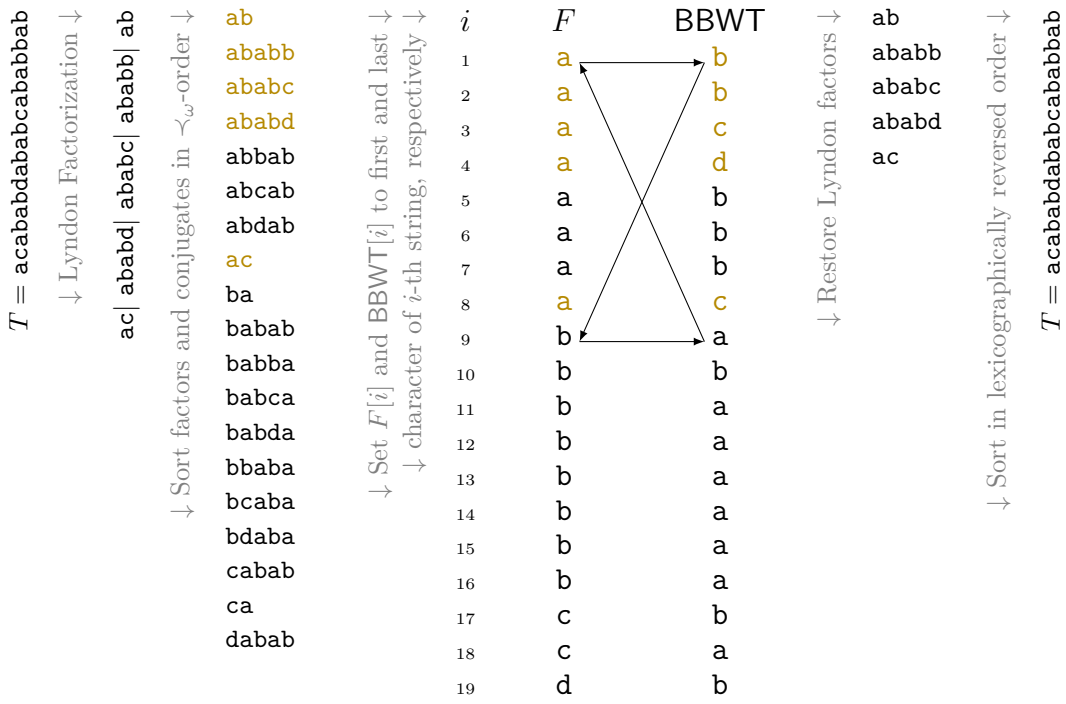


Figure 1 Constructing BBWT and restoring the original input. The Lyndon factors are colored in dark yellow. Middle: Restoring the Lyndon factor `ab` with the backward search, where the array F is defined by $F[i] := \mathtt{c}$ if $C[c-1]+1 \leq i \leq C[c]$. Right: Lyndon factors of T restored by visiting all cycles of BBWT.

3 Backward Search Algorithm

For finding patterns, our index applies the same backward search as the FM-index [9], which we briefly review. Prior to that, we define some necessary data structures:

Text Data Structures. Let SA and ISA denote the *suffix array* [18] and the *inverse suffix array* of T, respectively. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[\mathsf{SA}[i]..] \prec T[\mathsf{SA}[i+1]..]$ for all integers i with $1 \leq i \leq n-1$. The *Burrows-Wheeler transform (BWT)* [6] of T is the string BWT with $\mathsf{BWT}[i] = T[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ otherwise, for every i with $1 \leq i \leq n$.

FM-Index. The FM-index uses BWT with the following auxiliary data structures:

- an array C with $\sigma \lg n$ bits, where $C[c]$ is the number of occurrences of those characters in T that are smaller than c (for each character c with $1 \leq c \leq \sigma$), and
- a data structure that supports rank queries on BWT.

Given a pattern P whose characters are drawn from Σ, the occurrences of P in T are represented by $\mathsf{range}(P)$ storing an interval of SA such that $\mathsf{SA}[i]$ is a starting position of an occurrence of P for each $i \in \mathsf{range}(P)$. More formally, $\mathsf{range}(P)$ denotes the range in BWT such that

$$T[\mathsf{SA}[j]..\mathsf{SA}[j]+|P|-1] = P \text{ if and only if } j \in \mathsf{range}(P). \tag{1}$$

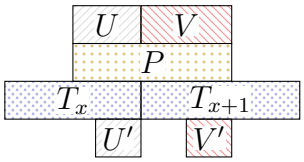


Figure 2 Setting of the proof of Lemma 6.

We obtain $R_i = \mathsf{range}(P[i..])$ from $R_{i+1} = \mathsf{range}(P[i+1..])$ with a *backward search* step

$$\mathsf{b}(R_i) = C[P[i]] + \mathsf{BWT.rank}_{P[i]}(\mathsf{b}(R_{i+1}) + 1) \text{ and } \mathsf{e}(R_i) = C[P[i]] + \mathsf{BWT.rank}_{P[i]}(\mathsf{e}(R_{i+1})). \tag{2}$$

We stipulate that the range of the empty string is $[1..n]$. Starting with the range of the empty string $\mathsf{range}(P[|P|+1..])$ and applying Equation (2) iteratively, we can find all occurrences of the pattern P in T with $|P|$ rank operations.

If we exchange BWT with BBWT, we need to take special care of a so-called *rewinding*. Suppose that we matched an occurrence of $P[i+1..]$ starting at position $j+1$ in T.

- If both text positions j and $j+1$ are contained in a Lyndon factor T_x for an integer x with $1 \le x \le t$, the backward search step

$$C[P[i]] + \mathsf{BBWT.rank}_{P[i]}(\mathsf{ISA}[i+1]) \tag{3}$$

  yields the occurrence of $P[i..]$ starting at position j in T.
- Otherwise, j and $j+1$ are contained in two different Lyndon factors. Let T_x be the Lyndon factor with $\mathsf{b}(T_x) = j+1$ (and hence the text position j is contained in T_{x-1}). Then the backward search gives $\mathsf{ISA}[\mathsf{e}(T_x)]$, i.e., the starting position of the last conjugate of T_x (in SA-order, cf. the cycle representing the Lyndon factor `ab` in Figure 1).

We call the second case *rewinding*, as the backward search counts down from the i-th conjugate to the $(i-1)$-th conjugate, but *rewinds* from the zeroth-conjugate (i.e., T_x itself) to the last conjugate. Whenever we expect that no rewinding will happen, we can find a pattern with the backward search of the FM-index:

▶ **Lemma 4.** *Given a text T and a pattern P such that each occurrence of P in T is contained in a Lyndon factor of T, we can compute these occurrences with the backward search of the FM-index on the* BBWT *with $|P|$ rank operations.*

Proof. Since all occurrences of P are contained in Lyndon factors of T, the backward search finds no occurrence of $P[i..]$ starting at the beginning $\mathsf{b}(T_x)$ of a Lyndon factor T_x in T, for $2 \le i \le |P|$ and $1 \le x \le t$. ◀

3.1 Lyndon Patterns

We first focus on the special case that the pattern itself is a Lyndon word. Subsequently, we show the general case (Section 3.2) by applying the Lyndon factorization to the pattern P and introduce an enhancement to the backward search for obtaining $\mathsf{range}(P[i..])$ from $\mathsf{range}(P[i+1..])$ in the case that the suffix $P[i+1..]$ starts with a Lyndon factor of T. For all this we need a little helper lemma:

▶ **Lemma 5** ([8, Prop. 1.10])**.** *The longest prefix of T that is a Lyndon word is the first Lyndon factor T_1 of T. Given* $\mathrm{LynF}(T) = \{T_1, \ldots, T_t\}$, $\mathrm{LynF}(T) = \{T_1\} \cup \mathrm{LynF}(T_2 \cdots T_t)$.

Figure 3 Suffix $P[i..]$ of pattern P matches the beginnings of some Lyndon factors of T. These Lyndon factors $T_x, \ldots, T_z$ are all consecutive.

▶ **Lemma 6.** *Let T be a string with* $\mathrm{LynF}(T) = \{T_1, \ldots, T_t\}$*, and let P be a pattern. If P is a Lyndon word, then there is no occurrence of P in T that crosses the border of two Lyndon factors, i.e., each occurrence of P in T is contained in a Lyndon factor T_x $(1 \leq x \leq t)$.*

Proof. Assume to the contrary that $P = UV$, where $U \in \Sigma^+$ is a suffix of T_x and $V \in \Sigma^+$ is a prefix of $T_{x+1} \cdots T_t$ for an integer x with $1 \leq x < t$. This setting is illustrated in Figure 2.

Since T_x is the longest Lyndon prefix of $T_x \cdots T_t$ (see Lemma 5), it is not possible that $U = T_x$ (otherwise we could extend T_x to UV to form a longer Lyndon word). We conclude that U is a proper suffix of T_x. Since T_x is a Lyndon word, we have $T_x \prec U'$ for every proper suffix U' of T_x (including U). This implies that $T_xV \prec U'V$, and in particular $T_xV \prec UV$. Since the pattern P is a Lyndon word, we have $V' \succ P = UV \succ T_xV$ for every suffix V' of V (including V itself).

Putting everything together, we have that T_xV is lexicographically smaller than its proper suffixes, and T_xV thus is a Lyndon word. However, this again contradicts the setting that T_x is the longest Lyndon prefix of $T_x \cdots T_t$. ◀

Combining this result with Lemma 4 yields:

▶ **Corollary 7.** *Given a pattern P that is a Lyndon word, we can find all its occurrences with $|P|$ rank operations of the FM-index built on* BBWT*.*

3.2 General Case

To find arbitrary patterns, we need to understand what happens during the rewinding. Suppose that we matched $P[i..]$ in T with the backward search. Further suppose that an occurrence of $P[i..]$ starts at position $\mathsf{b}(T_y)$ in T. Then we claim that T_y belongs to a consecutive set of Lyndon factors $T_x, \ldots, T_z$ with $x \leq y \leq z$ such that there is an occurrence of $P[i..]$ starting at position $\mathsf{b}(T_{y'})$ in T for each Lyndon factor $T_{y'}$ with $x \leq y' \leq z$. Figure 3 visualizes this setting. Assume that our claim is not true. Then there is an index y' with $x \leq y' \leq z$ and there is no occurrence of $P[i..]$ starting at position $\mathsf{b}(T_{y'})$ in T. This contradicts $T_x \succeq T_{y'} \succeq T_z$. We conclude our observation with the following lemma.

▶ **Lemma 8.** *If a string P is a prefix of T_x and T_z, then P is a prefix of T_y for each integer y with $x \leq y \leq z$.*

Suppose that we matched $P[i..]$ and that there are occurrences of $P[i..]$ starting with Lyndon factors of T. These Lyndon factors are consecutive according to Lemma 8. Let these Lyndon factors be $T_x, \ldots, T_z$. Moreover, $P[i..]$ starts with a Lyndon factor of P according to Lemma 6, i.e., $P[i..] = \mathrm{lfs}_P(w)$ for an integer w with $1 \leq w \leq p$. A further backward search step causes a rewinding for all occurrences of $P[i..]$ starting at $\mathsf{b}(T_x), \ldots, \mathsf{b}(T_z)$, where the following cases can occur:

- If $T[\mathsf{e}(T_z)] = P[i-1]$, but $T[\mathsf{b}(T_{z+1})..]$ does not have $\mathrm{lfs}_P(w)$ as a prefix, then the backward search carries on a *false occurrence*.
- If $T[\mathsf{b}(T_x) - 1] = P[i-1]$, we would expect that the backward search reports that an occurrence of $P[i-1..]$ starts at $T[\mathsf{b}(T_x) - 1]$ (we assume that $T[\mathsf{e}(T_x)] = P[i-1]$).

However, this is not the case because of the rewinding, either reporting the text position $\mathsf{e}(T_x)$ or dismissing this occurrence if $\mathsf{e}(T_x) \neq P[i-1]$. In either case, we say that there is a *missed occurrence* of $P[i-1..]$ starting at $\mathsf{b}(T_x) - 1$.

- If $T[\mathsf{e}(T_y)] = P[i-1]$ but $T[\mathsf{e}(T_{y+1})] \neq P[i-1]$ for an integer y with $x \leq y \leq z-1$, then the rewinding discards the occurrence of $P[i..]$ starting at $T[\mathsf{b}(T_{y+1})]$ although $T[\mathsf{b}(T_{y+1})-1] = T[\mathsf{e}(T_y)] = P[i-1]$. This looks like that the occurrence of $P[i..]$ starting at $T[\mathsf{b}(T_{y+1})]$ becomes a missed occurrence. However, since $T[\mathsf{b}(T_y)]$ and $T[\mathsf{b}(T_{y+1})]$ are the starting positions of occurrences of $P[i..]$, the occurrence of $P[i..]$ starting at $T[\mathsf{b}(T_y)]$ takes over the job from the occurrence starting at $T[\mathsf{b}(T_{y+1})]$ after the rewinding, i.e., we obtain the starting position $T[\mathsf{e}(T_y)] = T[\mathsf{b}(T_{y+1}) - 1]$ of the occurrence of $P[i-1..]$ after rewinding it.
- Similarly, the setting $T[\mathsf{e}(T_y)] \neq P[i-1]$ but $T[\mathsf{e}(T_{y+1})] = P[i-1]$ for an integer y with $x \leq y \leq z-1$ seems to cause a false occurrence after rewinding the occurrence of $P[i..]$ starting at $\mathsf{b}(T_y)$, but actually this occurrence takes over the job from the occurrence starting at $T[\mathsf{b}(T_{y+1})]$.
- In all other cases, for $x+1 \leq y \leq z$, $T[\mathsf{e}(T_{y-1})]\mathrm{lfs}_P(w) = T[\mathsf{e}(T_y)]\mathrm{lfs}_P(w)$, i.e., the rewound positions are beginning positions of occurrences of $P[i-1..]$, where the occurrence of $P[i..]$ starting at $\mathsf{b}(T_{y-1})$ takes the job from the occurrence starting at $\mathsf{b}(T_y)$ after the rewinding.

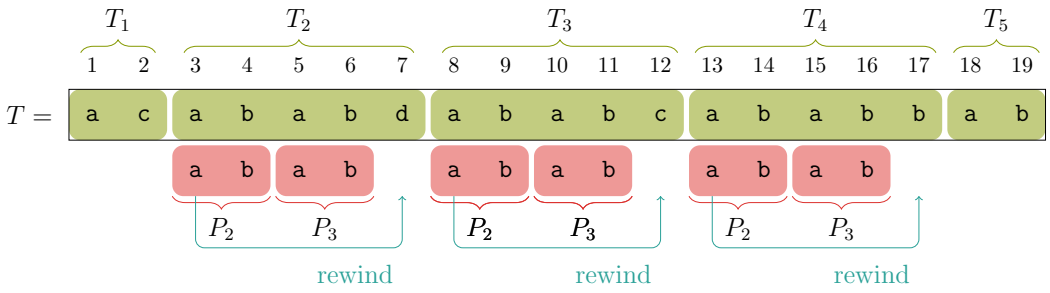


Figure 4 Backward search of a pattern P with $\mathrm{LynF}(P) = \{P_1, P_2 = \mathtt{ab}, P_3 = \mathtt{ab}\}$ in our running example $T = \mathtt{acababdababcababbab}$ after $|P_2P_3|$ steps. The sub-pattern P_2P_3 has occurrences starting at the starting positions of the Lyndon factors T_2, T_3, and T_4 of the text. The effects of the rewinding depend on P_1. If P_1 ends with $\mathtt{c}$, then we derive a *missed* occurrence from $\mathrm{lfs}_P(2)$ and T_2. If P_1 ends with $\mathtt{b}$, then we derive a *false* occurrence from $\mathrm{lfs}_P(2)$ and T_4.

In what follows, we study ways to limit the number of false and missed occurrences. We say that a false (resp. missed) occurrence of P is *derived from* P_w and T_z (resp. T_x) if it emerges on the rewinding at $T[\mathsf{b}(T_z)]$ (resp. $T[\mathsf{b}(T_x)]$). See Figure 4 for an example. According to Lemma 6 there are at most p rewindings, and hence at most p false and missed occurrences. (We lower this upper bound in the subsequent section). The false occurrences can be easily maintained in a separate list, in which each element corresponds to a false occurrence (more precisely, applying SA to such an element yields its corresponding starting position in the text). Each element of the list is subject to the backward search (Equation (3)) like the range itself (Equation (2)). Whenever a backward search step of an element of the list yields not an occurrence (e.g., we obtain the element $\mathsf{ISA}[j]$ by a backward search step from $P[i+1..]$ to $P[i..]$, but find out that $T[j] \neq P[i]$), then the false occurrence will also vanish from the range such that we no longer need to manage that element. Similarly, we keep track of the missed occurrences. For that, we take advantage of the fact that the entries of BBWT corresponding to Lyndon factors are lexicographically sorted (see the dark

yellow marked entries in Figure 1). To move from the beginning of a Lyndon factor to the end of its preceding Lyndon factor, it suffices to locate the previously larger Lyndon factor and apply a backward search step on it (to intentionally cause a rewinding). For that, we add a bit vector B_{L} marking the entries in BBWT corresponding to a Lyndon factor (and not to one of its conjugates) with '1'. Then $B_{\mathrm{L}}.\mathsf{select}_1(t-x+1)$ corresponds to T_x and the position $\mathsf{ISA}[\mathsf{b}(T_x)-1] = \mathsf{ISA}[\mathsf{e}(T_{x-1})]$ is found by applying a backward search step to $B_{\mathrm{L}}.\mathsf{select}_1(t-x)$. Again, we keep the missed occurrences in a list whose elements are (each individually) subject to the backward search. Finally, when we want to report all occurrences of the complete pattern, we take the computed range $\mathsf{range}(P)$, add all elements of the list of missed occurrences, and remove all elements of the list of the false occurrences. By doing so, we can restore the property of Equation (1). With the lists for the missed and false occurrences and the bit vector B_{L}, we can state the following theorem generalizing the backward search for arbitrary patterns.

▶ **Theorem 9.** *Given a text T and a pattern P, we can compute all occurrences of P in T with the FM-index built on* BBWT *with $\mathcal{O}(|P|p)$ rank operations, where p is the number of Lyndon factors of P.*

In the following, we improve the $\mathcal{O}(|P|p)$ bound on the number of rank operations. There is a problem with matching a pattern whose Lyndon factorization consists of the same Lyndon factor that is equal to some Lyndon factors of the text. An example for such a case is given by $T = P = \mathtt{a}^n$. Here, P has n Lyndon factors, and therefore our current upper bound on the number of rank operations stated in Theorem 9 is only $\mathcal{O}(n^2)$. Multiple occurrences of the same Lyndon factors (a) in the text as well as (b) in the pattern make the matching difficult. However, as we will see, we can cope with both individually.

First, we start with (a) the text; (b) is treated in Section 3.3. Our solution is to build the bijective BWT on all *distinct* Lyndon factors of T (along with their conjugates), remembering the number of occurrences of a Lyndon factor, such that the Lyndon factorization $T = T_1 \cdots T_t$ becomes $T = \tilde{T}_1^{\tau_1} \cdots \tilde{T}_{t'}^{\tau_{t'}}$, where $\tilde{T}_1, \cdots \tilde{T}_{t'}$ are distinct Lyndon words with $\tilde{T}_x \prec \tilde{T}_{x+1}$ for $1 \le x \le t'-1 \le t-1$, and for every $1 \le x \le t'$ it holds that (a) $\tau_j \ge 1$ and (b) there is an integer y with $y \ge x$ such that $\tilde{T}_x = T_y$. The set $\{\tilde{T}_1^{\tau_1}, \ldots, \tilde{T}_{t'}^{\tau_{t'}}\}$ is called the *composed* Lyndon factorization of T. Given $\tilde{T}_x = T_y$, we stipulate that $\mathsf{b}(\tilde{T}_x)$ is the starting position of the leftmost Lyndon factor $T_{y-\tau_x+1}$ with $T_{y-\tau_x+1} = \tilde{T}_x$. For instance, the composed Lyndon factorization of $T = \mathtt{bbabababa}$ is $T = \tilde{T}_1^2\, \tilde{T}_2^3\, \tilde{T}_3$ with $\tilde{T}_1 = \mathtt{b}$, $\tilde{T}_2 = \mathtt{ab}$, and $\tilde{T}_3 = \mathtt{a}$. The starting position $\mathsf{b}(\tilde{T}_2)$ is 3.

Now suppose that the Lyndon factor P_w occurs k_w times in $\mathsf{LynF}(P)$, and suppose that P_w is the rightmost occurrence of them, i.e., $P_{w-k_w} \neq P_{w-k_w+1} = \ldots = P_{w-1} = P_w \neq P_{w+1}$. Whenever we match $\mathrm{lfs}_P(w)$ with the beginning of the rightmost Lyndon factor T_y equal to $\tilde{T}_x$ with $|\tilde{T}_x| \le |\mathrm{lfs}_P(w)|$ occurring τ_x times in T, we can directly match $P_w^{k_w-1}\mathrm{lfs}_P(w)$ if $\tau_x \ge k_w$, skipping the backward search for $P[\mathsf{b}(P_{w-k_w+1})..\mathsf{b}(P_w)]$ such that we directly match $P[\mathsf{b}(P_{w-k_w+1})..]$ for one occurrence O starting at $T[\mathsf{b}(\tilde{T}_x)]$. For that, we assumed that $\tilde{T}_x = P_{w-j}$ for every integer j with $0 \le j \le k_w - 1$. This is true due to the following lemma:

▶ **Lemma 10.** *Given an occurrence of P_w that starts at position $\mathsf{b}(T_x)$ in T, $\mathrm{lfs}_P(w)$ is not a proper prefix of T_x if and only if $P_w = T_x$.*

Proof. Assume that $\mathrm{lfs}_P(w)$ is not a proper prefix of T_x. Switching the roles of P and T in Lemma 6, we yield that T_x cannot cross the border between P_w and P_{w+1}. Since $|\mathrm{lfs}_P(w)| \ge |T_x|$, it holds that $P_w = T_x$ (otherwise we could extend T_x to a longer Lyndon factor). ◀

The further matching of the occurrence O is conducted separately to the backward search with the range of occurrences $\mathsf{range}(P[\mathsf{b}(P_w)..])$. If $\tau_x < k_w$, then we cannot extend the currently matched occurrence, and thus can ignore to follow this occurrence. We call this technique of skipping consecutive Lyndon factors a *composed jump*.

The composed jump allows us to proceed as follows: We only count missed occurrences O that were not derived from a missed occurrence (i.e., an occurrence belonging to the range and not part of the list of missed occurrences), which we call in the following *freshly* missed occurrences. We do not count a missed occurrence O' that is derived from a missed occurrence O. Instead, we only update the position of O to O' in the list of missed occurrences. This is justified as we cannot create a freshly missed occurrence during a later backward search step:

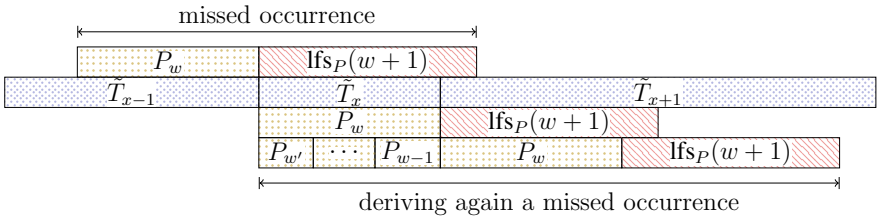


Figure 5 Setting of the proof of Lemma 11 where a false occurrence from $\mathrm{lfs}_P(w')$ and $\tilde{T}_x$ is derived after a false occurrence was derived from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$ with $w' < w$. However, this is not possible since then $\tilde{T}_{x-1} = \tilde{T}_x$.

▶ **Lemma 11.** *Let an occurrence of* $\mathrm{lfs}_P(w)$ *start at position* $\mathsf{b}(\tilde{T}_x)$ *in* T *and let* $|\tilde{T}_x| < |\mathrm{lfs}_P(w)|$. *If there is a missed occurrence derived from* $\mathrm{lfs}_P(w)$ *and* $\tilde{T}_x$, *there is no* $w' < w$ *such that* $\mathrm{lfs}_P(w')$ *and* $\tilde{T}_x$ *derive a* freshly *missed occurrence.*

Proof. By Lemma 10, $P_w = \tilde{T}_x$. Assume that there is a freshly missed occurrence derived from $\mathrm{lfs}_P(w')$ and $\tilde{T}_x$ for the largest such w' with $w' < w$. Then $P_{w'} \cdots P_{w-1} = P_w$ and $\mathrm{lfs}_P(w') = P_w\mathrm{lfs}_P(w) = P_wP_w\mathrm{lfs}_P(w+1)$. Hence, $P_w = \tilde{T}_x$ is a suffix of $\tilde{T}_{x-1}$ (cf. Figure 5). Since $\tilde{T}_{x-1}$ is a Lyndon word with $\tilde{T}_x \succeq \tilde{T}_{x-1}$, $\tilde{T}_{x-1} = \tilde{T}_x = P_w = P_{w-1}$ must hold. However, this contradicts the distinctness of $\tilde{T}_x$ in the composed Lyndon factorization. ◀

3.3 Improving the Number of Ranks

In this section, we study the case of multiple occurrences of the same Lyndon factor in the pattern to improve the bound to $\mathcal{O}(|P|p')$ rank operations, where p' is the number of *different* Lyndon factors of P. For that we show two lemmas:

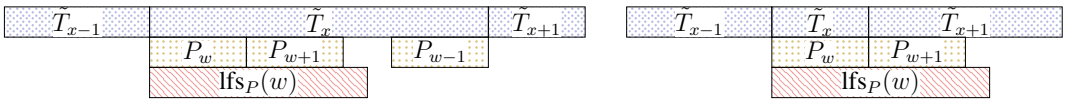


Figure 6 Setting of the proof of Lemma 12 that seems to derive a false occurrence. A necessary condition to derive a false occurrence from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$ is that $\tilde{T}_x$ is the last Lyndon factor having $\mathrm{lfs}_P(w)$ as a prefix (left). Since $\tilde{T}_x$ must be border-free, $\tilde{T}_x = P_w$ holds (right).

▶ **Lemma 12.** *If* $P_{w-1} = P_w = P_{w+1}$, *then a* false *occurrence derived from* $\mathrm{lfs}_P(w)$ *disappears after matching* $|P_w|$ *characters.*

Proof. Assume that there is a false occurrence derived from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$. Then (a) an occurrence of $\mathrm{lfs}_P(w)$ starts at position $\mathsf{b}(\tilde{T}_x)$ in T and (b) P_{w-1} is a suffix of $\tilde{T}_x$. See also Figure 6. Since a Lyndon word is border-free, $\tilde{T}_x = P_w$. However, we derived a false occurrence from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$ such that $\tilde{T}_{x+1}$ cannot start with P_{w+1}. This is a contradiction, since we found an occurrence of $\mathrm{lfs}_P(w)$ starting at position $\mathsf{b}(\tilde{T}_x)$ in T, $\tilde{T}_x = P_w$, and therefore $\tilde{T}_{x+1}$ must start with P_{w+1}. ◀

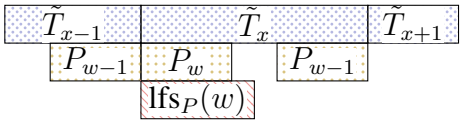


Figure 7 Setting of the proof of Lemma 13 that seems to derive a missed occurrence.

▶ **Lemma 13.** *Given an occurrence of P_w starts at position $\mathsf{b}(\tilde{T}_x)$ in T, a* missed *occurrence derived from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$ disappears after matching $|P_w|$ characters if $|\mathrm{lfs}_P(w)| \leq |\tilde{T}_x|$ and $P_{w-1} = P_w$.*

Proof. Suppose that there is a missed occurrence derived from $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$. We have the following setting, which is sketched in Figure 7:

- $\tilde{T}_x$ is the leftmost Lyndon factor of the composed Lyndon factorization of T that starts with $\mathrm{lfs}_P(w)$, and
- P_{w-1} is a suffix of $\tilde{T}_{x-1}$.

$$\begin{aligned} \text{Then } P_w \preceq \mathrm{lfs}_P(w) &\preceq \tilde{T}_x && (P_w \text{ is a prefix of } \mathrm{lfs}_P(w) \text{ and } \mathrm{lfs}_P(w) \text{ is a prefix of } \tilde{T}_x) \\ &\prec \tilde{T}_{x-1} && (\text{Definition of the composed Lyndon factorization}) \\ &\preceq P_{w-1}, && (\tilde{T}_{x-1} \text{ is a Lyndon word and } P_{w-1} \text{ one of its suffixes}) \end{aligned}$$

contradicting the assumption $P_{w-1} = P_w$. ◀

A conclusion is that all longest consecutive appearances of the same Lyndon factors $P_w = \ldots = P_{w+j}$ for integers $1 \leq w \leq p$ and $j \geq 0$ can cause at most one newly missed and one false occurrence in total (which we need to keep track of). In other words, we know that we only have to care about p' freshly missed and false occurrences. Thus, we can improve the number of rank operations to $\mathcal{O}(|P|p')$.

3.4 Longest Pre-Lyndon Word

To obtain $\mathcal{O}(|P| \lg |P|)$ rank operations, we need the notion of the *longest pre-Lyndon suffix* λ_P, which is the smallest integer such that $\mathrm{lfs}_P(w+1)$ is a prefix of P_w for every $\lambda_P \leq w \leq p$. In our running example (cf. Figure 1), $\lambda_T = 4$ since T_5 is a prefix of T_4, but T_4 is not a prefix of T_3.

$$\mathrm{lfs}_P(w)c^{|X|+1} =$$

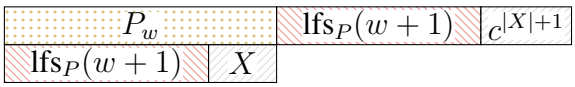


Figure 8 Setting of the proofs of Lemmas 14 and 18, where $\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word because $\mathrm{lfs}_P(w+1)$ is a prefix of P_w and $c^{|X|+1} \succ X$.

▶ **Lemma 14.** *The string* $\mathrm{lfs}_P(w)$ *is a pre-Lyndon word for every* $\lambda_P \le w \le p$.

Proof. Since $\mathrm{lfs}_P(w+1)$ is a prefix of P_w, there is a suffix X of P_w with $P_w = \mathrm{lfs}_P(w+1)X$ (cf. Figure 8). Given a $c \in \Sigma$ with $c^{|X|+1} \succ X$, $\mathrm{lfs}_P(w)c^{|X|+1} = \mathrm{lfs}_P(w+1)X\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word. ◀

We borrow the following facts from literature:

▶ **Lemma 15** ([13, Lemma 11])**.** P_w *is* not *a proper prefix of* $\mathrm{lfs}_P(w+1)$ *for every integer* w *with* $1 \le w \le \lambda_P - 1$.

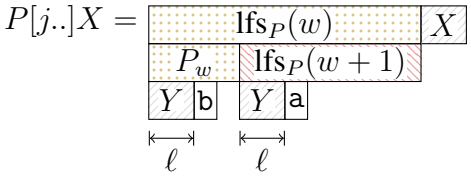


Figure 9 Setting of the proofs of Corollary 16 and Lemma 19, where a and b are characters with $\mathtt{a} \prec \mathtt{b}$, and $Y = P_w[1..\ell] = \mathrm{lfs}_P(w+1)[1..\ell]$. There is no such string X that $P[j..]X$ is a Lyndon word.

▶ **Corollary 16.** $\mathrm{lfs}_P(\lambda_P)$ *is the longest pre-Lyndon suffix of* P.

Proof. Assume that there is a longer pre-Lyndon suffix $P[j..]$. This suffix has to start with a Lyndon factor P_w for an integer w with $1 \le w \le p$, otherwise $\mathrm{lfs}_P(w) \prec P[j..]$ with w such that $\mathsf{b}(P_w) < j \le \mathsf{e}(P_w)$ (since every proper suffix of P_w is lexicographically larger than P_w), and therefore $\mathrm{lfs}_P(w)$ would be a longer pre-Lyndon suffix.

According to Lemma 15, there is an $\ell := \mathrm{lcp}(P_w, \mathrm{lfs}_P(w+1))$ with $\ell < \min(|P_w|, |\mathrm{lfs}_P(w+1)|)$. Then $P_w[\ell+1] > \mathrm{lfs}_P(w+1)[\ell+1]$ and therefore $\mathsf{conj}_{|P_w|}(\mathrm{lfs}_P(w)X) = \mathrm{lfs}_P(w+1)XP_w \prec P_w\mathrm{lfs}_P(w+1)X = \mathrm{lfs}_P(w)X$, regardless of the choice of the string X (cf. Figure 9). ◀

▶ **Lemma 17** ([13, Lemma 12])**.** $p' - \lambda_P = \mathcal{O}(\lg |P|)$ *where* p' *is the number of* distinct *Lyndon factors of* P.

The next lemmas show the usefulness of λ_P:

▶ **Lemma 18.** *Given an integer* w *with* $1 \le w \le \lambda_P - 1$, $\mathrm{lfs}_P(w+1)$ *is not a prefix of* P_w.

Proof. Assume to the contrary that $\mathrm{lfs}_P(w+1)$ is a prefix of P_w, and $P_w = \mathrm{lfs}_P(w+1)X$ for a string $X \in \Sigma^+$. Given a character $c \in \Sigma$ with $c^{|X|+1} \succ X$, $\mathrm{lfs}_P(w)c^{|X|+1} = \mathrm{lfs}_P(w+1)X\mathrm{lfs}_P(w+1)c^{|X|+1}$ is a Lyndon word, contradicting the fact that $\mathrm{lfs}_P(\lambda_P)$ is the longest pre-Lyndon word of P (cf. Corollary 16 and Figure 8). ◀

▶ **Lemma 19.** *Given an occurrence of* $\mathrm{lfs}_P(w)$ *with* $w < \lambda_P$ *that starts at position* $\mathsf{b}(\tilde{T}_x)$ *in* T *for an integer* x *with* $1 \le x \le t'$, *we have* $\tilde{T}_x = P_w$.

Proof. Since $\mathrm{lfs}_P(\lambda_P)$ is the longest pre-Lyndon suffix of P (see Corollary 16), $\mathrm{lfs}_P(w)$ is not a pre-Lyndon suffix of P. According to Lemma 18, $\mathrm{lfs}_P(w+1)$ is not a prefix of P_w. Let $\ell := \mathrm{lcp}(P_w, \mathrm{lfs}_P(w+1)) < \min(|P_w|, |\mathrm{lfs}_P(w+1)|)$. Then $P_w[\ell+1] > \mathrm{lfs}_P(w+1)[\ell+1]$ according to the Lyndon factorization of P, and therefore P_w is the longest Lyndon word of T having an occurrence that starts at position $\mathsf{b}(\tilde{T}_x)$ in T, i.e., $\tilde{T}_x = P_w$. That is because a longer Lyndon factor $\tilde{T}_x$ would contain $P_w\mathrm{lfs}_P(w+1)[1..\ell+1]$, which is lexicographically larger than $\mathsf{conj}_{|P_w|}(P_w\mathrm{lfs}_P(w+1)[1..\ell+1]) = \mathrm{lfs}_P(w+1)[1..\ell+1]P_w$ (cf. Figure 9). ◀

The above lemmas allow us to derive the following consequence:

▶ **Corollary 20.** *There is no freshly missed occurrence derived from* $\mathrm{lfs}_P(w)$ *for every integer* w *with* $w < \lambda_P$*. If one of those* $\mathrm{lfs}_P(w)$ *derives a false occurrence, then this false occurrence disappears until the range* $\mathsf{range}(P)$ *is matched.*

Proof. Suppose there is an occurrence of $\mathrm{lfs}_P(w)$ starting at position $\mathsf{b}(\tilde{T}_x)$ in T, for a composed Lyndon factor $\tilde{T}_x$ with $1 \leq x \leq t'$. According to Lemma 19, $P_w = \tilde{T}_x$. Since $w < \lambda_P \leq p$, we have that $|\tilde{T}_x| = |P_w| < |\mathrm{lfs}_P(w)|$. Then with Lemma 11 we obtain that $\mathrm{lfs}_P(w)$ and $\tilde{T}_x$ cannot derive a freshly missed occurrence. By Lemma 15, P_{w-1} is not a proper prefix of $\mathrm{lfs}_w(P)$, such that after matching $\mathsf{lcp}(P_{w-1}, \mathrm{lfs}_w(P))$ characters, a possibly derived false occurrence will disappear, and thus does not need to be tracked. ◀

With Lemma 17 we obtain:

▶ **Theorem 21.** *Given a text* T *and a pattern* P*, we can compute all occurrences of* P *in* T *with the FM-index built on the bijective BWT of the composed Lyndon factors of* T *with* $\mathcal{O}(|P| \lg |P|)$ *rank operations.*

Finally, we explain how to detect whether an occurrence of a suffix of the pattern starts at the beginning of a Lyndon factor of the text. For that, after each backward search step, we use the bit vector B_{L} introduced in Section 3.2 marking now the entries corresponding to the *composed* Lyndon factors in BBWT. If $\mathsf{range}(P[i..]) = [b..e]$ and $B_{\mathrm{L}}.\mathsf{rank}_1(e) - B_{\mathrm{L}}.\mathsf{rank}_1(b-1)$ is positive, then there is an occurrence of $P[i..]$ starting at position $\mathsf{b}(\tilde{T}_x)$ in T (after applying a composed jump) for every x with $t' - B_{\mathrm{L}}.\mathsf{rank}_1(e) + 1 \leq x \leq t' - B_{\mathrm{L}}.\mathsf{rank}_1(b-1)$. In this case, $P[i..] = \mathrm{lfs}_P(w)$ for an integer w with $1 \leq w \leq p$ due to Lemma 6.

4 Construction and Outlook

Having the backward search technique of Theorem 21, we can construct and use an online BBWT index data structure when combining the BBWT construction algorithm of Theorem 3 with a dynamic data structure for rank and select queries.

▶ **Theorem 22.** *Given a text* T *of length* n *whose characters are drawn from an alphabet of size* σ*, we can build a text index on the bijective BWT of* T *online in* $\mathcal{O}(n \lg n / \lg\lg n)$ *time. The text index supports searching for all occurrences of a pattern* P *in* $\mathcal{O}(|P| \lg |P| \lg n / \lg\lg n)$ *time. The index uses* $|\widetilde{\mathsf{BBWT}}|(H_0(\widetilde{\mathsf{BBWT}}) + H_0(B_{\mathrm{L}})) + o(n \lg \sigma) + \mathcal{O}(\sigma \lg n)$ *bits, where* $\widetilde{\mathsf{BBWT}}$ *is the bijective BWT of the composed Lyndon factorization. It returns a range and a list of positions in* SA *corresponding to starting positions of suffixes having* P *as a prefix.*

Proof. We use the dynamic representation of Navarro and Nekrich [21] for the wavelet tree of the FM-index, as well as for the bit vector B_{L}. This data structure can be constructed in $\mathcal{O}(n \lg n / \lg\lg n)$ time and can answer each type of query in optimal $\mathcal{O}(\lg n / \lg\lg n)$ time. Our space bound is due to this data structure. The array C is stored in a plain form using $\sigma \lg n$ bits. ◀

To actually report the matched positions in the text, we can use the approach of Mäkinen and Navarro [17] who apply run-length compression on the traditional BWT and store a suffix array entry for each run in the BWT, thus achieving $r \lg n$ bits additional cost for this suffix array sampling, where r is the number of runs in the BWT. It is straight-forward to adapt this technique for the bijective BWT: For that, we keep B_{L}, but run-length compress BBWT. The time bounds remain the same if $\sigma = \mathcal{O}(\lg^{\mathcal{O}(1)} n)$ [17].

Open Problems

We are unaware of an algorithm for which it is proven that it can build BBWT in linear time. It seems hard to find a relation to suffix array construction algorithms, since the context of the suffixes with respect to the lexicographic order and the context of the Lyndon words with respect to $\prec_\omega$ is different.

We are aware of a recently found redundancy [2] in the traditional BWT, and wonder whether this result translates to the bijective variant.

References

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

2 Uwe Baier. On Undetected Redundancy in the Burrows-Wheeler Transform. In *Proc. CPM*, volume 105 of *LIPIcs*, pages 3:1–3:15, 2018.

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

4 Stefan Böttcher, Alexander Bültmann, Rita Hartel, and Jonathan Schlüßler. Fast Insertion and Deletion in Compressed Texts. In *Proc. DCC*, page 393, 2012.

5 Stefan Böttcher, Alexander Bültmann, Rita Hartel, and Jonathan Schlüßler. Implementing Efficient Updates in Compressed Big Text Databases. In *Proc. DEXA*, volume 8056 of *LNCS*, pages 189–202, 2013.

6 M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

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

8 Jean-Pierre Duval. Factorizing Words over an Ordered Alphabet. *J. Algorithms*, 4(4):363–381, 1983.

9 Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proc. FOCS*, pages 390–398, 2000.

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

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-Time Text Indexing in BWT-runs Bounded Space. In *Proc. SODA*, pages 1459–1477, 2018.

12 Joseph Yossi Gil and David Allen Scott. A Bijective String Sorting Transform. *ArXiv 1201.3077*, 2012. `arXiv:1201.3077`.

13 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.

14 Masaru Ito, Hiroshi Inoue, and Kenjiro Taura. Fragmented BWT: An Extended BWT for Full-Text Indexing. In *Proc. SPIRE*, volume 9954 of *LNCS*, pages 97–109, 2016.

15 Manfred Kufleitner. On Bijective Variants of the Burrows-Wheeler Transform. In *Proc. PSC*, pages 65–79, 2009.

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

17 Veli Mäkinen and Gonzalo Navarro. Succinct Suffix Arrays based on Run-Length Encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

18 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

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

20 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007.

21 Gonzalo Navarro and Yakov Nekrich. Optimal Dynamic Sequence Representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014.

22 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A Faster Implementation of Online Run-Length Burrows-Wheeler Transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages 409–419, 2017.

23 Alberto Policriti and Nicola Prezza. Computing LZ77 in Run-Compressed Space. In *Proc. DCC*, pages 23–32, 2016.

On Maximal Repeats in Compressed Strings

Julian Pape-Lange
Technische Universität Chemnitz, Straße der Nationen 62, 09111 Chemnitz, Germany
julian.pape-lange@informatik.tu-chemnitz.de

Abstract

This paper presents and proves a new non-trivial upper bound on the number of maximal repeats of compressed strings. Using Theorem 1 of Raffinot's article "On Maximal Repeats in Strings", this upper bound can be directly translated into an upper bound on the number of nodes in the Compacted Directed Acyclic Word Graphs of compressed strings.

More formally, this paper proves that the number of maximal repeats in a string with z (self-referential) LZ77-factors and without q-th powers is at most $3q(z+1)^3 - 2$. Also, this paper proves that for $2000 \leq z \leq q$ this upper bound is tight up to a constant factor.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatoric problems

Keywords and phrases Maximal repeats, Combinatorics on compressed strings, LZ77, Compact suffix automata, CDAWGs

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.18

Acknowledgements I thank Professor Laurent Bartholdi for leading me to this research topic as well as for his helpful and inspiring pieces of advice.

1 Introduction

A repeat of a string S is a substring of S which occurs at least twice in S. A repeat P of S is a maximal repeat, if every string which properly contains P occurs less often in S than P itself. Usually there are much less maximal repeats than repeats. Nevertheless the set of maximal repeats still contains all of the information about the repeats. These repeats have, as shown by Gusfield in [9], many applications in computational biology. A good overview of the importance of repeats in computational biology together with a deeper analysis of local repeats is also given by Nicolas et al. in [10] on ResearchGate.

Maximal repeats are also closely linked to string compression and succinct data structures: Furuya et al. show in their recent arXiv-article [8] that there is a connection between maximal repeats and the grammar compression algorithm RePair and they use this connection to create an improved version of this algorithm. Raffinot proves in [12] that there is a natural one-to-one correspondence between the maximal repeats of a string and the number of internal nodes in its Compacted Directed Acyclic Word Graph (CDAWG).

The CDAWG of a string is a useful data structure which was introduced by Blumer et al. in [2] and has most advantages of suffix trees and acyclic directed word graphs while usually being much smaller than each of them. The CDAWG is therefore a powerful tool for string processing.

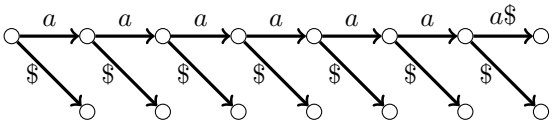


Figure 1 The suffix tree of a^7 $(= aaaaaaa)$.

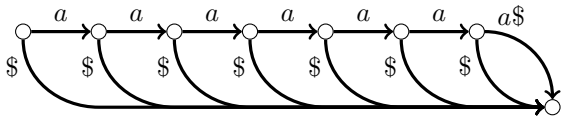


Figure 2 The CDAWG of a^7.

One might hope that well-compressible strings have highly structured suffix trees and thereby small CDAWGs. This, however, is unfortunately not the case. Even the arguably best compressible string, a^{q-1}, which does have the simple looking suffix tree shown in Figure 1, also has the CDAWG shown in Figure 2 with $q-2$ internal nodes. This shows that there is no non-trivial upper bound on either the number of nodes of a CDAWG of a string S or the number of maximal repeats of S which is only dependent on the compressed size of S. This fact may also explain the apparent lack of research regarding the number of nodes in CDAWGs of general compressible strings.

There are, however, some non-trivial bounds for the number of nodes in CDAWGs which take the structure of the underlying strings into account. For example Blumer et al. suggest in [2] that the number of nodes in a CDAWG of an English lower-case string S is between 0.26 times the length of S and 0.29 times the length of S. Blumer et al. prove in [3] formulas for the average size of the CDAWG of a random string. Stronger results have been found by Radoszewski and Rytter who prove in [11] that the number of nodes of the CDAWG of Thue-Morse words is linear in the compressed size of the word and thereby logarithmic in the size of the word itself. A similar result is shown by Epifanio et al. in [6] for Sturmian words.

Belazzougui et al. prove in [1] that the number of edges in the CDAWG of the string is bounded from below by the number of self-referential LZ77-factors. Therefore a string S over the alphabet Σ with z LZ77-factors has at least $\frac{z}{|\Sigma|}-1$ maximal repeats. This lower bound is met, for example, by a string in which every character occurs only once. While this string is not compressible, it does not have maximal repeats.

While Raffinot was motivated by the possibility of translating the better-known results for CDAWGs to maximal repeats, this paper's motivation was the other way round. The main goal was to find a new, more general upper bound for the number of nodes in the CDAWGs of compressed strings and it turned out to be very useful that Raffinot's result can be applied the other way round too.

The number z of LZ77-factors proved itself to be a very useful indicator of the complexity of strings in the past. For example Charikar et al. proved in [4] that even the minimal number of non-self-referential LZ77-factors is a lower bound for the smallest grammar compression. The self-referential version of LZ77 was used for example by Tanimura in [13] in order to show that the size of the t-truncated suffix tree is bounded by zt. Additionally, since high powers lead to CDAWGs with a high number of nodes, the additional structure of the string is measured by the highest power $q-1$ in the string.

Using these two variables, this paper gives an upper bound for the number of maximal repeats and the number of nodes in the CDAWG which is proven in section 3:

▶ **Theorem 1.** *Let S be a string. Let z be the number of (self-referential) LZ77-factors in an LZ77-decomposition of S. Let q be a number such that S does not contain q-th powers. Then the number of maximal repeats in S is bounded from above by $3q(z+1)^3-2$. Also, the Compacted Directed Acyclic Word Graph (CDAWG) of S has at most $3q(z+1)^3$ nodes.*

Additionally this paper shows:

▶ **Theorem 2.** *For $2000 \leq z \leq q$ there is a string S without q-th powers which can be expressed by z (self-referential) LZ77-factors and which has at least $\frac{1}{500}qz^3$ maximal repeats.*

This result, which is proven in section 4, shows that for $2000 \leq z \leq q$ the upper bound given by Theorem 1 is tight up to a constant factor.

2 Definitions

Let Σ be an *alphabet*. A *string* with *length* denoted by $|S|$ is the concatenation of *characters* $S[0]S[1]\cdots S[|S|-1]$ of Σ. For the sake of convenience we also define $S[-1] = \$$ and $S[|S|] = \$$ with $\$ \notin \Sigma$. The *substring* $S[i..j]$ with $0 \leq i \leq j \leq |S|-1$ is the concatenation $S[i]S[i+1]\cdots S[j]$. For $i > j$ the substring $S[i..j]$ is defined to be the empty string with length 0. A *prefix* is a substring of the form $S[0..j]$ and a *suffix* is a substring of the form $S[i..|S|-1]$.

A *maximal pair* of S is a triple $(n, m, l) \in \mathbb{N}^3$ with $l \geq 1$ such that $S[n..n+l-1]$ is equal to $S[m..m+l-1]$ and this property can not be extended to any side. More formally:

- $\forall i \in \mathbb{N}$ with $0 \leq i < l : S[n+i] = S[m+i]$ but
- $S[n-1] \neq S[m-1]$ and
- $S[n+l] \neq S[m+l]$.

Since for a maximal pair (n, m, l) the inequality $S[n-1] \neq S[m-1]$ holds, the indices n and m can not be equal. Furthermore, only $S[n..n+l-1]$ and $S[m..m+l-1]$ are required to be in S. The characters $S[n-1]$, $S[m-1]$, $S[n+l]$, $S[m+l]$ may be outside of S. This implies that $S[n..n+l]$ and $S[m..m+l]$ are in $S\$$.

The *distance* d of a maximal pair (n, m, l) is the distance $d = m - n$ of the two starting indices.

A *maximal repeat* of a string S is a substring $S[n..n+l-1]$ such that there is a maximal pair (n, m, l) for some indices n, m.

For example, in the string *banana*, the substring *na* is not a maximal repeat, because every occurrence of *na* is preceded by *a*. The substring *ana*, however, is a maximal repeat with maximal pair given by $(1, 3, 3)$. The distance of this maximal pair is 2.

A (self-referential) *LZ77-decomposition* of a string S is a factorization $S = F_1F_2\ldots F_z$ in *LZ77-factors* $F_1, F_2, \ldots, F_z$ such that for all $i \in 1, 2, \ldots, z$

- the factor F_i is a single character or
- the substring F_i occurs twice in $F_1F_2\ldots F_i$. (i.e. there is an occurrence of F_i in $F_1F_2\ldots F_i$ which does not use the last character of $F_1F_2\ldots F_i$)

In this paper, all LZ77-decompositions are allowed to be self-referential. Therefore we will only use the term LZ77-decomposition.

Normally the LZ77-definition requires the number of LZ77-factors of a string to be minimized. Since all theorems of this paper also hold for non-minimized LZ77-decompositions, this minimization is not required in this paper.

For example, see the following strings on the left-hand side with possible corresponding LZ77-factors, separated by "$\cdot$", on the right-hand side:

- $01001010 = 0 \cdot 1 \cdot 0 \cdot 010 \cdot 10$,
- $banana = b \cdot a \cdot n \cdot ana$,
- $aaaa = a \cdot aaa$ and
- $aaaa = a \cdot a \cdot a \cdot a$ (not minimal).

A *period* of a string S is a number Δ such that all characters in S with distance Δ are equal. If the minimal period $\Delta_{\min}$ of a non-empty string S is at most $\frac{|S|}{2}$, the string S is a *fractional power* with *exponent* $\frac{|S|}{\Delta_{\min}}$.

Fractional powers are also called repetitions in the literature. However, in order to keep them apart from the maximal repeats, the name fractional power will be used.

3 Upper Bound

The main goal of this section is to prove that the number of maximal repeats of a string S that can be written with z LZ77-factors and that does not contain a q-th power is bounded from above by $3q(z+1)^3 - 2$.

While it is easier to count the number of maximal pairs than to count the number of maximal repeats directly, there might be many maximal pairs for a single maximal repeat. Therefore, it is necessary to choose a subset of the maximal pairs which presents every maximal repeat at least once and which does not contain too many elements.

The following two lemmata will lead to a suitable subset of the maximal pairs, by showing that it is sufficient to count the maximal pairs (n, m, l) in which n is smaller than m and n as well as m are close to the boundary between two LZ77-factors.

▶ **Lemma 3.** *The triple (n, m, l) is a maximal pair if and only if (m, n, l) is a maximal pair.*

Proof. This lemma follows directly from the symmetry of the definition of maximal pairs. ◀

▶ **Lemma 4.** *Let S be a string. Let $F_1F_2 \dots F_zF_{z+1} = S\$$ be an LZ77-decomposition of $S\$$ and $s_1, s_2, \dots, s_z, s_{z+1}$ be the starting indices of the LZ77-factors in $S\$$. Let (n, m, l) be a maximal pair in S. Then there is a maximal pair (n', m', l) such that the equation $S[n..n+l-1] = S[n'..n'+l-1]$ holds and the intervals $[n', n'+l]$ and $[m', m'+l]$ contain starting indices s_j and s_k respectively.*

Proof. Let n' and m' the minimal indices such that $S[n-1..n+l] = S[n'-1..n'+l]$ and $S[m-1..m+l] = S[m'-1..m'+l]$. By construction (n', m', l) is a maximal pair and $S[n..n+l-1] = S[n'..n'+l-1]$ holds.

Assume the interval $[n'-1, n'+l]$ is inside an interval $[s_i, s_i+|F_i|-1]$ and thereby inside the LZ77-factor F_i.

Since the interval contains more than one character, every substring of F_i has an earlier occurrence. This contradicts the minimality of n'.

Therefore the last index in the interval $[n'-1, n'+l]$ lies inside another LZ77-factor than the first index in this interval. This implies the interval $[n', n'+l]$ contains some starting index s_j. Similarly, the interval $[m', m'+l]$ contains some starting index s_k. ◀

The next two lemmata will show some properties of maximal pairs with overlap. These properties will be important in the proof of the upper bound for the subset of maximal pairs.

▶ **Lemma 5.** *Let S be a string. Let further (n_a, m_a, l_a) and (n_b, m_b, l_b) be different maximal pairs in S such that there is an index c with $c \in [n_a, n_a+l_a]$ and $c \in [n_b, n_b+l_b]$. Then the distances $d_a = m_a - n_a$ and $d_b = m_b - n_b$ are unequal.*

Proof. By contradiction:

Assume The equation $d_a = d_b$ holds:

This implies $n_a - n_b = m_a - m_b$

Case 1: $n_a = n_b$ (see for example Figure 3):

Using $n_a = n_b$ and thereby $m_a = m_b$, it follows that $l_a \neq l_b$ holds. This, however implies

$$0 < \min(l_a, l_b) < \max(l_a, l_b) \tag{1}$$

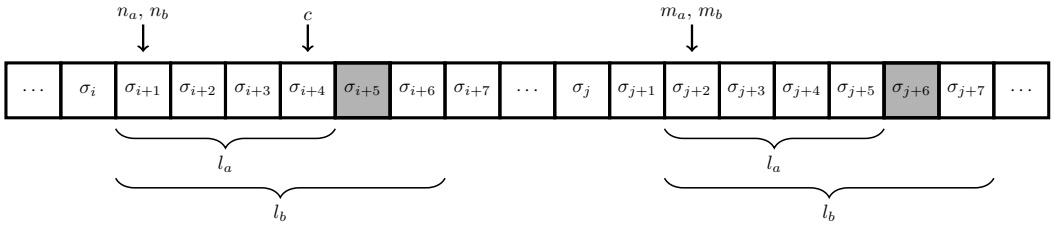


Figure 3 Case 1 of Lemma 5. The characters σ_{i+5} and σ_{j+6} have to be unequal because of their position just outside (n_a, m_a, l_a) and have to be equal because of their position in (n_b, m_b, l_b).

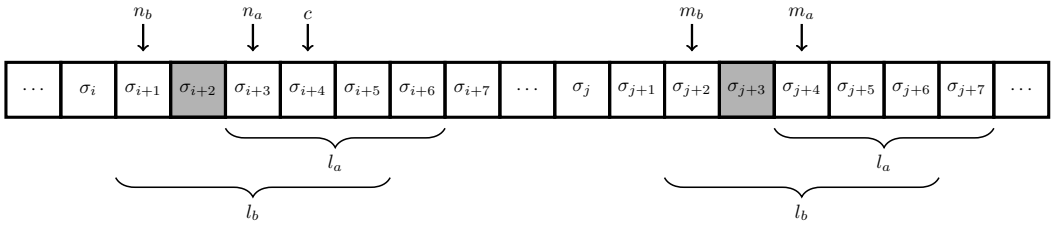


Figure 4 Case 2 of Lemma 5. The characters σ_{i+2} and σ_{j+3} have to be unequal because of their position just outside (n_a, m_a, l_a) and have to be equal because of their position in (n_b, m_b, l_b).

and thereby

$$S[n_a + \min(l_a, l_b)] \overset{1}{=} S[m_a + \min(l_a, l_b)].$$

Hence, either (n_a, m_a, l_a) or (n_b, m_b, l_b) is not a maximal pair.
Therefore this case is not possible.

Case 2: $n_a \neq n_b$ (see for example Figure 4):
Without loss of generality $n_a > n_b$ holds. Since $n_a \leq c$ and $c \leq n_b + l_b$ hold, the inequality

$$0 \leq n_a - n_b - 1 < l_b \tag{2}$$

follows and using $n_a - n_b = m_a - m_b$ we get

$$\begin{aligned} S[n_a - 1] = S[n_b + (n_a - n_b - 1)] &\overset{2}{=} S[m_b + (n_a - n_b - 1)] \\ &= S[m_b + (m_a - m_b - 1)] = S[m_a - 1] \end{aligned}$$

Hence, (n_a, m_a, l_a) is not a maximal pair.
Therefore this case is not possible.

Since all cases contradict the assumption, the distances d_a and d_b are unequal. ◀

▶ **Lemma 6.** *Let S be a string. Let further (n_a, m_a, l_a) and (n_b, m_b, l_b) be maximal pairs in S with distances $d_a \neq d_b$. Define the difference of the distances $\Delta_d = d_a - d_b$. Then $S[\max(n_a, n_b)..\min(n_a + l_a, n_b + l_b) - 1]$ is $|\Delta_d|$-periodic.*

Proof. Without loss of generality $n_a \geq n_b$ holds. Then $\max(n_a, n_b) = n_a$ holds and the string $S[\max(n_a, n_b)..\min(n_a + l_a, n_b + l_b) - 1]$ has length $\min(l_a, n_b - n_a + l_b)$.

Let x be a natural number such that $0 \leq x < x + |\Delta_d| < \min(l_a, n_b - n_a + l_b)$ holds.

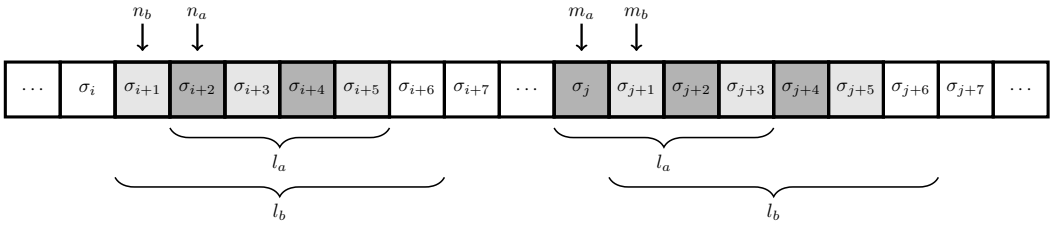


Figure 5 Case 1 of Lemma 6. The characters σ_{i+2} and σ_{j+2} have to be equal because of their position in (n_b, m_b, l_b) and the characters σ_{j+2} and σ_{i+4} have to be equal because of their position in (n_a, m_a, l_a).

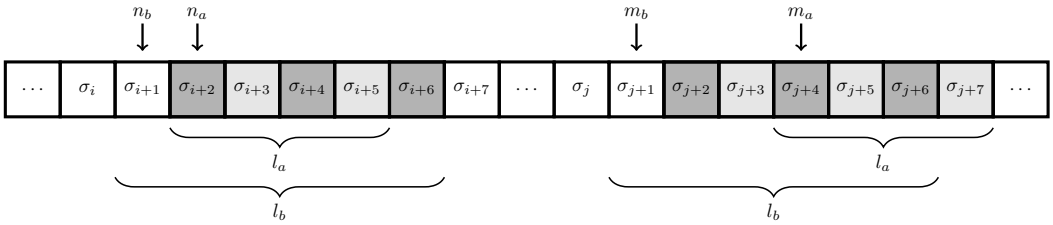


Figure 6 Case 2 of Lemma 6. The characters σ_{i+2} and σ_{j+4} have to be equal because of their position in (n_a, m_a, l_a) and the characters σ_{j+4} and σ_{i+4} have to be equal because of their position in (n_b, m_b, l_b).

Case 1: $\Delta_d < 0$ (see for example Figure 5):

In this case

$$0 \leq n_a - n_b \leq x + (n_a - n_b)$$
$$x + (n_a - n_b) < n_b - n_a + l_b + (n_a - n_b) = l_b \tag{3}$$

and

$$0 < x + |\Delta_d| < l_a \tag{4}$$

hold. Therefore

$$\begin{aligned} S[n_a + x] &= S[n_b + x + (n_a - n_b)] \\ &\overset{3}{=} S[m_b + x + (n_a - n_b)] \\ &= S[m_a + x - (m_a - m_b) + (n_a - n_b)] \\ &= S[m_a + x - \Delta_d] \\ &= S[m_a + x + |\Delta_d|] \\ &\overset{4}{=} S[n_a + x + |\Delta_d|] \end{aligned}$$

holds.

Case 2: $\Delta_d > 0$ (see for example Figure 6):

In this case

$$0 \leq x < l_a \tag{5}$$

and

$$0 < x + |\Delta_d| = x + \Delta_d = x + (m_a - m_b) - (n_a - n_b) \leq x + (m_a - m_b)$$
$$\begin{aligned} x + (m_a - m_b) &= x + \Delta_d + (n_a - n_b) \\ &= x + |\Delta_d| + (n_a - n_b) < n_b - n_a + l_b + (n_a - n_b) = l_b \end{aligned} \tag{6}$$

hold. Therefore

$$\begin{aligned} S[n_a + x] &\overset{5}{=} S[m_a + x] \\ &= S[m_b + x + (m_a - m_b)] \\ &\overset{6}{=} S[n_b + x + (m_a - m_b)] \\ &= S[n_a + x + (m_a - m_b) - (n_a - n_b)] \\ &= S[n_a + x + \Delta_d] \\ &= S[n_a + x + |\Delta_d|] \end{aligned}$$

holds.

Therefore for all numbers x with $0 \leq x < x + |\Delta_d| < \min(l_a, n_b - n_a + l_b)$ the equation $S[n_a+x] = S[n_a+x+|\Delta_d|]$ holds. Therefore the string $S[\max(n_a, n_b)..\min(n_a+l_a, n_b+l_b)-1]$ is $|\Delta_d|$-periodic. ◀

To use the periodicities we will utilize the following lemma. The simplification used here was presented in the book of Crochemore and Rytter in [5]. The original Lemma comes from the article [7] of Fine and Wilf.

▶ **Lemma 7** (Weak Periodicity Lemma). *Let P be a string with periods Δ_1 and Δ_2 such that $\Delta_1 + \Delta_2 \leq |P|$. Then $\gcd(\Delta_1, \Delta_2)$ is a period of P.*

With all this preparation it is now possible to count maximal pairs around given indices:

▶ **Theorem 8.** *Let S be a string. Let $F_1F_2 \dots F_zF_{z+1} = S\$$ be an LZ77-decomposition of $S\$$. Let $s_1, s_2, \dots, s_z, s_{z+1}$ be the starting indices of the LZ77-factors in $S\$$. Let $q \in \mathbb{N}_{\geq 2}$ and $i, j \in \{1, 2, \dots, z, z+1\}$ be natural numbers.*
Then the number of different maximal pairs (n_k, m_k, l_k) such that for all k

- *the substring $S[n_k..s_i - 1]$ is not a fractional power with exponent greater than or equal to q,*
- *the substring $S[s_i..n_k + l_k - 1]$ is not a fractional power with exponent greater than or equal to q,*
- *the starting index s_i is contained in the interval $[n_k, n_k + l_k]$,*
- *the starting index s_{i+1} is not contained in the interval $[n_k, n_k + l_k]$ and*
- *the starting index s_j is contained in the interval $[m_k, m_k + l_k]$*

is bounded from above by $18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil$

Proof. By contradiction:

Assume there are at least $(18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1$ different maximal pairs with the restrictions given by the prerequisites:

We will now use the pigeonhole principle until we get two pairs of maximal pairs which have a huge overlap and similar distances.

For each of these maximal pairs (n_k, m_k, l_k) at least one of the following options hold:

- At least half of the interval $[n_k, n_k + l_k - 1]$ lies before s_i (i.e. $n_k + \frac{l_k}{2} \leq s_i$), or
- At least half of the interval $[n_k, n_k + l_k - 1]$ lies after $s_i - 1$ (i.e. $n_k + \frac{l_k}{2} \geq s_i$).

Since there are two options and at least $(18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1$ maximal pairs at least one of these options hold for

$$\left\lceil \frac{(18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1}{2} \right\rceil = (9q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1$$

maximal pairs. By symmetry we can assume without loss of generality that there are $(9q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1$ of the given maximal pairs satisfying $n_k + \frac{l_k}{2} \leq s_i$.

Since all of these $(9q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1$ maximal pairs (n_k, m_k, l_k) satisfy both $s_i \in [n_k, n_k + l_k]$ and $s_{i+1} \notin [n_k, n_k + l_k]$, the inequality $l_k \leq |F_1F_2 \dots F_i|$ holds.

Taking the logarithm yields

$$0 = log_q(1) \leq log_q(l_k) \leq log_q(|F_1F_2 \dots F_i|) \leq \lceil \log_q(|F_1F_2 \dots F_i|) \rceil.$$

Since every $log_q(l_k)$ lies in at least one of the $\lceil \log_q(|F_1F_2 \dots F_i|) \rceil$ intervals $[h, h+1]$ with $0 \leq h < \lceil \log_q(|F_1F_2 \dots F_i|) \rceil$, the pigeonhole principle yields that there has to be a natural number L' such that

$$\left\lceil \frac{(9q\lceil \log_q(|F_1F_2 \dots F_i|) \rceil) + 1}{\lceil \log_q(|F_1F_2 \dots F_i|) \rceil} \right\rceil \geq 9q + 1$$

of these maximal pairs have length $L' \leq log_q(l_k) \leq 1 + L'$.

For $L = q^{L'}$ this gives a natural number L such that $L \leq l_k \leq qL$ holds for these $9q + 1$ maximal pairs.

Therefore there is a real number θ such that

- for at least $3q + 1$ of these $9q + 1$ maximal pairs $L \leq l_k \leq \theta L$ holds and
- for at least $6q + 1$ of these $9q + 1$ maximal pairs $\theta L \leq l_k \leq qL$ holds.

With the given restrictions $s_i \in [n_k, n_k + l_k]$ and $s_j \in [m_k, m_k + l_k]$ from the main assumption as well as $n_k + \frac{l_k}{2} \leq s_i$ from the application of the pigeonhole principle it follows that $n_k + \frac{l_k}{2} \leq s_i \leq n_k + l_k$ and $m_k \leq s_j \leq m_k + l_k$ hold. Therefore

$$s_j - s_i - \frac{l_k}{2} \leq (m_k + l_k) - \left(n_k + \frac{l_k}{2}\right) - \frac{l_k}{2} = m_k - n_k = d_k \text{ and}$$

$$d_k = m_k - n_k = m_k - (n_k + l_k) + l_k \leq s_j - s_i + l_k$$

hold and d_k lies in the interval $[s_j - s_i - \frac{l_k}{2}, s_j - s_i + l_k]$.

Of the $6q + 1$ maximal pairs (n_k, m_k, l_k) with $\theta L \leq l_k \leq qL$, each d_k is in at least one of the $6q$ intervals $[s_j - s_i - \frac{qL}{2} + h \cdot \frac{1}{4}L, s_j - s_i - \frac{qL}{2} + (h+1)\frac{1}{4}L]$ with $0 \leq h < 6q$. Therefore, the pigeonhole principle yields that at least

$$\left\lceil (6q+1)\frac{\frac{1}{4}L}{\frac{3}{2}qL} \right\rceil = \left\lceil (6q+1)\frac{1}{6q} \right\rceil = 2$$

of these maximal pairs have distances d_a, d_b with $|d_a - d_b| \leq \frac{1}{4}L$. Using Lemma 5 and Lemma 6 as well as $n_k + \frac{\theta L}{2} \leq n_k + \frac{l_k}{2} \leq s_i$ and $s_i \leq n_k + l_k$ for all these maximal pairs, we obtain that there is a maximal pair $(n_\alpha, m_\alpha, l_\alpha)$ such that $n_\alpha \leq s_i - \frac{\theta}{2}L$ and such that $S[n_\alpha..s_i - 1]$ has a period of $0 < \Delta_\alpha \leq \frac{1}{4}L$.

Similarly it can be shown that of the $3q + 1$ maximal pairs (n_k, m_k, l_k) with $L \leq l_k \leq \theta L$, there is a maximal pair $(n_\beta, m_\beta, l_\beta)$ such that $n_\beta \leq s_i - \frac{1}{2}L$ and such that $S[n_\beta..s_i - 1]$ has a period of $0 < \Delta_\beta \leq \frac{\theta}{2q}L$.

Since $S[n_\alpha..s_i - 1]$ is not a fractional power with exponent greater than or equal to q, we obtain $\frac{s_i - n_\alpha}{\Delta_\alpha} < q$. With $\frac{\theta L}{2} \leq \frac{l_\alpha}{2} \leq s_i - n_\alpha$ and $\Delta_\alpha \leq \frac{1}{4}L$ it follows that $\theta < \frac{q}{2}$ and hence $\Delta_\beta \leq \frac{1}{4}L$ hold.

Since $S[\max(n_\alpha, n_\beta)..s_i - 1]$ has length of at least $\frac{1}{2}L$ and is Δ_α-periodic as well as Δ_β-periodic with $\Delta_\alpha + \Delta_\beta \leq \frac{1}{2}L$ the periodicity lemma is applicable and shows that $S[\max(n_\alpha, n_\beta)..s_i - 1]$ is $\gcd(\Delta_\alpha, \Delta_\beta)$-periodic. This implies that $S[s_i - \Delta_\alpha..s_i - 1]$ is $\gcd(\Delta_\alpha, \Delta_\beta)$-periodic. Since $S[n_\alpha..s_i - 1]$ is Δ_α-periodic and at least one substring with length Δ_α is $\gcd(\Delta_\alpha, \Delta_\beta)$-periodic, even $S[n_\alpha..s_i - 1]$ is $\gcd(\Delta_\alpha, \Delta_\beta)$-periodic.

However with $\gcd(\Delta_\alpha, \Delta_\beta) \leq \Delta_\beta \leq \frac{\theta}{2q}L$ this implies that the substring $S[n_\alpha..s_i - 1]$ with at least $\frac{\theta}{2}L$ characters has a period with length of at most $\frac{\theta}{2q}L$. Therefore $S[n_\alpha..s_i - 1]$ is a fractional power with exponent greater than or equal to q.

This however contradicts the assumption and thereby proves the theorem. ◀

Now it is time to prove Theorem 1 which was stated in the introduction:

▶ **Theorem 1.** *Let S be a string. Let z be the number of (self-referential) LZ77-factors in an LZ77-decomposition of S. Let q be a number such that S does not contain q-th powers. Then the number of maximal repeats in S is bounded from above by $3q(z+1)^3 - 2$. Also, the Compacted Directed Acyclic Word Graph (CDAWG) of S has at most $3q(z+1)^3$ nodes.*

Proof. Lemma 3 shows that it is sufficient to count maximal pairs (n_k, m_k, l_k) with $n_k < m_k$. Lemma 4 shows that we can additionally require $s_i \in [n_k, n_k + l_k]$ and $s_j \in [m_k, m_k + l_k]$ for some starting indices s_i and s_j of the $z+1$ LZ77-factors of the string $S\$$.

Since the first LZ77-factor is always a single character, the equation $|F_1| = 1 = q^0$ holds. Since S does not contain a q-th power, every LZ77-factor can at most multiply the length of the string by the factor q. Therefore $|F_1F_2 \dots F_i| \leq q|F_1F_2 \dots F_{i-1}|$ holds. Induction therefore yields $|F_1F_2 \dots F_i| \leq q^{i-1}$. This implies $\lceil \log_q(|F_1F_2 \dots F_i|) \rceil \leq i - 1$

Since \$ does not occur in S, the last LZ77-factor of $S\$$ consists of only the character $\$ = S[|S|]$. Since $n_k < m_k \leq |S| - l_k$ holds, the inequality $n_k + l_k < |S|$ holds as well. This implies that s_{z+1} is not contained in $[n_k, n_k + l_k]$.

Using Theorem 8 and summing up over all pairs (s_i, s_j) with $s_i \leq s_j$ and $s_i \leq s_z$ yield that there are at most

$$\begin{aligned}
\sum_{i=1}^{z}\sum_{j=i}^{z+1}\left(18q \cdot \lceil \log_q(|F_1F_2 \dots F_i|) \rceil\right) &\leq 18q\sum_{i=1}^{z}\sum_{j=i}^{z+1}(i-1) \\
&= 18q\sum_{i=1}^{z}(i-1)(z+2-i) \\
&= 18q\sum_{i=1}^{z}(-i^2 + i(z+3) - (z+2)) \\
&= 3q(z^3 + 3z^2 - 4z) \\
&\leq 3q(z+1)^3 - 2
\end{aligned}$$

maximal repeats in S.

Raffinot shows in Theorem 1 of [12] that the maximal repeats of a string S are exactly the representatives of the internal states of the CDAWG of S. This implies that the CDAWG of S has at most $3q(z+1)^3$ states. ◀

4 Tightness

The goal of this section is to prove that for every q, z with $2000 \leq z \leq q$ there are strings without q-th powers which can be described with z LZ77-factors and which have at least $\frac{1}{500}qz^3$ maximal repeats. This also proves that the upper bound given in the last section can not be improved by more than a constant factor.

The proof of Theorem 8 suggests that high powers are necessary in order to have many maximal repeats. We therefore create a string $V_{v,q,q}$ consisting of nested $2q$-th powers first and then build a bigger string consisting of $V_{v,q,q}$ and some shortened copies of $V_{v,q,q}$.

We therefore define for natural numbers v, d, q and c with $v \geq 1$ and $d \leq q$:

$$V_{0,*,*} := \sigma_0,$$
$$V_{v,c,q} := (V_{v-1,q,q})^c \sigma_v (V_{v-1,q,q})^c,$$
$$L_{v,c,q} := (V_{v-1,q,q})^c \sigma_v (V_{v-1,q,q})^c,$$
$$R_{v,c,q} := (V_{v-1,q,q})^q \sigma_v (V_{v-1,q,q})^c,$$
$$C_{v,c,q} := L_{1,c,q} L_{2,c,q} \dots L_{v-1,c,q} V_{v,c,q} R_{v-1,c,q} \dots R_{2,c,q} R_{1,c,q} \text{ and}$$
$$S_{v,d,q} := V_{v,q,q} \left(\prod_{i=1}^{d} \$ C_{v,q-i,q} \right).$$

In order to find the highest power and the number of LZ77-factors, it is first necessary to show, that the $C_{v,q-i,q}$ are indeed proper substrings of $V_{v,q,q}$.

▶ **Lemma 9.** *For $c \leq q-1$ the string $L_{1,c,q} L_{2,c,q} \dots L_{w,c,q}$ is a proper suffix of $V_{w,q,q}$ and the string $R_{w,c,q} \dots R_{2,c,q} R_{1,c,q}$ is a proper prefix of $V_{w,q,q}$.*

Proof. This can easily be shown with an induction over w. ◀

▶ **Corollary 10.** *For $c \leq q-1$ the string $C_{v,c,q}$ is a proper substring of $V_{v,q,q}$*

This corollary leads to an upper bound for the highest power as well as for the necessary number of LZ77-factors of $S_{v,d,q}$.

▶ **Lemma 11.** *The string $S_{v,d,q}$ does not contain a $(2q+1)$-th power.*

Proof. by contradiction:

Assume there is a $q+1$-th power P in $S_{v,d,q}$.

The power P can not contain a \$ because the character \$ occurs only $d \leq q$ times in $S_{v,d,q}$. Therefore, using the previous lemma, the power P has to be a substring of $V_{v,q,q}$.

The power P can not contain a σ_v because the character σ_v occurs only once in $V_{v,q,q}$. Therefore the power P has to be a substring of $(V_{v-1,q,q})^q$.

The power P can not contain a σ_{v-1} because the character σ_{v-1} occurs only q times in $(V_{v-1,q,q})^q = ((V_{v-2,q,q})^q v_{v-1} (V_{v-2,q,q})^q)^q$. Therefore the power P has to be a substring of $(V_{v-2,q,q})^{2q}$.

It can be inductively shown that P can not contain σ_j for $j \in \{v-2, v-3, \dots, 1\}$ because the character σ_j occurs only $2q$ times in $(V_{j,q,q})^{2q} = ((V_{j-1,q,q})^q v_j (V_{j-1,q,q})^q)^{2q}$. Therefore the power P has to be a substring of $(V_{j-1,q,q})^{2q}$.

Since there are no characters left, this is a contradiction.

Therefore the string $S_{v,d,q}$ does not contain a $(2q+1)$-th power. ◀

▶ **Lemma 12.** *The string $S_{v,d,q}$ can be written with at most $1+3v+2d$ LZ77-factors.*

Proof. Since the string $V_{0,*,*}$ consist of a single letter, it can be written with a single LZ77-factor. By induction, the string $V_{v,c,q} = V_{v-1,q,q} \cdot (V_{v-1,q,q})^{c-1} \cdot \sigma_v \cdot (V_{v-1,q,q})^c$ can be written with at most $1+3v$ LZ77-factors. Using Corollary 10 yields that the string $S_{v,d,q} := V_{v,q,q} \cdot \left(\prod_{i=1}^{d} \$ \cdot C_{v,q-i,q} \right)$ can be written with at most $1+3v+2d$ LZ77-factors. ◀

In order to give a lower bound of the maximal repeats of $S_{v,d,q}$, we show that for natural numbers w, l, m and r with $1 \leq w \leq v-1$ and $1 \leq m+1 \leq q-d \leq l, r \leq q-1$

$$M_{w,l,m,r,q} := L_{1,l,q} L_{2,l,q} \dots L_{w,l,q} (V_{w,q,q})^m R_{w,r,q} \dots R_{2,r,q} R_{1,r,q}.$$

are maximal repeats of $S_{v,d,q}$.

▶ **Lemma 13.** *For*

$$\begin{aligned} w &\leq v-1 \\ m+1 \leq l, r &\leq q-1 \end{aligned}$$

the string $M_{w,l,m,r,q}$ is a proper prefix of $C_{v,l,q}$ and a proper suffix of $C_{v,r,q}$.

Proof. Using Lemma 9 the string $(V_{w,q,q})^m R_{w,r,q} \dots R_{2,r,q} R_{1,r,q}$ is a prefix of $(V_{w,q,q})^{m+1}$ which is a proper prefix of $L_{w+1,l,q}$. Therefore $M_{w,l,m,r,q}$ is a proper prefix of $C_{v,l,q}$. Similarly $M_{w,l,m,r,q}$ is a proper suffix of $C_{v,r,q}$. ◀

▶ **Corollary 14.** *If $1 \leq w \leq v-1$ and $1 \leq m+1 \leq q-d \leq l, r \leq q-1$ hold, the string $M_{w,l,m,r,q}$ is a maximal repeat of $S_{v,d,q}$.*

Proof. Since $M_{w,l,m,r,q}$ is a proper prefix of $C_{v,l,q}$, the string $\$M_{w,l,m,r,q}\sigma_*$ appears in $S_{v,d,q}$. Since $M_{w,l,m,r,q}$ is a proper prefix of $C_{v,r,q}$, the string $\sigma_* M_{w,l,m,r,q}\$$ appears in $S_{v,d,q}$. These two occurrences form a maximal pair. Therefore, the string $M_{w,l,m,r,q}$ is a maximal repeat of $S_{v,d,q}$. ◀

▶ **Corollary 15.** *The string $S_{v,d,q}$ has at least $(v-1)(q-d)d^2$ maximal repeats*

Combining Lemma 11, Lemma 12 and Corollary 15 yields Theorem 2 as given in the introduction:

▶ **Theorem 2.** *For $2000 \leq z \leq q$ there is a string S without q-th powers which can be expressed by z LZ77-factors and which has at least $\frac{1}{500} q z^3$ maximal repeats.*

Proof. Define $S = S_{\lfloor \frac{z}{9} \rfloor, \lfloor \frac{z}{3} \rfloor - 1, \lfloor \frac{q-1}{2} \rfloor}$. Using Lemma 11 the string S has no q-th power. Using Lemma 12 the string S can be described with $1 + 3\lfloor \frac{z}{9} \rfloor + 2\left(\lfloor \frac{z}{3} \rfloor - 1\right) \leq z$ LZ77-factors. Using Corollary 15 the string S has at least

$$\begin{aligned} &\left(\left\lfloor \frac{z}{9} \right\rfloor - 1\right)\left(\left\lfloor \frac{q-1}{2} \right\rfloor - \left(\left\lfloor \frac{z}{3} \right\rfloor - 1\right)\right)\left(\left\lfloor \frac{z}{3} \right\rfloor - 1\right)^2 \\ &\quad \geq \left(\frac{z}{9} - 2\right)\left(\frac{q}{2} - \frac{3}{2} - \frac{z}{3}\right)\left(\frac{z}{3} - 2\right)^2 \\ &\quad \geq \left(\frac{z}{9} - 2\frac{z}{2000}\right)\left(\frac{q}{2} - \frac{3}{2}\frac{q}{2000} - \frac{z}{3}\cdot\frac{q}{z}\right)\left(\frac{z}{3} - 2\frac{z}{2000}\right)^2 \\ &\quad = \left(\frac{1}{9} - 2\frac{1}{2000}\right)\left(\frac{1}{2} - \frac{3}{2}\frac{1}{2000} - \frac{1}{3}\right)\left(\frac{1}{3} - 2\frac{1}{2000}\right)^2 q z^3 \\ &\quad \geq \frac{1}{500} q z^3 \end{aligned}$$

maximal repeats. ◀

5 Conclusion

Since Theorem 1 suggests that well-compressed strings with many maximal repeats also have high powers and Theorem 8 even suggests that these high powers are not hidden inside the maximal repeats but are either a prefix or a suffix of them, it seems promising to do some more research on the maximal repeats of strings with high powers.

It might be possible to derive a data structure from the CDAWG by merging nodes stemming from similar powers of the same base. This data structure and its size as well as its usability will be determined in future work.

There are three more problems which should be researched:

The upper bound for the number of maximal repeats and the maximal repeats of the string given in section 4 differ by a factor of almost 1500. Even for strings with very high powers the factor is almost 500. This huge gap leaves room for further investigation.

The string in section 4 uses that the highest power is bigger than the parameter d. If the highest power is smaller than the number of LZ77-factors, the number of maximal repeats is only cp^3z for some constant c. It is an open question, whether the upper bound given by Theorem 1 is still tight up to constant for strings without high powers.

While the upper bound for the number of maximal repeats $3q(z+1)^3$ presented in this paper is tight up to a constant factor, the string $\sigma_1\sigma_2\ldots\sigma_{z-2}(\sigma_{z-1})^{q-1}$ has z LZ77-factors, no q-th power but a $(q-1)$-power and has only the $q-2$ maximal repeats $(\sigma_{z-1})^i$ with $1 \leq i \leq q-2$. Therefore, some additional structures should be taken into account in order to get a good estimate for the number of maximal repeats in a string.

References

1 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite Repetition-Aware Data Structures. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

2 Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

3 Anselm Blumer, Andrzej Ehrenfeucht, and David Haussler. Average sizes of suffix trees and DAWGs. *Discrete Applied Mathematics*, 24(1-3):37–45, 1989. doi:10.1016/0166-218X(92)90270-K.

4 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

5 Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994. URL: http://www-igm.univ-mlv.fr/%7Emac/REC/B1.html.

6 Chiara Epifanio, Filippo Mignosi, Jeffrey Shallit, and Ilaria Venturini. On Sturmian graphs. *Discrete Applied Mathematics*, 155(8):1014–1030, 2007. doi:10.1016/j.dam.2006.11.003.

7 N. J. Fine and H. S. Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: http://www.jstor.org/stable/2034009.

8 Isamu Furuya, Takuya Takagi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Takuya Kida. MR-RePair: Grammar Compression based on Maximal Repeats. *CoRR*, abs/1811.04596, 2018. arXiv:1811.04596.

9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

10 Jacques Nicolas, Christine Rousseau, Anne Siegel, Pierre Peterlongo, François Coste, Patrick Durand, Sebastien Tempel, Anne-Sophie Valin, and Frédéric Mahé. Local and Maximal Repeats. URL: https://www.researchgate.net/publication/228940275_Local_and_Maximal_Repeats.

11 Jakub Radoszewski and Wojciech Rytter. On the structure of compacted subword graphs of Thue-Morse words and their applications. *J. Discrete Algorithms*, 11:15–24, 2012. doi:10.1016/j.jda.2011.01.001.

12 Mathieu Raffinot. On maximal repeats in strings. *Inf. Process. Lett.*, 80(3):165–169, 2001. doi:10.1016/S0020-0190(01)00152-1.

13 Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-Space LCE Data Structure with Constant-Time Queries. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.10.

Dichotomic Selection on Words: A Probabilistic Analysis

Ali Akhavi
GREYC (Normandie Université, Unicaen, EnsiCaen, Cnrs), 14000, Caen, France
ali.akhavi@unicaen.fr

Julien Clément
GREYC (Normandie Université, Unicaen, EnsiCaen, Cnrs), 14000, Caen, France
julien.clement@unicaen.fr

Dimitri Darthenay
GREYC (Normandie Université, Unicaen, EnsiCaen, Cnrs), 14000, Caen, France
dimitri.darthenay@gmail.com

Loïck Lhote
GREYC (Normandie Université, Unicaen, EnsiCaen, Cnrs), 14000, Caen, France
loick.lhote@unicaen.fr

Brigitte Vallée
GREYC (Normandie Université, Unicaen, EnsiCaen, Cnrs), 14000, Caen, France
brigitte.vallee@unicaen.fr

Abstract

The paper studies the behaviour of selection algorithms that are based on dichotomy principles. On the entry formed by an ordered list L and a searched element $x \notin L$, they return the interval of the list L the element x belongs to. We focus here on the case of words, where dichotomy principles lead to a selection algorithm designed by Crochemore, Hancart and Lecroq, which appears to be "quasi-optimal". We perform a probabilistic analysis of this algorithm that exhibits its quasi-optimality on average.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Randomness, geometry and discrete structures; Theory of computation → Sorting and searching; Theory of computation → Pattern matching; Mathematics of computing → Combinatorics on words; Mathematics of computing → Discrete mathematics

Keywords and phrases dichotomic selection, text algorithms, analysis of algorithms, average case analysis of algorithms, trie, suffix array, lcp–array, information theory, numeration process, sources, entropy, coincidence, analytic combinatorics, depoissonization techniques

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.19

Funding Projects: DynA3S, CoDys, MetaConc (ANR), AleaEnAmSud (STICAmSud).

Acknowledgements We thank Pablo Rotondo for many interesting discussions. We also thank anonymous reviewers for pointing us to bibliographic references.

1 Introduction

The dichotomic search [18] is one of the most basic tool for locating the position of a target value x within a sorted list of n elements. This scheme, that has a classical "divide-and-conquer" flavour, is a good algorithmic compromise in many situations, because it is straightforward to implement and nonetheless guarantees a $\Theta(\log n)$ number of comparisons between x and the n elements of the list.

General context. In this paper, we are interested in the case when the list and the target values are *words*, that are emitted by a *source*. Then, the comparison between two words is the usual (lexicographic) comparison, and the performance of the dichotomic selection on words is measured by the total number of *symbol comparisons*. More precisely, we study algorithms that are designed in the book of Crochemore, Hancart, and Lecroq [7, chapter 4]. In this context, the list is fixed and many target values are searched: it is then natural to consider the list sorted (thanks to a precomputation step), and use dichotomic principles[1]. Such algorithms are notably the basis for efficient implementations of searching in the suffix array [19, 2, 22, 16, 21], which is a widely used index structure in text algorithmics [14, 7].

We focus on a particular algorithm described in [7, Chapter 4], and called here CLEVER-DICHO-SELECT. The authors explain there that the sorting precomputation is not actually sufficient to build an efficient search procedure: one needs to use a supplementary structure, called the LCP–array in [17, 20, 15, 12, 10] that stores the length of longest common prefixes between consecutive strings of the list[2]. The precomputation step which determines the LCP–array has a worst-case complexity linear in the total number of symbols of all words in the list. With these two precomputation steps at hand (the sorting precomputation and the LCP–array precomputation), the technique becomes very efficient for searching for a string of length m in a list of n strings with $O(m + \log n)$ symbol comparisons (in the worst-case). Remark that the usual algorithm that reads the word x and uses a *lexicographic tree* or *trie* for finding the longest common prefix of the target value x within the list of strings (which is the minimal information needed to locate the position of x) yields $O(m)$ symbol comparisons.

Motivations. There are three motivations for such a work.

(a) We wish to perform a precise analysis of variants of the dichotomic search algorithm on strings in the following framework described with four features : (i) the performance is measured in terms of the number of *symbol comparisons*; (ii) we adopt a *probabilistic* or *average-case* point of view which is significantly different from the worst-case one; (iii) we are interested in the *asymptotics*, when the number n of strings become large, and accordingly choose to deal with infinite words (to be detailed later); (iv) finally, we wish to determine the precise constants involved in the analysis.

(b) A main motivation is also to better understand how the structure of data influence the performance of algorithms. In many real applications, such data, of highly composite nature, are often aggregates of elementary symbols, (such as bits, bytes, characters...). It is then legitimate to consider such data as *words*, and the elementary cost is now the comparisons between symbols. This present work follows a series of recent works of the authors [27, 6, 4] where they revisited the *probabilistic* analysis of a panel of some classic sorting and selecting algorithms from this point of view (for instance Quicksort). Here we take the same perspective and wish to analyze the general dichotomic strategy, from the basic algorithm to the more clever version of [7, Chapter 4], when it operates on complex data.

(c) Our probabilistic modelling first involves the concept of a source, that describes how the input words (the elements of the list L, and the searched element x) are emitted. Such a source denoted by $\mathcal{M}$ extends the notion of a numeration process (see [25]), and

[1] Other strategies are possible apart from the dichotomic principles for the problem searching in a sorted list of strings (see [1, 11]).

[2] The used structure is in fact the LCP–table which is a slight variation of the usual LCP–array, as we will describe later.

any (infinite) word emitted by the source $\mathcal{M}$ can be viewed as the expansion of a real number of the unit interval in "base" $\mathcal{M}$. Besides this source $\mathcal{M}$, that emits the input words, there appears also a second *implicit* source (the regular binary source $\mathcal{B}$), which models the dichotomy process. The interplay of these two sources is then central in our analyses. Even though such an interplay between two sources often arises in numeration contexts, and is well studied there (see for instance [3, 8]), notably in relation to the various Changing Base algorithms, it came as a surprise for us that it also arises in our analyses.

Main results. In the context of our analysis, defined by the four features described in (a), we analyze in Theorem 9 the CLEVER-DICHO-SELECT algorithm. We first analyze the precomputation step, that builds (and stores) the LCP–table (an integer array of length $2n+1$ which is a slight variation of the usual LCP–array), and prove that the mean complexity cost for building the table is of order $\Theta(n)$. Then, we analyse the mean complexity cost of the algorithm itself and prove that it is of order $\Theta(\log n)$. It is thus quasi-optimal[3] *on average*. We precisely study the constants hidden in the Θ, and relate them to the main characteristics of the source, and, in particular, to the interplay between the two sources described in (c): the *input source* that emits the words, and the *dichotomic source* which models the dichotomic process.

Methodology. We first exhibit the main costs for evaluating the time complexity of the algorithm, that are not highlighted in the previous work [7]. These costs are read on the two main structures, the dichotomic tree $D(L)$ which underlies the partitioning process, together with *trie* $T(L)$ built on the list L (which was not explicit in [7]). We also introduce another strategy for computing the LCP–table related to an original parameter called `dic`. Then, we perform the average-case analysis of the main costs: we mainly deal with what we call *costs with toll* on the trie $T(L)$ (defined in Eq. (5)), and we adapt the analytic combinatorics methodology for such an analysis, as described both in [5] and [26].

Plan of the paper. Section 2 first recalls the general framework of the dichotomic strategy. Then, Section 3 focuses on the selection problems on words; it presents the CLEVER-DICHO-SELECT algorithm, and exhibits its main costs of interest. Finally, Section 4 is devoted to the probabilistic analysis of these costs, and proves the quasi-optimality (on average) of the CLEVER-DICHO-SELECT algorithm.

2 Basic Dichotomic Selection

Let us consider an ordered set $\mathcal{X}$. The problem SELECT (L, x) takes as input a list L of n distinct elements of $\mathcal{X}$, together with an element $x \notin L$, and determines the rank of x in L, namely the integer i for which x will be the i-th smallest element in the sorted set $L \cup \{x\}$. When the list $L = (L_1, \ldots, L_n)$ is already sorted, the well-known dichotomic strategy may be used. It deals with the *extended list* $\overline{L} := (L_0, L_1, \ldots, L_n, L_{n+1})$, where two *guards* are added to the initial list L, a *strict infimum* L_0 and a *strict supremum* L_{n+1} satisfying the strict inequalities $L_0 < y < L_{n+1}$ for any $y \in \mathcal{X}$. The dichotomy principle uses the function $\text{mid} : (b, e) \mapsto \lfloor (b+e)/2 \rfloor$, and determines the *rank* of x in $L \cup \{x\}$, i.e., the index $i \in [1\,..\,n+1]$ for which $L_{i-1} < x < L_i$. The algorithm DICHO-SELECT is described in Fig. 1 (left).

[3] compared to the bound $\Theta(\log n)$ obtained *on average* when dealing with a precomputed trie.

```
DICHO-SELECT (L, x)
b := 0
e := n + 1
while b + 1 < e do
    m := ⌊(b + e)/2⌋
if  L_m < x then
    b := m
else
    e := m
return e
```

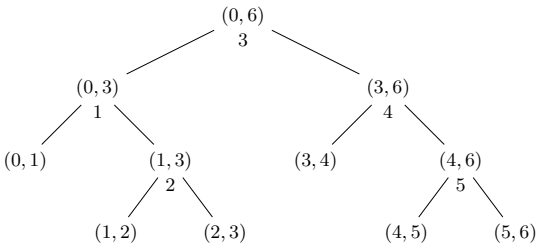


Figure 1 The DICHO-SELECT algorithm (left). The dichotomic tree D_n ($n = 5$) (right).

With each cardinality n, the dichotomy strategy associates a binary tree D_n, called the *dichotomic tree* (Fig. 1, right). Such a tree describes the partitioning process on any ordered list of cardinality n, with indices in $[1\,..\,n]$, and deals with extended indices in $[0\,..\,n+1]$.

▶ **Definition 1** (Dichotomic tree). *The* dichotomic tree D_n *associated with a cardinality* n *is a binary tree, with* n *internal nodes and* $n+1$ *external nodes. Each node is labelled by a pair* (b, e) *with* $0 \le b < e \le n+1$. *Moreover,*

- *if* $b + 1 < e$, *the node labelled by* (b, e) *is* internal*: it contains the index* $m = \mathrm{mid}(b, e)$ *and has two children, labelled by* (b, m) *(on the left) and* (m, e) *(on the right);*
- *if* $b + 1 = e$, *the node labelled by* (b, e) *is* external*: it has no children.*

An integer pair (c, d) *is dichotomic it appears as a label of a node in* D_n.

When the DICHO-SELECT algorithm is called on a sorted list L of cardinality n, the dichotomic tree D_n is adapted to the list L (and its extended version $\overline{L}$): an internal node (b, e) contains the key L_m with $m = \mathrm{mid}(b, e)$. The tree is thus called the dichotomic tree of the list L, and denoted by $D(L)$. During the execution of DICHO-SELECT on the input (L, x), the tree $D(L)$ is used as a binary search tree, and a comparison is performed at node (b, e) between the key x and the key L_m with $m = \mathrm{mid}(b, e)$.
The sequence of visited nodes forms a path in $D(L)$, that is called the *execution path* $\Pi(L, x)$. It leads to an external node with label $(i-1, i)$ such that $L_{i-1} < x < L_i$. The part $\Pi(L, x)$ that excludes the last (external) node and thus only gathers its internal nodes, is called the *internal execution path* and denoted by $\underline{\Pi}(L, x)$. The length of this path, denoted by $|\underline{\Pi}(L, x)|$ and defined as the number of its nodes, is a central parameter of DICHO-SELECT(L, x):

▶ **Lemma 2.** *The total number* $G(L, x)$ *of key comparisons performed by* DICHO-SELECT *on the input* (L, x) *is equal to the length* $|\underline{\Pi}(L, x)|$ *of the internal execution path* $\underline{\Pi}(L, x)$ *of* DICHO-SELECT *in* $D(L)$ *on input* x. *When the cardinality* n *of* L *belongs to* $[2^{p-1}\,..\,2^p - 1]$, *the length of the path* $\underline{\Pi}(L, x)$ *satisfies* $|\underline{\Pi}(L, x)| \in \{p-1, p\}$. *When* $n = 2^p - 1$, *one has* $|\underline{\Pi}(L, x)| = p$, *for any* x.

3 Dichotomic selection for words

3.1 General context

The DICHO-SELECT algorithm is straightforwardly adapted to words, as shown in [7]. The book describes the framework that appears in practical issues, where one deals with a list L of finite words (called strings), that may be possibly prefixes of other strings inside L, and where the searched string x may belong to L. This makes the pseudo code of [7] rather subtle and sometimes difficult to understand.

As we focus on asymptotic features, when both the cardinality of the list, and the length of strings tend to ∞, we are led to the case when the keys (both the elements of the list L and the searched element x) are infinite words. Furthermore, we suppose that all words in L are distinct and the word x does not belong to L. This makes the precise structure of algorithms more readable and their main parameters more apparent: this leads to a clearer asymptotic complexity analysis, notably on average.

We will design various algorithms of dichotomic flavour that adapt the DICHO-SELECT algorithm to the context of infinite words, built over some finite ordered alphabet Σ. We now deal with two different orderings, the partial prefix order $\preceq$ (defined on prefixes), and the lexicographic order $\leq$ (defined on infinite words). For two finite words w, w' on the alphabet Σ, we denote $w \preceq w'$ (or $w' \succeq w$) if w is a prefix of w'. We denote by $\Gamma(x, y)$ the *longest common prefix* between two words x and y and by $\gamma(x, y)$ its length, here called the *coincidence*[4] between x and y. Then, the number of symbol comparisons needed to compare the words x and y is equal to $\gamma(x, y) + 1$, and the main complexity parameter is the total number of symbol comparisons performed by the algorithm.

3.2 The trie structure

The *trie* structure (see [13, 23]) is the main tree structure for solving problems on words. Fig. 2 gives an example of a trie built on a list L with seven words on $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

▶ **Definition 3.** *Let L be a list of words on the ordered alphabet $\Sigma := \{a_1, a_2, \ldots, a_r\}$. The* trie *$T(L)$ is defined as follows*

- *if $L = \emptyset$, then $T(L) = \emptyset$; if $L = \{w\}$, then $T(L)$ is an* external node *which contains w;*
- *for $|L| \geq 2$, $T(L)$ is an* internal node *with r subtrees $T(L \setminus a_1), \ldots, T(L \setminus a_r)$, where $L \setminus u$ is the list formed with words of L that start with the symbol u stripped of u; the edge between the root and the i-th subtree is labelled with the symbol a_i.*

The trie only maintains the minimal prefix set $P(L)$ of symbols that is needed to distinguish all the elements of L (written underlined in blue in Fig. 2). Without any prior knowledge about the set of the words, this is the most efficient tool for comparing (and sorting) words.

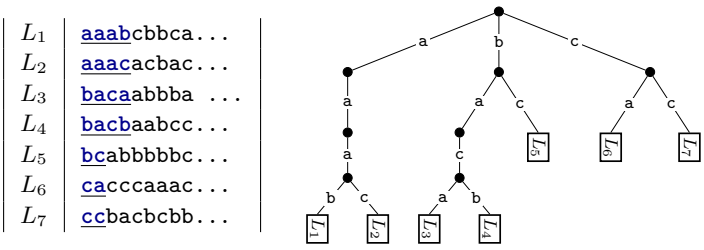


Figure 2 On the left, a list L with seven words, with its minimal prefix set $P(L)$ (underlined). On the right, the trie $T(L)$.

▶ **Proposition 4.** *The minimal number of symbol comparisons needed to solve the selection problem* SELECT *(L, x) on words (without prior knowledge on the input (L, x)) is exactly the length of the branch $B(L, x)$ of the trie $T(L \cup \{x\})$ leading to the external node associated with x.*

[4] This parameter is usually denoted as LCP(x, y) and reads as (length of) the least common prefix. As we both deal with the prefix itself, and its length, we prefer the notations Γ and γ.

The trie structure has been deeply analyzed in various contexts (see for instance [18, 9, 24]). In a quite general probabilistic framework of a source $\mathcal{M}$ with entropy $h(\mathcal{M})$ (defined in [25, 6] and later recalled in Section 4.1), the following holds, in terms of number of symbol comparisons, and on average, when the cardinality n of the list L tends to ∞ (see [5]):

- the asymptotic cost of *building the trie* $T(L)$ is equivalent to $[1/h(\mathcal{M})]\, n \log n$;
- the asymptotic size of the minimal prefix set $P(L)$ is equivalent to $[1/h(\mathcal{M})]\, n \log n$;
- the asymptotic cost $|B(L,x)|$ of *inserting* x in $T(L)$ is equivalent to $[1/h(\mathcal{M})] \log n$.

But, in the present context, the list L is assumed to be ordered. The main questions are now as follows: *Is it possible to take advantage of the ordering of the list L, with a clever use of the dichotomic strategy? Can this be done without explicitly building or storing the trie $T(L)$? Are these dichotomic algorithms close to the lower bound $|B(L,x)|$?*

The CLEVER-DICHO-SELECT algorithm proposed in [7] (and recalled later in Section 3.4) answers all these questions in an affirmative way.

Even though the trie $T(L)$ is not explicitly built during the execution of our further algorithms of type DICHO-SELECT on words, our analyses "read" operations made by the algorithms simultaneously on the two trees, the dichotomic tree $D(L)$ and the trie $T(L)$. With a node labelled by (b,e) in the dichotomic tree $D(L)$, we first consider the two elements L_e and L_b of the list $\overline{L}$ (that may be fictive guards for $e=0$ or $b=n+1$), then the prefix $\Gamma[b,e] := \Gamma(L_b, L_e)$ which is the largest common prefix between L_b and L_e, and finally its length $\gamma[b,e]$. In the case when $b=0$ or $e=n+1$, we let $\Gamma[b,e] := \varepsilon$ (the empty word) and $\gamma[b,e]=0$. This gives rise to the $\Gamma(L)$ tree that is a decoration[5] of the dichotomic tree $D(L)$ and contains at each node (b,e) of $D(L)$ the prefix $\Gamma[b,e]$ (see Fig. 5, right). This prefix $\Gamma[b,e]$ also identifies a node of the trie $T(L)$ and the associated branch in $T(L)$, from the root to this node.

A path $\Pi(L,x)$ of $D(L)$ leading to an external node of label $(i-1,i)$ gives rise to the branch labelled with $\Gamma[i-1,i]$ in the trie $T(L)$. Then, inserting x in $T(L)$ yields the branch $B(L,x)$ in the trie $T(L \cup \{x\})$. The sequence of prefixes $\Gamma[m,x] := \Gamma(L_m, x)$ is increasing (for the prefix order $\preceq$) along the path $\Pi(L,x)$, and the strategy of the algorithm defines how $\Gamma[m,x]$ is computed along $\Pi(L,x)$. We describe in the following three different strategies; we explain why the first two ones, defined in Section 3.3, are not efficient, and how they can be adapted in Section 3.4 to lead to the CLEVER-DICHO-SELECT Algorithm that is proven quasi-optimal in Section 4.2.

3.3 First two variants of dichotomic search on strings

Substituting the key-comparison "$L_m < x$" in DICHO-SELECT(L,x) with the red block of Fig. 3 (left) implementing the comparison between words gives rise to the algorithm WORD-DICHO-SELECT(L,x) described in Fig. 3 (left). The WORD-DICHO-SELECT algorithm appears to be quite naive, since it recomputes from scratch the prefix $\Gamma[m,x]$ along the path $\Pi(L,x)$. It does not use the important following property due to the ordering of the list:

▶ **Lemma 5.** *For a node of $\underline{\Pi}(L,x)$ with label (b,e) and index $m := \mathrm{mid}(b,e)$, and for any $x \in [L_b, L_e]$, the inequality holds: $\Gamma[m,x] \succeq \min(\Gamma[b,x], \Gamma[e,x]) = \Gamma[b,e]$.*

Lemma 5 leads to a more efficient algorithm, the DICHO-SELECT-WMIN algorithm described in Fig. 3 (right), where the added blue lines maintain along the path $\underline{\Pi}(L,x)$ the

[5] This means that $\Gamma(L)$ has the same structure as $D(L)$ with some extra information at nodes.

current coincidences $\ell_b := \gamma[b, x]$ and $\ell_e := \gamma[e, x]$ between the searched key x and the two ends of the interval $[L_b, L_e]$ it belongs to. However, the DICHO-SELECT-WMIN algorithm is not optimal. The following indeed holds along the execution path $\underline{\Pi}(L, x)$:

- At node (b, e), the algorithm deals with prefixes w that satisfy $\Gamma[b, e] \preceq w \preceq \Gamma[m, x]$;
- At the successor node with label (b', e'), the strict inequality $\Gamma[b', e'] \prec \Gamma[m, x]$ may hold; thus, the algorithm may go *backwards* along the branch $B(L, x)$, *repeating* the same symbol comparisons;

```
WORD-DICHO-SELECT (L, x);
b := 0 ; e := n + 1;
while b + 1 < e
      do m := mid(b, e) ;
ℓ := 0 ;
while x[ℓ] = L_m[ℓ] do
      ℓ := ℓ + 1;
if  L_m[ℓ] < x[ℓ] then
      b := m
else
      e := m
return e
```

```
DICHO-SELECT-W-MIN (L, x);
b := 0 ; e := n + 1 ;
ℓ_b := 0; ℓ_e := 0;
while b + 1 < e do
      m := mid(b, e);
ℓ := min(ℓ_b, ℓ_e);
while x[ℓ] = L_m[ℓ] do
      ℓ := ℓ + 1 ;
if  L_m[ℓ] < x[ℓ] then
      b := m; ℓ_b := ℓ ;
else
      e := m; ℓ_e := ℓ ;
return e
```

Figure 3 The two algorithms: the WORD-DICHO-SELECT (left), the DICHO-SELECT-WMIN (right).

3.4 A clever version that uses the LCP–table

This is why the authors of [7] propose to precompute the set $\{\gamma[b, e] \mid (b, e) \in D(L)\}$ that only depends on L (and not on x). When stored in an array (see Section 3.5), it is called in [7] the LCP–table and it slightly extends the notion of LCP–array which restricts to pairs of consecutive strings in L. Then, the authors prove that precise comparisons, at each node (b, e) of the path $\underline{\Pi}(L, x)$, between the four values

$$\gamma[b, x],\ \gamma[e, x],\ \gamma[b, m],\ \gamma[e, m], \quad \text{with} \quad m := \mathrm{mid}(b, e)$$

avoids backwards steps and thus redundant symbol comparisons, as it is now stated.

▶ **Lemma 6.** *Consider a node of $\underline{\Pi}(L, x)$ with label (b, e), together with the four coincidences $\ell_b = \gamma[b, x], \ell_e = \gamma[e, x], \gamma[b, m], \gamma[e, m]$. There are five (exclusive) cases in order to determine $\ell_m = \gamma[m, x]$, the successor pair (b', e') (and thus the next pair $(\ell_{b'}, \ell_{e'})$):*

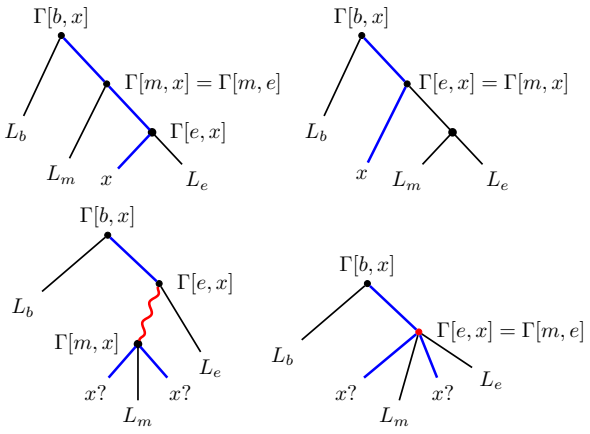
- *In the first four cases, the following holds:*
  (1) *if $\gamma[b, x] > \gamma[b, m]$, then $\ell_m = \gamma[b, m], (b', e') = (b, m)$.*
  (2) *if $\gamma[e, x] > \gamma[e, m]$, then $\ell_m = \gamma[m, e], (b', e') = (m, e)$.*
  (3) *if $\gamma[b, x] < \gamma[b, m]$, then $\ell_m = \ell_b, (b', e') = (m, e)$.*
  (4) *if $\gamma[e, x] < \gamma[e, m]$, then $\ell_m = \ell_e, (b', e') = (b, m)$.*

  The equality $\max(\ell_{b'}, \ell_{e'}) = \max(\ell_b, \ell_e)$ holds; the CLEVER-DICHO-SELECT algorithm does not perform any comparison and does not discover any new symbol;
- *In the remaining case (5), when $\gamma[b, x] = \gamma[b, m]$ and $\gamma[e, x] = \gamma[m, e]$, then a computation is needed for ℓ_m and (b', e'). The inequality $\max(\ell_{b'}, \ell_{e'}) \geq \max(\ell_b, \ell_e)$ holds;*
  The algorithm performs $[\max(\ell_{b'}, \ell_{e'}) - \max(\ell_b, \ell_e) + 1]$ symbol comparisons and discovers $[\max(\ell_{b'}, \ell_{e'}) - \max(\ell_b, \ell_e)]$ new symbols. Case (5) corresponds to a node (b, e) of the path $\underline{\Pi}(L, x)$ (called a forward node*) for which one of the two conditions (i) or (ii) hold*

(i) $\gamma[m,x] > \max(\gamma[b,x], \gamma[e,x])$
(ii) $\gamma[m,x] = \max(\gamma[b,x], \gamma[e,x]) = \max(\gamma[b,m], \gamma[e,m])$

The set of forward nodes (called the forward set*) of* $\underline{\Pi}(L,x)$ *is denoted as* $F(L,x)$.

Fig. 4 (left) leads to an easy proof of Lemma 6, explicitly based on the trie structure $T(L)$: it represents the branch $B(L,x)$ of the trie $T(L)$ (in blue) with the relative possible positions of the three branches that respectively lead to L_b, L_e, L_m. It focuses on the case where $\ell_b \le \ell_e$, which leads to four possible cases (2), (4) $(5i)$, $(5ii)$ of Lemma 6.




```
CLEVER-DICHO-SELECT(L, x);
(b, ℓ_b) := (0, 0); (e, ℓ_e) := (n + 1, 0)
while b + 1 < e do
  m := ⌊(b + e)/2⌋;
  if γ[m, e] < ℓ_e then
    (b, ℓ_b) := (m, γ[m, e])
  else if ℓ_e < γ[m, e] then
    e := m
  else if  γ[b, m] < ℓ_b then
    (e, ℓ_e) := (m, γ[b, m])
  else if ℓ_b < γ[b, m] then
    b := m
  else
    ℓ := max(ℓ_b, ℓ_e)
    while x[ℓ] = L_m[ℓ] do
      ℓ := ℓ + 1
    if L_m[ℓ] < x[ℓ] then
      (b, ℓ_b) := (m, ℓ)
    else
      (e, ℓ_e) := (m, ℓ)
return e.
```

Figure 4 On the left, description of the four cases (2), (4), $(5(i))$, $(5(ii))$ of Lemma 6 that occur when $\gamma[e,x] \ge \gamma[b,x]$. On the right, description of the CLEVER-DICHO-SELECT Algorithm.

Proof of Lemma 6. When the inequality $\Gamma[e,x] \succeq \Gamma[b,x]$ holds, we deal with the branch L_e, and consider the positions (defined by the prefixes $\Gamma[m,e]$ and $\Gamma[e,x]$) where the branches L_m and x are hung on the branch L_e; As the two words L_m and x satisfy (with respect to the lexicographic order) $L_m < L_e$ and $x < L_e$, these branches are hung on the left of the branch L_e: there are three cases for their relative positions:

$$\text{case (2): } \Gamma[e,x] \succ \Gamma[e,m]\ ; \quad \text{case (4): } \Gamma[e,x] \prec \Gamma[e,m]\ ; \quad \text{case (5): } \Gamma[e,x] = \Gamma[e,m]\,.$$

(2) The equality $\Gamma[m,x] = \Gamma[m,e]$ holds. As the branch x holds on the left of L_e, this shows that $x \in [L_m, L_e]$. Then the next (b',e') is (m,e).

(4) The equality $\Gamma[m,x] = \Gamma[e,x]$ holds. As the branch x holds on the left of L_e, this shows that $x \in [L_b, L_m]$. Then the next (b',e') is (b,m).

(5) The two branches x and L_m are hung at the same position on L_e, and both lie on the left of L_e. Then there exists a *red* path, (empty in the case $(5ii)$) which begins at prefix $\Gamma[e,x] = \Gamma[m,e]$ and represents the common suffix between L_m and x (empty in the case $(5ii)$). The comparison between L_m and x has to be performed, and begins at prefix $\Gamma[e,x]$. Remark that the equality $\Gamma[b,x] = \Gamma[b,m]$ also holds in this case. ◀

Then, Lemma 6 gives rise to the CLEVER-DICHO-SELECT algorithm described in Fig. 4 (right), whose complexity is now summarized in the following Proposition.

▶ **Proposition 7.** *Consider a list L, with its precomputed set $\gamma(L)$. Then, the total number of symbol comparisons performed by* CLEVER-DICHO-SELECT *on the input (L, x) is*

$$C(L, x) = |B(L, x)| + |F(L, x)|,$$

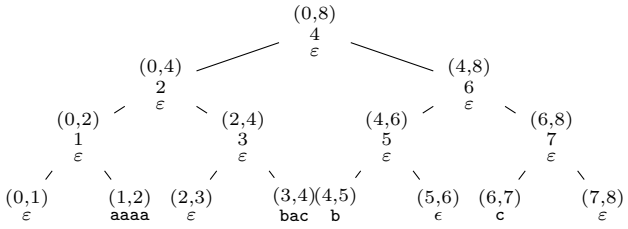
where $B(L, x)$ is the branch that leads to x in the trie $T(L \cup \{x\})$, and $F(L, x)$ is the subset of forward nodes in the internal execution path $\underline{\Pi}(L, x)$ of the dichotomic tree $D(L)$.

3.5 Precomputing the LCP–table

In Section 3.2, the $\Gamma(L)$ tree was defined as a decoration of the tree $D(L)$. Here, we only need the knowledge of on array of length $2n+1$ (Fig. 5, bottom right). This array (called the LCP–table in [7]) contains the value $\gamma[b, e]$ at index $i(b, e) \in [0\,..\,2n]$ defined as: $i(b, e) = b$, if (b, e) is external, and $i(b, e) = n + \text{mid}(b, e)$, if (b, e) is internal.

The computation of the LCP–table proposed in the book [7] is based on a bottom-up strategy, and has a complexity closely related to the path length of the trie $T(L)$, of order $\Theta(n \log n)$ (on average). We present here a new algorithm, based a top-down strategy. When called on the triple $(0, n+1, 0)$ the algorithm BUILD-LCP-TABLE, described in Fig. 5 (left), computes the LCP–table and the number $A(L)$ of symbol comparisons performed clearly involves the difference $\gamma[b, m] + \gamma[m, e] - 2\gamma[b, e]$ at each node $[b, e]$ of $D(L)$.

```
BUILD-LCP-TABLE(b, e, ℓ)
if b = 0 or e = n + 1 then
    ℓ' = 0
else
    ℓ' := ℓ
    while L_b[ℓ'] = L_e[ℓ'] do
        ℓ' := ℓ' + 1
γ[b, e] := ℓ'
if b + 1 < e then m := mid(b, e)
    BUILD-LCP-TABLE(b, m, ℓ')
    BUILD-LCP-TABLE(m, e, ℓ')
```



$$\text{LCP–table}(L) = [0, 4, 0, 3, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$$
$$\texttt{Int(a)} = \texttt{Int(aa)} = \texttt{Int(aaa)} = [1, 2],$$
$$\texttt{Int(b)} = [3, 5] = [3, 4] \cup [4, 5], \quad \texttt{Int(c)} = [6, 7]$$
$$\texttt{Dic(a)} = \texttt{Dic(aa)} = \texttt{Dic(aaa)} = \texttt{Dic(ba)} = \{[1, 2]\},$$
$$\texttt{Dic(b)} = \{[3, 4], [4, 5]\}$$

Figure 5 On the left, the algorithm for computing the LCP–table. On the right, the $\Gamma(L)$-tree and the LCP–table related to the list L of Fig. 2, together with some examples for $\texttt{Int}(w)$ and $\texttt{Dic}(w)$.

We now provide an alternative expression for the cost $A(L)$. Consider indeed an internal node of the trie $T(L)$ related to the prefix w and associate with it the largest integer interval $\texttt{Int}(w) := [a, b]$ for which $w = \Gamma[a, b]$. The cardinality of $\texttt{Int}(w)$ is the number N_w of elements of L which begin with w. The interval $\texttt{Int}(w)$ is not *dichotomic*[6] (in general). We

[6] A pair (a, b) is dichotomic if it labels a node in the dichotomic tree $D(L)$.

denote by $\texttt{Dic}(w)$ its decomposition into largest possible dichotomic intervals (see Fig. 5 for some examples), and by $\texttt{dic}(w)$ the cardinality of the decomposition $\texttt{Dic}(w)$. There are now two results: – first, the parameter $\texttt{dic}$ arises as a basic cost for $A(L)$ – second, there is a relation between the two parameters $\texttt{dic}(w)$ and the cardinality N_w .

▶ **Proposition 8.** *The number $A(L)$ of symbol comparisons used by Algorithm* Build-lcp-table *for computing the* lcp*–table for list L is expressed as*

$$A(L) = (2n + 1 - E_n) + \sum_{\substack{w \in T(L) \\ w \neq \varepsilon}} \texttt{dic}(w), \tag{1}$$

and involves the number E_n of nodes on the extreme (left and right) branches of $D(L)$ together with the cardinality $\texttt{dic}(w)$ of the dichotomic decomposition for every internal node w of the trie $T(L)$. One has $E_n = \Theta(\log n)$ and the following estimate holds,

$$\texttt{dic}(w) \leq 2 + 2\log_2 N_w \,. \tag{2}$$

4 Probabilistic analyses of the dichotomic process

The algorithm clever-Dicho-Select algorithm involves thus three parameters:

- The length $|B(L, x)|$ of the branch which leads to x in the trie $T(L \cup \{x\})$; this is a classical parameter, and an instance of a *cost with toll* (see Eq. (5)). It only depends on probabilistic properties of the input source and does not involve the dichotomic process.
- The other two parameters are more difficult to analyze, because they involve both the trie $T(L)$ and the dichotomic tree $D(L)$, and thus two sources (as Section 4.3 will explain).
  - We only propose trivial bounds for the number $|F(L, x)|$ of forward nodes in the dichotomic path $\underline{\Pi}(L, x)$, namely $1 \leq |F(L, x)| \leq |\underline{\Pi}(L, x)|$
  - As Proposition 8 shows, the cost $A(L)$ for computing the lcp–table for L is defined from the cost $\texttt{dic}$. This is *not* a *cost with toll*, but it can bounded from above and below with two costs with toll.

Moreover, the basic structure that underlies the whole dichotomic strategy on the list L is the $\Gamma(L)$ tree, that contains at each node (b, e) the prefix $\Gamma[b, e]$ of length $\gamma[b, e]$. The mean value of the coincidence $\gamma[b, e]$ at a random node (b, e) of depth ℓ is also a central parameter, and we give some hints in Section 4.3 for a possible further study.

4.1 The input source and its parameterization

We consider a source $\mathcal{M}$ on an ordered finite alphabet Σ which will both produce the elements of the list L and the searched word x (called the input source). A source is a (probabilistic) mechanism which emits a symbol from the alphabet Σ, at each discrete time $t = 0, 1, \ldots$. It produces infinite words in $\Sigma^{\mathbb{N}}$ and is defined by the set of probabilities

$$p_w := \Pr[\text{a word begins with the prefix } w], \quad (w \in \Sigma^\star) \,.$$

Figure 6 (top) recalls the parameterization of a source (see also [6]): using, for each k, the equality $\sum_{w \in \Sigma^k} p_w = 1$ and the lexicographic order on Σ^k, builds a sequence of quasi-disjoint intervals $\mathcal{I}_w$, with $|\mathcal{I}_w| = p_w$. When k tends to ∞, each word y is written (almost everywhere) as $y = M(t)$, with $t \in [0, 1]$. The increasing mapping $M : [0, 1] \to \Sigma^{\mathbb{N}}$ maps reals to infinite words. Using an analogy with numeration, we say that the infinite word $M(t)$ is the expansion of the real t in "base" $\mathcal{M}$, and (conversely) the real t is the *parameter* of the word y.

Classical sources are obtained in this general framework: all the memoryless sources – the *regular* ones, for which all the probabilities p_w of the prefixes $w \in \Sigma^k$ are equal, as well as the non regular ones– , but also Markov chains, and finally a class of more correlated sources, as described in [25]. Figure 6 (bottom) describes the parameterization of two regular sources: the binary one (in blue), and the ternary one (in green).

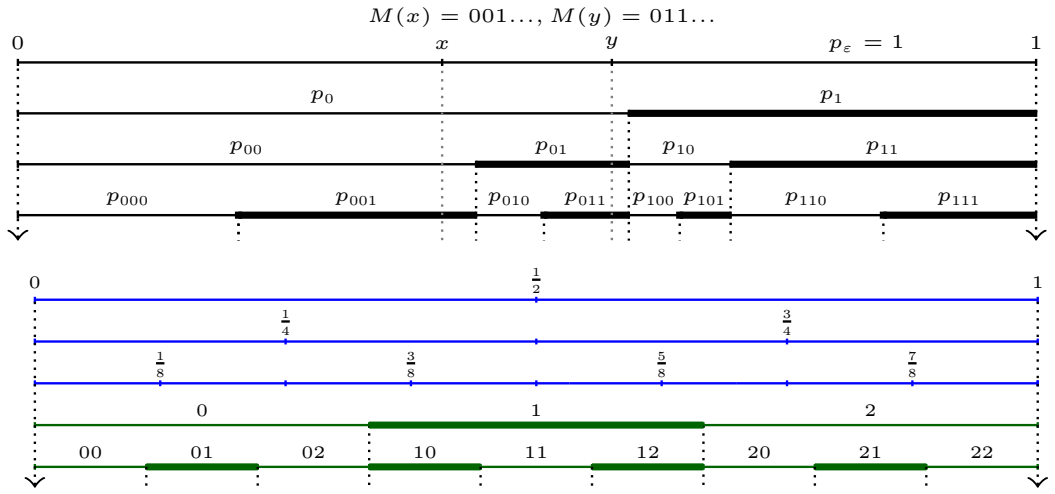


Figure 6 Parameterization of a source and fundamental intervals (top). Two instances of parameterization for regular sources (bottom).

In this context, the Dirichlet generating series of the sources, defined as

$$\Lambda(s) := \sum_{w \in \Sigma^\star} p_w^s, \qquad \Lambda_k(s) := \sum_{w \in \Sigma^k} p_w^s\,, \tag{3}$$

play a prominent role, as highlighted in [25]. The source is said to be *tame* if these functions considered as functions of a complex variable s have good properties near the vertical line $\Re s = 1$ (see [6] and details in the Appendix); it notably possesses an entropy,

$$h(\mathcal{M}) := \lim_{k\to\infty} \frac{1}{k} \sum_{w \in \Sigma^k} p_w \log p_w = \lim_{k\to\infty} \frac{1}{k} \frac{d}{ds} \Lambda_k(s)\Big|_{s=1}\,. \tag{4}$$

The source is said to be *periodic* if the series $\Lambda(s)$ is periodic.
The Dirichlet series of a regular source associated with an alphabet of cardinality b admit nice expressions,

$$\Lambda_k(s) = b^k \cdot b^{-ks} = b^{k(1-s)}, \quad \Lambda(s) = (1 - b^{1-s})^{-1}\,.$$

This shows that a b-regular source is tame and periodic of period $2\pi i/(\log b)$, with an entropy equal to $\log b$.

The intervals that appear at depth k in Figure 6 (top) define the parameterization M. They are called the fundamental intervals (of depth k). They satisfy $\mathcal{I}_w := \{u \mid M(u) \text{ begins with } w\}$. They may be defined via the coincidence γ, as $I_k(t) = \{u \in [0,1] \mid \gamma(M(t), M(u)) \geq k\}$.

Both the elements of the list L and the searched word x will be produced by the source $\mathcal{M}$. We first draw $(n+1)$ real numbers $(t, u_1, \ldots, u_n)$ in the $[0,1]$ interval, in a uniform and independent way; then, we sort $(u_1, u_2, \ldots, u_n)$, and rewrite this n-uple as an ordered n-uple

$(t_1 < t_2 < \cdots < t_n)$; we let $x = M(t)$, and $L_i := M(t_i)$ for $i \in [1 \dots n]$. As the mapping M is increasing, the list L is sorted in the increasing order. The parameter t_i of the word L_i is the i-th smallest element (also called the i-th statistics) of the random n-uple $(u_1, u_2, \dots, u_n)$.

4.2 Probabilistic analysis of the Clever-Dicho-Select algorithm

With a list L and a finite prefix w, we have associated in Section 3.5 the parameter $N_w := N_w(L)$ that is the number of elements of L which begin with w. The internal nodes of the trie $T(L)$ are labelled with prefixes w for which $N_w(L) \geq 2$. The variable N_w plays a central role in the analysis of the trie parameters, notably those that are associated with a *toll* function $x : \mathbb{N} \to \mathbb{N}$ and obtained as a sum taken over the internal nodes:

$$X(L) := \sum_{w | N_w \geq 2} x(N_w)\,. \tag{5}$$

This class of parameters X (called *costs with toll*) contains for instance the path length (for $x(k) = k$) and the number of internal nodes (for $x(k) = 1$.) The asymptotic mean value of such parameters has been deeply studied, notably in [5], and it depends both on the toll and the characteristics of the source, in particular its entropy. Even though the cost $\texttt{dic}(w)$ is *not* a cost with toll, the inequality $1 \leq \texttt{dic}(w) \leq 2 \log_2 N_w + 2$ stated in Proposition8 bounds it with two costs with toll. We then obtain the main result of the paper, for which a sketch of proof is given in the Appendix.

▶ **Theorem 9.** *Consider a* good *input source $\mathcal{M}$ and a random input (L, x) with $|L| = n$ as described in Section 4.1. Then asymptotically:*

- *the mean value B_n of the length $|B(L, x)|$ satisfies $B_n \sim (1/h(\mathcal{M})) \log n$;*
- *the mean value F_n of the cardinality of the forward set satisfies $1 \leq F_n \leq \log_2(n+1)$;*
- *the mean value A_n of the cost $A(L)$ for building the $\gamma(L)$ array satisfies*

$$\frac{n}{h(\mathcal{M})}[1 + P(\log n)] \leq A_n \leq \frac{n}{h(\mathcal{M})}[1 + Q(\log n)] \left[\frac{2}{\log 2} \sum_{k \geq 2} \frac{\log k}{k(k-1)} \right],$$

where P and Q are two periodic functions of very small amplitude which only appear when the source is periodic.

4.3 Probabilistic analysis of the $\Gamma(L)$ tree: the second source

We now consider the probabilistic behaviour of the main structure that underlies the algorithm, namely the $\Gamma(L)$ tree. We focus here on the case of *pure dichotomy* where the cardinality n of the list satisfies $n + 1 = 2^p$ for some integer $p > 0$, and, thus, the dichotomic tree D_n (denoted by $\widehat{D}_p$) is *complete*.

A node (b, e) at depth ℓ in $\widehat{D}_p$ satisfies $b = a\, 2^{p-\ell}$, $e = (a+1)\, 2^{p-\ell}$ for some integer $a \in [0 \,..\, 2^\ell - 1]$. This node is associated in the $\Gamma(L)$ tree with the pair (L_b, L_e), whose parameters t_b and t_e are resp. the b-th and the e-th statistics of the n-uple $(u_1, u_2, \dots, u_n)$, i.e., the elements of respective ranks b and e in the sorted n-uple $(t_1, t_2, \dots, t_n)$. They thus satisfy, with classical results on i-th statistics recalled in the Appendix,

$$[t_b, t_e] \approx [v_b, v_e]\,, \quad \text{with} \quad v_b = b\, 2^{-p} = a\, 2^{-\ell}, \quad v_e := e\, 2^{-p} = (a+1)\, 2^{-\ell}\,. \tag{6}$$

This estimate holds for $p \to \infty$ and any $\ell \leq p$. Then, any node (b, e) of depth ℓ in $\widehat{D}_p$ is (asymptotically for $p \to \infty$) associated with a fundamental interval $[v_b, v_e]$ of depth ℓ of the regular binary source $\mathcal{B}$.

This provides an (asymptotic) geometric interpretation of the coincidence $\gamma[b, e]$ (when $p \to \infty$): this is the largest integer k for which there exists a fundamental interval $\mathcal{I}_w$ of depth k (of the source $\mathcal{M}$) that contains the given fundamental interval $[v_b, v_e]$ of the source $\mathcal{B}$. Such an interplay between two sources –here the sources $\mathcal{M}$ and $\mathcal{B}$– is deeply studied in the context of dynamical systems (see for instance the results of [3, 8]), at least when the depth ℓ tends to ∞. A more precise view of this (asymptotic) interplay between these two sources will be central in a further analysis of the $\Gamma(L)$ tree.

The parameter `Dic` may be also described thanks to the interplay of the two sources: it is indeed related to the decomposition of a fundamental interval of the source $\mathcal{M}$ in terms of fundamental intervals of the source $\mathcal{B}$.

Conclusion. The present study has introduced the $\Gamma(L)$ tree and the parameter `Dic`. The $\Gamma(L)$ tree underlies any algorithm based on dichotomic process, and we feel that the parameter `Dic` plays an important role in this process and should be further studied for itself.

References

1 Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. Tight Bounds for Searching a Sorted Array of Strings. *SIAM J. Comput.*, 30(5):1552–1578, 2000. doi:10.1137/S0097539797329889.

2 Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. 40 Years of Suffix Trees. *Commun. ACM*, 59(4):66–73, March 2016. doi:10.1145/2810036.

3 Wieb Bosma, Karma Dajani, and Cor Kraaikamp. Entropy quotients and correct digits in number-theoretic expansions. In *Dynamics & stochastics*, volume 48 of *IMS Lecture Notes Monogr. Ser.*, pages 176–188. Inst. Math. Statist., Beachwood, OH, 2006. doi:10.1214/074921706000000202.

4 Julien Clément, James Allen Fill, Thu Hien Nguyen Thi, and Brigitte Vallée. Towards a Realistic Analysis of the QuickSelect Algorithm. *Theory of Computing Systems*, 58(4):528–578, May 2016. doi:10.1007/s00224-015-9633-5.

5 Julien Clément, Philippe Flajolet, and Brigitte Vallée. Dynamical sources in information theory: A general analysis of trie structures. *Algorithmica*, 29(1):307–369, February 2001. doi:10.1007/BF02679623.

6 Julien Clément, Thu-Hien Nguyen-Thi, and Brigitte Vallée. Towards a Realistic Analysis of Some Popular Sorting Algorithms. *Combinatorics, Probability and Computing*, 24(1):104–144, 2015. doi:10.1017/S0963548314000649.

7 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

8 Karma Dajani and Adam Fieldsteel. Equipartition of interval partitions and an application to number theory. *Proc. Amer. Math. Soc.*, 129(12):3453–3460, 2001. doi:10.1090/S0002-9939-01-06299-2.

9 Luc Devroye. A Probabilistic Analysis of the Height of Tries and of the Complexity of Triesort. *Acta Inf.*, 21(3):229–237, October 1984. doi:10.1007/BF00264248.

10 Johannes Fischer. Inducing the LCP-Array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 374–385, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

11 Gianni Franceschini and Roberto Grossi. No Sorting? Better Searching! In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 491–498, 2004. doi:10.1109/FOCS.2004.43.

12 Simon Gog and Enno Ohlebusch. Fast and Lightweight LCP-array Construction Algorithms. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX '11, pages 25–34, Philadelphia, PA, USA, 2011. Society for Industrial and Applied Mathematics.

13 G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: In Pascal and C (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

14 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

15 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted Longest-Common-Prefix Array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

16 Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

17 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

18 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

19 U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

20 Giovanni Manzini. Two Space Saving Tricks for Linear Time LCP Array Computation. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004*, pages 372–383, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

21 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In James A. Storer and Michael W. Marcellin, editors, *DCC*, pages 193–202. IEEE Computer Society, 2009.

22 Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Comput. Surv.*, 39(2), July 2007. doi:10.1145/1242471.1242472.

23 Robert Sedgewick. *Algorithms in C*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

24 Wojciech Szpankowski. *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

25 Brigitte Vallée. Dynamical sources in information theory: Fundamental intervals and word prefixes. *Algorithmica*, 29(1):262–306, February 2001. doi:10.1007/BF02679622.

26 Brigitte Vallée. The Depoissonisation Quintet: Rice-Poisson-Mellin-Newton-Laplace. In *Proceedings of the AofA 2018 Conference*, volume LIPICS Dagstuhl 110, pages 35:1–35:20, June 2018. (long version: https://arxiv.org/pdf/1802.04988). doi:10.4230/LIPIcs.AofA.2018.35.

27 Brigitte Vallée, Julien Clément, James Allen Fill, and Philippe Flajolet. The Number of Symbol Comparisons in QuickSort and QuickSelect. In *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 750–763, 2009. doi:10.1007/978-3-642-02927-1_62.

A Appendix

A.1 Proof of Proposition 8. Relation (1)

Proof. Consider a node with prefix w in the trie $T(L)$. We denote by $[A_w, B_w]$ the maximal interval $[A, B]$ for which $w = \Gamma[A, B]$, and the decomposition of $[A_w, B_w]$ into largest dichotomic intervals, of the form

$$\texttt{Dic}(w) := \{[b_1, e_1], [b_2, e_2], \dots [b_k, e_k]\}, \quad \text{with } b_1 = A_w,\ e_{j-1} = b_j \text{ for } j \in [2\,..\,k], e_k = B_w\,.$$

We denote by $\texttt{dic}(w)$ the cardinality of $\texttt{Dic}(w)$, namely the number of dichotomic intervals it contains (with the previous notations, one has $\texttt{Dic}(w) = k$) .
This decomposition is easily built in a bottom-up strategy. We begin with the decomposition of $[A_w, B_w]$ into dichotomic intervals of length one in $D(L)$. As soon as there exists in the decomposition two adjacent nodes in $D(L)$ with the same predecessor, we replace them by their common predecessor. The recursive procedure halts at depth ℓ in $D(L)$ with two possible cases: one node of depth ℓ, or two adjacent nodes of depth ℓ with a different predecessor. For each dichotomic interval $[b, e]$ in $D(L)$ (distinct of the root), denote the predecessor of $[b, e]$ by $\texttt{pred}(b, e)$. We now prove that the two conditions are equivalent

- the prefix w satisfies the inequality $\Gamma[\texttt{pred}(b, e)] \prec w \preceq \Gamma[b, e]$;
- the dichotomic interval $[b, e]$ belongs to the set $\texttt{Dic}(w)$;

Indeed, when $[b, e] \in \texttt{Dic}(w)$, then first $\Gamma[b, e] \succeq w$, and second the interval $[b, e]$ is a largest dichotomic interval, then its predecessor $\Gamma[\texttt{pred}(b, e)]$ satisfies the strict inequality $\Gamma[\texttt{pred}[b, e]] \prec w$.
Conversely, when the prefix w satisfies the inequality $\Gamma[\texttt{pred}(b, e)] \prec w \preceq \Gamma[b, e]$, the interval $[b, e]$ is a largest dichotomic interval for which $\Gamma[b, e] \succeq w$, and thus the inequality $\Gamma[\texttt{pred}(b, e)] \prec w \preceq \Gamma[b, e]$ holds.

There is thus a bijection between the two multi-sets

$$\{\texttt{Dic}(w) \mid w \text{ internal node of } T(L)\}, \quad \{w \mid \exists (b, e) \in D(L), \ \ \Gamma[\texttt{pred}(b, e)] \prec w \preceq \Gamma[b, e]\}.$$

As the cardinality of the last multi-set coincides with the number of (positive) comparisons that are needed to compute the $\gamma(L)$ array, this ends the proof. ◀

A.2 Proof of Proposition 8. Bound for $\texttt{dic}(w)$ given in Eq. (2)

This bound will be established thanks to the next lemmas 10, 12 and 13.

▶ **Lemma 10.** *Let $n > 0$ and D_n be a dichotomic tree of height $h = \lceil \log_2 n \rceil$. Each node $(b, e) \in D_n$ corresponds to a dichotomic interval $[b, e]$ of length $e - b$. At depth $0 \leq \ell \leq h$, all nodes in D_n correspond with intervals of length either $\lfloor \frac{n}{2^\ell} \rfloor$ or $\lfloor \frac{n}{2^\ell} \rfloor + 1$.*

Proof. The root interval $[0, n]$ has length n. If n is even, left and right intervals children have both length $n/2$ and, if n is odd, left and right intervals children have respectively length $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$. Hence the property of the lemma holds at depth $\ell = 1$. By recursion, suppose now that at a given depth ℓ and posing $N = \lfloor \frac{n}{2^\ell} \rfloor$, all intervals have length either N or $N+1$. We can check easily (for instance using binary expansion of n) that $\lfloor N/2 \rfloor = \lfloor n/2^{\ell+1} \rfloor$, and also $\lfloor (N+1)/2 \rfloor = \lfloor n/2^{\ell+1} \rfloor$ if N is even and $\lfloor (N+1)/2 \rfloor = \lfloor n/2^{\ell+1} \rfloor + 1$ if N is odd. Hence the property holds at depth $\ell + 1$. ◀

So at a given depth, dichotomic intervals can only take two lengths. Abusing notation, for a fixed dichotomic tree D_n, we note $\texttt{Dic}(i, f)$ the decomposition of the interval $[i, f]$ as a union of minimal cardinality of disjoint (apart from their extremities) dichotomic intervals. We also define $\texttt{dic}(i, f)$ as this minimal cardinality. We now want to upper bound the parameter $\texttt{dic}(i, f)$. We will use the following definition:

▶ **Definition 11.** *For a dichotomic tree D_n, an interval (i, f) is said to be left (resp. right) aligned if there exists a dichotomic interval (b, e) such that $i = b$ and $e \geq f$ (resp. $f = e$ and $b \leq i$).*

First we examine the case of an interval (i, f) which is aligned on the left.

▶ **Lemma 12.** *Let $n > 0$ and a dichotomic tree D_n. Let consider a pair (b, f) with $0 \leq b < f \leq n$ such that $[b, f]$ is left aligned on a dichotomic interval $[b, e]$. We have*

$$\texttt{dic}(b, f) \leq 1 + \log_2(f - b).$$

Proof. If $[b, f]$ is dichotomic, $\texttt{dic}(b, f) = 1$ and the lemma holds. We assume now that $[b, f]$ is not dichotomic. There exists $e < e'$ such that

$$[b, e] \subsetneqq [b, f] \subsetneqq [b, e'],$$

and $[b, e], [b, e']$ are dichotomic intervals with $(b, e') = \texttt{pred}(b, e)$ (meaning $\lfloor (b + e')/2 \rfloor = e$). Consequently $[e, e']$ is also a dichotomic interval of D_n of length $b - e + \epsilon$ (for some $\epsilon \in \{0, 1\}$, see Lemma 10). The first and largest interval of $\texttt{Dic}(b, f)$ is the dichotomic interval $[b, e]$. We can thus write $\texttt{dic}(b, f) = 1 + \texttt{dic}(e, f)$. Since $[e, f]$ is left aligned on the dichotomic interval $[e, e']$ we can iterate. Using the fact that in the next iteration we decompose an interval $[e, f]$ of length $f - e \leq f/2$, we get the bound $\texttt{dic}(b, f) \leq 1 + 2\log_2(f - b)$. ◀

The lemma can be adapted when the interval is right aligned. We can now examine the general case (i, f) with $0 \leq i < f \leq n$ as illustrated in Fig. 7.

▶ **Lemma 13.** *Let $n > 0$ and a dichotomic tree D_n. Let consider a pair (i, f) with $0 \leq i < f \leq n$. We have*

$$\texttt{dic}(i, f) \leq 2 + 2\log_2(f - i).$$

Proof. Let $[b, e]$ the dichotomic interval of maximal length such that $[i, f] \subset [b, e]$. Then posing $m = \lfloor (b + e)/2 \rfloor$, we decompose the interval $[i, f]$ according to this splitting point m into two intervals $[i, m] \cup [m, f]$. Those two intervals are respectively aligned on the right and the left of two dichotomic consecutive intervals $[b, m]$ and $[m, e]$ of length less than $f - i$. We can also check that $\texttt{Dic}(i, f) = \texttt{Dic}(i, e) + \texttt{Dic}(e, f)$. Finally we apply the previous lemma to prove the result. ◀

Now going back to usual notation of the paper, this yields the estimate of Eq. (2) in Proposition 8.

A.3 Exact expression for $\texttt{dic}(w)$ in the pure dichotomic case

In the case of a pure dichotomy, there is an exact expression for $\texttt{dic}(w)$, in terms of the parameter $|x|_1$ that denotes the number of '1' in the binary writing of the integer x. This makes more precise the estimate given in Lemma 12.

▶ **Lemma 14.** *In the pure dichotomic case, the following holds:*

- *Consider a pair (b, f) such that the interval $[b, f]$ is left aligned. Then $\texttt{dic}(b, f) = |f - b|_1$.*
- *Consider a pair (i, e) such that the interval $[i, e]$ is right aligned on a dichotomic interval $[b, e]$. Then $\texttt{dic}(i, e) = |i - b|_1$*

Proof. The proof is omitted. ◀

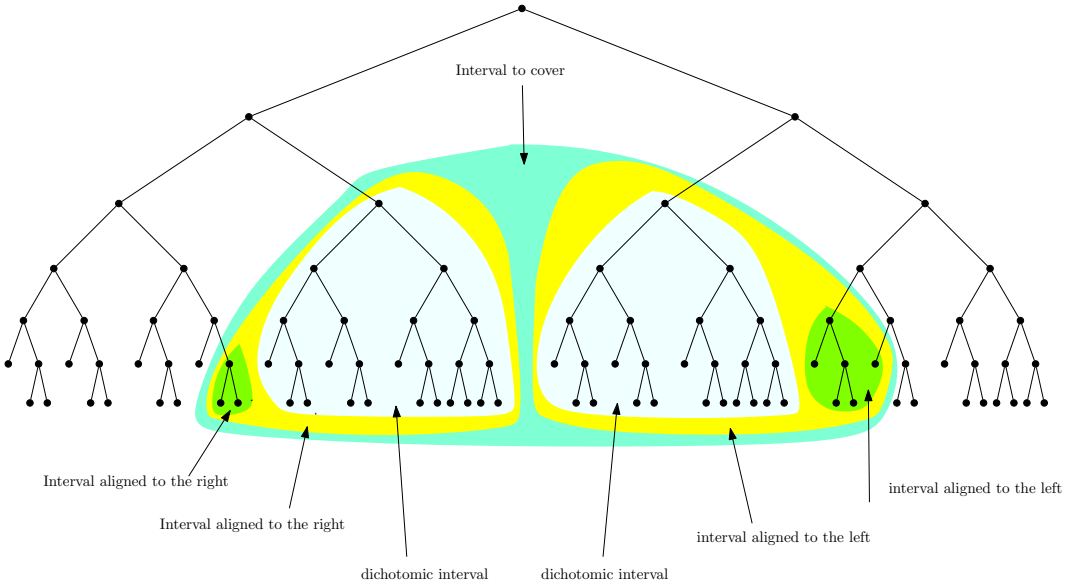


Figure 7 A dichotomic tree with an interval $\mathcal{I}$ (in light green). It decomposes in two intervals (yellow) which are respectively aligned on the left and on the right.

A.4 Proof of Theorem 9: Average-case analysis of a cost with toll in a trie built on a good general source

In this section, $x : \mathbb{N} \to \mathbb{R}$ denotes a toll function, and, for a trie built on a list of words L, X is the total cost defined in (5)

$$X(L) := \sum_{w|N_w \geq 2} x(N_w)\,,$$

where N_w is the number of elements of L which begin with w.

There are three main steps in the analysis of the expectation of parameter X.

First step. We begin to deal with the Poisson model $\mathcal{P}_z$, where the cardinality of the list N is a random variable which follows the law

$$\Pr[N = n] = e^{-z}\frac{z^n}{n!}\,.$$

With a toll x, we associate its Poisson transform

$$P_x(z) := e^{-z}\sum_{n\geq 0} x(n)\frac{z^n}{n!}\,,$$

which coincides with the expectation $\mathbb{E}_z[x]$ of the cost x in the Poisson model. In the model $\mathcal{P}_z$, the cardinality N_w follows a Poisson law of rate $z\,p_w$ that involves the fundamental probability p_w of the source. This is the main advantage of the Poisson model.

We then deal with the Poisson transforms $P_X(z)$ and $P_x(z)$ that resp. coincide with the expectations of X and x in the Poisson model, and averaging Relation (5) in the model $\mathcal{P}_z$ entails a relation between the two Poisson transforms

$$P_X(z) := \mathbb{E}_z[X] = \sum_{w\in\Sigma^\star} \mathbb{E}_z[x(N_w)] = \sum_{w\in\Sigma^\star} P_x(z\,p_w). \tag{7}$$

Then, the function $P_X(z)$ writes as a harmonic sum built on the function P_x. The Mellin transform is a good tool for dealing with harmonic sums and the Mellin transform $P_X^*(s)$ factorizes and involves the Λ generating function of the source (defined in (3)),

$$P_X^*(s) = \Lambda(-s) \cdot P_x^*(s)\,.$$

On the other side, the Mellin transform of P_x satisfies,

$$P_x^*(s) = \sum_{k\geq 2} \frac{x(k)}{k!} \int_0^\infty e^{-z} z^k z^{s-1} dz = \sum_{k\geq 2} \frac{x(k)}{k!} \Gamma(k+s) = \sum_{k\geq 2} \frac{x(k)}{k} \frac{\Gamma(k+s)}{\Gamma(k)}. \tag{8}$$

Second step. We now wish to return first to $P_X(z)$ (the expectation of X in the Poisson model), then extract the coefficients of order n in P_X to obtain the expectation $E_{[n]}[X]$ of X in the usual model where N is fixed and equal to n (this is called the Bernoulli model). This step is called the Depoissonisation step and may be performed with Depoissonisation techniques or via the Rice Formula, as it is explained in [26]. In [26], the author shows that the Rice method may be applied as soon as the function

$$\psi(s) := \frac{P_X^*(-s)}{\Gamma(-s)} = \frac{P_x^*(-s)}{\Gamma(-s)} \cdot \Lambda(-s)$$

is *tame* on a domain $\mathcal{R}$ (meaning "meromorphic on $\mathcal{R}$ and of polynomial growth when $|\Im s| \to \infty$ inside $\mathcal{R}$).

For the two costs that bound the function `Dic`, namely, the function $x(k) = 1$ (associated with the size of the trie) but also the function $x(k) = \log k$, the ratio $P_x^*(s)/\Gamma(s)$ is tame on a domain $\mathcal{R} := \{s \mid \Re s > 1 - a\}$ for some $a > 0$ and even analytic (without poles). In our context, the Λ function of the source is also tame on such a domain[7] $\mathcal{R}$.
The Rice method deals with the product of the two functions

$$\psi(s) := \frac{P_X^*(-s)}{\Gamma(-s)} = \frac{P_x^*(-s)}{\Gamma(-s)} \cdot \Lambda(-s), \qquad L_n(s) := \frac{(-1)^{n+1}\, n!}{s(s-1)(s-2)\dots(s-n)}$$

and provides the estimate for the expectation $E_{[n]}[X]$, as

$$E_{[n]}[X] = -\left[\sum_{k \mid s_k \in \mathcal{R}} \operatorname{Res}\left[L_n(s) \cdot \psi(s); s = s_k\right] + \frac{1}{2i\pi} \int_{\mathcal{C}} L_n(s) \cdot \psi(s)\, ds \right],$$

where the sum is taken over the poles s_k of ψ inside a region $\mathcal{R}$ of tameness. We have now to choose a region $\mathcal{R}$ for tameness, exhibit the poles that appear in this region and compute the associated residues.

Periodicity of the source. In this general context of good sources, the series $\Lambda(s)$ has always a simple pole at $s = 1$ inside the domain $\mathcal{R}$. And there are two possibilities

(*i*) The pole $s = 1$ is the unique pole on the vertical line $\Re s = 1$, with a residue equal to the inverse of the entropy, $1/h(\mathcal{M})$;

(*ii*) There exists another pole $s \neq 1$ on the line $\Re s = 1$, and then there is an infinite family of poles regularly spaced on this line, of the form $s_k = 1 + 2ik\rho\pi$ for some ρ and any $k \in \mathbb{Z}$.

[7] We say here that the source is good.

In the first case, the source is aperiodic; in the second case, the function $s \mapsto \Lambda(s)$ is periodic, and the source itself is said to be periodic. At a first glance, the case (ii) seems to be quite particular. However, as we explain in Section 4.1, this cases arises as soon as the source is *regular*, i.e. associated with an alphabet of size b, and probabilities $p_i = 1/b$. In this case, one has $\Lambda(s) = (1 - b^{1-s})^{-1}$ and the parameter ρ equals $1/(\log b)$.

The residue sum involves the function $P_x^\star$ defined in (8), and there are two possibilities for the residue sum:

$$\frac{n}{h(\mathcal{M})} \cdot P_x^*(1) \quad (\text{case } (i)), \qquad \frac{n}{h(\mathcal{M})} \left[\sum_{k \in \mathbb{Z}} n^{s_k - 1} \cdot P_x^*(s_k) \right] \quad (\text{case } (ii)).$$

Remark that the factor of the second sum provides a periodic function of the variable $\log_\rho n$. This ends the proof.

A.5 Towards the analysis of coincidence $\gamma(b, e)$

Consider a node (b, e) of depth ℓ in $D(L)$, the longest common prefix $\Gamma[b, e] := \Gamma[L_b, L_e]$, its length denoted as $\gamma(b, e)$, and the two parameters t_b, t_e of L_b, L_e. With any integer k, we associate the set $\mathcal{V}_k$ that gathers all the ends of fundamental intervals of the source $\mathcal{M}$ of depth k. There are three cases for the cardinality $J_k(b, e)$ of the intersection $\mathcal{V}_k \cap [t_b, t_e]$, i.e., $J_k(b, e) = 0, 1$ or $J_k(b, e) \geq 2$.

We first consider the toy case where : (a) the intervals $[t_b, t_e]$ exactly coincide with the binary intervals $[v_b, v_e]$ defined in Eq. (6) – (b) the source $\mathcal{M}$ is b-regular. Then, for each possible cardinality $J_k(b, e)$, we can relate first $\gamma(b, e)$ and k, then k and ℓ, and we obtain the estimate

$$\theta\ell - 1 \leq \gamma(b, e) \leq \theta(\ell + 2) \qquad \text{with} \quad \theta := \log 2/(\log b).$$

Of course, the toy case described in items (a) and (b) is not the actual situation; but the actual situation – *at least on average* and for $\ell \to \infty$– appears to be close to this "simplification", due to the two following facts:

(a') The interval $[t_b, t_e]$ is close to the interval $[v_b, v_e]$. For an interval (b, e) at depth ℓ, the pair (t_b, t_e) indeed admits the joint distribution $f_{b,e,n}$ on the triangle $\mathcal{T} := \{(x, y) \mid 0 \leq x \leq y \leq 1\}$ defined as

$$f_{b,e,n}(x, y) = \frac{n!}{(b-1)!(e-b-1)!(n-e)!} x^{b-1}(y-x)^{e-b-1}(1-y)^{n-e} \qquad (n = 2^p)\,.$$

It is then easy to exactly compute the expectation and the variance of the main parameters $t_b, t_e, |t_b - t_e|, (1/2)(t_b + t_e)$ which allows to compare the two intervals $[t_b, t_e]$ and $[v_b, v_e]$. These variances are always negligible with respect to their expectations (for $n = 2^p \to \infty$), and there is a *concentration* phenomenon for these distributions.

(b') For a good source, there is an asymptotic Gaussian law on the logarithms $\ell_k(\cdot)$ of the lengths of the fundamental intervals $I_k(\cdot)$ (for $k \to \infty$) (see [25]).

Finding a Small Number of Colourful Components

Laurent Bulteau
Université Paris-Est, LIGM (UMR 8049), CNRS, ENPC, UPEM, ESIEE Paris, France
laurent.bulteau@u-pem.fr

Konrad K. Dabrowski
Department of Computer Science, Durham University, Durham, UK
konrad.dabrowski@durham.ac.uk

Guillaume Fertin
Université de Nantes, LS2N (UMR 6004), CNRS, Nantes, France
guillaume.fertin@univ-nantes.fr

Matthew Johnson
Department of Computer Science, Durham University, Durham, UK
matthew.johnson2@durham.ac.uk

Daniël Paulusma
Department of Computer Science, Durham University, Durham, UK
daniel.paulusma@durham.ac.uk

Stéphane Vialette
Université Paris-Est, LIGM (UMR 8049), CNRS, ENPC, UPEM, ESIEE Paris, France
stephane.vialette@u-pem.fr

Abstract

A partition $(V_1, \ldots, V_k)$ of the vertex set of a graph G with a (not necessarily proper) colouring c is colourful if no two vertices in any V_i have the same colour and every set V_i induces a connected graph. The Colourful Partition problem, introduced by Adamaszek and Popa, is to decide whether a coloured graph (G, c) has a colourful partition of size at most k. This problem is related to the Colourful Components problem, introduced by He, Liu and Zhao, which is to decide whether a graph can be modified into a graph whose connected components form a colourful partition by deleting at most p edges.

Despite the similarities in their definitions, we show that Colourful Partition and Colourful Components may have different complexities for restricted instances. We tighten known NP-hardness results for both problems by closing a number of complexity gaps. In addition, we prove new hardness and tractability results for Colourful Partition. In particular, we prove that deciding whether a coloured graph (G, c) has a colourful partition of size 2 is NP-complete for coloured planar bipartite graphs of maximum degree 3 and path-width 3, but polynomial-time solvable for coloured graphs of treewidth 2.

Rather than performing an ad hoc study, we use our classical complexity results to guide us in undertaking a thorough parameterized study of Colourful Partition. We show that this leads to suitable parameters for obtaining FPT results and moreover prove that Colourful Components and Colourful Partition may have different parameterized complexities, depending on the chosen parameter.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Colourful component, colourful partition, tree, treewidth, vertex cover

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.20

Related Version A full version of this paper is available from `https://arxiv.org/abs/1808.03561` [3].

Funding Supported by EPSRC (EP/K025090/1) and the Leverhulme Trust (RPG-2016-258).

1 Introduction

Research in comparative genomics, which studies the structure and evolution of genomes from different species, has motivated a number of interesting graph colouring problems. In this paper we focus on the multiple genome alignment problem, where one takes a set of sequenced genomes, lets the genes be the vertex set of a graph G, and joins by an edge any pair of genes whose *similarity* (determined by their nucleotide sequences) exceeds a given threshold. The vertices are also coloured to indicate the species to which each gene belongs. This leads to a *coloured graph* (G, c), where $c : V(G) \to \{1, 2, \ldots\}$ denotes the colouring of G. We emphasize that c is not necessarily proper (that is, adjacent vertices may have the same colour).

One seeks to better understand the evolutionary processes affecting these genomes by attempting to partition similar genes into *orthologous* sets (that is, collections of genes that originated from the same ancestral species but diverged following a speciation event). This translates into partitioning $V(G)$ such that each part

(i) contains no more than one vertex of each colour, and

(ii) induces a connected component.

These two conditions ensure that each part contains vertices representing an orthologous set of similar genes. In addition, one seeks to find a partition that is in some sense optimal.

When Zheng et al. [14] considered this model, one approach they followed was to try to delete as few edges as possible such that the connected components of the resulting graph G' are *colourful*, that is, contain no more than one vertex of any colour; in this case the connected components give the partition of $V(G)$. This led them to the Orthogonal Partition problem, introduced in [8], also known as the Colourful Components problem [1, 2, 7]. (Note that a graph is *colourful* if each of its components is colourful.)

Colourful Components	
Instance:	A coloured graph (G, c) and an integer $p \geq 1$
Question:	Is it possible to modify G into a colourful graph G' by deleting at most p edges?

The focus of this paper is on the companion problem Colourful Partition, which was introduced by Adamaszek and Popa [1]. A partition is *colourful* if every partition class induces a connected colourful graph. The *size* of a partition is its number of partition classes.

Colourful Partition	
Instance:	A coloured graph (G, c) and an integer $k \geq 1$
Question:	Does (G, c) have a colourful partition of size at most k?

Let us immediately demonstrate with an example that Colourful Components and Colourful Partition are distinct: a colourful partition with the fewest parts might require the deletion of more edges than the minimum needed to obtain a colourful graph. This has not previously been clearly established. Later, we will find a family of instances on which the problems have different complexities (see Corollary 11).

▶ **Example 1.** Let G have vertices $u_1, \ldots, u_k$, $v_1, \ldots, v_k$, w, w' and edges ww' and u_iw, u_iw', v_iw, v_iw' for $i \in \{1, \ldots, k\}$. Let c assign colour i to each u_i and v_i, colour $k+1$ to w and colour $k+2$ to w'. Then $(\{u_1, \ldots, u_k, w\}, \{v_1, \ldots, v_k, w'\})$ is a colourful partition for (G, c) of size 2, which we obtain by deleting $2k+1$ edges. However, deleting the $2k$ edges v_iw and v_iw' for $i \in \{1, \ldots, k\}$ also yields a colourful graph (with a colourful partition of size $k+1$). See Figure 1 for an illustration.

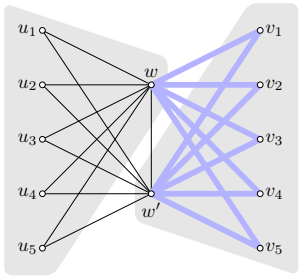


Figure 1 The graph G of Example 1 with $k = 5$.

Known Results. Adamaszek and Popa [1] proved, among other results, that Colourful Partition does not admit a polynomial-time approximation within a factor of $n^{\frac{1}{14}-\epsilon}$, for any $\epsilon > 0$ (assuming P $\neq$ NP). A coloured graph (G, c) is *ℓ-coloured* if $1 \leq c(u) \leq \ell$ for all $u \in V(G)$. Bruckner et al. [2] proved the following two results for Colourful Components. The first result follows from observing that for $\ell = 2$, the problem becomes a maximum matching problem in a bipartite graph after removing all edges between vertices coloured alike. This observation can also be used for Colourful Partition.

▶ **Theorem 2** ([2]). Colourful Partition *and* Colourful Components *are polynomial-time solvable for* 2*-coloured graphs.*

▶ **Theorem 3** ([2]). Colourful Components *is* NP*-complete for* 3*-coloured graphs of maximum degree* 6*.*

The situation for trees is different than for general graphs (see Example 1). A tree T has a colourful partition of size k if and only if it can be modified into a colourful graph by at most $k-1$ edge deletions. Hence, the problems Colourful Partition and Colourful Components are equivalent for trees. The following hardness and FPT results are due to Bruckner et al. [2] and Dondi and Sikora [7]. Note that trees of diameter at most 3 are stars and double stars (the graph obtained from two stars by adding an edge between their central vertices), for which both problems are readily seen to be polynomial-time solvable. A *subdivided star* is the graph obtained by subdividing the edges of a star.

▶ **Theorem 4** ([2]). Colourful Partition *and* Colourful Components *are polynomial-time solvable for coloured trees of diameter at most* 3*, but* NP*-complete for coloured trees of diameter* 4*.*

▶ **Theorem 5** ([7]). Colourful Partition *and* Colourful Components *are polynomial-time solvable for coloured paths (which have path-width at most* 1*), but* NP*-complete for coloured subdivided stars (which are trees of path-width at most* 2*).*

▶ **Theorem 6** ([2]). Colourful Partition *and* Colourful Components *are* FPT *for coloured trees, when parameterized by the number of colours.*

▶ **Theorem 7** ([7]). Colourful Partition *and* Colourful Components *are* FPT *for coloured trees, when parameterized by the number of colourful components (or equivalently, by the size of a colourful partition).*

In addition to Theorem 6, Bruckner et al. [2] showed that Colourful Components is FPT for general coloured graphs when parameterized by the number of colours ℓ and the number of edge deletions p. Misra [10] considered the size of a minimum vertex cover as the parameter.

▶ **Theorem 8** ([10]). Colourful Components *is* FPT *when parameterized by vertex cover number.*

Our Contribution.[1] Our main focus is the Colourful Partition problem. We consider particular graph classes and, for FPT results, parameters. Our choices have a clear motivation as, guided by past work, we seek the key to understanding the problem's (in)tractability. That is, our aims are: **(i)** to improve known NP-hardness results by obtaining tight results and to generalize tractability results for Colourful Partition (Section 3); **(ii)** to prove new results analogous to those known for Colourful Components (Section 3) and to show that the two problems are different (Sections 3 and 4); and **(iii)** to use the new results to determine suitable parameters for obtaining FPT results (Section 4). Along the way we also prove some new results for Colourful Components.

First, we show an analogue of Theorem 3 by proving that Colourful Partition is NP-complete even for 3-coloured 2-connected planar graphs of maximum degree 3. The bounds on this result are best possible, as Colourful Partition is polynomial-time solvable for 2-coloured graphs (Theorem 2) and for graphs of maximum degree 2 (trivial). We show that it also gives us a family of instances on which Colourful Components and Colourful Partition have different complexities.

Second, we focus on coloured trees. Due to Theorem 6, Colourful Partition is polynomial-time solvable for ℓ-coloured trees for every fixed ℓ. Hence the coloured trees in the hardness proofs of Theorems 4 and 5 use an arbitrarily large number of colours. They also have arbitrarily large degree. We define the *colour-multiplicity* of a coloured graph (G, c) as the maximum number of vertices in G with the same colour. We prove that Colourful Partition is NP-complete even for coloured trees of maximum degree 6 and colour-multiplicity 2. As both problems are equivalent on trees, we obtain the same result for Colourful Components (note that the graphs in the proof of Theorem 3 are not trees).

Third, we fix the number k of colourful components, which gives us the k-Colourful Partition problem. For every $k \geq 1$, this problem is polynomial-time solvable for coloured trees (due to Theorem 7) and for ℓ-coloured graphs for every fixed ℓ (such graphs have at most $k\ell$ vertices). We prove that 2-Colourful Partition is NP-complete for split graphs and for coloured planar bipartite graphs of maximum degree 3 and path-width 3, but polynomial-time solvable for coloured graphs of treewidth at most 2. The latter two results form our main results. They complement Theorem 5, which implies NP-completeness of Colourful Partition for path-width 2.

In Section 4 we show that Colourful Partition and Colourful Components are FPT when parameterized by the treewidth and the number of colours. This generalizes Theorem 6. Our choice for this combination of parameters was guided by our results from Section 3. Our results imply that for the six natural parameters: number of colourful components; maximum degree; number of colours; colour-multiplicity; path-width; treewidth; and *all* their possible combinations, we have obtained either para-NP-completeness or an FPT algorithm for Colourful Partition. This motivates the search for other parameters for this problem. As the vertex cover number of the subdivided stars in the proof of Theorem 5 can be arbitrarily large, it is natural to consider this parameter. As an analogue to Theorem 8, we prove that Colourful Partition is FPT when parameterized by vertex cover number. A vertex u of a coloured graph (G, c) is *uniquely coloured* if its colour $c(u)$ is not used

[1] Since this paper was submitted, Chlebikova and Dallard have independently extended some of our results [5].

on any other vertex of G. It is easy to show NP-hardness for Colourful Partition and Colourful Components for instances with no uniquely coloured vertices (see also Theorem 12). By uncovering a surprising connection with the Robertson-Seymour graph minors project, we also prove that, when parameterized by the number of non-uniquely coloured vertices, Colourful Components is para-NP-hard and Colourful Partition is FPT. Thus there are families of instances on which the two problems have different parameterized complexities.

2 Preliminaries

All our graphs are simple, with no loops or multiple edges. Let $G = (V, E)$ be a graph. A subset $U \subseteq V$ is *connected* if it induces a connected subgraph of G. For a vertex $u \in V$, $N(u) = \{v \mid uv \in E\}$ is the *neighbourhood* of u, and $\deg(u) = |N(u)|$ is the *degree* of u. A graph is *cubic* if every vertex has degree exactly 3. A connected graph on at least three vertices is 2-*connected* if it has no vertex whose removal disconnects the graph. A graph $G = (V, E)$ is *split* if V can be partitioned into two (possibly empty) sets K and I, where K is a clique and I is an independent set. A mapping $c : E \to \{1, 2, 3\}$ is a *proper* 3-*edge colouring* of G if $c(e) \neq c(f)$ for any two distinct edges e and f with a common end-vertex. A set $S \subseteq V$ is a *vertex cover* of G if $G - S$ is an independent set. The Vertex Cover problem asks if a graph has a vertex cover of size at most s for a given integer s. The *vertex cover number* $\mathrm{vc}(G)$ of a graph G is the minimum size of a vertex cover in G. We use the following lemma.

▶ **Lemma 9** ([4]). Vertex Cover *is* NP*-complete for* 2*-connected cubic planar graphs with a proper* 3*-edge colouring given as input.*

A *tree decomposition* of a graph G is a pair $(T, \mathcal{X})$ where T is a tree and $\mathcal{X} = \{X_i \mid i \in V(T)\}$ is a collection of subsets of $V(G)$, called *bags*, such that

1. $\bigcup_{i \in V(T)} X_i = V(G)$;
2. for every edge $xy \in E(G)$, there is an $i \in V(T)$ such that $x, y \in X_i$; and
3. for every $x \in V(G)$, the set $\{i \in V(T) \mid x \in X_i\}$ induces a connected subtree of T.

The *width* of $(T, \mathcal{X})$ is $\max\{|X_i| - 1 \mid i \in V(T)\}$, and the *treewidth* of G is the minimum width over all tree decompositions of G. If T is a path, then $(T, \mathcal{X})$ is a *path decomposition* of G. The *path-width* of G is the minimum width over all path decompositions of G.

For some of our proofs we use variants of the Satisfiability problem. Note here only that in instances of the NP-complete problem Not-All-Equal Positive 3-Satisfiability [12], each clause contains three positive literals and the problem is to find a truth assignment τ where each clause contains at least one true literal and at least one false literal. In this context we call such a τ *satisfying*.

In the remainder of this paper, proofs of results and claims marked (♠) are omitted.

3 Classical Complexity

We will prove four hardness results and one polynomial-time result on Colourful Partition. Note that Colourful Partition belongs to NP. We start by proving the following result.

▶ **Theorem 10.** Colourful Partition *is* NP*-complete for* 3*-coloured* 2*-connected planar graphs of maximum degree* 3*.*

Proof Sketch. We use a reduction from Vertex Cover. By Lemma 9 we may assume that we are given a 2-connected cubic planar graph G with a proper 3-edge colouring c. From G and c we construct a coloured graph (G', c') as follows.

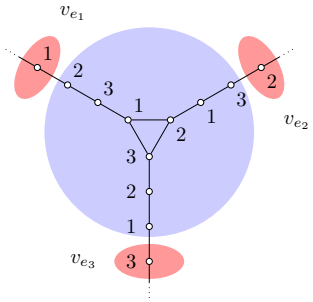


Figure 2 The blue 9-vertex gadget for a vertex v incident to edges e_1, e_2, e_3 in G, connected to the three red vertices $v_{e_1}, v_{e_2}, v_{e_3}$ in G' in the proof of Theorem 10.

- For each $e \in E(G)$ with $c(e) = i$, create a vertex v_e with colour $c'(v_e) = i$ in G' (a *red* vertex).
- For every vertex $v \in V(G)$ with incident edges e_1, e_2, e_3 (with colours 1, 2, 3, respectively), create a copy of the 3-coloured 9-vertex gadget shown in Figure 2 (a *blue* set), and connect it to the red vertices v_{e_1}, v_{e_2} and v_{e_3}, as shown in the same figure.

Note that G' is 3-coloured, 2-connected, planar and has maximum degree 3. We claim that G has a vertex cover of size at most s if and only if G' has a colourful partition of size at most $3n + s$ (♠). ◀

The coloured graph (G', c') constructed in the proof of Theorem 10 can be modified into a colourful graph by omitting exactly one edge adjacent to each red vertex and exactly three edges inside each blue component. It can be readily checked that this is the minimum number of edges required. Hence Colourful Components is polynomial-time solvable on these coloured graphs (G', c'), whereas Colourful Partition is NP-complete. Thus we have:

▶ **Corollary 11.** *There exists a family of instances on which* Colourful Components *and* Colourful Partition *have different complexities (assuming* P ≠ NP*).*

We now present our second, third and fourth NP-hardness results. The last condition in Theorem 12 shows that the number of uniquely coloured vertices is not a useful parameter.

▶ **Theorem 12** (♠). Colourful Partition *and* Colourful Components *are* NP*-complete for coloured trees with maximum degree at most* 6*, colour-multiplicity* 2*, and no uniquely coloured vertices.*

▶ **Theorem 13** (♠). 2-Colourful Partition *is* NP*-complete for coloured split graphs.*

▶ **Theorem 14** (♠). 2-Colourful Partition *is* NP*-complete for coloured planar bipartite graphs of maximum degree* 3 *and path-width* 3*.*

We complement Theorem 14 with a positive result (Theorem 15) whose bounds are best possible due to Theorem 14. The idea behind it is that the sets V_1 and V_2 of a colourful partition of size 2 form connected subtrees in a tree decomposition. Branching over all options, we "guess" two vertices a and b of one bag assigned to different sets V_i. By exploiting the treewidth-2 assumption we can translate the instance into an equivalent instance of 2-Satisfiability.[2]

[2] A graph has treewidth at most 2 if and only if every 2-connected component is a series-parallel graph. This fact can be used for an alternative proof of Theorem 15.

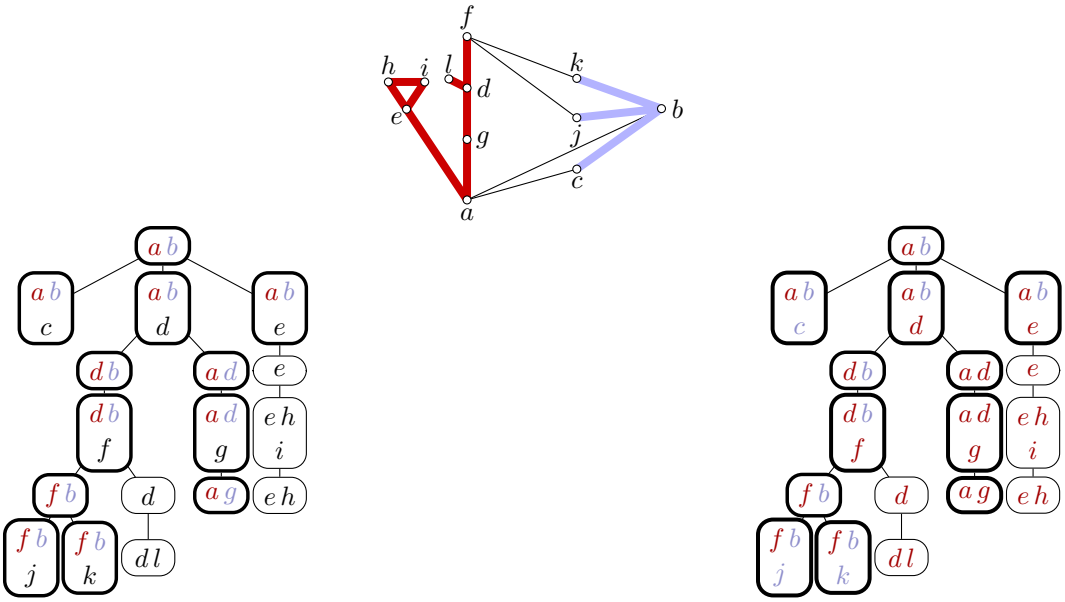


Figure 3 Illustration of the polynomial-time algorithm for treewidth 2. Top: an input graph (the vertex colouring is not represented), with a partition cutting (a, b), i.e. a is in one part, b is in the other. Bottom-left: a tree-decomposition of the graph satisfying Properties 1, 2 and 3. The bags in the head subtree are in bold. Precuts computed as in Claim 19 are shown using colours on the top line of bags that have precuts. For example, the bag $\{d, b, f\}$ is precut on (d, b), as can be verified directly in the graph (if d, b and f are not all in the same part of an (a, b)-partition, then d must be in the same part as a). Here f is attached to b, and e is attached to a but not to b. Bottom-right: the tree decomposition showing a possible output of the 2-Satisfiability formula, where the colours represent the values of each x_u.

▶ **Theorem 15.** 2-Colourful Partition *is polynomial-time solvable for coloured graphs of treewidth at most* 2.

Proof. Let (G, c) be a coloured graph on n vertices such that G has treewidth at most 2. Without loss of generality we may assume that G is connected. We may assume that G is not colourful, otherwise we are trivially done. If G has treewidth 1, then it is a tree and we apply Theorem 7. Hence we may assume that G has treewidth 2. Let $(T, \mathcal{X})$ be a tree decomposition of G of width 2 (so all bags of $\mathcal{X}$ have size at most 3). We can obtain this tree decomposition in linear time [13]. Let us state and then explain two properties that we may assume hold for $(T, \mathcal{X})$. These properties, along with the main definitions necessary for our algorithm, are illustrated in Figure 3.

1. For any two adjacent nodes i, j in T, one of X_i and X_j strictly contains the other.
2. All bags are pairwise distinct.

In fact, what we will show is that if we find that $(T, \mathcal{X})$ does not have these two properties, then we can make simple changes to obtain a tree decomposition that does. First, if there are two adjacent nodes i, j in T such that neither X_i nor X_j contains the other, we remove the edge ij from T, create a new node k that is adjacent to both i and j and let $X_k = X_i \cap X_j$. Note that $X_k \neq \emptyset$, as G is connected. We now have a tree decomposition that has Property 1 unless $X_i = X_j$ for a pair of adjacent nodes i and j. But now for *every* pair of identical bags X_i and X_j, we delete j and make each of its neighbours adjacent to i and so obtain a tree decomposition with Properties 1 and 2.

Let a and b be a fixed pair of adjacent vertices in G. Almost all of the remainder of this proof is concerned with describing an algorithm that decides whether or not G has a colourful partition of size 2 in which a and b belong to different parts. Clearly such an algorithm suffices: we can apply it to each of the $O(n^2)$ pairs of adjacent vertices in G to determine whether G has *any* colourful partition of size 2 (since such a partition must separate at least one pair of adjacent vertices).

As a and b are adjacent, there is at least one bag that contains both of them. We may assume that this bag is $X_0 = \{a, b\}$; otherwise we add a new node 0 to T with $X_0 = \{a, b\}$ and make 0 adjacent to i such that X_i is a (larger) bag that contains both a and b.

We orient all edges of T away from 0 and think of T as being rooted at 0. We write $i \to j$ to orient an edge from i to j. If $i \to j$ is present in T, then i is the *parent* of j and j is a *child* of i. The *head subtree* of T is the subtree obtained by removing all 1-vertex bags along with all their descendants. For any oriented edge $i \to j$ in T, we have either $X_i \subsetneq X_j$ or $X_j \subsetneq X_i$ by Property 1. The tree $T[i]$ is the subtree of T rooted at i; in particular $T = T[0]$. The set $V[i]$ denotes $\bigcup_{j \in V(T[i])} X_j$. As we shall explain, we may assume the following property.

3. For any node i of T, the subgraph $G[V[i]]$ induced by $V[i]$ is connected.

It is possible that we must again modify the tree decomposition to obtain this property. Suppose that it does not hold for some node i. That is, the vertices of $V[i]$ can be divided into two sets U and W such that there is no edge from U to W in G. We create two trees $T_{i,U}$ and $T_{i,W}$ that are isomorphic to $T[i]$: for each vertex j in $T[i]$, we let j_U and j_W be the corresponding nodes in $T_{i,U}$ and $T_{i,W}$ respectively and let $X_{j_U} = X_j \cap U$, $X_{j_W} = X_j \cap W$. Then the tree decomposition is modified by replacing $T[i]$ by $T_{i,U}$ and $T_{i,W}$ and making each of i_U and i_W adjacent to the parent of i. (The vertex i certainly has a parent since $i = 0$ would imply that G is not connected.) If at any point we create a node whose associated bag is empty or identical to that of its parent, we delete it and make its children (if it has any) adjacent to its parent. Note that although the number of bags may increase through this operation, the sum of the sizes of the bags of T can only decrease, and at least one bag gets its size reduced, so overall this operation only needs to be applied a polynomial number of times. In this way we obtain a decomposition that now satisfies each of Properties 1, 2 and 3.

We say that a colourful partition $P = (V_1, V_2)$ of G is an (a, b)-*partition* if $a \in V_1$ and $b \in V_2$ and we say that P *cuts* a pair (u_1, u_2) if $u_1 \in V_1$ and $u_2 \in V_2$. We emphasize that the order is important. A colourful partition P *respects* a bag X if $X \subseteq V_1$ or $X \subseteq V_2$. Note that every colourful partition respects all 1-vertex bags. Let X be a bag that contains vertices u and v. Then X is *precut* on (u, v) if every (a, b)-partition either cuts (u, v) or respects X. Note that X_0 is precut on (a, b) by definition. We now prove three structural claims.

▷ Claim 16. If an (a, b)-partition P respects a bag X_i, then it respects every X_j with $i \to j$.

Proof. Consider the set $\mathcal{C}$ of bags that are not respected by P. Then $\mathcal{C}$ is the intersection of the set of bags containing at least one vertex from V_1 and the set of bags containing at least one vertex from V_2. Since both V_1 and V_2 are connected in G, the set of nodes whose bags contain at least one vertex from V_1 and the set of nodes whose bags contain at least one vertex from V_2 induce subtrees of T. Hence their intersection, $\mathcal{C}$, is a set of nodes that also induce a subtree of T. Since (a, b) is not respected by P, we know $0 \in \mathcal{C}$. This means that if i is not in $\mathcal{C}$, then for every other vertex $j \in T[i]$, j cannot be in $\mathcal{C}$. ◁

▷ Claim 17. Let $i \to j$ be an oriented edge of T with $X_i = \{u, v\}$ and $|X_j| = 3$. If X_i is precut on (u, v), then X_j is precut on (u, v).

Proof. By Property 1, we know that $X_i \subsetneq X_j$, so X_j contains u and v. Suppose (u, v) is a precut on X_i and let P be an (a, b)-partition. If P cuts (u, v) then we are done. Otherwise, P respects $X_i = \{u, v\}$, so by Claim 16, P also respects X_j. Thus (u, v) is a precut on X_j. ◁

▷ Claim 18. Let $i \to j$ be an oriented edge of T with $X_i = \{u, v, w\}$ and $|X_j| = 2$ such that X_i is precut on (u, v). If $X_j = \{v, w\}$, then X_j is precut on (w, v). If $X_j = \{u, w\}$, then X_j is precut on (u, w).

Proof. Let $P = (V_1, V_2)$ be an (a, b)-partition of G. Suppose P does not respect X_j. Then if X_j contains w, we must have that w is not in the same part of the partition as the other vertex – either u or v – of X_j. By Claim 16, P does not respect X_i either, so we know $u \in V_1$ and $v \in V_2$. Thus if $X_j = \{v, w\}$, then $w \in V_1$, and if $X_j = \{u, w\}$, then $w \in V_2$. ◁

By Claim 16 and the fact that 1-vertex bags are respected, for every node i not in the head subtree of T, X_i is respected by every (a, b)-partition of G.

▷ Claim 19. For every node i in the head subtree, X_i is precut on some pair of its vertices, and these precuts can be computed in linear time.

Proof. We show that Claim 19 holds by proving a slightly stronger statement: for all d, for each node j in the head subtree at distance d from 0, X_j is precut on a pair of its vertices, and if X_j contains three vertices, then it is precut on some pair (u, v) such that $X_i = \{u, v\}$ where i is the parent of j. We prove this by induction on d. The base case holds as X_0 is precut on (a, b). For the inductive case, suppose that j is some node at distance $d > 0$ with parent i. As i and j are in the head subtree, each of X_i and X_j has either two or three vertices. Suppose first that $X_j = \{u, v, w\}$, and then we may assume that X_i is, say, $\{u, v\}$. (Recall that $X_i \subsetneq X_j$ or $X_i \subsetneq X_j$ by Property 1.) By the induction hypothesis, we know that X_i is precut on (u, v) or on (v, u) and so, by Claim 17, X_j is precut in the same way. Now suppose that X_j has two vertices and $X_i = \{u, v, w\}$. Since X_0 has two vertices, it follows that $X_i \neq X_0$, so i has a parent h, and we can suppose that $X_h = \{u, v\}$. By the induction hypothesis, without loss of generality we may assume X_h is precut on (u, v), so X_i is precut on (u, v) by the same argument as above. Then, as $\mathcal{X}$ has no identical bags by Property 2, we find that $X_j = \{u, w\}$ or $X_j = \{v, w\}$. By Claim 18, X_j is precut. ◁

We say that two vertices u and v of G are *attached* if they are adjacent or $\mathcal{X}$ contains the bag $\{u, v\}$. If u and v are not attached, then they are *detached*.

▷ Claim 20. Let $X_i = \{u, v, w\}$ be a 3-vertex bag precut on (u, v). If w and u (or w and v) are detached, then in every (a, b)-partition of G the vertices w and v (respectively w and u) are in the same colourful set.

Proof. Let $P = (V_1, V_2)$ be an (a, b)-partition. If P respects X_i, then w is in the same colourful set as both u and v. Thus we may assume that P does not respect X_i and so $u \in V_1$ and $v \in V_2$, since X_i is precut on (u, v). We may assume without loss of generality that w is detached from u and so we must show that $w \in V_2$.

Let us assume instead that $w \in V_1$ and derive a contradiction. By the connectivity of $G[V_1]$, there exists an induced path Q on ℓ edges in $G[V_1]$ that connects u to w. As u and w are detached, they are not adjacent, so ℓ is at least 2. No internal vertex of Q is in X_i since u and w are its end-vertices and v is not in V_1. Let $X_{i_1}, \ldots, X_{i_\ell}$ be bags of $\mathcal{X}$ such that X_{i_j} is a bag that contains the pair of vertices joined by the jth edge of Q. We take a walk in T from i_1 to i_ℓ by stitching together paths from i_j to i_{j+1}, $1 \leq j \leq \ell - 1$. As X_{i_j} and $X_{i_{j+1}}$ correspond to incident edges of Q, and both X_{i_j} and $X_{i_{j+1}}$ contain the internal

vertex of Q incident with both these edges, every bag of the nodes along the path between them also contains this vertex. Thus every bag of the nodes along our walk from i_1 to i_ℓ contains an internal vertex of Q and so none of these bags is X_i. Therefore our walk must be contained within a connected component of $T \setminus i$. Let j be the node of this component adjacent to i in T. We note that X_i contains u and w, X_{i_1} contains u, X_{i_ℓ} contains w, and the paths from i to each of i_1 and i_ℓ go through j. Thus X_j must contain u and w, and so, by Property 1, it follows that $X_j = \{u, w\}$. As u and w are detached, we have a contradiction. ◁

We now build ϕ, an instance of 2-SATISFIABILITY, that will help us to find an (a, b)-partition of G (if one exists). For each vertex u of G, we create a variable x_u (understood as "$u \in V_1$"). We add clauses (or pairs of clauses) on two variables equivalent to the following statements:

$$x_a = \texttt{true} \tag{1}$$
$$x_b = \texttt{false} \tag{2}$$
$$x_u \neq x_v \text{ if } u \text{ and } v \text{ have the same colour} \tag{3}$$

For any bag X_i precut on some pair (u, v),

$$x_u \vee \neg x_v \tag{4}$$
$$\neg x_u \Rightarrow \neg x_w \quad \forall w \in V[i] \setminus \{u\} \tag{5}$$
$$x_v \Rightarrow x_w \quad \forall w \in V[i] \setminus \{v\} \tag{6}$$

For each bag $X_i = \{u\}$ of size 1,

$$x_w = x_u \quad \forall w \in V[i] \setminus \{u\} \tag{7}$$

For each bag $\{u, v, w\}$ of size 3 precut on (u, v)

$$x_w = x_u \text{ if } w \text{ is detached from } v \tag{8}$$
$$x_w = x_v \text{ if } w \text{ is detached from } u \tag{9}$$

We now claim that G has an (a, b)-partition if and only if ϕ is satisfiable.

First suppose that G has an (a, b)-partition P. For each vertex u in G, set $x_u = \texttt{true}$ if $u \in V_1$, and $x_u = \texttt{false}$ otherwise. As P cuts (a, b), Statements (1) and (2) are satisfied. As P is colourful, Statement (3) is satisfied.

Consider a bag X_i precut on (u, v). Two cases are possible: either P cuts (u, v), or X is respected by P. In the first case we have $x_u = \texttt{true}$ and $x_v = \texttt{false}$, which is enough to satisfy Statements (4), (5) and (6). In the second case, $x_u = x_v$ (which satisfies Statement (4)), and by Claim 16, for each j in $T[i]$, X_j is respected by P. Thus all vertices in $V[i]$ are in the same subset V_1 or V_2, hence they have the same value of x_w, which satisfies Statements (5) and (6).

For Statement (7), note that each bag X_i of size 1, and, by Claim 16, every bag X_j such that j is a descendent of i is always respected, so all values of x_w for $w \in V[i]$ are identical. Statements (8) and (9) follow from Claim 20.

Now suppose ϕ is satisfiable. We construct two disjoint sets V_1 and V_2 by putting a vertex u in V_1 if $x_u = \texttt{true}$ in V_1 and in V_2 otherwise. We show (V_1, V_2) is an (a, b)-partition of G. By Statements (1) and (2), $a \in V_1$ and $b \in V_2$. By Statement (3), two vertices of the same colour cannot belong to the same set. It remains to prove that both sets are connected.

In fact, we shall prove by induction on d, that for the set of vertices belonging to bags of nodes at distance at most d from 0 in T, the two subsets found by dividing the set according to membership of V_1 or V_2 each induce a connected subgraph of G. For the base case, we consider $X_0 = \{a, b\}$ and the two subsets contain a single vertex so we are done. For the inductive case, we consider a node j at distance $d > 0$ from 0. Let i be the parent of j in T. It is enough to show that any vertex in $X_j \setminus X_i$ is in the same component of $G[V_1]$ or $G[V_2]$ as a vertex of X_i assigned to the same set (V_1 or V_2).

Let us first assume that X_j is in the head subtree. If $|X_j| = 2$, then $X_j \subsetneq X_i$ so we may assume that $X_j = \{u, v, w\}$ and also that $X_i = \{u, v\}$, and X_i and X_j are both precut on (u, v) (using Claims 17 and 19).

We distinguish three cases.

▷ Case 1. $x_u = x_v = \texttt{true}$.

Considering Statement (6) for bag X_i, we have $x_w = \texttt{true}$, as well as $x_{w'} = \texttt{true}$ for every vertex $w' \in V[i] \setminus X_i$. Hence $V[i] \subseteq V_1$. By Property 3, $G[V[i]]$ is connected, so w is in the same component of $G[V_1]$ as v.

▷ Case 2. $x_u = x_v = \texttt{false}$.

Symmetrically to Case 1, we can show that w is in the same component of $G[V_2]$ as u.

▷ Case 3. $x_u \neq x_v$.

By Statement (4), in this case we have $x_u = \texttt{true}$ and $x_v = \texttt{false}$. Suppose that $x_w = \texttt{true}$ (the case for $x_w = \texttt{false}$ again follows symmetrically). Since $x_w \neq x_v$, by Statement (9), w must be attached to u. If there is an edge from u to w, then we are done. If there is no such edge, then $\mathcal{X}$ has a bag $X_k = \{u, w\}$, and k must be a child of j. As k is in the head subtree, X_k is precut on (u, w) or (w, u) by Claim 19. By Statements (5) and (6), since $x_u = x_w = \texttt{true}$, $x_{w'} = \texttt{true}$ for every $w' \in V[k] \setminus \{u, w\}$ and thus $V[k] \subseteq V_1$. Since $G[V[k]]$ is connected and contains both u and w, we are done.

Now we consider the case when X_j is not in the head subtree. Thus X_j is in a subtree that has at its root a bag containing a single vertex u and, by Statement (7), every vertex in the bags of the subtree are in the same subset V_1 or V_2 as u and these vertices together induce a connected subgraph. Thus, as u is also in the parent of the root of the subtree (since G is connected, the parent is not empty), we are done.

Thus we have a polynomial-time algorithm to decide 2-Colourful Partition on the instance (G, c). First we compute in polynomial time a tree-decomposition $(T, \mathcal{X})$ of G with Properties 1, 2 and 3. Then for every pair of adjacent vertices a and b in G, we check whether there is a colourful (a, b)-partition of size 2 in polynomial time using the corresponding 2-Satisfiability formula ϕ. ◀

4 Parameterized Complexity

Guided by some observations on the aforementioned hardness constructions, we present three new FPT results in this section. The first one generalizes Theorem 6.

▶ **Theorem 21 (♠).** Colourful Partition *and* Colourful Components *are* FPT *when parameterized by treewidth plus the number of colours.*

The proof of our second FPT result uses similar arguments to the proof sketch of Theorem 8 given in [10]. However, its details are different, as optimal solutions for Colourful Partition do not necessarily translate into optimal solutions for Colourful Components.

▶ **Theorem 22.** Colourful Partition *is* FPT *when parameterized by vertex cover number.*

Proof Sketch. In fact we will show the result for the optimization version of Colourful Partition. Let (G, c) be a coloured graph. We will prove that we can find the size of a *minimum* colourful partition (one with smallest size) in FPT time. By a simple greedy argument, we can find a vertex cover S of G that contains at most $2\,\mathrm{vc}(G)$ vertices. It is therefore sufficient to show that Colourful Partition is FPT when parameterized by $|S|$. If two vertices of G have the same colour then they will always be in different colourful components of G. Thus if two vertices with the same colour are adjacent, we can delete the edge that joins them, that is, we may assume that c is a proper colouring of G (note that deleting edges from G maintains the property that S is a vertex cover). Since S is a vertex cover, $T = V(G) \setminus S$ is an independent set. Let C be the set of colours used on G. We let $s = |S|$ and for a set $S' \subseteq S$, we let $T_i(S')$ be the set of vertices with colour i whose neighbourhood in S is S', that is, for all $u \in T_i(S')$, we have that $N_S(u) = S'$ and $c(u) = i$.

▷ Rule 1. If there is a colour $i \in C$ and a set $S' \subseteq S$ such that $|T_i(S')| \geq s + 1$, then delete $|T_i(S')| - s$ (arbitrary) vertices of $T_i(S')$ from G.

We claim that we can safely apply Rule 1 (♠) and apply it exhaustively. We again denote the resulting instance by (G, c) and let $T = V(G) \setminus S$. Note that now $|T_i(S')| \leq s$ for every $i \in C$ and every $S' \subseteq S$. Consequently, the number of vertices of T with colour i is at most $s2^s$. Note that this means that $|T| \leq |C|s2^s$ and thus the total number of vertices in G is at most $s + |C|s2^s$. Hence, it remains to bound the size of C by a function of s.

We let C_T denote the set of colours that appear on vertices of T but not on vertices of S. For two colours $i, j \in C_T$, if $T_i(S') = T_j(S')$ holds for all $S' \subseteq S$, then we say that i and j are *clones* and note that these colours are interchangeable. For a colourful partition $P = (V_1, \ldots, V_k)$ for G, we let $P_S = (V_1 \cap S, \ldots, V_k \cap S)$ be the partition of S *induced by* P (note that in this case we allow some of the blocks to be empty).

We consider each partition Q of S. If a block of Q is not colourful, then we discard Q. Otherwise we determine a minimum colourful partition P for G with $P_S = Q$; note that such a partition P may not exist, as it may not be possible to make the blocks of Q connected (by using vertices T in addition to edges both of whose endpoints lie in S). We will choose the colourful partition for G that has minimum size overall. Let Q be a partition of S in which each block is colourful.

▷ Rule 2. If there are s distinct colours $i_1, \ldots, i_s$ in C_T that are pairwise clones, then delete all vertices with colour i_s from G.

We claim that we can safely apply Rule 2 (♠) and apply it exhaustively; again call the resulting graph G and define S and T as before.

▷ Claim 23 (♠). $|C_T| \leq (s-1)(s+1)^{2^s}$.

We continue as follows. The number of different colours used on vertices of S is at most s. Hence $C \setminus C_T$ has size at most s. Recall that for every colour $i \in C$, the number of vertices of T with colour i is at most $s2^s$. We combine these two facts with Claim 23 to deduce that $|V| = |S| + |T| \leq s + |C|s2^s = s + |C \setminus C_T|s2^s + |C_T|s2^s \leq s + s^2 2^s + (s-1)(s+1)^{2^s} s2^s$. Then by brute force we can compute a minimum colourful partition P for G subject to the restriction that $P_S = Q$ in $f(s)$ time for some function f that only depends on s.

The correctness of our FPT-algorithm follows from the above description. It remains to analyze the running time. Applying Rule 1 exhaustively takes $O(2^s|C|) = O(2^s n)$ time, as the number of different subsets $S' \subseteq S$ is 2^s. We then branch into at most s^s directions by

considering every partition of S. Applying Rule 2 exhaustively takes $O(2^s n^2)$ time, as for each colour $i \in C_T$ we first calculate the values of $|T_i(S')|$ for every $S' \subseteq S$, which can be done in $O(2^s n)$ time. Doing this for every colour takes a total of $O(2^s n^2)$ time and partitioning the colours into sets that are clones can be done in $O(2^{2s} n^2)$ time, and deleting colours can be done in $O(sn)$ time. As every auxiliary graph F has at most n vertices, we can compute a maximum matching in F in $O(n^{\frac{5}{2}})$ time by using the Hopcroft-Karp algorithm [9]. Finally, translating a minimum solution into a minimum solution for the graph in which we restored the vertices we removed due to exhaustive application of Rules 1 and 2 takes $O(n)$ time. This means that the total running time is $O(2^s n) + s^s(O(2^{2s} n^2) + O(s) + O(n^{\frac{5}{2}}) + f(s) + O(n)) = f'(s)O(n^{\frac{5}{2}})$ for some function f' that only depends on s, as desired. ◀

The Disjoint Connected Subgraphs problem takes as input a graph G with r pairwise disjoint subsets $Z_1, \ldots, Z_r$ of $V(G)$ for some $r \geq 1$. It asks whether we can partition $V(G) \setminus (Z_1 \cup \cdots \cup Z_r)$ into sets $S_1, \ldots, S_r$ such that every $S_i \cup Z_i$ induces a connected subgraph of G. Robertson and Seymour introduced this problem in their graph minor project and proved that it is cubic-time solvable as long as $Z_1 \cup \cdots \cup Z_r$ has constant size [11].

▶ **Theorem 24.** *When parameterized by the number of non-uniquely coloured vertices,* Colourful Components *is para-*NP*-complete, but* Colourful Partition *is* FPT.

Proof. The Multiterminal Cut problem is to test for a graph G, integer p and terminal set S, if there is a set E' with $|E'| \leq p$ such that every terminal in S is in a different component of $G - E'$. This problem is NP-complete even if $|S| = 3$ [6]. To prove the first part, give each of the three vertices in S colour 1 and the vertices in $G - S$ colours $2, \ldots, |V| - 2$.

To prove the second part, let (G, c) be a coloured graph and k be an integer. We assume without loss of generality that G is connected. Let Q with $|Q| = q$ be the set of non-uniquely coloured vertices. If $k \geq q$, then place each of the q vertices of Q in a separate component and assign the uniquely coloured vertices to components in an arbitrary way subject to maintaining connectivity of the q components. This yields a colourful partition of (G, c) of size at most k. Now assume that $k \leq q - 1$. We consider every possible partition of Q into k sets $Z_1, \ldots, Z_k$, where some of the sets Z_i may be empty. It remains to solve Disjoint Connected Subgraphs on the input $(G, Z_1, \ldots, Z_k)$. Note that $|Z_1| + \cdots + |Z_k|$ has size q. Hence, by the above result of Robertson and Seymour [11], solving Disjoint Connected Subgraphs takes cubic time. As there are $O(q^q)$ partitions to consider, the result follows. ◀

5 Conclusions

We showed that Colourful Partition and Colourful Components are NP-complete for coloured trees of maximum degree at most 6 (and colour-multiplicity 2). What is their complexity for coloured trees of maximum degree d for $3 \leq d \leq 5$? Colourful Components is known to be NP-complete for 3-coloured graphs of maximum degree 6 (Theorem 3); we also ask if one can prove a result analogous to Theorem 10: what is its complexity for coloured graphs of maximum degree 3? Our main result is that 2-Colourful Partition is NP-complete for coloured (planar bipartite) graphs of path-width 3 (and maximum degree 3), but polynomially solvable for coloured graphs of treewidth 2. We believe that the latter result can be extended to k-Colourful Partition ($k \geq 3$), but leave this for future research. A more interesting question is if the problem is FPT for treewidth 2 when parameterized by k.

References

1 Anna Adamaszek and Alexandru Popa. Algorithmic and Hardness Results for the Colorful Components Problems. *Algorithmica*, 73(2):371–388, 2015.

2 Sharon Bruckner, Falk Hüffner, Christian Komusiewicz, Rolf Niedermeier, Sven Thiel, and Johannes Uhlmann. Partitioning into Colorful Components by Minimum Edge Deletions. *Proc. CPM 2012, LNCS*, 7354:56–69, 2012.

3 Laurent Bulteau, Konrad K. Dabrowski, Guillaume Fertin, Matthew Johnson, Daniël Paulusma, and Stéphane Vialette. Finding a Small Number of Colourful Components. Manuscript, 2018. `arXiv:1808.03561`.

4 Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Maximal strip recovery problem with gaps: Hardness and approximation algorithms. *Journal of Discrete Algorithms*, 19:1–22, 2013.

5 Janka Chlebikova and Clément Dallard. A Complexity Dichotomy for Colourful Components Problems on k-caterpillars and Small-Degree Planar Graphs. Manuscript, 2019. `arXiv:1902.11191`.

6 Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

7 Riccardo Dondi and Florian Sikora. Parameterized complexity and approximation issues for the colorful components problems. *Theoretical Computer Science*, 739:1–12, 2018.

8 George He, Jiping Liu, and Cheng Zhao. Approximation Algorithms for Some Graph Partitioning Problems. *Journal of Graph Algorithms and Applications*, 4(2):1–11, 2000.

9 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

10 Neeldhara Misra. On the Parameterized Complexity of Colorful Components and Related Problems. *Proc. IWOCA 2018, LNCS*, 10979:237–249, 2018.

11 Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

12 Thomas J. Schaefer. The Complexity of Satisfiability Problems. *Proc. STOC 1978*, pages 216–226, 1978.

13 Joseph A. Wald and Charles J. Colbourn. Steiner trees, partial 2–trees, and minimum IFI networks. *Networks*, 13(2):159–167, 1983.

14 Chunfang Zheng, Krister M. Swenson, Eric Lyons, and David Sankoff. OMG! Orthologs in multiple genomes – competing graph-theoretical formulations. *Proc WABI 2011, LNBI*, 6833:364–375, 2011.

Streaming Dictionary Matching with Mismatches

Paweł Gawrychowski
University of Wrocław, 50-137 Wrocław, Poland
gawry@cs.uni.wroc.pl

Tatiana Starikovskaya
DIENS, École normale supérieure, PSL Research University, 75005 Paris, France
tat.starikovskaya@gmail.com

Abstract

In the k-mismatch problem we are given a pattern of length m and a text and must find all locations where the Hamming distance between the pattern and the text is at most k. A series of recent breakthroughs have resulted in an ultra-efficient streaming algorithm for this problem that requires only $\mathcal{O}(k \log \frac{m}{k})$ space [Clifford, Kociumaka, Porat, SODA 2019]. In this work, we consider a strictly harder problem called dictionary matching with k mismatches, where we are given a dictionary of d patterns of lengths $\leq m$ and must find all their k-mismatch occurrences in the text, and show the first streaming algorithm for it. The algorithm uses $\mathcal{O}(kd \log^k d \operatorname{polylog} m)$ space and processes each position of the text in $\mathcal{O}(k \log^k d \operatorname{polylog} m + occ)$ time, where occ is the number of k-mismatch occurrences of the patterns that end at this position. The algorithm is randomised and outputs correct answers with high probability.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Streaming, multiple pattern matching, Hamming distance

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.21

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving this paper.

1 Introduction

The pattern matching problem is the fundamental problem of string processing and has been studied for more than 40 years. Most of the existing algorithms are deterministic and assume the word-RAM model of computation. Under these assumptions, we must store the input in full, which is infeasible for modern massive data applications. The streaming model of computation was designed to overcome the restrictions of the word-RAM model. In this model, we assume that the text arrives as a stream, one character at a time. The characters are assumed to be integers that fit in $\mathcal{O}(\log n)$-bit machine words, where n is the length of the stream. Each time a new character of the text arrives, we must update the output. The space complexity of an algorithm is defined to be all the space used, including the space we need to store the information about the pattern(s) and the text. The time complexity of an algorithm is defined to be the time we spend to process one character of the text. The streaming model of computation aims for algorithms that use as little space and time as possible. All streaming algorithms we discuss in this paper are randomised by necessity. They can err with probability inverse-polynomial in the length of the input.

The first sublinear-space streaming algorithm for exact pattern matching was suggested by Porat and Porat in FOCS 2009 [26]. For a pattern of length m, their algorithm uses $\mathcal{O}(\log m)$ space and $\mathcal{O}(\log m)$ time per character. Later, Breslauer and Galil gave a $\mathcal{O}(\log m)$-space and $\mathcal{O}(1)$-time algorithm [8].

The first algorithm for dictionary matching was developed by Aho and Corasick [1]. The algorithm assumes the word-RAM model of computation, and for a dictionary of d patterns of length at most m, uses $\Omega(md)$ space and $\mathcal{O}(1 + occ)$ amortised time per

character, where *occ* is the number of the occurrences ending at this position. Apart from the Aho–Corasick algorithm, other word-RAM algorithms for exact dictionary matching include [3, 4, 7, 13, 14, 16, 19, 21, 22, 27]. In ESA 2015, Clifford et al. [9] showed a streaming dictionary matching algorithm that uses $\mathcal{O}(d \log m)$ space and $\mathcal{O}(\log\log(m+d) + occ)$ time per character. In ESA 2017, Golan and Porat [18] showed an improved algorithm that uses the same amount of space and $\mathcal{O}(1 + occ)$ time per character for constant-size alphabets.

In the k-mismatch problem we are given a pattern of length m and a text and must find all alignments of the pattern and the text where the Hamming distance is at most k. By reduction to the streaming exact pattern matching, Porat and Porat [26] showed the first streaming k-mismatch algorithm with space $\mathcal{O}(k^3 \log^7 m/\log\log m)$ and time $\mathcal{O}(k^2 \log^5 m/\log\log m)$. The complexity has been subsequently improved in [10, 11, 17]. The current best algorithm uses only $\mathcal{O}(k \log \frac{m}{k})$ space and $\mathcal{O}(\log \frac{m}{k}(\sqrt{k \log k} + \log^3 m))$ time per character [11].

In the problem of dictionary matching with k mismatches, we are given a set (dictionary) of d patterns of maximal length m and must find all their k-mismatch occurrences in the text. This problem is strictly harder than both k-mismatch and dictionary matching, and on the other hand, it is well-motivated by practical applications in cybersecurity and bioinformatics. In the word-RAM model, dictionary matching with k mismatches has been addressed in [2, 24, 25]. Muth and Manber [24] gave a randomised algorithm for $k = 1$, and Baeza-Yates and Navarro [2] and Navarro [25] gave the first algorithms for a general value of k. The time complexity of the algorithms is good on average, but in the worst case can be $\Omega(md)$ per character.

1.1 Our results

In this work, we commence a study of dictionary matching with k mismatches in the streaming model of computation. Our contribution is twofold. First, we show a streaming dictionary matching algorithm that uses space sublinear in m and time sublinear in both m and d (Section 4). Similar to previous work on streaming pattern matching, we assume that we receive the dictionary first, preprocess it (without accounting for the preprocessing time), and then receive the text.

▶ **Theorem 1.** *For any $k \geq 1$, there is a streaming algorithm that solves dictionary matching with k mismatches in $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ worst-case time per arriving character. The algorithm is randomised and its answers are correct w.h.p.*[1] *Both false-positive and false-negative errors are allowed.*

Hereafter *occ* is the number of k-mismatch occurrences of the patterns that end at the currently processed position of the text, i.e., it is at most d and typically it is much smaller than the total number of the occurrences of the patterns in the text. Our algorithm makes use of a new randomised variant of the k-errata tree (Section 3), a famous data structure of Cole, Gottlieb, and Lewenstein for dictionary matching with k mismatches [12]. This variant of the k-errata tree allows to improve both the query time and the space requirements and can be considered as a generalisation of the z-fast tries [5, 6], that have proved to be useful in many streaming applications.

Compare our result to a streaming algorithm that can be obtained by a repeated application of the k-mismatch algorithm [11]:

[1] $\tilde{\mathcal{O}}$ hides a multiplicative factor polynomial in $\log m$ and w.h.p. means that the error probability is at most $1/n^c$ for an arbitrary given constant c.

▶ **Corollary 2** (of Clifford, Kociumaka, and Porat [11]). *For any $k \geq 1$, there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(dk)$ space and $\tilde{\mathcal{O}}(d\sqrt{k})$ time per character. The algorithm is randomised and its answers are correct w.h.p.*

As it can be seen, the time complexity of Corollary 2 depends on d in linear way, which is prohibitive for applications where the stream characters arrive at a high speed and the size of the dictionary is large, up to several thousands of patterns, as we must be able to process each character before the next one arrives to benefit from the space advantages of streaming algorithms.

Our second contribution is a space lower bound for streaming dictionary matching with mismatches. In Section 6 we show the following claim by reduction from the Index problem (see the proof for the definition):

▶ **Lemma 3.** *Any streaming algorithm for dictionary matching with k mismatches such that its answers are correct w.h.p. requires $\Omega(kd)$ bits of space.*

2 Preliminaries: Fingerprints and sketches

In this section, we give the definitions of two hash functions that we use throughout the paper. We first give the definition of Karp–Rabin fingerprints that let us decide whether two strings are equal.

▶ **Definition 4** (Karp–Rabin fingerprints, Karp–Rabin [20]). *For a fixed prime p and $r \in [0, p-1]$ chosen uniformly at random, the Karp—Rabin fingerprint of a string $S = S[1]S[2]\dots S[m]$ is defined as a quadruple $\Phi(S) = (\varphi(S), \varphi^R(S), r^{|S|} \bmod p, r^{-|S|} \bmod p)$, where $\varphi(S) = \sum_{i=1}^{m} S[i] \cdot r^{m-i} \bmod p$ and $\varphi^R(S) = \sum_{i=1}^{m} S[i] \cdot r^{i-1} \bmod p$.*

▶ **Fact 5.** *For $r \in [0, p-1]$ chosen uniformly at random, the probability of two distinct strings of equal lengths $\ell \leq m$ over the integer alphabet $[0, p-1]$ to have equal Karp–Rabin fingerprints is at most m/p.*

Consider a string Z that is equal to the concatenation of two strings X and Y of length at most m, that is $Z = XY$. We can compute in $\mathcal{O}(1)$ time $\varphi(Z)$ given $\varphi(X)$ and $\varphi(Y)$, and $\varphi(Y)$ given $\varphi(Z)$ and $\varphi(X)$. Furthermore, given the Karp–Rabin fingerprint of $S[1]S[2]\dots S[m]$, we can compute the Karp–Rabin fingerprint of $S[m]S[m-1]\dots S[1]$ in $\mathcal{O}(1)$ time.

We now remind the definition of k-mismatch sketches that will allow us to decide whether two strings are at Hamming distance at most k.

▶ **Definition 6** (k-mismatch sketch, Clifford, Kociumaka, and Porat [11]). *For a fixed prime p and $r \in [0, p-1]$ chosen uniformly at random, the k-mismatch sketch $sk_k(S)$ of a string $S = S[1]S[2]\dots S[m]$ is defined as a tuple $(\phi_0(S), \dots, \phi_{2k}(S), \phi'_0(S), \dots, \phi'_k(S), \Phi(S))$, where $\phi_j(S) = \sum_{i=1}^{i=m} S[i] \cdot i^j \bmod p$ and $\phi'_j(S) = \sum_{i=1}^{i=m} S[i]^2 \cdot i^j \bmod p$ for $j \geq 0$.*

▶ **Lemma 7** (Clifford, Kociumaka, and Porat [11]). *Given the sketches $sk_k(S_1)$ and $sk_k(S_2)$ of two strings of equal lengths $\ell \leq m$, in $\tilde{\mathcal{O}}(k)$ time one can decide (with high probability) whether the Hamming distance between S_1 and S_2 is at most k. If so, one can report each mismatch p between S_1 and S_2 as well as $S_1[p]$ and $S_2[p]$. The algorithm uses $\mathcal{O}(k)$ space.*

▶ **Lemma 8** (Clifford, Kociumaka, and Porat [11]). *We can construct one of the sketches $sk_k(S_1)$, $sk_k(S_2)$, or $sk_k(S_1S_2)$ given the other two in $\tilde{\mathcal{O}}(k)$ time using $\mathcal{O}(k)$ space, provided that all the processed strings are over the alphabet $[0, p-1]$ and are of length at most m. Furthermore, we can compute $sk_k(S^m)$, where S^m is a concatenation of m copies of S, in $\tilde{\mathcal{O}}(k)$ time as well under the same assumption.*

3 Algorithm based on the randomised k-errata tree

In this section, we show a streaming algorithm for dictionary matching with k mismatches based on a new randomised implementation of the k-errata tree, a data structure introduced by Cole, Gottlieb, and Lewenstein [12].

▶ **Lemma 9.** *There is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(k \cdot (m + d \log^k d))$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character, where occ is the number of the occurrences. On request, the algorithm can output the mismatches in $\tilde{\mathcal{O}}(k)$ time per occurrence. The algorithm is randomised and its answers are correct w.h.p.*

We start by showing a randomised version of the k-errata tree. The k-errata tree [12] is a data structure that supports dictionary look-up with k mismatches queries: Given a query string Q, find all patterns in the dictionary that are at the Hamming distance at most k from it. The k-errata tree is a collection of compact tries that can answer a dictionary look-up with k mismatches for a string Q of length m in time $\mathcal{O}(m + \log^k d \log\log m + occ)$. The query algorithm consists of $\mathcal{O}(\log^k d)$ calls to a procedure called PrefixSearch. This procedure takes three arguments, a compact trie τ, a node u (or a position on an edge) in τ, and a string S, and must find the longest path in τ that starts at u and is labelled by a prefix of S. In our case, τ is always one of the compact tries of the k-errata tree, and S is always one of the suffixes of Q. Cole, Gottlieb, and Lewenstein [12] showed that one can use the suffix tree on the patterns in the dictionary to answer the PrefixSearch queries deterministically in $\mathcal{O}(\log\log m)$ time after $\mathcal{O}(m)$-time shared preprocessing. Unfortunately, this solution uses too much space and time for our purposes. In the randomised version of the k-errata tree, we implement each of the compact tries as a z-fast trie:

▶ **Fact 10** (z-fast tries, Belazzougui et al. [6])**.** *Consider a string S and suppose that we can compute the Karp—Rabin fingerprint of any prefix of S in t_φ time. A compact trie on a set of r strings of length at most m can be implemented in $\mathcal{O}(r)$ space to support the following queries in $\mathcal{O}(t_\varphi \cdot \log m)$ time: Given S, find the highest node v such that the longest prefix of S present in the trie is a prefix of the label of the root-to-v path. The answers are correct w.h.p.*[2]

This gives an efficient implementation of all PrefixSearch queries if u is the root of a compact trie, but there are more details for the general case. We provide full details, as well as the definition of the k-errata tree, in Appendix A.

▶ **Lemma 11.** *A dictionary of d patterns of maximal length m can be preprocessed into a data structure which we call* randomised k-errata tree *that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and allows retrieving all the patterns that are within Hamming distance k from Q or one of its prefixes in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time, assuming that we know the k-mismatch sketches of all prefixes of Q. The answers are correct w.h.p.*

We are now ready to give the proof of Lemma 9.

Proof of Lemma 9. During the preprocessing step, the algorithm builds the k-errata tree for the reverses of the patterns. During the main step, the algorithm maintains the Karp–Rabin fingerprints and the k-mismatch sketches of the m longest prefixes of the text in a round-robin fashion updating them in $\tilde{\mathcal{O}}(k)$ time when a new character arrives (Lemma 8). If the text

[2] Error probability comes from the collision probability for Karp–Rabin fingerprints.

ends with a k-mismatch occurrence of some pattern P_i, there is a suffix of the text of length $|P_i| \leq m$ such that the Hamming distance between it and some pattern in the dictionary is bounded by k. It means that we can retrieve all occurrences of such patterns by using the randomised k-errata tree for the reverse of the m-length suffix of the text. We can retrieve the fingerprint and the k-mismatch sketch of any substring of this suffix in $\tilde{\mathcal{O}}(k)$ time (Lemma 8), and therefore perform the dictionary look-up query in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time. In total, the algorithm uses $\tilde{\mathcal{O}}(k \cdot (m + d \log^k d))$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. ◀

4 Improving space

The algorithm of Corollary 2 is efficient in terms of space, but not in terms of time. The algorithm of Lemma 9 is efficient in terms of time, but not in terms of space. In this section, we show that it is possible to achieve sublinear dependency on m for space, and in m and d for time:

▶ **Theorem 12.** *There is a streaming algorithm that solves the problem of dictionary matching with k mismatches, for any $k \geq 1$, in $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p. Both false-positive and false-negative errors are allowed.*

Note that the time complexity of the algorithm is amortised. In Appendix 5 we show how to de-amortise the running time to obtain our main result, Theorem 1. The techniques that we use have flavour similar to [9–11, 18], but make a significant step forward to allow *both* mismatches and multiple patterns.

▶ **Definition 13** (k-period, Clifford et al. [10])**.** *The k-period of a string $S = S[1]S[2] \dots S[m]$ is the minimal integer $\pi > 0$ such that the Hamming distance between $S[\pi + 1, m]$ and $S[1, m - \pi]$ is at most $2k$.*

▶ **Observation 14.** *If the k-period of S is larger than d, there can be at most one k-mismatch occurrence of S per d consecutive positions of the text.*

Hereafter we assume $k \log \log d < \log^k d$ (all logs are base two), which is true for any $d \geq 3$ and $k \geq 1$. For $d = 1, 2$ we can use Corollary 2 to achieve the complexities of Theorem 1. Furthermore, we assume that the lengths of the patterns are at least $3d$, for shorter patterns we can use the algorithm of Lemma 9. We partition the dictionary into two smaller dictionaries: the first dictionary $\mathcal{D}_1$ contains the patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is larger than d, and the second dictionary $\mathcal{D}_2$ contains patterns P_i such that the k-period of their suffix τ_i is at most d. In Section 4.2 we show a streaming algorithm that finds all k-mismatch occurrences of the patterns in $\mathcal{D}_1$, and in Section 4.3 a streaming algorithm for $\mathcal{D}_2$. We run the two algorithms in parallel to obtain Theorem 12.

4.1 Reminder: The k-mismatch algorithm of Porat and Porat

We first give an outline of the k-mismatch algorithm of Porat and Porat [26] and explain how it can be applied to the dictionary matching setting.

Porat and Porat showed that the k-mismatch problem for a pattern P can be reduced to exact pattern matching in the following way. Let $Q = \{q_1, q_2, \dots, q_{\log m / \log \log m}\}$ be the set of the first $\log m / \log \log m$ primes larger than $\log m$, and $R = \{r_1, r_2, \dots, r_{k \log m / \log \log m}\}$

be the set of the first $k \log m / \log\log m$ primes larger than $\log m$. A subpattern $(P_i)^{\ell}_{q,r}$ of a pattern P_i is defined by two primes $q \in Q, r \in R$ and an integer $1 \leq \ell \leq q \cdot r$, namely, $P^{\ell}_{q,r} = P[\ell]P[q \cdot r + \ell]P[2q \cdot r + \ell] \dots$ and so on until the end of P. The prime number theorem implies that $q \in \tilde{\mathcal{O}}(1)$ and $r \in \tilde{\mathcal{O}}(k)$, and therefore for a fixed q, r there are $\tilde{\mathcal{O}}(k)$ subpatterns.

▶ **Lemma 15** (Porat–Porat [26]). *Consider an alignment of the pattern P_i and the text. Given the subset of the subpatterns of P_i that match exactly at this alignment, there is a deterministic $\tilde{\mathcal{O}}(k^2)$-time algorithm that outputs "No" if the Hamming distance between P_i and T is larger than k, and the true value of the Hamming distance otherwise.*

Using this reduction, we show a streaming algorithm that uses $\tilde{\mathcal{O}}(k^3 d)$ space and processes each character of the text in $\tilde{\mathcal{O}}(k^2 \log\log(m + d))$ time. On request, the algorithm can tell in $\tilde{\mathcal{O}}(k^2)$ time if there is a k-mismatch occurrence of a pattern P_i that ends at the current position of text. During the preprocessing step, for each pair of primes $q \in Q, r \in R$, we build a compact trie on the reverses of the subpatterns $(P_i)^{\ell}_{q,r}$. Furthermore, we preprocess each trie (using a depth-first traversal) to be able to tell in $\mathcal{O}(1)$ time if the reverse of the subpattern $(P_i)^{\ell}_{q,r}$ is a prefix of the reverse of the subpattern $(P_{i'})^{\ell'}_{q,r}$.

During the main stage, for each pair of primes $q \in Q, r \in R$ and an integer $1 \leq \ell \leq q \cdot r$ we define a text substream $T^{\ell}_{q,r} = T[\ell]T[q \cdot r + \ell]T[2q \cdot r + \ell] \dots$ and so on until the end of T. We then run the streaming dictionary matching algorithm of Clifford et al. [9] for the substream $T^{\ell}_{q,r}$ and the dictionary of subpatterns $(P_i)^{\ell'}_{q,r}$, where $i = \{1, 2, \dots, d\}$ and $1 \leq \ell' \leq q \cdot r$. At each position, the streaming dictionary matching algorithm outputs the id of the longest subpattern that matches at this position. In total for each pair of primes there are $\tilde{\mathcal{O}}(k)$ substreams and $\tilde{\mathcal{O}}(kd)$ subpatterns per substream, and therefore the algorithm uses $\tilde{\mathcal{O}}(k^3 d)$ space and $\tilde{\mathcal{O}}(k \log\log(m + kd))$ time per character, the latter is because each time a new character $T[p]$ arrives, where $p = q \cdot r + \ell$, we must update exactly one substream $T^{\ell}_{q,r}$ (and over all $q \in Q, r \in R$, there are $\tilde{\mathcal{O}}(k \log\log(m + kd))$ substreams to update).

Using the compact tries built at the preprocessing step, we can then check, for any subpattern $(P_i)^{\ell}_{q,r}$, if it matches at this position in $\mathcal{O}(1)$ time and therefore can decide if there is a k-mismatch occurrence of P_i in $\tilde{\mathcal{O}}(k^2)$ by Lemma 15.

4.2 Streaming algorithm for patterns with large periods

In this section, we show a streaming algorithm for the dictionary $\mathcal{D}_1$ that contains patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is at least d.

▶ **Lemma 16.** *If for each pattern in the dictionary $\mathcal{D}_1$ the k-period of its $2d$-length suffix is larger than d, then there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p.*

Note that any k-mismatch occurrence of a pattern P_i ends with a k-mismatch occurrence of τ_i. The first step of our algorithm is to retrieve the occurrences of τ_i. To do so, we run the streaming algorithm of Lemma 9. At each position of the text, the algorithm outputs all indices i such that there is a k-mismatch occurrence of τ_i ending at this position. After having found the occurrences of τ_i, our second step is to check if they can be extended into full occurrences of P_i which we do with the help of the streaming algorithm explained in Section 4.1.

We now analyse the complexity of the algorithm. To find occurrences of the suffixes τ_i, we need $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. The algorithm of Section 4.1 uses $\tilde{\mathcal{O}}(k^3 d)$ space and $\tilde{\mathcal{O}}(k \log\log(m + kd))$ time per character. To test if an

occurrence of τ_i can be extended into an occurrence of P_i, we need $\tilde{\mathcal{O}}(k^2)$ time. Importantly, by Observation 14, there is at most one k-mismatch occurrence of τ_i per d positions of the text. Hence, we will need $\tilde{\mathcal{O}}(k^2 d)$ time to test all k-mismatch occurrences of the suffixes τ_i that end at any d consecutive positions of the text. Lemma 16 follows.

4.3 Streaming algorithm for patterns with small periods

In this section, we show a streaming algorithm for the second dictionary $\mathcal{D}_2$ that contains patterns P_i such that the k-period of their suffix $\tau_i = P_i[|P_i| - 2d + 1, |P_i|]$ is at most d.

▶ **Lemma 17.** *If for each pattern in the dictionary $\mathcal{D}_2$ the k-period of its $2d$-length suffix is smaller than d, then there is a streaming algorithm for dictionary matching with k mismatches that uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. The algorithm is randomised and its answers are correct w.h.p.*

We define τ'_i, $|\tau'_i| \geq |\tau_i|$, to be the longest suffix of P_i with the k-period at most d. Two cases are possible:

1. The suffix τ'_i equals P_i (in other words, the k-period of P_i is at most d);
2. The suffix τ'_i is a proper suffix of P_i.

We first assume that Case 1 holds for all the patterns in $\mathcal{D}_2$, and then extend the algorithm to Case 2 as well. We start by showing a simple but important property of patterns with small periods.

▶ **Lemma 18.** *Consider a position r of the text. Let $j \cdot d$ be the largest multiple of d that is smaller than r and L be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ with the $2k$-period at most d. Every k-mismatch occurrence of $P_i \in \mathcal{D}_2$ in T that ends at the position r is fully contained in $L\,T[j \cdot d + 1, r]$.*

Proof. Consider an occurrence $T[\ell, r]$ of a pattern $P_i \in \mathcal{D}_2$ that ends at the position r. Since the length of P_i is at most m, $\ell \geq r - m + 1 > j \cdot d - m + 1$. Now, let $\rho \leq d$ be the k-period of P_i. Since the Hamming distance between $T[\ell, r]$ and P_i is at most k, the $2k$-period of $T[\ell, r]$ is at most ρ. Indeed, the Hamming distance between $T[\ell + \rho - 1, r]$ and $T[\ell, r - \rho + 1]$ is at most $2k$ plus the Hamming distance between $P_i[\rho, |P_i|]$ and $P_i[1, |P_i| - \rho + 1]$ which can be upper bounded by $2k$ in its turn. Therefore, the $2k$-period of $T[\ell, j \cdot d]$ is at most ρ and hence it is contained in L. ◀

4.3.1 Algorithm for Case 1

We are now ready to describe the algorithm for the Case 1. During the preprocessing stage, we build the k-errata tree for the reverses of all the patterns in $\mathcal{D}_2$. During the main stage of the algorithm, we maintain the suffix L and an associated data structure D. The data structure D will be used to answer the following queries in $\tilde{\mathcal{O}}(k)$ time: Given a suffix of L defined by its starting and ending positions, return its $4k$-mismatch sketch.

Let us first explain how we maintain L. We initialize L with an empty string and update it each d characters. While reading the next d characters of the text, that is a substring $T[(j-1) \cdot d + 1, j \cdot d]$, we compute the $4k$-mismatch sketches of its d prefixes in $\tilde{\mathcal{O}}(kd)$ time (Lemma 8). After having reached $T[j \cdot d]$, we update L. It suffices to compute the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ such that the Hamming distance between it and its copy shifted by ρ positions, for $\rho = 1, \ldots, d$, is at most $2k$. For a fixed value of ρ, we use binary search and the $4k$-mismatch sketches. Suppose we want to decide whether the Hamming distance between $T[\ell, j \cdot d - \rho + 1]$ and $T[\ell + \rho, j \cdot d]$ is at most $4k$. First note that we must only consider the case when $T[\ell, j \cdot d]$ is fully contained in $L\,T[(j-1) \cdot d + 1, j \cdot d]$.

▶ **Observation 19.** *If $T[\ell, j \cdot d]$ is longer than $L\,T[(j-1) \cdot d + 1, j \cdot d]$, then its $2k$-period is larger than d.*

Proof. If the $2k$-period of $T[\ell, j \cdot d]$ is at most d, the $2k$-period of $T[\ell, (j-1) \cdot d]$ is at most d. If $T[\ell, (j-1) \cdot d]$ is longer than L, we obtain a contradiction. ◀

Since we are only interested in the case when $T[\ell, j \cdot d]$ is fully contained in $L\,T[(j-1) \cdot d + 1, j \cdot d]$, both $T[\ell, j \cdot d - \rho + 1]$ and $T[\ell + \rho, j \cdot d]$ can be represented as a concatenation of a suffix of L and a substring of $T[(j-1) \cdot d + 1, j \cdot d]$. We can retrieve the $4k$-mismatch of any suffix of L in $\tilde{\mathcal{O}}(k)$ time using the data structure D and the $4k$-mismatch sketch of any substring of $T[(j-1) \cdot d + 1, j \cdot d]$ using Lemma 8. Therefore, we can compute the $4k$-mismatch sketches of both strings and hence the Hamming distance between them in $\tilde{\mathcal{O}}(k)$ time using Lemma 7. In total, we need $\tilde{\mathcal{O}}(dk)$ time to update L, or $\tilde{\mathcal{O}}(k)$ amortised time per character.

We now define the data structure D and explain how we update it. Suppose that after the latest update the $2k$-period of L is $\rho \leq d$ and consider a partitioning of L into non-overlapping blocks of length ρ. We say that a block contains a mismatch if, for some i, its i-th character is different from the i-th character of the preceding block. For convenience, we also say that the first block in L is mismatch-containing.

▶ **Observation 20.** *The total number of the blocks containing a mismatch is $\mathcal{O}(k)$.*

Proof. By definition, the Hamming distance between $L[1, |L| - \rho + 1]$ and $L[\rho + 1, |L|]$ is at most $4k$, and it upper bounds the number of the blocks containing a mismatch. ◀

D consists of two parts. First, we store a binary search tree on the set of the starting positions of all blocks containing a mismatch. Secondly, for each block $L[(j-1) \cdot \rho + 1, j \cdot \rho]$ containing a mismatch we store the $4k$-mismatch sketch of each of its suffixes, as well as the sketch of the suffix of L that starts at the position $(j-1) \cdot \rho + 1$. In total, D occupies $\tilde{\mathcal{O}}(k^2 d)$ space.

▶ **Lemma 21.** *We can update D in $\tilde{\mathcal{O}}(k^2)$ amortised time per character. After it has been updated, we can compute the $4k$-mismatch sketch of any suffix of L in $\tilde{\mathcal{O}}(k)$ time.*

Proof. Using the $4k$-mismatch sketches for $L[\rho, |L|]$ and $L[1, |L| - \rho + 1]$, we can find the $\mathcal{O}(k)$ blocks containing a mismatch in $\tilde{\mathcal{O}}(k)$ time. We can then re-build the binary search tree in $\tilde{\mathcal{O}}(k)$ time and compute the sketches for the $\mathcal{O}(k)$ mismatch-containing blocks in $\tilde{\mathcal{O}}(k^2 d)$ time.

Given a starting position ℓ of a suffix of L, we use the binary search tree to determine the streak of blocks without mismatches it belongs to, and retrieve the sketch of the suffix starting just after the streak in $\tilde{\mathcal{O}}(k)$ time. The remaining part consists of a number of repetitions of the block containing the position ℓ prepended with the suffix of the block. We can compute the sketch of the block and of its suffix in $\tilde{\mathcal{O}}(k)$ time, and therefore we can compute the sketch of the remaining part in $\tilde{\mathcal{O}}(k)$ time using Lemma 8. ◀

Let $T[r]$ be the latest arrived character of the text. To retrieve the k-mismatch occurrences that end at the position r, we use the k-errata tree for the reverses of the patterns in $\mathcal{D}_2$ that we build during the preprocessing stage. Let $j \cdot d$ be the largest multiple of d that is at most r and let L be defined as above. By Lemma 18, any k-mismatch occurrence of pattern $P_i \in \mathcal{D}_2$ that ends at r must be equal either to a suffix of $T[j \cdot d + 1, r]$, or to the concatenation of some suffix of L and $T[j \cdot d + 1, r]$. The data structure D allows to compute the $4k$-mismatch sketch (and therefore k-mismatch) of any suffix of L in $\tilde{\mathcal{O}}(k)$ time. We can

also compute the $4k$-mismatch sketch of any of the d latest suffixes of the text in $\tilde{\mathcal{O}}(k)$ time. Therefore, we can retrieve the k-mismatch occurrences of the patterns for a current position in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time using the k-errata tree. In total, the algorithm for Case 1 uses $\tilde{\mathcal{O}}(kd \log^k d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character.

4.3.2 Extension to Case 2 and wrapping up

Consider now Case 2. Note first that the $2k$-period of a string $P_i[|P_i| - |\tau_i'|, |P_i|]$, which is τ_i' extended by one character, must be at least d, and therefore by Observation 14 there can be at most one k-mismatch occurrence of $P_i[|P_i| - |\tau_i'|, |P_i|]$ per d positions of the text. We use the techniques of the algorithm for Case 1 to retrieve the occurrences of $P_i[|P_i| - |\tau_i'|, |P_i|]$, and then use the techniques of the algorithm for patterns with large periods (Lemma 16) to extend the retrieved occurrences.

In more detail, consider a position r of the text. As before, let $j \cdot d$ be the largest multiple of d that is smaller than r and L be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ with the $2k$-period at most d. Let now L' be the suffix L extended by one character to the left, i.e. $L' = T[j \cdot d - |L|, j \cdot d]$. By definition, the $(2k+1)$-period of L' is at most $d-1$. Furthermore, similar to Lemma 18, we can show that any k-mismatch occurrence of $P_i[|P_i| - |\tau_i'|, |P_i|]$ ending at the position r must be fully contained in $L'T[j \cdot d + 1, r]$.

Similarly to the previous section, we can maintain L' and the associated data structure D' using $\tilde{\mathcal{O}}(k^2 d)$ space and $\tilde{\mathcal{O}}(k^2 d)$ time per character. Using D', we can compute the $4k$-mismatch (and therefore k-mismatch) sketch of any suffix of $L'T[j \cdot d + 1, r]$ in $\tilde{\mathcal{O}}(k)$ time and hence we can find the occurrences of $P_i[|P_i| - |\tau_i'|, |P_i|]$ using the k-errata tree in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time per character. We now need to decide which of the found occurrences can be extended into full occurrences of P_i. In order to do this, we run the algorithm of Section 4.1. When we find an occurrence of $P_i[|P_i| - |\tau_i'|, |P_i|]$, we test it in $\tilde{\mathcal{O}}(k^2)$ time.

In total, the algorithm for Case 2 uses $\tilde{\mathcal{O}}(kd \log^k d + k^2 d)$ space and $\tilde{\mathcal{O}}(k \log^k d + occ)$ amortised time per character. Lemma 17 and Theorem 12 follow.

5 Proof of Theorem 1 – de-amortisation

Recall that the streaming algorithm of Theorem 12 is comprised of the algorithms of Lemma 16 and of Lemma 17 ran in parallel. Below we explain how to de-amortise these two algorithms. We use a standard approach called the *tail trick* that was already used in [9–11].

5.1 De-amortised algorithm with a delay

First, note that there is an easy way to de-amortise the algorithm of Lemma 16 if we allow delaying the occurrences by d characters. In order to do that, we divide the text into non-overlapping blocks of length d, and de-amortise the processing time of a block over the next block, by running $\tilde{\Theta}(k + \log d)$ steps of the computation per character. We will need to memorize the occurrences that end at the last $2d$ positions of the text, but this requires only $\mathcal{O}(d)$ space and we can afford it.

We now show how to de-amortise the algorithm for Case 1 of Lemma 17. This time, we will not need the delay. The only step of the algorithm that requires de-amortisation is updating L and D. We can de-amortise this step in a standard way. Namely, we de-amortise the time we need for an update by running $\tilde{\Theta}(k \log d)$ steps of the computation per each of the next d characters of text. We also maintain the sketches of the $2d$ longest prefixes of the text in a round-robin fashion using $\tilde{\mathcal{O}}(kd)$ space and $\mathcal{O}(k)$ time. If we need to extract the

sketch of some suffix of L before the update is finished, we use the previous version of the data structure and the sketches of the $2d$ latest suffixes of the text to compute the required values using Lemma 8.

Finally, we show how to de-amortise the algorithm of Case 2 of Lemma 17, again with a delay of d characters. Recall that this algorithm first finds the k-mismatch occurrences of the suffixes $P_i[|P_i| - |\tau_i'|, |P_i|]$ using an algorithm similar to the algorithm for Case 1 of Lemma 17, which can be de-amortised with no delay as explained above, and then tests these occurrences using the algorithm of Section 4.1, which can be de-amortised with a delay of d characters. Importantly, there are at most d occurrences that need to be tested per d characters, so we can memorize them until we can test them. The claim follows.

5.2 Removing the delay

We now show how to remove the delay. Recall that we assume the patterns to have lengths larger than $3d$. We partition each pattern $P_i = H_i\, Q_i$, where Q_i is the suffix of P_i of length d, and H_i is the remaining prefix. The idea is to find occurrences of the prefixes H_i and of the suffixes Q_i independently, and then to see which of them form an occurrence of P_i.

As above, we have three possible cases: the k-period of $H_i[|H_i| - 2d + 1, |H_i|]$ is larger than d; the k-period of H_i is at most d; the k-period of H_i is larger than d but the k-period of $H_i[|H_i| - 2d + 1, |H_i|]$ is at most d.

In the second case, we do not need to change much. For the current position r of the text we consider the largest $j \cdot d$ such that $r - j \cdot d \geq d$ and define L to be the longest suffix of $T[j \cdot d - m + 1, j \cdot d]$ such that its $2k$-period is at most d. We store the k-errata tree on the reverses of $P_i = H_i\, Q_i$ and run the de-amortised algorithm described in the previous section that maintains the suffix L. Any k-mismatch occurrence of a pattern P_i is fully contained in the concatenation of L and a suffix of the text of length $3d$, and therefore we can find all such occurrences using the k-errata tree as above.

We now explain how we remove the delay in the first and third cases. To find the occurrences of Q_i we use the streaming algorithm of Lemma 9. To find the occurrences of H_i we use the de-amortised version of the algorithm of Lemma 16 or of Lemma 17, as appropriately, that report the occurrences with a delay of at most d characters. It means that at the time when we find an occurrence of Q_i, the corresponding occurrence of H_i is already reported, so it is easy to check whether they form an occurrence of P_i. The only technicality is that we need to store the occurrences of H_i that we found while processing the last d characters of the text.

To this end, we use a dynamic hashing scheme [15]. The scheme allows to store a dynamic dictionary in linear space and with high probability guarantees constant look-up and update times. The answers to the look-up queries are always correct. Note that we can modify the data structure slightly to have constant time per operation if we allow the answers to be correct only with high probability (which we can afford), namely, if an operation takes too much time, we can simply abandon it.

We use the scheme for each of the last d positions of the text. Namely, consider a position p of the text and suppose that we found a set of k-mismatch occurrences of the prefixes H_i that end at p. Consider one of the prefixes, H_i, and let the Hamming distance between a prefix H_i and the text be $h \leq k$. Recall that by Fact 23 there are $\mathcal{O}(\log^k d)$ nodes of the k-errata tree labelled by Q_i. For each such node u of the k-errata tree, we insert a pair (u, h) into the dictionary. In case we insert a pair (u, h) several times for different prefixes H_i's, we associate (u, h) with the set of such prefixes. Note that at any moment the total size of the dictionaries is $\tilde{\mathcal{O}}(d \log^k d)$ as each of the patterns H_i has at most one k-mismatch occurrence over each d consecutive positions of the text.

Suppose we are at a position p of the text and we have run a dictionary look-up query and found the $\mathcal{O}(\log^k d)$ nodes in the tries of the k-errata tree corresponding to the suffixes Q_i that occur at this position with at most k mismatches. For each such node u we know the Hamming distance h' between the occurrences and the text. We then go to the dictionary at the position $(p-2d)$ and look up pairs $(u, k-h'), (u, k-h'-1), \ldots, (u, 0)$. If they are in the dictionary, we report all H_i's associated with these pairs. This step takes $\mathcal{O}(k \log^k d + occ)$ time.

6 Proof of Lemma 3 – space lower bound

In the communication complexity setting the Index problem is stated as follows. We assume that there are two players, Alice and Bob. Alice holds a binary string of length n, and Bob holds an index i encoded in binary. In a one-round protocol, Alice sends Bob a single message (depending on her input and on her random coin flips) and Bob must compute the i-th bit of Alice's input using her message and his random coin flips correctly with probability $> 2/3$. The length of Alice's message (in bits) is called the randomised one-way communication complexity of the problem. The randomised one-way communication complexity of the Index problem is $\Omega(n)$ [23].

Given a streaming algorithm for dictionary matching with k mismatches, we can construct a randomised one-way communication complexity protocol for the Index problem as follows. As above, let d be the size of the dictionary, and assume that $n = kd$. Split Alice's string into d blocks of length k. Let $\#, \$, \$_1, \ldots, \$_d$ be distinct characters different from $\{0, 1\}$. For the j-th block B_j create a string $P_j = (\$_j)^{k+1} \# B_j$, where $(\$_j)^{k+1}$ means that we repeat the character $\$_j$ $(k+1)$ times. For Bob's input $i = k \cdot q + r$ we create a string T which is equal to $(\$_q)^{k+1}$ concatenated with a string of length $k+1$ obtained from $\$^{k+1}$ by changing the $(r+1)$-th bit to 0. A streaming dictionary matching with k mismatches for the set of patterns P_i and T will output a k-mismatch occurrence of B_q at the position $2k+2$ of the text iff the r-th bit of Alice's input is equal to 0. Therefore, if Alice preprocesses P_j as in the streaming algorithm and sends the result to Bob, Bob will be able to continue to run the streaming algorithm on T to decide the i-th bit of Alice's input. Therefore, the lower bound for communication complexity of the Index problem is a space lower bound for any streaming algorithm for dictionary matching with mismatches. Lemma 3 follows.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communincations of the ACM*, 18(6):333–340, June 1975. doi:10.1145/360825.360855.

2 Ricardo Baeza-Yates and Gonzalo Navarro. Multiple approximate string matching. In *Proc. of the 5th Workshop on Algorithms and Data Structures*, pages 174–184, 1997. doi:10.1007/3-540-63307-3_57.

3 Djamal Belazzougui. Succinct Dictionary Matching with No Slowdown. In *Proc. of the 21st Annual Symposium on Combinatorial Pattern Matching*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

4 Djamal Belazzougui. Worst-case efficient single and multiple string matching on packed texts in the word-RAM model. *Journal of Discrete Algorithms*, 14:91–106, 2012. doi:10.1007/978-3-642-19222-7_10.

5 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone Minimal Perfect Hashing: Searching a Sorted Table with $O(1)$ Accesses. In *Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 785–794, 2009. doi:10.1137/1.9781611973068.86.

6 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic Z-Fast Tries. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval*, pages 159–172, 2010. doi:10.1007/978-3-642-16321-0_15.

7 Djamal Belazzougui and Mathieu Raffinot. Average Optimal String Matching in Packed Strings. In *Proc. of the 8th International Conference on Algorithms and Complexity*, pages 37–48, 2013. doi:10.1007/978-3-642-38233-8_4.

8 Dany Breslauer and Zvi Galil. Real-Time Streaming String-Matching. *ACM Transactions on Algorithms*, 10(4):22:1–22:12, August 2014. doi:10.1145/2635814.

9 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary Matching in a Stream. In *Proc. of the 23rd Annual European Symposium on Algorithms*, pages 361–372, 2015. doi:10.1007/978-3-662-48350-3_31.

10 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In *Proc. of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2039–2052, 2016. doi:10.1137/1.9781611974331.ch142.

11 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proc. of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1125, 2019. doi:10.1137/1.9781611975482.68.

12 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. of the 36th Annual ACM Symposium on Theory of Computing*, pages 91–100, 2004. doi:10.1145/1007352.1007374.

13 Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proc. of the 6th International Colloquium on Automata, Languages and Programming*, pages 118–132, 1979. doi:10.1007/3-540-09510-1_10.

14 Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3):107–113, 1999. doi:10.1016/S0020-0190(99)00092-7.

15 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. *Dynamic Hashing in Real Time*, pages 95–119. Vieweg+Teubner Verlag, Wiesbaden, 1992. doi:10.1007/978-3-322-95233-2_7.

16 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via Small-Space Multiple-Pattern Matching. In *Proc. of the 23rd European Symposium on Algorithms*, pages 533–544, 2015. doi:10.1007/978-3-662-48350-3_45.

17 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards Optimal Approximate Streaming Pattern Matching by Matching Multiple Patterns in Multiple Streams. In *Proc. of the 45th International Colloquium on Automata, Languages, and Programming*, pages 65:1–65:16, 2018. doi:10.4230/LIPIcs.ICALP.2018.65.

18 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In *Proc. of the 25th Annual European Symposium on Algorithms*, volume 87, pages 41:1–41:15, 2017. doi:10.4230/LIPIcs.ESA.2017.41.

19 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster Compressed Dictionary Matching. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval*, pages 191–200, 2010. doi:10.1007/978-3-642-16321-0_19.

20 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987. doi:10.1147/rd.312.0249.

21 Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In *Proc. of the 27th Annual Symposium on Combinatorial Pattern Matching*, volume 54, pages 6:1–6:13, 2016. doi:10.4230/LIPIcs.CPM.2016.6.

22 Dmitry Kosolobov and Nikita Sivukhin. Compressed Multiple Pattern Matching. *CoRR*, abs/1811.01248, 2018. arXiv:1811.01248.

23 Ilan Kremer, Noam Nisan, and Dana Ron. On Randomized One-round Communication Complexity. In *Proc. of the 27th Annual ACM Symposium on Theory of Computing*, pages 596–605, 1995. doi:10.1007/s000370050018.

24 Robert Muth and Udi Manber. Approximate multiple string search. In *Proc. of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 75–86, 1996. doi:10.1007/3-540-61258-0_7.

25 Gonzalo Navarro. Multiple Approximate String Matching by Counting. In *Proc. of the 4th South American Workshop on String Processing*, pages 95–111, 1997.

26 Benny Porat and Ely Porat. Exact And Approximate Pattern Matching In The Streaming Model. In *Proc. of the 50th Annual Symposium on Foundations of Computer Science*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

27 Sun Wu and Udi Manber. Agrep - A Fast Approximate Pattern-Matching Tool. In *Proc. of the USENIX Technical Conference*, pages 153–162, 1992.

A Proof of Lemma 11 – randomised k-errata tree

We will first remind the definition of the k-errata tree of Cole et al. [12], and then show a randomised implementation of this data structure.

A.1 Reminder: the k-errata tree

Consider a dictionary $\mathcal{D}$ of d patterns of maximal length m. We start with the compact trie $\mathcal{T}$ for the dictionary $\mathcal{D}$, and decompose it into heavy paths.

▶ **Definition 22.** *The heavy path of $\mathcal{T}$ is the path that starts at the root of $\mathcal{T}$ and at each node v on the path branches to the child with the largest number of leaves in its subtree (*heavy *child), with ties broken arbitrarily. The heavy path decomposition is defined recursively, namely, it is defined to be a union of the heavy path of $\mathcal{T}$ and the heavy path decompositions of the off-path subtrees of the heavy path.*

During the recursive step, we construct a number of new compact tries. For each heavy path H, and for each node $u \in H$ consider the off-path trees hanging from u. First, we create a *vertical substitution trie* for u. Let a be the first character on the edge $(u, v) \in H$. Consider an off-path tree hanging from u, and let $b \neq a$ be the first character on the edge from u to this tree. For each pattern in this off-path tree, we replace b by a. We consider a set of patterns obtained by such a substitution for all off-path trees hanging from u and build a new compact trie for this set. Next, we create *horizontal substitution tries* for the node u. We create a separate horizontal substitution trie for each off-path tree hanging from u. To do so, we take the patterns in it and cut off the first characters up to and including the first character on the edge from u to this tree, and then build a compact trie on the resulting set of patterns. To finish the recursive step we build the $(k-1)$-errata trees for each of the new vertical and horizontal tries.

From the construction, it follows that the k-errata tree is a set of compact tries, and each string S in the tries originates from a pattern in the dictionary $\mathcal{D}$. We mark the end of the path labelled by S by the id of the pattern it originates from.

Queries. A dictionary look-up with k mismatches for a string Q is performed in a recursive way as well. We will make use of a procedure called PrefixSearch. This procedure takes three arguments: a compact trie, a starting node u (or a position on an edge) in this trie, and a query string Q', and must output a pointer to the end of the longest path starting at u and labelled by a prefix of Q'. For the purposes of recursion, we introduce a mismatch credit –

the number of mismatches that we are still allowed to make. We start with the mismatch credit $\mu = k$. The algorithm first runs a PrefixSearch in the trie $\mathcal{T}$ for the query string Q starting from the root. If $\mu = 0$ and the path is labelled by Q, the algorithm returns the ids of the patterns in $\mathcal{D}$ that are associated with the end of the path. Otherwise, we consider the heavy paths $H_1, H_2, \ldots, H_j$ traversed by the PrefixSearch. Let u_i be the position where the PrefixSearch leaves the heavy path H_i, $1 \leq i \leq j$. Note that for $i < j$, u_i is necessarily a node of $\mathcal{T}$, and for $i = j$ it can be a position on an edge. We can divide all the patterns in $\mathcal{D}$ into four groups: (I) Patterns hanging off some node u in a heavy path H_i, where u is located above u_i, $1 \leq i \leq j$; (II) Patterns in the subtrees of u_i's children not in the heavy path H_{i+1}, for $1 \leq i < j$; (III) Patterns in the subtree of the position in H_j that is just below u_j; (IV) If u_j is a node, then patterns in the subtrees of u_j's children not in the heavy path H_j.

We process each of these groups of patterns independently. Consider a pattern P in group I, and let it hang from a node $u \in H_i$, where u is above u_i. Let ℓ be the length of the label of u, then Q and any pattern P in this subtree have a mismatch at the position $\ell + 1$. When creating vertical substitution tries, we removed this mismatch. Therefore, we can retrieve all such patterns that are at the Hamming distance $\leq k$ from Q by running the algorithm recursively with mismatch credit $\mu - 1$ in the $(k-1)$-errata tree that we created for the vertical substitution trie for the node u. The patterns of groups II and IV are processed in a similar way but using the $(k-1)$-errata trees for the horizontal substitution trees. Finally, to process the patterns of group III, we run the algorithm with mismatch credit $\mu - 1$ starting from the position that follows u_j in H_j.

This algorithm correctly retrieves the subset of the patterns in $\mathcal{D}$ that are at Hamming distance $\leq k$ from Q but can be slow as it makes many recursive calls. Cole et al. showed that the number of recursive calls can be reduced to logarithmic by introducing grouping on the substitution tries. In more detail, for each heavy path we consider its vertical substitution tries and build a weight-balanced tree, where the leaves of the weight-balanced tree are the vertical substitution tries, in the top-down order, and for each node of the tree, we create a new trie by merging the tries below it. For each of these group vertical substitution tries we build the $(k-1)$-errata tree. We group the horizontal substitution tries in a similar way, namely, we consider each node u and build a weight-balanced tree on the horizontal substitution tries that we created for the node u.

▶ **Fact 23** (Cole et al. [12]). *The id of any pattern in $\mathcal{D}$ occurs in the compact tries of the k-errata tree $\mathcal{O}(\log^k d)$ times, and as a corollary the total size of the tries is $\mathcal{O}(d \log^k d)$.*

To speed up the algorithm, we search a logarithmic number ($\mathcal{O}(\log d)$) of group substitution tries instead of searching each substitution trie individually. In total, we run $\mathcal{O}(\log^k d)$ PrefixSearch operations.

▶ Remark 24. We will use the k-errata tree to retrieve the patterns that are within Hamming distance k from the query string Q or from one of its prefixes. Recall that we mark each node of the k-errata tree corresponding to an end of a dictionary pattern. Furthermore, during the preprocessing step, we compute a pointer from each node to its nearest marked ancestor. At the end of each PrefixSearch we follow the pointers and retrieve the patterns corresponding to the marked nodes between the end and the start of the PrefixSearch. The number of the PrefixSearch operations that we perform does not change.

It remains to explain how we perform the PrefixSearch operations. Cole et al. gave a deterministic implementation of PrefixSearch that requires $\mathcal{O}(md)$ extra space and $\mathcal{O}(m)$ time of preprocessing, which is too much for our purposes. In the next section, we will show a randomised implementation of PrefixSearch which requires both less space and less time.

A.2 Randomised implementation of the k-errata tree

Recall from above that the k-errata tree is a collection of compact tries. In the randomised version of the k-errata tree, we replace each of them with a z-fast trie (see Fact 10). We also store the k-mismatch sketch of the label of every node of the tries, which requires $\tilde{\mathcal{O}}(kd \log^k d)$ space in total.

We now explain how we answer dictionary look-up with k mismatches. Recall that each dictionary look-up with k mismatches is a sequence of calls to the PrefixSearch procedure, and therefore it suffices to give an efficient implementation of PrefixSearch. We first explain how to implement this operation if it starts at the root of some compact trie of the k-errata tree. Assuming that we can retrieve the Karp–Rabin fingerprint of any substring of Q in $\mathcal{O}(1)$ time, Fact 10 immediately implies that a PrefixSearch starting at the root of a compact trie can be implemented in $\mathcal{O}(\log m)$ time. Note that if the end of the PrefixSearch is a position on an edge of the trie, then the functionality of the z-fast tries will allow us retrieving only the edge this position belongs to, but not the position itself. As we show below, it is sufficient for our purposes.

We now give an implementation of a PrefixSearch starting at an arbitrary position of a compact trie by reducing it first to a PrefixSearch that starts at a node of the trie and then to a PrefixSearch that starts at the root of the trie. We first show a reduction from a PrefixSearch that starts at an arbitrary position on an edge to a PrefixSearch that starts at a node. As we explained above, we might know the edge this starting position belongs to, but the position itself. However, from the description of the query algorithm in Section A.1 it follows that the algorithm will continue along the edge by running PrefixSearch operations until it either runs out of the mismatch credit or reaches the lower end of the edge. We will fast-forward to the lower end of the edge using the k-mismatch sketches. Namely, let Q' be the query string when we entered the current tree (note that we do not change the tree when retrieving patterns of group III). Importantly, the string Q' is a suffix of Q. We want to check whether we can reach the lower end of the edge and not run out of the mismatch credit. In other words, we want to compare the number of mismatches between the label S of the lower end of the edge and the prefix S' of Q' of length $|S|$, and the mismatch credit. We use the k-mismatch sketches for this task. We store the sketch of S, and the sketch of S' can be computed in $\tilde{\mathcal{O}}(k)$ time as it is a substring of Q. Having computed the sketches, we can compute the Hamming distance between S and S' using Lemma 7. If the Hamming distance is larger than the available mismatch credit, we stop, otherwise, we continue the PrefixSearch from the lower end of the edge.

Finally, we show an implementation of a PrefixSearch for a string Q' that starts at a node u of a trie. Let S be the label of u. Our task is equivalent to performing a PrefixSearch starting from the root of a trie for a string $S\,Q'$. Recall that Fact 10 assumes that we can extract the Karp–Rabin fingerprint of any prefix of $S\,Q'$. We do not know the Karp–Rabin fingerprints of the prefixes of $S\,Q'$, but we can compute them as follows. First, we use the k-mismatch sketches similar to above to compute the at most k mismatches that occurred on the way from the root of the trie to u. After having computed the mismatches, we can compute any of the fingerprints in $\tilde{\mathcal{O}}(k)$ time by taking the fingerprint of the corresponding substring of Q and "fixing" it in at most k positions.

So, we can answer a dictionary look-up with k mismatches query in $\tilde{\mathcal{O}}(k \log^k d + occ)$ time, and to compute the mismatches for each of the retrieved patterns in $\tilde{\mathcal{O}}(k)$ time per pattern if requested.

Quasi-Periodicity in Streams

Paweł Gawrychowski
University of Wrocław, 50-137 Wrocław, Poland
gawry@cs.uni.wroc.pl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, 02-097 Warsaw, Poland
jrad@mimuw.edu.pl

Tatiana Starikovskaya
DIENS, École normale supérieure, PSL Research University, 75005 Paris, France
tat.starikovskaya@gmail.com

Abstract

In this work, we show two streaming algorithms for computing the length of the shortest cover of a string of length n. We start by showing a two-pass algorithm that uses $\mathcal{O}(\log^2 n)$ space and then show a one-pass streaming algorithm that uses $\mathcal{O}(\sqrt{n \log n})$ space. Both algorithms run in near-linear time. The algorithms are randomized and compute the answer incorrectly with probability inverse-polynomial in n. We also show that there is no sublinear-space streaming algorithm for computing the length of the shortest seed of a string.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Streaming algorithms, quasi-periodicity, covers, seeds

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.22

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.

1 Introduction

One of the major tasks in processing data streams is trend analysis. In this work, we focus on a specific representative trend of streaming data, namely, *periodicity*. The motivation for analyzing the periodicity trend is that it can be used for detecting anomalies in streams, for example, in streams of financial data.

The study of periodicity in data streams was initiated by Ergün et al. in [20]. For a stream of length n, they showed a one-pass streaming algorithm to compute the length of the shortest period of the stream in polylogarithmic space assuming that the length of the period is at most $n/2$. On the other hand, they showed that there is no sublinear-space algorithm for computing periods of length larger than $n/2$. Motivated by real-life applications, where data streams are almost never exactly periodic, this work was later extended to allow approximate periods with mismatches and wildcards [18, 19].

We consider a different relaxation of the notion of periods, that of *quasi-periodicity*. Quasi-periodicity has been extensively studied in the RAM model of computation, starting from the early works of Apostolico and Ehrenfeucht [7], and by now is a well-established approach to detecting repetitive structure of a string when the classical definition of periodicity fails. There are two basic definition of quasi-periods that allow detecting different kinds of repetitive structure of a string: *covers* and *seeds*. Informally, a cover of a string T is a substring C of T such that every position of T lies within some occurrence of C. A string S is said to be a seed of T if S is a cover of some string containing T as a substring. The shortest cover and the shortest seed of a string can be computed in linear time; see [8] and [30, 31] (and

a much older $\mathcal{O}(n \log n)$-time algorithm [27]), respectively. Other works include computing all covers [37, 38] and shortest covers of all prefixes of a string [2, 11, 34]; left and right seeds being notions intermediate between covers and seeds [14, 16]; combinatorial studies on covers [3, 15]; computing approximate quasi-periodicities called enhanced covers [1, 22], partial covers [32, 36], partial seeds [33], approximate covers [4–6, 39], approximate seeds [13], and other variations [25, 35]; as well as quasi-periodicities that consist of multiple strings, called λ-covers, k-covers, and λ-seeds [17, 23, 24, 26, 41].

In this work, we commence a study of quasi-periodicity in streams. Recall that in the streaming model of computation, the input arrives as a stream, one character at a time, and we must account for all the space used, including the space used to store the input. We show two streaming algorithms for computing the shortest cover of a string of length n. We start by showing a two-pass algorithm that uses $\mathcal{O}(\log^2 n)$ space (Section 3, Theorem 12) and then show a one-pass streaming algorithm that uses $\mathcal{O}(\sqrt{n \log n})$ space (Section 4, Theorem 18). Both algorithms run in near-linear time. The algorithms are randomized and compute the answer incorrectly with probability inverse-polynomial in n. We also show that there is no sublinear-space one-pass streaming algorithm for computing the shortest seed of a string (Section 6).

2 Preliminaries

We assume that the characters of a string T are numbered 1 through $|T|$. By $T[i]$ we denote the i-th character of T and by $T[i, j]$ we denote $T[i] \dots T[j]$ which we call a substring of T. If $i = 1$, the substring $T[i, j]$ is called a prefix of T, and if $j = |T|$, a suffix of T. For strings T and X, we denote $\mathsf{Occ}_T(X) = \{i : T[i, i + |X| - 1] = X\}$.

2.1 Periods and quasi-periods

▶ **Definition 1** (Periods, borders). *We say that a positive integer p is a* period *of a string T if $T[i] = T[i + p]$ for all $i = 1, \dots, |T| - p$. By $\mathsf{per}(T)$ we denote the smallest period of T. The string T is called* periodic *if $2\,\mathsf{per}(T) \le |T|$. We say that a string B is a border of T if B is both a prefix and a suffix of T.*

▶ **Lemma 2** (Fine and Wilf's periodicity lemma [21]). *If a string Q has two periods p and q and $p + q \le |Q|$, then Q also has a period $\gcd(p, q)$.*

Let $\mathsf{Occ}_T(X) = \{a_1, \dots, a_m\}$ such that $a_1 < \dots < a_m$. We say that $a_j, \dots, a_k$, for $1 \le j \le k \le m$, form a *chain* of occurrences of X in T if for every $i = j + 1, \dots, k$, we have $a_i - a_{i-1} \le \frac{|X|}{2}$. A chain is called *maximal* if $k = m$ or $a_{k+1} - a_k > \frac{|X|}{2}$.

▶ **Corollary 3** (of Lemma 2). *Every chain of occurrences of X in T is an arithmetic progression with difference $\mathsf{per}(X)$.*

Proof. Assume to the contrary that $a_i - a_{i-1} \neq \mathsf{per}(X) = d$ for some chain $a_j, \dots, a_k$ and $j < i \le k$. Then $a_i - a_{i-1}$ is a period of X. By the periodicity lemma, it is a multiple of d. Hence, $a_{i-1} + d \in \mathsf{Occ}_T(X)$, a contradiction. ◀

For a set of integers $A = \{a_1, \dots, a_m\}$, $a_1 < a_2 < \dots < a_m$, by $\mathsf{maxgap}(A)$ we denote the maximum distance between consecutive elements of A: $\mathsf{maxgap}(A) = \max\{a_i - a_{i-1} : i = 2, \dots, m\}$.

▶ **Definition 4** (Covers, seeds). *A string C is a* cover *of a string T if*

$$\mathsf{maxgap}(\mathsf{Occ}_T(C) \cup \{-|C|+1, |T|+1\}) = |C|.$$

We say that a string C is a seed *of T if $|C| \leq |T|$ and C is a cover of some string containing T as a substring.*

▶ **Example 5.** *aabaa* is the shortest cover and *aba* is a shortest seed of *aabaabaa*.

▶ **Observation 6.** *Any cover of a string T is a border of T. Moreover, the shortest cover of T is not periodic.*

2.2 Reminder: Streaming pattern matching

▶ **Definition 7** (Rabin–Karp fingerprint [28, 40]). *The Rabin–Karp fingerprint of a string $X = X[1] \dots X[\ell]$ is defined as $\varphi(X) = (\sum_{i=1}^{\ell} X[i] \cdot r^i) \bmod p$, where p is a prime and r is a random integer in $\mathbb{F}_p$. We assume that together with the Rabin–Karp fingerprint we store $r^\ell \bmod p$ and $r^{-\ell} \bmod p$.*

If we choose p to be large enough, then the collision probability of any two ℓ-length strings X and Y, where $\ell \leq n$, will be at most $1/n^{\mathcal{O}(1)}$ [28, 40]. We will also need the following fact which follows immediately from the definition.

▶ **Fact 8.** *Let X, Y be two strings and $Z = XY$ be their concatenation. From the Rabin–Karp fingerprints of two of the strings X, Y, Z one can compute the Rabin–Karp fingerprint of the third string in $\mathcal{O}(1)$ time using the formula:*

$$\varphi(Z) = (\varphi(X) + r^{|X|} \cdot \varphi(Y)) \bmod p.$$

▶ **Corollary 9.** *Let X be a string and $Z = X^\alpha$ be the α-th power of X for positive integer α, i.e. the concatenation of α copies of X. From the Rabin–Karp fingerprint of X one can compute the Rabin–Karp fingerprint of Z in $\mathcal{O}(\log \alpha)$ time.*

We recall the main idea of the streaming pattern matching algorithm by Porat and Porat [40] in a simplified form by Breslauer and Galil [12]. The algorithm takes as an input a pattern Q and a streaming text T of length n, and outputs all occurrences of Q in T. The algorithm is randomized and can output an incorrect answer with probability inverse-polynomial in n. It uses $\mathcal{O}(\log |Q|)$ space and takes $\mathcal{O}(\log |Q|)$ time per text character.

We assume that the algorithm receives the pattern first, in a form of a stream, and computes the Rabin–Karp fingerprints of Q and of the prefixes $Q[1, 2^j]$ for all j. During the main stage, the algorithm stores $\mathcal{O}(\log |Q|)$ levels of positions of T. Positions in level j are occurrences of $Q[1, 2^j]$ in the suffix of the current text T of length 2^{j+1}. The algorithm stores the Rabin–Karp fingerprints of the prefixes of T up to each of these positions. If there are at least 3 such positions at one level, then, by the periodicity lemma, all the positions form a chain (that is, a single arithmetic progression with difference $\mathsf{per}(Q[1, 2^j])$; see Corollary 3). This allows storing the aforementioned information very compactly, using only $\mathcal{O}(\log |Q|)$ space in total. Finally, the algorithm stores the Rabin–Karp fingerprint of the current text. When a new character $T[i]$ arrives, the algorithm considers the leftmost position ℓ_j in each level j. If $i - \ell_j + 1$ is smaller than 2^{j+1}, the algorithm does nothing. Otherwise if the fingerprints imply that ℓ_j is an occurrence of $Q[1, 2^{j+1}]$, the algorithm promotes it to the next level, and if ℓ_j is not an occurrence of $Q[1, 2^{j+1}]$, the algorithm discards it. When a position reaches the level $j = \lfloor \log |Q| \rfloor + 1$, which corresponds to the whole pattern Q, it is an occurrence of Q and the algorithm outputs it. For more details, see [40].

The complexity of [40] has been later improved by [12] to use $\mathcal{O}(\log |Q|)$ space and $\mathcal{O}(1)$ time per character of the text. Given a prefix of a streaming text T, we can use either of the algorithms to find all occurrences of the prefix in T.

3 Two-pass algorithm for shortest cover

In this section we give a two-pass streaming algorithm for computing the length of the shortest cover of a stream T of given length n. We start with the following simple observation.

▶ **Lemma 10.** *There is a streaming algorithm that checks if T ($|T| = n$) has a cover of length ℓ in $\mathcal{O}(\log \ell)$ space and $\mathcal{O}(1)$ time per character. The algorithm is randomized and has error probability inverse-polynomial in n.*

Proof. We apply the streaming pattern matching algorithm [12] to compute the subsequent elements of the set $A = \mathsf{Occ}_T(T[1, \ell])$. It suffices to check if

$$\mathsf{maxgap}(A \cup \{-\ell + 1, n + 1\}) \leq \ell. \qquad \blacktriangleleft$$

In the first pass of the algorithm, we identify $\mathcal{O}(\log n)$ candidates for the length of the shortest cover. In the second pass, we apply Lemma 10 to verify the candidates.

For a positive integer x, let $I(x) = [x, \frac{3}{2}x)$. Let us consider a sequence $x_1, x_2, \ldots, x_r$ of length $r = \mathcal{O}(\log n)$ such that $\{1, \ldots, n\} \subseteq \bigcup_{i=1}^{r} I(x_i)$. From now on let us focus on a single $x = x_i$. In the first pass, our goal is to find a candidate for the shortest cover of length in the interval $I(x)$. To this end, we run streaming pattern matching for $X = T[1, x]$ in T. We choose the candidate using the following algorithm:

Algorithm 1: Find-Candidate($\boldsymbol{x}$).

1. Find the maximal chain of occurrences of $X = T[1, x]$ in T that starts at position 1. Let d be its length.
2. Let $i \in \mathsf{Occ}_T(X)$ be the last position that starts a maximal chain of occurrences of length d. Choose $n - i + 1$ as the candidate.

If the candidate is outside the interval $I(x)$, we discard it.

▶ **Lemma 11.** *If the shortest cover C of T ($|T| = n$) has length $|C| \in I(x)$, then $|C| = n - i + 1$ where i is the last position that starts a maximal chain of occurrences of X of length d and d is the length of the maximal chain of occurrences of X that contains position 1.*

Proof. Let $a_1, \ldots, a_d$ be the maximal chain of occurrences of X in T that starts with $a_1 = 1$. Corollary 3 asserts that every chain is an arithmetic progression with difference $\mathsf{per}(X)$. Hence, by Observation 6, $|C| \geq a_d + x - 1$ ($|C| > a_d + x - 1$ if $d > 1$), so C contains the whole chain. Clearly, any occurrence of C at position j in T implies a chain of length at least d starting at j. Moreover, it is easy to show that in this case the maximal chain starting at position j has length exactly d. Indeed, assume to the contrary that it has length at least $d + 1$ and j' is its $(d + 1)$-th element. If $j' + x - j \leq |C|$, then this would imply that the chain starting at position 1 also has length at least $d + 1$. Otherwise, C would be periodic with period $\mathsf{per}(X)$, contradicting Observation 6.

Let $i' < i$ be two positions where a maximal chain of length d starts. We will show that i' cannot be the last occurrence of C. Then $i - i' > \frac{x}{2}$, so $n - i' + 1 > n - i + 1 + \frac{x}{2} \geq \frac{3}{2}x$. Hence, $n - i' + 1 \notin I(x)$. This shows that only the last position where a maximal chain of length d starts may be the last occurrence of the shortest cover of length in $I(x)$. $\blacktriangleleft$

▶ **Theorem 12.** *The length of the shortest cover of a string of length n can be computed by a two-pass streaming algorithm which uses $\mathcal{O}(\log^2 n)$ space and $\mathcal{O}(n \log n)$ time in each pass. The algorithm is randomized and has error probability inverse-polynomial in n.*

Proof. In the first pass we run Find-Candidate(x_i) for each $i = 1, \ldots, r$. The algorithm requires only $\mathcal{O}(1)$ time and space in addition to the streaming pattern matching algorithm [12] for pattern $X_i = T[1, x_i]$ in text T. Indeed, at each moment it suffices to store the rightmost inclusion-maximal chain of occurrences of X_i, represented in $\mathcal{O}(1)$ space as an arithmetic sequence. We compute the length d of the first chain. Afterwards, when a chain ends, we can compute the d-th element from its end (if any) and store it until the next such element is encountered.

This produces at most $r = \mathcal{O}(\log n)$ candidates for the length of the shortest cover. In the second pass, we use Lemma 10 to verify them. The complexities of the algorithm follow. The error probability follows from the union bound. ◀

4 One-pass algorithm for shortest cover

In this section we give a one-pass streaming algorithm for computing the length of the shortest cover of a stream T of given length n. It consists of two processes that we run in parallel. The first one finds the length of the shortest cover if it is at most $\sqrt{n \log n}$ and the second if it is greater than $\sqrt{n \log n}$.

4.1 Covers of small length

Our first algorithm computes the length of the shortest cover if it is at most $\sqrt{n \log n}$. It is based on the online algorithm of Breslauer [11] which we briefly recall below.

▶ **Definition 13** (Super-primitivity). *A string is called* super-primitive *if it is equal to its shortest cover.*

The algorithm makes use of three arrays B, C, R, where $B[i]$ is the length of the longest proper border of the prefix $T[1, i]$, $C[i]$ is the length of the shortest cover of $T[1, i]$, and $R[i]$ is not defined if $T[1, i]$ is not super-primitive and otherwise stores the length of the longest prefix of T, up to the latest arrived character $T[k]$, such that $T[1, i]$ is its cover. In the end, $C[n]$ contains the length of the shortest cover of the text.

The algorithm applies the Knuth–Morris–Pratt algorithm [29] that computes B in $\mathcal{O}(n)$ space and time.

Algorithm 2: Compute-Shortest-Cover.

1. **for** $k = 1$ **to** n **do**
   - $B[k]$ = the length of the longest proper border of $T[1, k]$
   - **if** $B[k] > 0$ and $R[C[B[k]]] \geq k - C[B[k]]$ **then**
     #If the shortest cover of $B[k]$ covers $T[1, k]$,
     #then it is the shortest cover of $T[1, k]$.
     - $C[k] = C[B[k]]$; $R[C[k]] = k$
   - **else** #If $B[k] = 0$ or $C[B[k]]$ does not cover $T[1, k]$, then $T[1, k]$ is
     #super-primitive.
     - $C[k] = k$; $R[k] = k$

We now explain our streaming algorithm that computes the length of the shortest cover if it is at most $\sqrt{n \log n}$. Similar to the algorithm of Breslauer, it uses three arrays B', C', and R'. $B'[i]$ is defined to be the largest $0 \leq i' \leq \sqrt{n \log n}$, $i' < i$, such that $T[1, i]$ has a border of length i', $C'[i]$ is the length of the shortest cover of $T[1, i]$ if it is at most $\sqrt{n \log n}$ (and otherwise undefined), and $R'[i]$, for $1 \leq i \leq \sqrt{n \log n}$ is not defined if $T[1, i]$ is not super-primitive and otherwise stores the length of the longest prefix of T, up to the latest arrived character $T[k]$, such that $T[1, i]$ is its cover.

Our algorithm proceeds exactly as the algorithm of Breslauer except that it uses C', R', B' instead of C, R, B. To compute B', we use the following straightforward corollary of [29].

▶ **Corollary 14** (of Knuth, Morris, Pratt [29]). *There is a (deterministic) streaming algorithm that computes B' using $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n)$ time.*

Proof. We take as a basis the Knuth–Morris–Pratt algorithm. We then note that to compute $B'[k]$ it suffices to know only $B'[k-1]$ and the first $\sqrt{n \log n}$ entries of the array B'. The complexities follow. ◀

Let $T[k]$ be the last arrived character of T. The algorithm first computes $B'[k]$. If $B'[k] > 0$, it checks if the shortest cover of $T[1, B'[k]]$ covers $T[1, k]$ (using the condition $R'[C'[B'[k]]] \geq k - C'[B'[k]]$) and, if so, sets $C'[k] = C'[B'[k]]$ and $R'[C'[k]] = k$. Otherwise, if $k \leq \sqrt{n \log n}$, it sets $C'[k] = R'[k] = k$, else it leaves $C'[k]$ undefined. The final answer is $C'[n]$.

The algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n)$ time in total. We now argue that the algorithm is correct. From Observation 6 it follows that any cover of $T[1, k]$ must be its border. That is, the only possible candidates for the shortest cover of $T[1, k]$ that have length $\leq \sqrt{n \log n}$ are the borders of $T[1, k]$ of length $\leq \sqrt{n \log n}$. Correctness of our algorithm for the case $B'[k] = 0$ follows. For the case $B'[k] > 0$ we use the following claim:

▶ **Fact 15** (Breslauer [11]). *If a string W is a cover of a string Z, and another string V, such that $|W| \leq |V|$, is a border of Z, then W is a cover of V. Hence, if a string Z has two covers W and V, such that $|W| \leq |V|$, then W is a cover of V.*

It follows that if the length of the shortest cover of $T[1, k]$ is at most $\sqrt{n \log n}$, then it is the length of the shortest cover of $T[1, B'[k]]$, which concludes the proof.

4.2 Covers of large length

We now give an algorithm that computes the length of the shortest cover of T if it is larger than $\sqrt{n \log n}$. Let $x_1, \ldots, x_r$ and $I(x)$ be defined as in Section 3. For all $x_{i+1} > \sqrt{n \log n}$, we seek for the shortest cover of length in $I(x_i)$ (if any). From now on we focus on a single $x = x_i$. Let $X = T[1, x]$.

Let a be the last element of the maximal chain of occurrences of X in T that starts at position 1 and $x' = a + x - 1$, $X' = T[1, x']$. We obtain the following lemma as a corollary of Lemma 11.

▶ **Lemma 16.** *If the shortest cover C of T has length $|C| \in I(x)$, then* $\max \mathsf{Occ}_T(C) = \max \mathsf{Occ}_T(X')$.

Proof. It suffices to note that the last position that starts a maximal chain of occurrences of X of length d is exactly the last occurrence of X'. Indeed, any occurrence of X' starts a chain of occurrences of X of length at least d (and conversely). If the length of the chain was greater than d, then X' would also occur $\mathsf{per}(X)$ positions later. ◀

In the beginning, we compute the maximal chain of occurrences of X in T that starts at position 1 and compute x' and $\varphi(X')$. We then continue with streaming pattern matching to find all occurrences of X' in T. We note that we can do the preprocessing for the streaming pattern matching algorithm simultaneously with computing X'.

One small technical difficulty here is that x' is known only once we know that the initial chain of occurrences of X does not extend, which can take place at position $x' + \frac{x}{2}$. To overcome this, whenever a new occurrence of X in the chain is found, say at position j, we assume that $x' = j + x - 1$ and start the pattern matching algorithm for $X' = T[1, x']$. If the next occurrence in the same chain is found, x' is overwritten. Note that the pattern matching algorithm for X' always looks for occurrences of prefixes of X' of lengths being powers of two, so it can be easily updated when the chain is extended.

We use the following key observation.

▶ **Observation 17.** *For every maximal chain of occurrences of X' in T, only the last position in it can be an occurrence of the shortest cover C.*

X' can be periodic and there can be many occurrences of X' in T, but by Observation 17, only the last position in every maximal chain can be an occurrence of C. Occurrences that satisfy this condition are henceforth called *relevant*. We maintain a stack of relevant occurrences (the topmost one can be non-relevant). When a new occurrence p of X' arrives, we check if it is in the same maximal chain at the occurrence in the top of the stack. If it is, we first pop from the stack, and then push p to the stack (in other words, we replace the occurrence in the top of the stack with p). Otherwise, we simply push p to the stack.

Suppose that in the end, the stack contains occurrences $p_1, p_2, \ldots, p_k$. Let $l_j = p_{j+1} - p_j$. Note that $l_j > \frac{x}{2}$ for every $j = 1, \ldots, k-1$, as the relevant occurrences are in different maximal chains. Therefore, there are $\mathcal{O}(n/x) = \mathcal{O}(\sqrt{n/\log n})$ occurrences in total. For each p_j, we memorize the fingerprint $\varphi(T[1, p_j - 1])$. By Lemma 16, there is only one possible candidate for the shortest cover in $I(x)$, the suffix C of T that starts at p_k. Then $\varphi(C)$ can be computed from $\varphi(T)$ and $\varphi(T[1, p_k - 1])$ via Fact 8.

If we have a way to test whether p_j is a starting position of an occurrence of C, then we can check if C is a cover of T using the maxgap. We now explain how we test p_j. If any of the differences l_j satisfies $l_j > |C|$, we immediately return NO. If $l_j = |C|$, we can check whether p_j is a starting position of an occurrence of C by computing $\varphi(T[p_j, p_{j+1} - 1])$ and comparing it with $\varphi(C)$. From now on, we focus on j such that $l_j < |C|$. From $|C| < \frac{3}{2}x$, it follows that $|C| - l_j \le x'$ and therefore $T[p_{j+1}, p_{j+1} + |C| - l_j - 1] = T[1, |C| - l_j]$. Consequently, if we know $\varphi(T[1, |C| - l_j])$, we can check whether p_j is a starting position of an occurrence of C in $\mathcal{O}(1)$ time in three steps: first, compute $\varphi(T[p_j, p_{j+1} - 1]) = \varphi(T[p_j, p_j + l_j - 1])$, second, compute $\varphi(T[p_j, p_j + |C| - 1])$ from $\varphi(T[p_j, p_j + l_j - 1])$ and $\varphi(T[1, |C| - l_j])$ via Fact 8, and finally, compare $\varphi(T[p_j, p_j + |C| - 1])$ and $\varphi(C)$. If the fingerprints are equal, p_j is an occurrence of C with high probability.

We compute the fingerprints $\varphi(T[1, |C| - l_j])$ as follows. Assume that we know the fingerprints $\varphi(T[1, n - l_j])$ and recall that C starts at p_k. By Fact 8, we can compute $\varphi(T[1, |C| - l_j])$ for all $j = 1, 2, \ldots, k-1$ from $\varphi(T[1, p_k - 1])$. Namely, we compute $\varphi(T[p_k, n - l_j]) = \varphi(T[1, n - l_j - p_k + 1])$, and

$$n - l_j - p_k + 1 = n - p_k + 1 - l_j = |C| - l_j.$$

We now explain how we compute the fingerprints $\varphi(T[1, n - l_j])$. First, consider all j such that $n - l_j \ge p_{j+1} + x' - 1$. Note that since $l_j < |C| < \frac{3}{2}x$ and the distance between any two consecutive positions is greater than $\frac{x}{2}$, this inequality holds for all $j \le k - 4$:

$$n - l_j > n - \tfrac{3}{2}x \ge p_k + x' - 1 - \tfrac{3}{2}x > p_{j+1} + (k - j - 1)\tfrac{x}{2} + x' - 1 - \tfrac{3}{2}x \ge p_{j+1} + x' - 1.$$

We maintain a balanced BST of all l_j. Say that the streaming pattern matching algorithm reports a new occurrence p of X' (which happens when $T[p + x' - 1]$ arrives). If the previous occurrence p_j was more than $\frac{x}{2}$ positions earlier, we assume for now that $p_{j+1} = p$, compute $l_j = p_{j+1} - p_j$ and insert it to the BST. If the previous position p_j was at most $\frac{x}{2}$ positions earlier, it is popped from the stack, we assume that $p_j = p$ and l_{j-1} is recomputed. Furthermore, when a new position i of the text arrives, we check whether the BST contains $l_j = n - i$ and, if so, memorize the fingerprint of $T[1, i] = T[1, n - l_j]$. The algorithm will indeed compute $\varphi(T[1, n - l_j])$ for all j such that $n - l_j \geq p_{j+1} + x' - 1$.

It remains to compute $\varphi(T[1, n - l_j])$ for all $j \leq k - 1$ such that $n - l_j < p_{j+1} + x' - 1$. As explained above, this inequality can hold only for $j \in \{k - 3, k - 2, k - 1\}$. We note that, additionally, we have $p_{j+1} \leq n - l_j$. Indeed, from $l_j < |C|$, we have

$$n - l_j \geq n - |C| + 1 = p_k \geq p_{j+1}.$$

For all such j we use a different subroutine. The subroutine is initialised at the position $p_j + x' - 1$ and must output $\varphi(T[1, n - l_j])$ at the position $p_{j+1} + x' - 1$ if $p_{j+1} \leq n - l_j < p_{j+1} + x' - 1$. In Section 5 we will show such a subroutine that takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character of the text.

We use the subroutine in the following way. When we find a new occurrence p of X' such that $n < p + x' - 1 + 2 \cdot \frac{3}{2}x$, we initialize a new instance of the subroutine. If later we find out that p is not a relevant occurrence, we kill the process. Note that at any time, there will be a constant number of instances of the subroutine running (because the number of survived processes is equal to the number of occurrences p_j satisfying $n < p_j + x' - 1 + 3x$, and every two occurrences p_j are at least $\frac{x}{2}$ positions apart). Therefore, in total this step takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character.

In conclusion, we can compute $\varphi(T[1, n - l_j])$ for all $j \leq k - 1$ using $\mathcal{O}(\sqrt{n/\log n})$ space and $\mathcal{O}(\log n)$ time per character of the text.

The whole algorithm for a given x can be stated as below.

1. Perform streaming pattern matching for X in T. Compute x' and X'.
   #x' can be updated several times due to "guessing"
2. Start streaming pattern matching for X' in T:
   - Using a stack, compute subsequent relevant occurrences of X'. When occurrence p_{j+1} is found, store $\varphi(T[1, p_{j+1} - 1])$ and insert $l_j = p_{j+1} - p_j$ to a BST.
     #l_j can be updated several times due to "guessing" of p_{j+1}
   - When $T[i]$ is read and the BST contains $l_j = n - i$, memorize the fingerprint of $T[1, i] = T[1, n - l_j]$.
   - For every occurrence p_j that satisfies $n < p_j + x' - 1 + 3x$, start the subroutine of Section 5 to compute $T[1, n - l_j]$.
     #If the occurrence turns out not to be relevant, kill the subroutine.
3. When the end of the text is reached:
   - If $l_j > |C|$ for any $j = 1, \ldots, k - 1$, where $C = T[p_k, n]$, return NO.
   - Compute $\varphi(C)$ from $\varphi(T[1, p_k - 1])$ and $\varphi(T)$.
   - Compute $\varphi(T[p_j, p_j + |C| - 1])$ for all $j = 1, \ldots, k - 1$ using $\varphi(T[p_j, p_{j+1} - 1])$ and $\varphi(T[1, n - l_j])$ (if $l_j < |C|$).
   - Let $P = \{p_j : \varphi(T[p_j, p_j + |C| - 1]) = \varphi(C)\}$. If $\mathsf{maxgap}(P \cup \{-|C| + 1, n + 1\}) = |C|$, return $|C|$. Otherwise, return NO.

The algorithm uses $\mathcal{O}(k + \log n) = \mathcal{O}(\sqrt{n/\log n})$ space and spends $\mathcal{O}(\log n)$ time per text character, apart from the last position of the text where it spends $\mathcal{O}(\sqrt{n/\log n})$ time. Over

all $x = x_i$ for $i = 1, \ldots, r$ such that $\frac{3}{2}x_i > \sqrt{n \log n}$, the algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^2 n)$ time in total. Combining this algorithm with the algorithm of Section 4.1, we arrive at the following result.

▶ **Theorem 18.** *The length of the shortest cover of a string of length n can be computed by a one-pass streaming algorithm which uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^2 n)$ time in total. The algorithm is randomized and has error probability inverse-polynomial in n.*

5 Computing the fingerprint of $T[1, n - l_j]$ for $j \in \{k-3, k-2, k-1\}$

We will show how to solve a more general problem.

▶ **Problem 1.** *Let T be a streaming text of a given length n and $y \leq n$ be a positive integer. For some position* start*, $y \leq \mathsf{start} \leq n$, when we have read $T[\mathsf{start}]$, we are given an integer z. Let $\mathsf{len}(q) = z - 2q$. For each q such that $T[q, q+y-1] = T[1, y]$, at the moment when we have read $T[q + y - 1]$, we are to report $\psi(q) = \varphi(T[q, q + \mathsf{len}(q)])$ provided that $\mathsf{start} \leq q + \mathsf{len}(q)$ and $0 \leq \mathsf{len}(q) < y$.*

If we take $\mathsf{start} = p_j + x' - 1$, $y = x'$, $z = n + p_j$, then for $q = p_{j+1}$ we have $q + \mathsf{len}(q) = z - q = n + p_j - p_{j+1} = n - l_j$. We also have $p_j + x' - 1 = \mathsf{start} \leq q + \mathsf{len}(q) = n - l_j$ since $p_j + x' - 1 + l_j = p_{j+1} + x' - 1 \leq n$. Finally, we require that $p_{j+1} \leq n - l_j < p_{j+1} + x' - 1$ which translates to $0 \leq \mathsf{len}(q) < y - 1$. Therefore, using an algorithm for Problem 1 we can compute $\varphi(T[p_{j+1}, n - l_j])$, and therefore, by Fact 8, $\varphi(T[1, n - l_j])$ from $\varphi(T[1, p_{j+1} - 1])$ that is stored by the exact pattern matching algorithm for X' and T.

We now show an algorithm for Problem 1 that takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character of the text. Denote $Y = T[1, y]$ and let

$$\mathsf{Occ}_T(j) = \{q \in \mathsf{Occ}_T(Y[1, 2^j]) \,|\, \mathsf{start} \leq q + \mathsf{len}(q) \text{ and } 2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}\}.$$

For $j = \lceil \log y \rceil$, we additionally assume that $\mathsf{len}(q) < y$. Note that the second condition in the above definition can be written equivalently as

$$\tfrac{z+1}{2} - 2^j < q \leq \tfrac{z+1}{2} - 2^{j-1}.$$

which concludes, in particular, that $\mathsf{Occ}_T(j)$ either contains at most one element or is a single chain of occurrences of $Y[1, 2^j]$.

▶ **Observation 19.** *The set of all $q \in \mathsf{Occ}_T(Y)$, $\mathsf{start} \leq q + \mathsf{len}(q)$, such that $0 \leq \mathsf{len}(q) < y$, is a subset of $\bigcup_{0 \leq j \leq \lfloor \log y \rfloor} \mathsf{Occ}_T(j)$.*

We will show how to compute and store a small amount of additional information for each set $\mathsf{Occ}_T(j)$ that we will use to retrieve the fingerprints $\psi(q)$ for $q \in \mathsf{Occ}_T(j) \cap \mathsf{Occ}_T(Y)$.

We will build our solution upon the streaming pattern matching algorithm for Y in T. Recall that the algorithm stores, for every $j = 0, \ldots, \lfloor \log y \rfloor$, a subset of $\mathsf{Occ}_T(Y[1, 2^j])$ that corresponds to occurrences in the suffix of length 2^{j+1} of $T[1, i]$. Let us denote this subset by $S_j(i)$. The set $S_j(i)$ is stored as either at most two single occurrences or a chain with period $\Delta_j = \mathsf{per}(Y[1, 2^j])$, so that one can check in $\mathcal{O}(1)$ time if $q \in S_j(i)$. Moreover, for every $q \in S_j(i)$, we can recover $\varphi(T[1, q - 1])$; we also store $\varphi(T[1, i])$.

For every position $i \geq \mathsf{start}$, we can compute the position q that satisfies $q + \mathsf{len}(q) = i$; we have $q = z - i$. Let j be such that $2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}$. If $q \notin S_j(i)$, then $q \notin \mathsf{Occ}_T(Y)$ and we can ignore it. Otherwise, at this point we can compute $\psi(q)$ using Fact 8. If we could

store $\psi(q)$ for every $q \in \mathsf{Occ}_T(j)$, then, when the streaming pattern matching algorithm outputs a position $q \in \mathsf{Occ}_T(Y)$, we could in $\mathcal{O}(\log n)$ time find j such that $q \in \mathsf{Occ}_T(j)$ and then output $\psi(q)$, as desired.

Unfortunately, if $|\mathsf{Occ}_T(j)|$ is large, we cannot afford to store $\psi(q)$ for every position $q \in \mathsf{Occ}_T(j)$. Moreover, at the time when q is processed, we actually do not know if q will be an occurrence of Y and only this information makes this value relevant. We will use periodicity to overcome these setbacks and design a small representation of values $\psi(q)$ of all potentially relevant elements $q \in \mathsf{Occ}_T(j)$; see Example 20 for an illustration.

For a given j, let us consider the first moment $i = i'$ (if any) when $q' = z - i' \in \mathsf{Occ}_T(j)$, for any $i \geq \mathsf{start}$. We then have $q' = \max \mathsf{Occ}_T(j)$. If $q \in \mathsf{Occ}_T(j)$, then $q + \mathsf{len}(q) \geq i'$ but $\mathsf{len}(q) < 2^{j+1}$, so $q \in S_j(i')$. In other words, $\mathsf{Occ}_T(j) \subseteq S_j(i')$. Let $\{q_1, \ldots, q_r\}$, with $q_1 < \ldots < q_r$, be the subsequence of consecutive elements of $S_j(i')$ such that q_1 is the smallest element of $S_j(i')$ and $q_r = q'$. There is an index $\ell \in \{1, \ldots, r\}$ such that $\mathsf{Occ}_T(j) = \{q_\ell, \ldots, q_r\}$. If $r - \ell + 1 \leq 2$, we will store $\psi(q_k)$ for every $k = \ell, \ldots, r$ explicitly. Otherwise, $q_\ell, \ldots, q_r$ is a chain of occurrences of $Y[1, 2^j]$ with step Δ_j.

In the beginning, the pattern matching algorithm computes the set $S_j(2^{j+1})$. Let d_j be the length of the longest chain of occurrences of $Y[1, 2^j]$ in $S_j(2^{j+1})$ that starts at position 1. We will store this value in the algorithm.

Let $i_k = z - q_k$. When i increases from i_r to i_ℓ, the algorithm computes $\psi(q_k)$ for subsequent $k = r, \ldots, \ell$. We will store $\psi(q_r)$ as well as $\psi(q_k)$ for the last element q_k that was considered. Let us now consider $i = i_k$ for $k < r$. Let $q_{r+1}, \ldots, q_{t_k}$ be the subsequent elements of the chain that contains q_r in $S_j(i_k)$. If $q_{t_k} + 2^j - 1 \geq i_{k+1}$, then $T[q_{k+1}, i_{k+1}]$ is periodic with period Δ_j. Hence, for every $k' > k$,

$$T[q_{k'}, i_{k'}] = T[1, \Delta_j]^{2(r-k')}\, T[q_r, i_r].$$

Thus $\psi(q_{k'})$ for any $k' > k$ can be computed from $\psi(q_r)$ (which is stored) and $\varphi(T[1, \Delta_j])$ (which can be computed from $\varphi(T[1, q_r - 1])$ and $\varphi(T[1, q_{r-1} - 1])$ using Fact 8) in $\mathcal{O}(\log n)$ time via Corollary 9.

As for the opposite case, let us consider the first $k < r$ for which $q_{t_k} + 2^j - 1 < i_{k+1}$. If it exists, let us denote this value of k by k_0. This means that the chain of occurrences of $Y[1, 2^j]$ that contains q_r definitely ends at position $q_{t_{k_0}}$. Then Y can occur at position q_p for $p \in \{\ell, \ldots, r\}$ only if $t_{k_0} - p + 1 = d_j$. If $p > k_0 + 1$, $\psi(q_p)$ can be restored as shown above. If $p = k_0 + 1$, $\psi(q_p)$ is still stored from the previous step. Otherwise, we will store $\psi(q_p)$ when the position i_p is processed.

If the position k_0 is not found, we also store $\psi(q_\ell)$ since $T[q_\ell, i_\ell]$ might not have period Δ_j.

▶ **Example 20.** Let us consider the setting from Fig. 1 with $j = 5$. Assume that z is such that $\mathsf{Occ}_T(j) = \{q_2, \ldots, q_6\}$ (i.e., $\ell = 2$ and $r = 6$) and $i_k = q_k + \mathsf{len}(q_k)$ for $k = 2, \ldots, 6$. Further assume that $\mathsf{start} \leq i_6$.

When $i = i_6$, only the occurrences $q_1, \ldots, q_6$ are known. At this moment we compute and store $\psi(q_6)$.

When $i = i_5$, only the occurrences $q_1, \ldots, q_7$ are known (i.e., $t_5 = 7$) and $T[q_6, i_6]$ has period Δ_j (since $q_7 + 2^j - 1 \geq i_6$), so $k_0 \neq 5$. At this moment we compute $\psi(q_5)$ and store it until the next step. We also compute and store $\varphi(T[1, \Delta_j])$.

When $i = i_4$, all the occurrences $q_1, \ldots, q_8$ are known (i.e., $t_4 = 8$) and $T[q_5, i_5]$ has period Δ_j (since $q_8 + 2^j - 1 \geq i_5$), so $k_0 \neq 4$. We compute $\psi(q_4)$ and store it until the next step.

When $i = i_3$, there is no new occurrence in the chain (i.e., $t_3 = 8$), so $q_8 + 2^j - 1 < i_4$ and $k_0 = 3$ (note that $T[q_4, i_4]$ still could have period Δ_j, if the character $\mathtt{x}$ of T was equal to $\mathtt{c}$; see Fig. 1). At this moment we know that, among $\psi(q_k)$ for $k = 2, \ldots, 6$, at most one

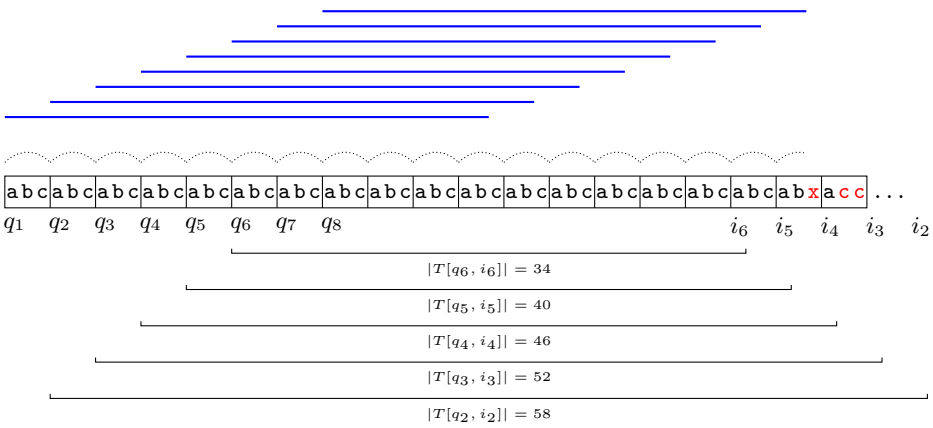


Figure 1 Let $j = 5$ and assume that $Y[1, 32] = (\mathtt{abc})^{10}\mathtt{ab}$ and $y \geq 64$. The positions $q_1, \ldots, q_8$ form a maximal chain of occurrences of $Y[1, 32]$ in T (the occurrences are depicted as blue lines) and the period $Y[1, \Delta_j]$ is shown by dotted arcs.

value can be relevant, for $8 - k + 1 = d_j$. If $k > 4$, this value can be computed from $\psi(q_6)$ and $\varphi(T[1, \Delta_j])$. Otherwise, it is either $\psi(q_4)$ that is still stored or one of the values $\psi(q_k)$ for $k \leq 3$ that will be computed in this or subsequent steps.

E.g., if Y starts with $(\mathtt{abc})^{14}\mathtt{abx}$, then $d_j = 5$ and we only need to store $\psi(q_4)$.

In conclusion, the algorithm for a given $j \in \{0, \ldots, \lfloor \log z \rfloor\}$ works as follows:

1. Compute d_j, the length of the longest chain of occurrences starting from position 1 in $S_j(2^{j+1})$.
2. Starting from position start, look for the first $i \geq \mathsf{start}$ such that $q = z - i \in \mathsf{Occ}_T(j)$. Let $q_1, \ldots, q_r$ be the maximal increasing subsequence of $S_j(i)$ that ends at $q = q_r$. Let q_ℓ be the minimal element in $\mathsf{Occ}_T(j)$ (it can be computed in $\mathcal{O}(1)$ time). Compute $\psi(q_r)$ using Fact 8 and store it. If $r = \ell$, no more data is stored.
3. For each subsequent position i of the text, if $i = i_k$ for some $k \in \{\ell, \ldots, r-1\}$, then:
   - Compute $\psi(q_k)$ using Fact 8 and store it until position i_{k-1}.
   - Assume that $k = r - 1$. If $k = \ell$, store this fingerprint. Otherwise, compute $\varphi(T[1, \Delta_j])$.
   - If $q_{t_k} + 2^j - 1 \geq i_{k+1}$, continue.
   - When the first $k < r$ for which $q_{t_k} + 2^j - 1 < i_{k+1}$ is encountered, $k_0 = k$, store only $\psi(q_p)$ for $p = tk_0 - d_j + 1$. If $p > k_0$, this value can be restored from the previous steps in $\mathcal{O}(\log n)$ time. Otherwise, continue the algorithm until i_p.
   - If there was no such k_0, store $\psi(q_\ell)$.

When $q \in \mathsf{Occ}_T(Y)$ is encountered, $\mathsf{start} \leq q + \mathsf{len}(q)$, and $2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}$, compute k such that $q = q_k$ and:

- If $r - \ell \leq 1$ or k_0 exists, return the stored value of $\psi(q_k)$ (for $k \in \{q, r\}$) in $\mathcal{O}(1)$ time;
- If k_0 does not exist, return the stored value if $k = \ell$ or compute $\psi(q_k)$ from $\psi(q_r)$ and $\varphi(T[1, \Delta_j])$ in $\mathcal{O}(\log n)$ time if $k > \ell$.

In total, for a fixed value of j, we use $\mathcal{O}(1)$ space and spend $\mathcal{O}(\log n)$ time per text character in the worst case. The value q such that $q + \mathsf{len}(q) = i$ is always unique, so at every position of the text, computations for only one index j take place. Overall, we spend $\mathcal{O}(\log n)$ time per text character. This completes the description of the algorithm for Problem 1.

6 Hardness of computation of seeds in a stream

While above we showed an $\mathcal{O}(\sqrt{n \log n})$-space one-pass streaming algorithm for computing the shortest cover, there is no sublinear-space one-pass streaming algorithm for computing the shortest seed. The proof follows the lines of the proof of the space lower bound for computing the shortest period of a stream by Ergün et al. [20].

Consider the communication game between Alice and Bob who hold two strings T_1 and T_2 of length $n/2$ each, where the goal is to compute the shortest seed of $T = T_1T_2$. By a standard reduction, the lower bound on the communication complexity for this problem is a space lower bound for any streaming algorithm computing the shortest seed of a string T of length n. We can show the lower bound of $\Omega(n)$ bits for the communication complexity of the problem by a reduction from the augmented indexing problem: Suppose Alice is given a string $X \in \{0,1\}^{n/2}$, and Bob is given an index $i \in [1, n/2]$ and a string $Y \in \{0,1\}^{i-1}$ such that $Y = X[1, i-1]$. Bob must decide whether $X[i] = 1$. The randomized communication complexity of this problem is $\Omega(n)$ bits [9, 10]. On the other hand, for $i < n/2$ we can reduce it to the problem of computing the shortest seed by taking $T_1 = X$ and $T_2 = \$^{n/2-i}\ Y\ 1$, where \$ is a special character different from $0, 1$. It is easy to see that the shortest seed of $T = T_1T_2$ is equal to $n - i$ iff $X[i] = 1$. Therefore, the communication game of computing the shortest seed requires $\Omega(n)$ bits of communication, and any streaming algorithm for computing the shortest seed of a string of length n requires $\Omega(n)$ bits of space as well.

7 Conclusion and open questions

In this work, we give the first sublinear-space streaming algorithms for computing the length of the shortest cover of a stream. Our two-pass streaming algorithm uses $\mathcal{O}(\log^2 n)$ space and $\mathcal{O}(n \log n)$ time, and our one-pass streaming algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^3 n)$ time. It is an interesting question if similar algorithms can be developed for computing the shortest covers of all prefixes of the stream, and if the complexity of the algorithms can be improved. We also state an open question concerning computation of seeds: Design a streaming algorithm with any number of passes and $\mathcal{O}(n^{1-\varepsilon})$ space, for $\varepsilon > 0$, for computing the shortest seed of a stream.

References

1 Ali Alatabbi, Abu Sayed Md. Sohidull Islam, Mohammad Sohel Rahman, Jamie Simpson, and William F. Smyth. Enhanced Covers of Regular and Indeterminate Strings Using Prefix Tables. *Journal of Automata, Languages and Combinatorics*, 21(3):131–147, 2016. doi:10.25596/jalc-2016-131.

2 Ali Alatabbi, M. Sohel Rahman, and William F. Smyth. Computing covers using prefix tables. *Discrete Applied Mathematics*, 212:2–9, 2016. doi:10.1016/j.dam.2015.05.019.

3 Amihood Amir, Costas S. Iliopoulos, and Jakub Radoszewski. Two strings at Hamming distance 1 cannot be both quasiperiodic. *Information Processing Letters*, 128:54–57, 2017. doi:10.1016/j.ipl.2017.08.005.

4 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can We Recover the Cover? In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 25:1–25:15, 2017. doi:10.4230/LIPIcs.CPM.2017.25.

5 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate Cover of Strings. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78, pages 26:1–26:14, 2017. doi:10.4230/LIPIcs.CPM.2017.26.

6 Amihood Amir, Avivit Levy, and Ely Porat. Quasi-Periodicity Under Mismatch Errors. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 4:1–4:15, 2018. doi:10.4230/LIPIcs.CPM.2018.4.

7 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

8 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

9 Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. The Sketching Complexity of Pattern Matching. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2004*, pages 261–272, 2004. doi:10.1007/978-3-540-27821-4_24.

10 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Information theory methods in communication complexity. In *Proceedings of the IEEE Annual Conference on Computational Complexity, CCC 2002*, pages 93–102, 2002. doi:10.1109/CCC.2002.1004344.

11 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

12 Dany Breslauer and Zvi Galil. Real-Time Streaming String-Matching. *ACM Transasctions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.

13 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *Journal of Automata, Languages and Combinatorics*, 10:609–626, 2005. doi:10.25596/jalc-2005-609.

14 Michalis Christou, Maxime Crochemore, Ondrej Guth, Costas S. Iliopoulos, and Solon P. Pissis. On left and right seeds of a string. *Journal of Discrete Algorithms*, 17:31–44, 2012. doi:10.1016/j.jda.2012.10.004.

15 Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. Quasiperiodicities in Fibonacci strings. *Ars Combinatoria*, 129:211–225, 2016.

16 Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Waleń. Efficient seed computation revisited. *Theoretical Computer Science*, 483:171–181, 2013. doi:10.1016/j.tcs.2011.12.078.

17 Richard Cole, Costas S. Iliopoulos, Manal Mohamed, William F. Smyth, and Lu Yang. The Complexity of the Minimum k-cover Problem. *Journal of Automata, Languages and Combinatorics*, 10(5–6):641–653, 2005. doi:10.25596/jalc-2005-641.

18 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming Periodicity with Mismatches. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2017*, pages 42:1–42:21, 2017. doi:10.4230/LIPIcs.APPROX-RANDOM.2017.42.

19 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Periodicity in Data Streams with Wildcards. In *Proceedings of the International Computer Science Symposium in Russia, CSR 2018*, pages 90–105, 2018. doi:10.1007/978-3-319-90530-3_9.

20 Funda Ergün, Hossein Jowhari, and Mert Sağlam. Periodicity in Streams. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2010*, pages 545–559, 2010. doi:10.1007/978-3-642-15369-3_41.

21 Nathan J. Fine and Herbert S. Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.

22 Tomáš Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theoretical Computer Science*, 506:102–114, 2013. doi:10.1016/j.tcs.2013.08.013.

23 Qing Guo, Hui Zhang, and Costas S. Iliopoulos. Computing the λ-Seeds of a String. In *Proceedings of Algorithmic Aspects in Information and Management, AAIM 2006*, pages 303–313, 2006. doi:10.1007/11775096_28.

24 Qing Guo, Hui Zhang, and Costas S. Iliopoulos. Computing the λ-covers of a string. *Information Sciences*, 177(19):3957–3967, 2007. doi:10.1016/j.ins.2007.02.020.

25 Ondřej Guth. On approximate enhanced covers under Hamming distance. *Discrete Applied Mathematics*, 2019. doi:10.1016/j.dam.2019.01.015.

26 Costas S. Iliopoulos, Manal Mohamed, and William F. Smyth. New complexity results for the k-covers problem. *Information Sciences*, 181(12):2571–2575, 2011. doi:10.1016/j.ins.2011.02.009.

27 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, September 1996. doi:10.1007/BF01955677.

28 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

29 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:322–350, 1977. doi:10.1137/0206024.

30 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*, pages 1095–1112, 2012. URL: http://dl.acm.org/citation.cfm?id=2095116.2095202.

31 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation. *CoRR*, abs/1107.2422v2, 2019. arXiv:1107.2422v2.

32 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast Algorithm for Partial Covers in Words. *Algorithmica*, 73(1):217–233, September 2015. doi:10.1007/s00453-014-9915-3.

33 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient algorithms for shortest partial seeds in words. *Theoretical Computer Science*, 710:139–147, 2018. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). doi:10.1016/j.tcs.2016.11.035.

34 Yin Li and William F. Smyth. Computing the Cover Array in Linear Time. *Algorithmica*, 32(1):95–106, January 2002. doi:10.1007/s00453-001-0062-2.

35 Neerja Mhaskar and William F. Smyth. Frequency Covers for Strings. *Fundamenta Informaticae*, 163(3):275–289, 2018. doi:10.3233/FI-2018-1744.

36 Neerja Mhaskar and William F. Smyth. String covering with optimal covers. *Journal of Discrete Algorithms*, 51:26–38, 2018. doi:10.1016/j.jda.2018.09.003.

37 Dennis Moore and William F. Smyth. A Correction to "An Optimal Algorithm to Compute All the Covers of a String". *Information Processing Letters*, 54(2):101–103, April 1995. doi:10.1016/0020-0190(94)00235-Q.

38 Dennis W. G. Moore and William F. Smyth. An Optimal Algorithm to Compute all the Covers of a String. *Information Processing Letters*, 50:239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

39 Alexandru Popa and Andrei Tanasescu. Hardness and algorithmic results for the approximate cover problem. *CoRR*, abs/1806.08135, 2018. arXiv:1806.08135.

40 Benny Porat and Ely Porat. Exact And Approximate Pattern Matching In The Streaming Model. In *Proceedings of the Annual Symposium on Foundations of Computer Science, FOCS 2009*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

41 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Algorithms for Computing the λ-regularities in Strings. *Fundamenta Informaticae*, 84(1):33–49, 2008. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi84-1-04.

Computing Runs on a Trie

Ryo Sugahara
Department of Informatics, Kyushu University, Japan
sugahara.ryo.408@s.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

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

A maximal repetition, or run, in a string, is a maximal periodic substring whose smallest period is at most half the length of the substring. In this paper, we consider runs that correspond to a path on a trie, or in other words, on a rooted edge-labeled tree where the endpoints of the path must be a descendant/ancestor of the other. For a trie with n edges, we show that the number of runs is less than n. We also show an $O(n\sqrt{\log n}\log\log n)$ time and $O(n)$ space algorithm for counting and finding the shallower endpoint of all runs. We further show an $O(n\log n)$ time and $O(n)$ space algorithm for finding both endpoints of all runs. We also discuss how to improve the running time even more.

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

Keywords and phrases runs, Lyndon words

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.23

Related Version `https://arxiv.org/abs/1901.10633`

Funding *Yuto Nakashima*: Supported by JSPS KAKENHI Grant Number JP18K18002.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.
Hideo Bannai: Supported by JSPS KAKENHI Grant Number JP16H02783.
Masayuki Takeda: Supported by JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

Repetitions are fundamental characteristics of strings, and their combinatorial properties as well as their efficient computation has been a subject of extensive studies. Maximal periodic substrings, or *runs*, is one of the most important types of repetitions, since they essentially capture all occurrences of consecutively repeating substrings in a given string. One of the reasons which makes runs important and interesting is that the number of runs contained in a given string of length n is $O(n)$ [18], in fact, less than n [1], and can be computed in $O(n)$ time assuming a constant or integer alphabet [18, 1], or in $O(n\alpha(n))$ time for general ordered alphabets [9], where α is the inverse Ackermann function.

In this paper, we consider runs that correspond to a path on a trie, or in other words, on a rooted edge-labeled tree where the endpoints of the path must be a descendant/ancestor of the other. The contributions of this paper are as follows. For a trie with n edges, we show:

- that the number of such periodically maximal paths is linear, and in fact less than n.
- an $O(n\sqrt{\log n}\log\log n)$ time and $O(n)$ space algorithm for counting and finding the shallower endpoint of all such paths.
- an $O(n\sqrt{\log n}\log^2\log n)$ time and $O(n)$ space algorithm for finding both endpoints of such paths.

Furthermore, we also discuss how to improve the running time even more.

1.1 Related Work

A similar problem was considered in [16, 8, 17], but differs in three aspects: they consider *distinct* repetitions with *integer powers* on an *unrooted* (or undirected) tree. In this work, we consider *occurrences* of repetitions with maximal (possibly fractional) powers on a *rooted* (directed) tree.

2 Preliminaries

2.1 Strings, Periods, Maximal Repetitions, Lyndon Words

Let $\Sigma = \{1, \ldots, \sigma\}$ denote the alphabet. We consider an integer alphabet, i.e., $\sigma = n^c$ for some constant c. Σ^* is the set of strings over Σ. For any string $w \in \Sigma^*$, let $w[i]$ denote the ith symbol of w, and $|w|$ the length of w. For any $1 \leq i \leq j \leq |w|$, let $w[i..j] = w[i] \cdots w[j]$. For technical reasons, we assume that w is followed by a distinct character (i.e. $w[|w|+1]$) in Σ that does not occur in $w[1..|w|]$.

A string is *primitive*, if it is not a concatenation of 2 or more complete copies of the same string. A string $w = u^2$, for some string u, is called a *square*, and in particular, if u is primitive, then w is called a *primitively rooted square*. An integer $1 \leq p \leq |w|$ is called a period of w, if $w[i] = w[i+p]$ for all $1 \leq i \leq |w| - p$. The smallest period of w will be denoted by $per(w)$. For any period p of w, there exists a string x, called a *border* of w, such that $|x| = |w| - p$ and $w = xy = zx$ for some y, z. A string is a *repetition*, if its smallest period is at most half the length of the string. A *maximal repetition*, or *run*, is a maximal periodic substring that is a repetition, i.e., a maximal repetition of a string w is an interval $[i..j]$ of positions where $per(w[i..j]) \leq (j-i+1)/2$, and $per(w[i..j]) \neq per(w[i'..j'])$ for any $1 \leq i' \leq i$ and $j \leq j' \leq n$ such that $i' \neq i$ or $j' \neq j$. In other words, a run contains at least two consecutive occurrences of a substring of length p, and the periodicity does not extend to the left or right of the run. The smallest period of the run will be called the period of the run. The fraction $(j-i+1)/p \geq 2$ is called the *exponent* of the run.

Let $\prec_0$ denote an arbitrary total ordering on Σ, as well as the lexicographic ordering on Σ^* induced by this ordering. We also consider the reverse ordering $\prec_1$ on Σ (i.e., $\forall a, b \in \Sigma, a \prec_0 b \iff b \prec_1 a$), and the induced lexicographic ordering on Σ^*. For $\ell \in \{0, 1\}$, let $\bar{\ell} = 1 - \ell$. A string w is a *Lyndon word* w.r.t. to a given lexicographic ordering, if w is lexicographically smaller than any of its proper suffixes. A well known fact is that a Lyndon word cannot have a non-empty border.

Crochemore et al. observed that in any run $[i..j]$ with period p, and any lexicographic ordering, there exists a substring of length p in the run, that is a Lyndon word [10, 11]. Such Lyndon words are called L-roots. Below, we briefly review the main result of [1] which essentially tied longest Lyndon words starting at specific positions within the run, to L-roots of runs. This will be the basis for our new results for tries.

▶ **Lemma 1** (Lemma 3.2 of [1]). *For any position $1 \leq i \leq |w|$ of string w, let $\ell \in \{0,1\}$ be such that $w[k] \prec_\ell w[i]$ for $k = \min\{k' \mid w[k'] \neq w[i], k' > i\}$. Then, the longest Lyndon word that starts at position i is $w[i..i]$ w.r.t $\prec_\ell$, and $w[i..j]$ for some $j \geq k$ w.r.t. $\prec_{\bar{\ell}}$.*[1]

▶ **Lemma 2** (Lemma 3.3 of [1]). *Let $r = [i..j]$ be a run in w with period p, and let $\ell \in \{0,1\}$ be such that $w[j+1] \prec_\ell w[j+1-p]$. Then, any L-root $w[i'..j']$ of r with respect to $\prec_\ell$ is the longest Lyndon word with respect to $\prec_\ell$ that is a prefix of $w[i'..|w|]$.*

Since an L-root cannot be shared by two different runs, it follows from Lemma 2 that the number of runs is at most $2n$, since each position can be the starting point of at most two L-roots that correspond to distinct runs. In [1], a stronger bound of n was shown from the observation that each run contains at least one L-root that does not begin at the first position of the run, and that the two longest Lyndon words starting at a given position for the two lexicographic orders cannot simultaneously be such L-roots of runs. This is because if $w[i'..i']$ and $w[i'..j']$ were L-roots and the runs start before position i', then, from the periods of the two runs, it must be that $w[i'-1] = w[i'] = w[j']$ contradicting that $w[i'..j']$ is a Lyndon word and cannot have a non-empty border. In Section 3.1, we will see that the last argument does not completely carry over to the case of tries, but show that we can still improve the bound again to n.

The above lemmas also lead to a new linear time algorithm for computing all runs, that consists of the following steps:

1. compute the longest Lyndon word that starts at each position for both lexicographic orders $\prec_0$ and $\prec_1$,
2. check whether there is a run for which the longest Lyndon word corresponds to an L-root.

There are several ways to compute the first step in amortized constant time for each position, but it essentially involves computing the next smaller values (NSV) in the inverse suffix array of the string. We describe the algorithm in more detail in Section 3.2, and will see that the amortization of the standard algorithm does not carry over to the trie case. We give a new linear time algorithm using the static tree set-union data structure (more specifically, decremental nearest marked ancestor queries) [14], which *does* carry over to the trie case.

The second step can be computed in constant time per candidate L-root with linear time-preprocessing, by using longest common extension queries (e.g. [13]) in the forward and reverse directions of the string. Unfortunately again, this does not directly carry over to the trie case because, as far as we know, longest common extension queries on trees can be computed in constant time only in the direction toward the root of the trie, when space is restricted to linear in the size of the trie. In Section 3.3, we will show how to apply the range predecessor/successor data structure of [2] to this problem.

2.2 Common Suffix Trie

A *trie* is a rooted tree with labeled edges, such that each edge to the children of a node is labeled with distinct symbols. A trie can be considered as representing a set of strings obtained by concatenating the labels on a root to leaf path. Note that for a trie with n edges, the total length of such strings can be quadratic in n. An example can be given by the set of strings $X = \{xc_1, xc_2, \cdots xc_n\}$ where $x \in \Sigma^{n-1}$ is an arbitrary string and $c_1, \ldots, c_n \in \Sigma$

[1] Note that j becomes $|w|+1$ when $i = |w|$.

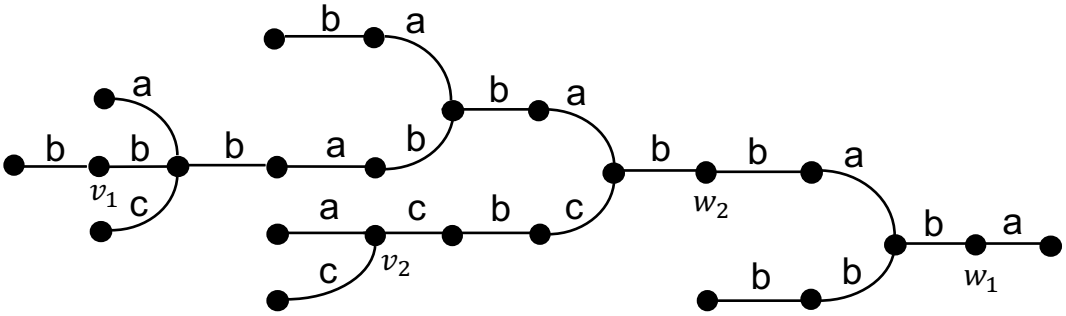


Figure 1 Example of runs in a trie. (v_1, w_1) is a run with period 3, and (v_2, w_2) is a run with period 2.

are pairwise distinct characters. Here, the size of the trie is $\Theta(n)$, while the total length of strings is $\Theta(n^2)$. Also notice that the total number of distinct suffixes of strings in X is also $\Theta(n^2)$. However if we consider the strings in the reverse direction, i.e., consider edges of the trie to be directed toward the root, the number of distinct suffixes is linear in the size of the tree. Such tries are called *common suffix tries* [7]. We will use the terms parent/child/ancestor/descendant with the standard meaning based on the undirected trie, e.g., the root is an ancestor of all nodes in the trie. For any node u of the trie, $par(u)$ will denote its parent node. Also, we consider a node to be an ancestor/descendant of itself.

For any nodes u, v of the trie where v is an ancestor of u, let $str(u, v)$ denote the string obtained by concatenating the labels on the path from u to v. For technical reasons, we assume that the root node has an auxiliary parent node $\perp$, where the edge is labeled by a distinct character in Σ that is not used elsewhere in the trie. We denote by $suf(v)$ the string obtained by concatenating the labels on the path from v to $\perp$, i.e., $suf(v) = str(v, \perp)$. Such strings will be called a *suffix* of the trie.

Note that a trie can be pre-processed in linear time so that for any node v, the ancestor of node v at an arbitrary specified depth d can be obtained in constant time (e.g. [4]).

3 Runs in a Trie

3.1 The Number of Runs in a Trie

We first define runs on a trie. A run (v_i, v_j) on a trie T is a maximal periodic path with endpoints v_i and v_j, where v_j is an ancestor of v_i and $str(v_i, v_j)$ is a repetition. More precisely, $per(str(v_i, v_j)) \leq |str(v_i, v_j)|/2$, and for any descendant $v_{i'}$ of v_i and ancestor $v_{j'}$ of v_j, $per(str(v_i, v_j)) \neq per(str(v_{i'}, v_{j'}))$ if $v_{i'} \neq v_i$ or $v_{j'} \neq v_j$.

Noticing that for any node in the trie its parent is unique, it is easy to see that analogies of Lemmas 1 and 2 hold for tries. Thus, we have the following.

▶ **Corollary 3.** *For any node v except the root or $\perp$, let $w_v = suf(v)$, and $\ell \in \{0, 1\}$ be such that $w_v[k] \prec_\ell w_v[1]$ for $k = \min\{k' \mid w_v[k'] \neq w_v[1], k' > 1\}$. Then, the longest Lyndon word that is a prefix of w_v is $w_v[1..1]$ w.r.t. $\prec_\ell$ and $w_v[1..j]$ w.r.t. $\prec_{\bar{\ell}}$ for some $j \geq k$.*

▶ **Corollary 4.** *Let $r = (v_i, v_j)$ be a run with period p in the trie, $w_v = suf(v_i)$, and $\ell \in \{0, 1\}$ be such that $w_v[x + 1] \prec_\ell w_v[x - p + 1]$, where $x = |str(v_i, v_j)|$. Then, any L-root $str(v_{i'}, v_{j'})$ of the run with respect to $\prec_\ell$ is the longest Lyndon word that is a prefix of $suf(v_{i'})$.*

Since we assumed that the edge labels of the children of a given node in a trie are distinct, a given L-root can only correspond to one distinct run; i.e., the extension of the period in both directions from an L-root is uniquely determined. Therefore, the argument for standard strings carries over to the trie case, and it follows that the number of runs must be less than $2n$. We further observe the following

▶ **Theorem 5.** *The maximum number of runs in a trie with n edges is less than n.*

Proof. Suppose $str(v_{i'}, par(v_{i'}))$ and $str(v_{i'}, v_{j'})$ are simultaneously L-roots of runs respectively w.r.t. $\prec_\ell$ and $\prec_{\bar{\ell}}$, and that they do not start at the beginning of the runs. Let $p = |str(v_{i'}, v_{j'})|$ and $w_{v_{i'}} = suf(v_{i'})$. If $v_{i'}$ has only one child, this leads to a contradiction using the same argument as the case for strings; i.e., if u is the child of $v_{i'}$, then, from the periods of the two runs, $str(u, v_{i'}) = w_{v_{i'}}[1] = w_{v_{i'}}[p]$ contradicting that $w_{v_{i'}}[1..p]$ is a Lyndon word and cannot have a non-empty border. Thus, $v_{i'}$ must have at least two children, u, u', where $str(u, v_{i'}) = w_{v_{i'}}[1]$ and $str(u', v_{i'}) = w_{v_{i'}}[p]$. Let k be the number of branching nodes in the trie. Then, the number of leaves is at least $k + 1$. Since a run cannot start before a leaf node, this means that a longest Lyndon word starting at a leaf cannot be an L-root that does not start at the beginning of the run. Therefore, although there can be at most k nodes such that both longest Lyndon words are such L-roots, there exist at least $k + 1$ nodes where both are not. Thus, the theorem holds. ◀

In a similar way to Theorem 3.6 of [1], we can bound the sum of exponents of all runs in a trie.

▶ **Corollary 6.** *The sum of exponents of all runs in a trie with n edges is less than $3n$.*

Proof. A given run r with exponent e_r contains at least $\lfloor e_r - 1 \rfloor \geq 1$ occurrences of its L-roots that do not start at the beginning of the run, each corresponding to a longest Lyndon word starting at that position. From the proof of Theorem 5, the total number of these L-roots is less than n. Let *Runs* denote the set of all runs in the trie. Then, $\sum_{r \in Runs}(e_r - 2) \leq \sum_{r \in Runs} \lfloor e_r - 1 \rfloor < n$, and the Corollary follows. ◀

3.2 Computing Longest Lyndon Words

Next, we consider the problem of computing, for any node v of the trie, the longest Lyndon word that is a prefix of $suf(v)$. We first describe the algorithm for strings, which is based on the following lemma.

▶ **Lemma 7.** *For any string w and position $1 \leq i < |w|$, the longest Lyndon word starting at i w.r.t. $\prec$ is $w[i..j-1]$, where j is such that $j = \min\{k > i \mid w[k..|w|] \prec w[i..|w|]\}$.*

Proof. Let $j = \min\{k > i \mid w[k..|w|] \prec w[i..|w|]\}$. By definition, we have $w[k..|w|] \succ w[i..|w|]$ for any $i < k < j$. For any such k, $w[k..|w|] = w[k..j-1]w[j..|w|] \succ w[i..i+(j-1-k)]w[i+(j-k)..j-1]w[j..|w|] = w[i..|w|]$. If the longest common prefix of $w[k..|w|]$ and $w[i..|w|]$ is longer than or equal to $w[k..j-1]$, this implies $w[j..|w|] \succ w[i+(j-k)..j-1]w[j..|w|] \succ w[i..|w|]$, a contradiction. Therefore, the longest common prefix of $w[k..|w|]$ and $w[i..|w|]$ must be shorter than $w[k..j-1]$, implying that $w[i..j-1] \prec w[k..j-1]$. Thus, $w[i..j-1]$ is a Lyndon word. Suppose $w[i..k]$ is a Lyndon word for some $k \geq j$. Then, $w[i..k] \prec w[j..k]$. Since $|w[i..k]| > |w[j..k]|$, $w[i..k]$ cannot be a prefix of $w[j..k]$ which implies $w[i..|w|] \prec w[j..|w|]$, contradicting the definition of j. Thus, $w[i..j-1]$ is the longest Lyndon word starting at i. ◀

From Lemma 7, the longest Lyndon word starting at each position of a string w can be computed in linear time, given the inverse suffix array of w. The inverse suffix array $ISA[1..|w|]$ of w is an array of integers such that $ISA[i] = j$ when $w[i..|w|]$ is the lexicographically jth

smallest suffix of w. That is, the j in Lemma 7 can be restated as $j = \min\{k > i \mid ISA[k] < ISA[i]\}$. This can be restated as the problem of finding the next smaller value (NSV) for each position of the ISA, for which there exists a simple linear time algorithm (e.g. [24]) as show in Algorithm 1. The linear running time comes from a simple amortized analysis;

Algorithm 1: Computing NSV on array A of integers.

```
   // assumes A[n+1] is smaller than all values in A.
1  NSV[n] = n + 1;
2  for i = n − 1 to 1 do
3  |  x = i + 1;
4  |  while A[i] ≤ A[x] do
5  |  |  x = NSV[x];
6  |  NSV[i] = x;
```

in the while loop, $NSV[x]$ is only accessed once for any position x since $NSV[i]$ is set to a larger value and thus will subsequently be skipped.

Since, as before, the parent of a node is unique, Lemma 7 carries over to the trie case. We can assign the lexicographic rank $ISA[v]$ of $suf(v)$ to each node v in linear time from the suffix tree of the trie, which is a compacted trie containing all and only suffixes of the trie.

▶ **Theorem 8** (suffix tree of a trie [7, 22]). *The suffix tree of a trie on a constant or integer alphabet can be represented and constructed in $O(n)$ time.*

The problem now is to compute, for each node v_i, the closest ancestor v_j of v_i such that the lexicographic rank of $suf(v_j)$ is smaller than that of $suf(v_i)$. Algorithm 1 can be modified to correctly compute the NSV values on the trie; the for loop is modified to enumerate nodes in some order such that the parent of a considered node is already processed, and line 3 can be changed to $x = par(x)$. However, the amortization will not work; the existence of branching paths means there can be more than one child of a given node, and the same position (node) x could be accessed in the while loop for multiple paths, leading to a super-linear running time.

To overcome this problem, we introduce a new, (conceptually) simple linear time algorithm based on nearest marked ancestor queries.

▶ **Theorem 9** (decremental nearest marked ancestor [14]). *A given tree can be processed in linear time such that all nodes are initially marked, and the following operations can be done in amortized constant time:*

- *$nma(v)$: return the nearest ancestor node of v that is marked.*
- *$unmark(v)$: unmark the node v.*

The pseudo-code of our algorithm is shown in Algorithm 2.

Algorithm 2: Computing NSV on trie with values ISA.

```
1  Preprocess trie for decremental nearest marked ancestor;
2  foreach node v in decreasing order of ISA[v] do
3  |  unmark(v);
4  |  NSV[v] = nma(v);
```

▶ **Theorem 10.** *Given a trie of size n, the longest Lyndon word that is a prefix of $suf(v)$ for each v can be computed in total $O(n)$ time and space.*

Proof. It is easy to see the linear running time of Algorithm 2. The correctness is also easy to see, because the nodes are processed in decreasing order of lexicographic rank, and thus, all and only nodes with larger lexicographic rank are unmarked. ◀

3.3 Computing Runs

To compute all runs in a trie, we extend the algorithm for strings to the trie case. After computing the longest Lyndon word that is a prefix of $suf(v)$ for each node v for the two lexicographic orderings $\prec_0$ and $\prec_1$, we must next see if they are L-roots of runs by checking how long the periodicity extends. Given a longest Lyndon word $y = str(v_i, v_j)$ w.r.t. $\prec_\ell$ that starts at v_i, we can compute the longest common extension from nodes v_i and v_j towards the root, i.e., the longest common prefix $z = str(v_j, v_k)$ between $suf(v_i)$ and $suf(v_j)$. To avoid outputting duplicate runs, y will be a candidate L-root only if $|z| < |y|$ and $str(v_i)[|y| + |z| + 1] \prec_\ell str(v_i)[|z| + 1]$. Using the suffix tree of the trie, this longest common extension query can be computed in constant time after linear time preprocessing, since it amounts to lowest common ancestor queries (e.g. [3]). The central difficulty of our problem is in computing the longest common extension in the opposite direction, i.e. towards the leaves, because the paths can be branching. We cannot solve this problem by simply considering longest common extensions on the common suffix trie for the reverse strings, since, as observed in Section 2, this can lead to a quadratic blow-up in the size of the trie.

We overcome this problem by reducing the longest common extension query in the leaf direction to several queries of finding the lexicographically closest suffix that is in a specific subtree and at a specific depth, combined with some naive traversal. We use the following result for range predecessor queries multiple times to achieve Lemma 12. A range predecessor problem is to pre-process n points on a $[1, n]^2$ grid where all points differ in both coordinates, so as to answer the query: given three integers x_1, x_2, y find the point $rpred(x_1, x_2, y) = (u, v)$ such that $x_1 \leq u \leq x_2$, and v is the largest such that $v \leq y$.

▶ **Theorem 11** (Range Predecessor Queries (Theorem 5.1 in [2])). *Given n points from the grid $[1, n]^2$, we can in $O(n\sqrt{\log n})$ time build a data structure that occupies $O(n \log n)$ bits of space and that answers range predecessor queries in $O(\sqrt{\log n} \log \log n)$ time. The construction uses $O(n \log n)$ bits of working space.*

By converting each of the y coordinates to $n - y + 1$, range successor queries $rsucc(x_1, x_2, y)$ can also be achieved in the same time/space bounds.

▶ **Lemma 12.** *A trie of size n can be pre-processed in $O(n\sqrt{\log n})$ time and $O(n)$ space so that given a node v and a positive integer $d > |suf(v)|$, we can answer in $O(\sqrt{\log n} \log \log n)$ time, the node v' such that v' is a descendant of v where $|suf(v')| = d$ and of all such nodes, has the longest common prefix with $suf(v)$.*

Proof. We construct several range predecessor/successor data structures of Theorem 11, where each point in the grid corresponds to a node in the trie. The first coordinate (which we will consider as the identifier of the node) is given by the order of nodes that would appear in a breadth-first traversal of the trie, where nodes of the same depth are ordered as they would appear in a depth-first traversal. Thus, for any depth d, all nodes at depth d can be represented in an interval $[i_d..j_d]$. The second coordinate is given by each of the following three types of values in the range $[1, n]$, and can be assigned to each node v by a simple depth-first traversal on the trie.

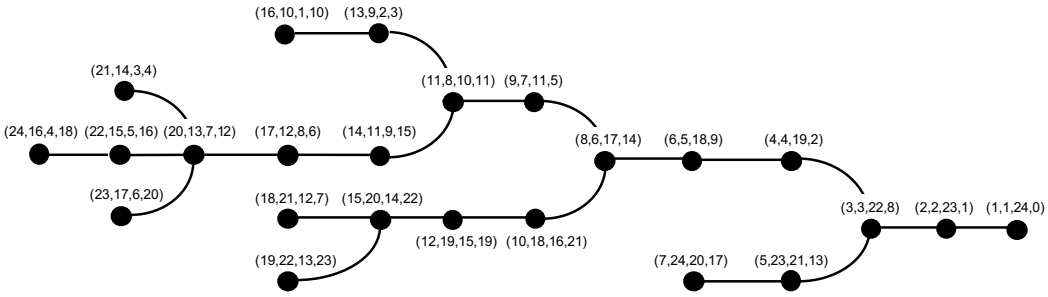


Figure 2 An example of values assigned for each node v of the trie for Figure 1. Each 4-tuple shows (v, b_v, e_v, l_v), where v is the identifier of the node, b_v is the pre-order rank, e_v is the post-order rank, and l_v is the lexicographic rank of $suf(v)$.

- b_v: pre-order rank of node v in a depth-first traversal.
- e_v: post-order rank of node v in a depth-first traversal.
- l_v: lexicographic rank of $suf(v)$ (i.e. $ISA[v]$).

The construction costs $O(n\sqrt{\log n})$ time and $O(n)$ (words of) space due to Theorem 11.

Now, the desired node u that is the answer to our query lies in the range $[i_d..j_d]$. The nodes corresponding to descendants of v further lie in a sub-range, namely, $[i_{d,v}..j_{d,v}]$, which can be computed by $(i_{d,v}, b') = rsucc(i_d, j_d, b_v)$ and $(j_{d,v}, e') = rpred(i_d, j_d, e_v)$, respectively using the successor and predecessor data structures for pre-order rank and post-order rank.

Finally, the node u which gives the longest common prefix must be one of the lexicographically closest ones in this range, i.e., it is one of $(u_1, l'_1) = rpred(i_{d,v}, j_{d,v}, l_v)$ or $(u_2, l'_2) = rsucc(i_{d,v}, j_{d,v}, l_v)$, using the predecessor/successor data structure for lexicographic rank. The longest common prefix between $suf(u_1)$ and $suf(v)$ as well as that between $suf(u_2)$ and $suf(v)$ can be computed in constant time with linear time pre-processing, as mentioned before. Thus, the total time is $O(\sqrt{\log n} \log\log n)$ for the four range queries. ◀

We first use Lemma 12, once for each candidate L-root, to determine whether the periodicity of the candidate L-root can extend long enough in the leaf direction to form a run.

▶ **Theorem 13.** *Given a trie of size n, we can in $O(n\sqrt{\log n}\log\log n)$ time and $O(n)$ space, count the total number of runs in the trie, as well as identify the shallower endpoint of all runs.*

Proof. Let $y = str(v_i, v_j)$ be a candidate L-root and let $z = str(v_j, v_k)$ be its extension in the root direction as described in the beginning of Section 3.3. Let $p = |y|$. Also, let v' be the node on the path from v_i and v_j at depth $|suf(v_k)| + p$. We use Lemma 12 in order to check if there exists a node v_l that is a descendant of v' and at depth $|suf(v_k)| + 2p$ such that the longest common prefix between $str(v_l)$ and $str(v')$ is at least p. If v_l does not exist, then the L-root cannot be extended to a run. If v_l does exist, this implies that $str(v_l, v') = str(v', v_k)$, and since $z = str(v_j, v_k) = str(v_i, v')$ is a suffix of $str(v', v_k)$ and thus of $str(v_l, v')$, v_l must be a descendant of v_i because labels on the edges to child nodes are distinct. Thus, the prefix of length $2p$ of $suf(v_l)$ has period p, contains $str(v_i, v_j)$, and the periodicity ends at v_k, i.e., it is an endpoint of a run with $str(v_i, v_j)$ as an L-root.

Such queries are conducted at most once for each candidate L-root, so the total time is $O(n\sqrt{\log n}\log\log n)$. ◀

Finally, we describe how to compute the other endpoint. We make use of Theorem 13 and the following static dictionary.

▶ **Theorem 14** (Static Dictionary (Theorem 1 in [21])). *Suppose that a set of n integers from the universe $\{0, 1, \ldots, n^{O(1)}\}$ is given. A static linear space dictionary on that set can be deterministically constructed on a word RAM in time $O(n \log \log n)$, so that lookups to the dictionary take constant time.*

▶ **Theorem 15.** *Given a trie of size n, we can compute in $O(n \log n)$ time and $O(n)$ space, all runs (v_i, v_j) in the trie.*

Proof. After confirming an occurrence of the run as described in Theorem 13, we further repeatedly use Lemma 12 to check if the periodicity can be further extended in the leaf direction by length p. The total number of times that this is repeated is bounded by the sum of exponents of all runs in the trie, which is linear (Corollary 6). Therefore, the total time for this is $O(n\sqrt{\log n} \log \log n)$.

It remains to find the remaining extension shorter than the period p of the run. We compute this extension naively edge by edge. We know which character we need to extend by, since we know $str(v_i, v_j)$ is what is repeating. To avoid the $O(\log \sigma)$ factor for finding the child node of a branching node, we use the static dictionary by Ruzic [21], so that each child node can be found in constant time. This can be done with $O(n \log \log n)$ time and $O(n)$ space pre-processing using Theorem 14; each edge can be considered as a pair of integers from 1 to n and 1 to n^c, representing a unique id of the node, and the label on the edge. The pair can be encoded as an integer from 1 to $n^{O(1)}$, where the encoding and decoding can be done in constant time. Given a node and edge label, the child can be obtained in constant time by looking up the dictionary.

Finally, since each naive extension implies an occurrence of a primitively rooted square, the total number of naive extensions is bounded by the total number of primitively rooted squares that occur in the trie. Due to the three squares lemma (Lemma 10 of [12]), it follows that for each node v, there can only be $O(\log |suf(v)|)$ primitively rooted squares that are prefixes of $suf(v)$. Thus, it follows that their total number is $O(n \log n)$.

From the above arguments, computing both endpoints of all runs can be done in $O(n \log n)$ time using $O(n)$ space. ◀

4 Discussion

Shortly after the submission of the paper, we realized that running time could be improved by using a doubling + binary search when extending the run in the leaf direction, rather than a naive traversal. If the remaining length of a run is t, the query of Lemma 12 is conducted $O(\log t)$ times. Let $Runs$ be the set of all runs in the trie, and let t_r denote the length of the remaining extension for run $r \in Runs$. The total number of queries is thus $\sum_{r \in Runs} \log t_r$, where $\sum_{r \in Runs} t_r = O(n \log n)$ as mentioned in the proof of Theorem 15. Since $\sum_{r \in Runs} \log t_r = \log(\prod_{r \in Runs} t_r)$, this is maximized when each t_r has the same length, i.e., $t_r = \Theta(\frac{n \log n}{|Runs|})$. Thus, $\sum_{r \in Runs} \log t_r = O(|Runs| \log \frac{n \log n}{|Runs|})$. Noticing that $|Runs| \log \frac{n}{|Runs|} = O(n)$, the total number of times Lemma 12 is used can be bounded by $O(n \log \log n)$. Therefore, the total running time would be $O(n\sqrt{\log n}(\log \log n)^2)$.

Furthermore, after posting our paper with the above improvement to arXiv [23], Tomohiro I pointed us to the paper by Bille et al. [5], where they consider LCE queries on tries in the leaf direction. The path-tree query in their paper can be used in place of Lemma 12 and is more powerful. A path-tree query, given nodes v_1, v_2 and w, where v_2 is a descendant of v_1, returns the longest common prefix of the path from v_1 to v_2, and any path from w to a descendant leaf.

▶ **Theorem 16** (Theorem 2 of [5]). *For a tree T with n nodes, a data structure of size $O(n)$ can be constructed in $O(n)$ time to answer path–tree LCE queries in $O((\log \log n)^2)$ time.*

The number of times the path-tree query is used can be bounded by the total sum of exponents of runs, and thus is linear. Therefore, the total computation time can be improved to $O(n(\log\log n)^2)$.

5 Conclusion

We generalized the notion of runs in strings to runs in tries, and showed that the analysis of the maximum number of runs, as well as algorithms for computing runs can be extended and adapted to the trie case, but with an increase in running time.

Our algorithm can output all primitively rooted squares in $O(n \log n)$ time, which is tight, since there can be $\Theta(n \log n)$ primitively rooted squares in a string, e.g., Fibonacci words [20], and thus in a trie.

An obvious open problem is whether there exists a linear time algorithm for computing all runs in a trie. For strings, there exists another linear time algorithm for computing all runs that is based on the Lempel-Ziv parsing [18]. It is not clear how this algorithm could be extended to the case of tries. The case for general ordered alphabets, instead of integer alphabets, is another open problem [6, 19, 15, 9].

References

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

2 Djamal Belazzougui and Simon J. Puglisi. Range Predecessor and Lempel-Ziv Parsing. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 2053–2071. SIAM, 2016. doi:10.1137/1.9781611974331.ch143.

3 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.

4 Michael A. Bender and Martin Farach-Colton. The Level Ancestor Problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

5 Philip Bille, Pawel Gawrychowski, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Longest common extensions in trees. *Theor. Comput. Sci.*, 638:98–107, 2016. doi:10.1016/j.tcs.2015.08.009.

6 Dany Breslauer. *Efficient String Algorithmics*. PhD thesis, Columbia University, 1992.

7 Dany Breslauer. The Suffix Tree of a Tree and Minimizing Sequential Transducers. *Theor. Comput. Sci.*, 191(1-2):131–144, 1998. doi:10.1016/S0304-3975(96)00319-2.

8 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Wojciech Tyczynski, and Tomasz Walen. The Maximum Number of Squares in a Tree. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 27–40. Springer, 2012. doi:10.1007/978-3-642-31265-6_3.

9 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20,*

2016, Proceedings, volume 9954 of *Lecture Notes in Computer Science*, pages 22–34, 2016. doi:10.1007/978-3-319-46049-9_3.

10 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. The maximal number of cubic runs in a word. *J. Comput. Syst. Sci.*, 78(6):1828–1836, 2012. doi:10.1016/j.jcss.2011.12.005.

11 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

12 Maxime Crochemore and Wojciech Rytter. Sqares, Cubes, and Time-Space Efficient String Searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

13 Johannes Fischer and Volker Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

14 Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

15 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster Longest Common Extension Queries in Strings over General Alphabets. 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 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.5.

16 Tomasz Kociumaka, Jakub Pachocki, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Efficient counting of square substrings in a tree. *Theoretical Computer Science*, 544:60–73, 2014.

17 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. String Powers in Trees. *Algorithmica*, 79(3):814–834, 2017. doi:10.1007/s00453-016-0271-3.

18 Roman M. Kolpakov and Gregory Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

19 Dmitry Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.

20 M. Lothaire. *Periodic Structures in Words*, page 430–477. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005. doi:10.1017/CBO9781107341005.009.

21 Milan Ruzic. Constructing Efficient Dictionaries in Close to Sorting Time. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

22 Tetsuo Shibuya. Constructing the Suffix Tree of a Tree with a Large Alphabet. In Alok Aggarwal and C. Pandu Rangan, editors, *Algorithms and Computation, 10th International Symposium, ISAAC '99, Chennai, India, December 16-18, 1999, Proceedings*, volume 1741 of *Lecture Notes in Computer Science*, pages 225–236. Springer, 1999. doi:10.1007/3-540-46632-0_24.

23 Ryo Sugahara, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing runs on a trie. *CoRR*, abs/1901.10633, 2019. arXiv:1901.10633.

24 Wikipedia. All nearest smaller values — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=All%20nearest%20smaller%20values&oldid=882430084, 2019. [Online; accessed 29-March-2019].

Linking BWT and XBW via Aho-Corasick Automaton: Applications to Run-Length Encoding

Bastien Cazaux
Department of Computer Science, University of Helsinki, Finland
L.I.R.M.M., CNRS, Université Montpellier, France
https://www.mv.helsinki.fi/home/cazaux/
bastien.cazaux@helsinki.fi

Eric Rivals
L.I.R.M.M., CNRS, Université Montpellier, France
https://www.lirmm.fr/~rivals/
rivals@lirmm.fr

Abstract

The boom of genomic sequencing makes compression of sets of sequences inescapable. This underlies the need for multi-string indexing data structures that helps compressing the data. The most prominent example of such data structures is the Burrows-Wheeler Transform (BWT), a reversible permutation of a text that improves its compressibility. A similar data structure, the eXtended Burrows-Wheeler Transform (XBW), is able to index a tree labelled with alphabet symbols. A link between a multi-string BWT and the Aho-Corasick automaton has already been found and led to a way to build a XBW from a multi-string BWT. We exhibit a stronger link between a multi-string BWT and a XBW by using the order of the concatenation in the multi-string. This bijective link has several applications: first, it allows one to build one data structure from the other; second, it enables one to compute an ordering of the input strings that optimises a Run-Length measure (i.e., the compressibility) of the BWT or of the XBW.

2012 ACM Subject Classification Mathematics of computing → Discrete mathematics; Theory of computation → Randomness, geometry and discrete structures; Theory of computation → Data structures and algorithms for data management

Keywords and phrases Data Structure, Algorithm, Aho-Corasick Tree, compression, RLE

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.24

Related Version A full version of the paper is available at https://arxiv.org/abs/1805.10070.

Supplement Material Source code at: https://framagit.org/bcazaux/compressbwt

Funding Support from the Institut de Biologie Computationnelle (ANR-11-BINF-0002).
Eric Rivals: thanks the GEM Flagship project funded from Labex NUMEV (ANR-10-LABX-0020).

1 Introduction

A seminal, key data structure, which was used for searching a set of words in a text, is the Aho-Corasick (AC) automaton [1]. Its states form a tree that indexes all the prefixes of the words, and each node in the tree is equipped with another kind of arc, called a Failure Link. A failure link of a node/prefix v points to the node representing the largest proper suffix of v in the tree. In a way, the Aho-Corasick automaton can be viewed as a multi-string indexing data structure.

In the early 90's, the Burrows-Wheeler Transform (BWT) of a text T, which is a reversible permutation of T, was introduced for the sake of compressing a text. Indeed, the BWT permutation tends to group identical symbols in runs, which favours compression [5]. However, the BWT can also be used as an index for searching in T, using the Backward Search

procedure [10]. In fact, the BWT of T is the last column of a matrix containing all cyclic-shifts of T sorted in lexicographical order. As sorting the cyclic shifts of T is equivalent to sorting its suffixes, there exists a natural link between the suffix array of T and the BWT of T. Starting in 2005, the radical increase in textual data and in biological sequences raised the need for multi-string indexes. In the multi-string case, a simple approach is to concatenate all input strings and separate them using the same termination symbol (which does not belong to the alphabet), and then to index the result with a traditional indexing data structure (*e.g.*, a suffix array or a FM-index). Another approach is to add a unique termination symbol for each string without concatenation [3, 21]. In both approaches, an initial order on the strings is given by the permutation of the strings in the first approach, or by the order between the termination symbols in the second. All our results are made with the first approach (which minimises the alphabet size) in mind, and can be adapted to the second. Such multi-string indexes are heavily exploited in bioinformatics: first, to index all chromosomes of a genome [17], or a large collection of similar genomes, which allows aligning sequence reads simultaneously to several reference genomes [19], or second, to store and mine whole sets of raw DNA/RNA sequence reads for the purpose of comparing biological conditions or of identifying splice junctions in RNA [7, 14]. In fact, managing compressed and searchable read data sets is now crucial for bioinformatic analyses.

Initially viewed as a simple extension of single-string BWT construction, the efficient construction of multi-string BWT is not trivial and has been investigated *per se*. Bauer *et al.* proposed, among others, a lightweight incremental algorithm for their construction [3]. Then, Holt *et al.* devised algorithms for directly merging several, already built multi-string BWTs efficiently [14, 15], which has recently been improved to build simultaneously the companion *Longest Common Prefix* (LCP) table [8], or to scale up to terabyte datasets [23].

The notion of BWT has been extended into the XBW to index trees whose arcs are labelled by alphabet symbols [9]. The XBW comprises two arrays, which compactly represent the tree and offer navigational operations.

Recently, Gagie *et al.* propose the notion of Wheeler graphs to subsume several variants of the BWT, including the XBW of a trie for a set of strings [12]. The relation between a multi-string BWT and the XBW representation of the Aho-Corasick automaton has already been studied and exploited. Hon *et al.* first use the XBW representation of the Aho-Corasick trie to speed-up dictionary matching [16] building up on [4]. Manzini gave an algorithm that computes the failure links for the trie using the Suffix Array and LCP tables, and an algorithm to build the XBW of the trie with failure links from the BWT of a multi-string [22]. However, none of these established a bijective link between a multi-string BWT and the XBW of the Aho-Corasick automaton. To generalise these results, one needs to consider the order in which the strings are concatenated to form the multi-string. This idea enables us to exhibit a bijection between a multi-string BWT and the XBW of the Aho-Corasick automaton, which allows building one structure from the other in either direction (from BWT to XBW or from XBW to BWT). Finally, we exploit this bijection to find an optimal string order that minimises a Run-Length Encoding (i.e., maximises the compressibility) of these two data structures. Numerous proofs are in appendix (A preprint version of this work is available as [6]).

2 Preliminaries

In this section, we introduce basic concepts and notation, present the Burrows-Wheeler Transform and an extension of it for a set of strings. Finally, we provide a notation for the Aho-Corasick automaton and the eXtended Burrows-Wheeler Transform.

2.1 General notation

Set and Permutation. Let i and j be two integers such that $i \leq j$. The *interval* $[\![i, j]\!]$ is the set of all integers between i and j. An *integer interval partition* of $[\![i, j]\!]$ is a set of intervals $\{[\![i_1, j_1]\!], \dots [\![i_n, j_n]\!]\}$ such that $i_1 = i$, $j_n = j$, and for all $k \in [\![1, n-1]\!]$, $j_k + 1 = i_{k+1}$. We also define the order $<$ on intervals such that for two intervals $u := [\![i, j]\!]$ and $v := [\![i', j']\!]$, $u < v$ iff $j < i'$. Let E be a finite set and let $\#E$ (or $\#(E)$) denote its cardinality. A *permutation* of E is an automorphism of E. A permutation σ of E is said to be *circular* iff for all i and $j \in E$, there exists a positive integer k such that $\sigma^k(i) = j$ (where $\sigma^1(i) = \sigma(i)$ and $\sigma^k(i) = \sigma(\sigma^{k-1}(i))$). For a circular permutation σ of E and an element $e \in E$, we denote by $\overline{\sigma_e}$ the function from $[\![1, \#E]\!]$ to E such that for all $i \in [\![1, \#E]\!]$, $\overline{\sigma_e}(i) := \sigma^{(i+\#E-1)}(e)$. Given a total order $<$ for E, we define $<_\sigma$ as the order on E such that $e <_\sigma f$ **iff** $\sigma(e) < \sigma(f)$ for any $e, f \in E$.

String. Let Σ be a finite alphabet. A *string* w of length n over Σ is a sequence of symbols $w[1] \dots w[n]$ where $w[i] \in \Sigma$ for all $i \in [\![1, n]\!]$. Σ^* is the set of all finite strings over Σ. The *length* of a string w is denoted by $|w|$. A *substring* of w is written as $w[i, j] := w[i] \dots w[j]$ for some $i \leq j$. A *prefix* of w is a substring of w starting at position 1, and a *suffix* of w is a substring of w ending at position $|w|$. A prefix x (or a suffix) of a string y is said *proper* is x is different from y. The *reverse* of a string w, denoted by $\overleftarrow{w}$, is the string $w[n] \dots w[2]w[1]$. We define the *lexicographic order* $<$ on strings as usual.

Ordered Set of Strings. Let $S = \{s_1, \dots, s_n\}$ be a set of strings. The norm of S, denoted $\|S\|$, is the sum of the length of strings of S, i.e. $\|S\| := \Sigma_{s_i \in S}|s_i|$. Let $\texttt{Prefix}(S)$, (respectively $\texttt{Suffix}(S)$) denote the set of all prefixes (resp. all suffixes) of strings of S. We denote by $\overleftarrow{S}$ the set of all reverse strings of strings of S, i.e. $\overleftarrow{S} := \{\overleftarrow{s_1}, \dots, \overleftarrow{s_n}\}$. An *ordered set of strings* P is a pair (S, σ) where S is a set of strings in lexicographic order, and σ a circular permutation of S. We denote by $P.S$ the set of strings S and by $P.\sigma$ the circular permutation σ. We denote by $\overleftarrow{P}$ the pair $(\overleftarrow{P.S}, P.\sigma)$.

2.2 Burrows-Wheeler Transform

BWT of a string. Let w be a string and i be an integer satisfying $1 \leq i \leq |w|$. The *Suffix Array* (SA) of w [20], denoted $\texttt{SA}(w)$, is the array of integers that stores the starting positions of the $|w|$ suffixes of w sorted in lexicographic order. The *Burrows-Wheeler Transform* (BWT) [5] of w, denoted $\texttt{BWT}(w)$, is the array containing a permutation of the symbols of w which satisfies $\texttt{BWT}(w)[i] := w[\texttt{SA}(w)[i] - 1]$ if $\texttt{SA}(w)[i] > 1$, and $\texttt{BWT}(w)[i] := w[|w|]$ otherwise. The *Longest Common Prefix* table (LCP) [5] of w, denoted $\texttt{LCP}(w)$, is the array of integers such that $\texttt{LCP}(w)[i]$ equals the length of the longest common prefix between the suffixes of w starting at positions $\texttt{SA}(w)[i]$ and $\texttt{SA}(w)[i-1]$ if $i > 1$, and 0 otherwise.

For any string $s \in \Sigma^*$ and any symbol $c \in \Sigma$, one defines the functions denoted `rank` and `select` as follows: $\texttt{rank}_c(s, i)$ is the number of occurences of c in $s[1, i]$, and $\texttt{select}_c(s, j)$ is the position of the j^{th} occurence of c in s. The arrays $\texttt{BWT}(w)$ and $\texttt{LCP}(w)$ can be computed in $O(|w|)$ time [5, 10]. Simultaneously, one can compute `rank` and `select` for $\texttt{BWT}(w)$ at no additional cost and implement them such that any `rank` or `select` query takes constant time [13, 11, 18]. We use such state-of-the-art data structure to store a BWT.

Let C denote the array of length $\#\Sigma$ such that $C[c]$ equals the number of symbols of w that are alphabetically strictly smaller than c. The *Last-to-First column mapping* (LF) [10] of w is the function such that for any $1 \leq i \leq |w|$ one has $\texttt{LF}(w)[i] := C[\texttt{BWT}(w)[i]] + \texttt{rank}_{\texttt{BWT}(w)[i]}(\texttt{BWT}(w), i)$. It is proved that $\texttt{SA}(w)[\texttt{LF}(w)[i]] = \texttt{SA}(w)[i] - 1$ if $\texttt{SA}(w)[i] \geq 2$ and $\texttt{SA}(w)[\texttt{LF}(w)[i]] = |w|$ otherwise [5, 10].

BWT of a set of strings. From now on, let $P := (S, \sigma)$ be an ordered set of strings. We assume the symbol \$ is not in Σ and is alphabetically smaller than all other symbols. We denote by m_P the string obtained by concatenating the strings of $P.S := \{s_1, \ldots, s_{\#(P.S)}\}$ separated by a \$ and following the order $\overline{P.\sigma}$. i.e., $m_P := s_{\overline{P.\sigma}(1)}\$s_{\overline{P.\sigma}(2)}\$\ldots\$s_{\overline{P.\sigma}(n)}\$$ (See Figure 1). We extend the notion of $\texttt{BWT}$ of a string to an ordered set of strings P: the $\texttt{BWT}$ of P is the $\texttt{BWT}$ of the string m_P, i.e., $\texttt{BWT}(P) := \texttt{BWT}(m_P)$. We extend similarly the $\texttt{LF}$ function by setting that $\texttt{LF}(P) := \texttt{LF}(m_P)$ (by using $\texttt{rank}$ on $\texttt{BWT}(P)$).

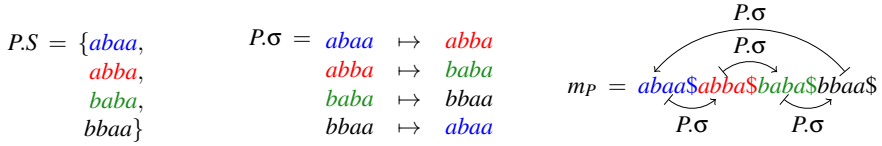


Figure 1 Running example with $P.S := \{abaa, abba, baba, bbaa\}$ and the corresponding m_P.

We define the *Longest Representative Suffix* table ($\texttt{LRS}$) of P as the array of $|m_P|$ integers satisfying: for any $i \in [\![1, |m_P|]\!]$ one has $\texttt{LRS}(P)[i] := \texttt{select}_\$(m_P[\texttt{SA}(m_P)[i] : |m_P|], 1) - 1$. The entry $\texttt{LRS}(P)[i]$ gives the length of the substring of m_P starting at position $\texttt{SA}(m_P)[i]$ up to, but not including the next \$. Using the $\texttt{LRS}$ table, we extend the notion of $\texttt{LCP}$ table to an ordered set of strings. For P, we set $\texttt{LCP}(P)[i] := \min(\texttt{LCP}(m_P)[i], \texttt{LRS}(P)[i])$ for any $i \in [\![1, |m_P|]\!]$. Using tables $\texttt{BWT}(P)$ and $\texttt{LCP}(m_P)$, we can compute the tables $\texttt{LRS}(P)$ and $\texttt{LCP}(P)$ in linear time in $|m_P|$ (see Appendix). Figure 2 illustrates the $\texttt{LCP}$ and $\texttt{LRS}$ tables.

2.3 Aho-Corasick automaton and eXtended Burrows-Wheeler Transform

Tree. Let $\mathcal{T}$ be a tree and u be a node of $\mathcal{T}$. Let $\perp$ denote the root of $\mathcal{T}$. We denote by $\texttt{Parent}_\mathcal{T}(u)$ the parent of u in $\mathcal{T}$, by $\texttt{Children}_\mathcal{T}(u)$ the set of children of u in $\mathcal{T}$, and by $\texttt{Leaves}_\mathcal{T}(u)$ the set of leaves in the subtree of u in $\mathcal{T}$. Let v be a leaf of $\mathcal{T}$; we denote by $\mathcal{B}_\mathcal{T}(v)$ the subtree of $\mathcal{T}$ containing all nodes comprised between $\perp$ and v included. As for a leaf v in the subtree of u in $\mathcal{T}$, $\#\texttt{Children}_{\mathcal{B}_\mathcal{T}(v)}(u) = 1$, we denote by $\texttt{Child}_{\mathcal{T}(v)}(u)$ the unique element of $\texttt{Children}_{\mathcal{B}_\mathcal{T}(v)}(u)$. Given a total order $\prec$ on $\texttt{Leaves}_\mathcal{T}(\perp)$, we extend this order on the set $\texttt{Children}_\mathcal{T}(v)$ for any node v of $\mathcal{T}$ by: for any x, y in $\texttt{Children}_\mathcal{T}(v)$, $x \prec y$ **iff** $\min_{x' \in \texttt{Leaves}_\mathcal{T}(x)} x' \prec \min_{y' \in \texttt{Leaves}_\mathcal{T}(y)} y'$.

Aho-Corasick tree. The *Aho-Corasick automaton* ($\texttt{AC}$) [1] of a set of strings S is a digraph whose set of nodes is the set of all prefixes of the strings of S. This graph is composed of two trees on the same node set. The first tree, which we called the *Aho-Corasick Tree* ($\texttt{ACT}$), has an arc from a prefix u to a different prefix v iff u is the longest proper prefix of v in $\texttt{Prefix}(S)$ (see Figure 4). The second tree, termed *Aho-Corasick Failure link* ($\texttt{ACFL}$), has an arc from a prefix u to a different prefix v iff v is the longest proper **suffix** of u in $\texttt{Prefix}(S)$.

eXtended Burrows-Wheeler Transform. Let $\mathcal{T}$ be an ordered tree (i.e., with a total order on the set of children of each node) such that every node of $\mathcal{T}$ is labelled with a symbol from an alphabet Σ. We define the functions δ and π on the set of nodes of $\mathcal{T}$ except for the root such that for a node v of $\mathcal{T}$, $\delta(v)$ is the label of v, and $\pi(v)$ is the string obtained by concatenating the labels from v's parent up to the root of $\mathcal{T}$. Let $\prec$ be the total order between the nodes of $\mathcal{T}$ such that for u and v two nodes of $\mathcal{T}$, $u \prec v$ iff $\pi(u)$ is strictly lexicographically smaller than $\pi(v)$ or u is before v in the order of $\mathcal{T}$.

LCP(P)	LRS(P)	SA(m_P)																			BWT(P)
0	0	20	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a
0	0	5	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a
0	0	10	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a
0	0	15	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a
0	1	19	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a
1	1	4	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a
1	1	9	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b
1	1	14	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b
1	2	18	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b
2	2	3	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b
1	3	12	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b
3	4	1	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$
2	4	6	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$
0	2	8	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b
2	2	13	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a
2	3	17	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b
3	3	2	b	a	a	$	a	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a
2	4	11	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$
1	3	7	b	b	a	$	b	a	b	a	$	b	b	a	a	$	a	b	a	a	$	a
3	4	16	b	b	a	a	$	a	b	a	a	$	a	b	b	a	$	b	a	b	a	$

Figure 2 Tables $\texttt{LCP}(P)$, $\texttt{LRS}(P)$, $\texttt{SA}(m_P)$ and $\texttt{BWT}(P)$ for the running example. One has $\texttt{LCP}(P)[10] = \texttt{LRS}(P)[10] = 2$ although the longest common prefix between suffixes of rank 9 and 10 is $aa\$ab$ of length 5 (i.e., although $\texttt{LCP}(m_P)[10] = 5$).

▶ **Example 1.** With the tree of Figure 3, we have $\delta(\textcircled{13}) = b$ and $\pi(\textcircled{13}) = aba = \delta(\textcircled{12})\delta(\textcircled{8})\delta(\textcircled{1})$, and also $\delta(\textcircled{4}) = b$ and $\pi(\textcircled{4}) = baa = \delta(\textcircled{3})\delta(\textcircled{2})\delta(\textcircled{1})$. Thus, $\textcircled{13} \prec \textcircled{4}$ since $\pi(\textcircled{13}) < \pi(\textcircled{4})$.

The *Prefix Array* (PA) of an ordered tree $\mathcal{T}$ is the array of pointers to the nodes of $\mathcal{T}$ (except the root of $\mathcal{T}$) sorted in $\prec$ order. The *eXtended Burrows-Wheeler Transform* (XBWT) [9][1] of a tree $\mathcal{T}$ is an array of symbols of Σ of length $\#\texttt{PA}(\mathcal{T})$, such that the entry at position i gives the label of the node $\texttt{PA}(\mathcal{T})[i]$. The *eXtended Burrows-Wheeler Last* (XBWL) [9][1] of a tree $\mathcal{T}$ is the bit array of length $\#\texttt{PA}(\mathcal{T})$ such that $\texttt{XBWL}(\mathcal{T})[i]$ equals 1 if the node $\texttt{PA}(\mathcal{T})[i]$ is the last child of its parent, and 0 otherwise. Figure 5a illustrates XBWT and XBWL.

eXtended Burrows-Wheeler Transform of Aho-Corasick automaton. For a set of strings $S := \{s_1, \ldots, s_n\}$, we denote by $S^\$$ the set $\{s_1\$, \ldots, s_n\$\}$. Let P be an ordered set of strings. We define $\texttt{ACT}(P)$ as the variant of the Aho-Corasick tree of $P.S^\$$ where the label of an arc is shifted on the deepest node of this arc (i.e., the label of a node v of $\texttt{ACT}(P)$ is the label of (u, v) in $\texttt{ACT}(P.S)$ where u is the parent of v in $\texttt{ACT}(P)$) and equipped with the order $\lhd_{P.\sigma}$. Indeed, $\lhd_{P.\sigma}$ is the order on the leaves satisfying: for u and v two leaves of $\texttt{ACT}(P)$, $u \lhd_{P.\sigma} v$

[1] In [9], the XBW-*transform* is defined as $\texttt{XBW}(T)[i] := \langle \texttt{XBWT}[i], \texttt{XBWL}[i] \rangle$, for any position i.

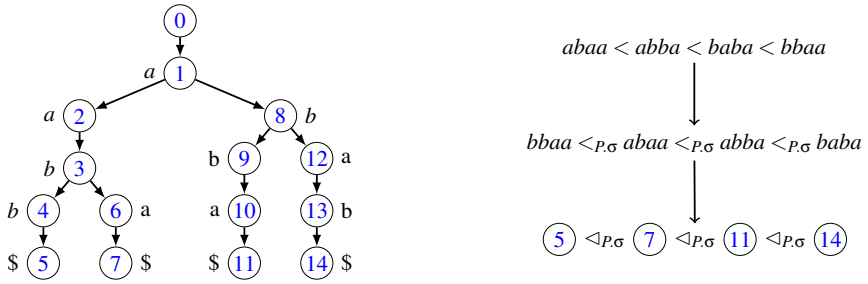


Figure 3 Tree $\texttt{ACT}(\overleftarrow{P})$ for the running example, and links between the orders $<$, $<_{P.\sigma}$, and $\lhd_{P.\sigma}$.

iff $\pi(u) <_{P.\sigma} \pi(v)$. We extend this order to the set of children of all nodes (see Figure 3). Note that $\texttt{ACT}(P)$ differs from $\texttt{ACT}(P.S)$ (defined above) to correspond to an input of the eXtended Burrows-Wheeler Transform.

3 Link between BWT and XBW of Aho-Corasick automaton

Here, we introduce a decomposition of a multi-string BWT that leads us to exhibit a bijection with the Aho-Corasick automaton (see Proposition 2). This builds on and extends Manzini's work [22]. We extend this bijection to the XBWT of the Aho-Corasick automaton (see Figure 4). For space reasons, many proofs are given in the Appendix (or the preprint [6]).

Link between BWT and AC. Given an ordered set of strings P, $\texttt{Decomp\_BWT}(P)$ is the integer interval partition of $[\![1, |m_P|]\!]$ such that

$$[\![i,j]\!] \in \texttt{Decomp\_BWT}(P) \quad \textbf{iff} \quad \begin{cases} \texttt{LCP}(P)[k] \neq \texttt{LRS}(P)[k], & \text{for } k \in \{i, j+1\} \\ \texttt{LCP}(P)[k] = \texttt{LRS}(P)[k], & \text{for } k \in [\![i+1, j]\!]. \end{cases}$$

We define Dec_Pre as the function from $\texttt{Decomp\_BWT}(P)$ to Σ^* such that

$$\texttt{Dec\_Pre}[u] := \overleftarrow{m_P[\texttt{SA}(m_P)[i] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]},$$
$$\text{for all } u := [\![i,j]\!] \in \texttt{Decomp\_BWT}(P).$$

▶ **Proposition 2.** Dec_Pre *is a bijection between* $\texttt{Decomp\_BWT}(P)$ *and* $\texttt{Prefix}(\overleftarrow{P.S})$.

Let us start the proof of Proposition 2 with the following Lemma.

▶ **Lemma 3.** *Let* $u = [\![i,j]\!] \in \texttt{Decomp\_BWT}(P)$. *For all* $k \in [\![i,j]\!]$,

$$\texttt{Dec\_Pre}[u] \;=\; \overleftarrow{m_P[\texttt{SA}(m_P)[k] : \texttt{SA}(m_P)[k] + \texttt{LRS}(P)[k] - 1]}.$$

Proof. Let us show by contraposition that $\texttt{LRS}(P)[k-1] = \texttt{LCP}(P)[k]$ for all $k \in [\![i+1, j]\!]$. Assume that there exists $k \in [\![i+1, j]\!]$ such that $\texttt{LRS}(P)[k-1] \neq \texttt{LCP}(P)[k]$. Whenever $\texttt{LRS}(P)[k-1] < \texttt{LCP}(P)[k]$, we get by definition that $\texttt{LRS}(P)[k] \geq \texttt{LCP}(P)[k]$, and thus $\texttt{LRS}(P)[k-1] < \texttt{LRS}(P)[k]$. By the definiton of $\texttt{LRS}(P)$, $m_P[\texttt{SA}(m_P)[k-1] + \texttt{LRS}(P)[k-1]] = \$$. By the definition of $\texttt{LCP}(P)$, for all $j \in [\![1, \texttt{LCP}(P)[k-1] - 1]\!]$, $m_P[\texttt{SA}(m_P)[k] + j] = m_P[\texttt{SA}(m_P)[k-1] + j]$. As $\texttt{LRS}(P)[k-1] < \texttt{LCP}(P)[k]$, we have $m_P[\texttt{SA}(m_P)[k] + \texttt{LRS}(P)[k-1]] = m_P[\texttt{SA}(m_P)[k-1] + \texttt{LRS}(P)[k-1]] = \$$, which is impossible since $\texttt{LRS}(P)[k-1] < \texttt{LRS}(P)[k]$. Whenever $\texttt{LRS}(P)[k-1] > \texttt{LCP}(P)[k]$, as $\texttt{LCP}(P)[k] = \texttt{LRS}(P)[k]$, the string $m_P[\texttt{SA}(m_P)[k] : |m_P|]$ is lexicographically strictly smaller than $m_P[\texttt{SA}(m_P)[k-1] : |m_P|]$, which is impossible. This concludes the proof. ◀

Proof. Let $u = [\![i, j]\!]$ be an interval of $\texttt{Decomp\_BWT}(P)$. First, we prove that $\texttt{Dec\_Pre}[u] \in \texttt{Prefix}(\overleftarrow{P.S})$, and then to prove the bijection, we show $\texttt{Dec\_Pre}$ is injective and surjective. By definition, $\texttt{LRS}(P)[i] = \texttt{select}_{\$}(m_P[\texttt{SA}(m_P)[i] : |m_P|], 1) - 1$, we get that $m_P[\texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i]] = \$$ and for any $j \in [\![\texttt{SA}(m_P)[i], \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]\!]$, $m_P[j] \neq \$$. Hence, $m_P[\texttt{SA}(m_P)[i] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]$ is a suffix of a string w of $P.S$, and thus $\texttt{Dec\_Pre}[u]$ is a prefix of $\overleftarrow{w}$ in $\overleftarrow{P.S}$.

Injectiveness. Let $u_1 = [\![i_1, j_1]\!]$ and $u_2 = [\![i_2, j_2]\!]$ be two elements of $\texttt{Decomp\_BWT}(P)$. Without loss of generality, we take $i_1 \leq i_2$. Assume that $\texttt{Dec\_Pre}[u_1] = \texttt{Dec\_Pre}[u_2]$, we have that $m_P[\texttt{SA}(m_P)[i_1] : \texttt{SA}(m_P)[i_1] + \texttt{LRS}(P)[i_1] - 1] = m_P[\texttt{SA}(m_P)[i_2] : \texttt{SA}(m_P)[i_2] + \texttt{LRS}(P)[i_2] - 1]$ and thus for all $k \in [\![i_1, i_2]\!]$, $m_P[\texttt{SA}(m_P)[i_1] : \texttt{SA}(m_P)[i_1] + \texttt{LRS}(P)[i_1] - 1] = m_P[\texttt{SA}(m_P)[k] : \texttt{SA}(m_P)[k] + \texttt{LRS}(P)[k] - 1]$. Hence, we have $\texttt{LCP}(P)[k] = \texttt{LRS}(P)[i_1]$ and $\texttt{LRS}(P)[k] = \texttt{LRS}(P)[i_1]$. Therefore by the definition of $\texttt{Decomp\_BWT}(P)$, we get $u_1 = u_2$.

Surjectiveness. Let v be a prefix of a string of $\overleftarrow{P.S}$. The string $\overleftarrow{v}$ is a suffix of a string of $P.S$. By the definition of m_P, $\overleftarrow{v}$ is a prefix of a suffix s of m_P such that $s[|\overleftarrow{v}| + 1] = \$$. By the definition of $\texttt{BWT}(P)$, the table $\texttt{SA}(m_P)$ gives, for a position i, the starting position of the i^{th} suffix of m_P in lexicographic order. Hence, there is a bijection between $\texttt{Suffix}(m_P)$ and the set of positions in $\texttt{SA}(m_P)$. Let $k \in [\![1, |m_P|]\!]$ such that $s = m_P[\texttt{SA}(m_P)[k] : |m_P|]$. As $\overleftarrow{v}$ is a suffix of a string of $P.S$ and a prefix of s, we have $\overleftarrow{v} = m_P[\texttt{SA}(m_P)[k] : \texttt{SA}(m_P)[k] + \texttt{LRS}(P)[k] - 1]$. We take $u = [\![i, j]\!] \in \texttt{Decomp\_BWT}(P)$ such that $k \in [\![i, j]\!]$. By Lemma 3, $\overleftarrow{v} = \texttt{Dec\_Pre}[u]$. ◀

By Proposition 2, there exists an integer interval partition of $[\![1, |m_P|]\!]$ (i.e., $\texttt{Decomp\_BWT}(P)$) that is in bijection with the set of nodes of $\texttt{AC}(\overleftarrow{P.S})$. The following theorem extends this bijection by building an isomorphic graph of $\texttt{AC}(P.S)$ whose set of nodes is $\texttt{Decomp\_BWT}(\overleftarrow{P})$. This states how to simulate an Aho-Corasick automaton using the $\texttt{BWT}$ (similarly to [22]).

▶ **Theorem 4.** *Using tables* $\texttt{BWT}(\overleftarrow{P})$, $\texttt{LCP}(\overleftarrow{P})$, $\texttt{LRS}(\overleftarrow{P})$ *and the function* $\texttt{LF}(\overleftarrow{P})$, *we can build the graph* $(\texttt{Decomp\_BWT}(\overleftarrow{P}), A_T(\overleftarrow{P}) \cup A_F(\overleftarrow{P}))$ *that is isomorphic to* $\texttt{AC}(P.S)$.

$$A_T(P) := \{(u, v) \in \texttt{Decomp\_BWT}(P)^2 |\ \exists x \in u \text{ such that } \texttt{LF}(P)[x] \neq 0 \text{ and } \texttt{LF}(P)[x] \in v\},$$

$$A_F(P) := \{(u, v) \in \texttt{Decomp\_BWT}(P)^2 |\ \Big(\max_{\substack{k<i \\ \texttt{LRS}(P)[k] = \min_{k<l\leq i}(\texttt{LCP}(P)[l])}} (k) \Big) \in v \text{ with } u := [\![i, j]\!]\}.$$

More precisely, the graph $(\texttt{Decomp\_BWT}(\overleftarrow{P}), A_T(\overleftarrow{P}))$ is isomorphic to the tree $\texttt{ACT}(P.S)$ and the graph $(\texttt{Decomp\_BWT}(\overleftarrow{P}), A_F(\overleftarrow{P}))$ is isomorphic to the tree $\texttt{ACFL}(P.S)$ (see Appendix).

Link between BWT and XBW. After giving a new proof of the relation between Aho-Corasick automaton and $\texttt{BWT}$, we exhibit a new (bijective) link between the $\texttt{BWT}$ and the $\texttt{XBW}$, which takes into account the order in the multi-string (Theorem 5). This leads to both, another construction algorithm of the $\texttt{XBW}$ from the $\texttt{BWT}$, and to a construction of the $\texttt{BWT}$ from the $\texttt{XBW}$, and thereby extends Manzini's results (Corollary 6).

Alike the definition of $\texttt{Decomp\_BWT}(P)$ for an ordered set of strings P, we define $\texttt{Decomp\_XBW}(\mathcal{T})$ of a tree $\mathcal{T}$ of t nodes as the integer interval partition of $[\![1, t]\!]$ such that

$$[\![i, j]\!] \in \texttt{Decomp\_XBW}(\mathcal{T}) \quad \textbf{iff} \quad \begin{cases} \texttt{XBWL}(\mathcal{T})[k] = 1, & \text{for } k \in \{i - 1, j\} \\ \texttt{XBWL}(\mathcal{T})[k] = 0, & \text{for } k \in [\![i, j - 1]\!]. \end{cases}$$

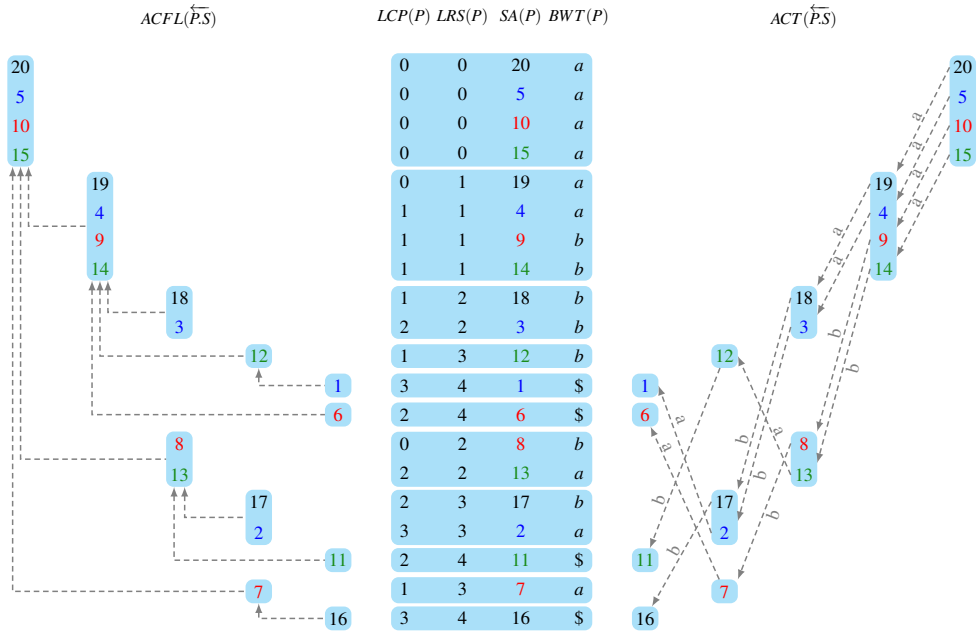


Figure 4 Link between $\texttt{BWT}(P)$, $\texttt{ACT}(\overleftarrow{P.S})$ and $\texttt{ACFL}(\overleftarrow{P.S})$ for the running example. The blocks in the tables SA, LCP, LRS and BWT show the decomposition $\texttt{Decomp\_BWT}(P)$.

▶ **Theorem 5** (See Figure 5b). *There exists a bijection, denoted* $\texttt{BWT\_XBW}$*, between* $\texttt{Decomp\_BWT}(P)$ *and* $\texttt{Decomp\_XBW}(\texttt{ACT}(\overleftarrow{P}))$ *such that for all* $u \in \texttt{Decomp\_BWT}(P)$ *with* $u := [\![i, j]\!]$, $\texttt{BWT\_XBW}(u) := [\![i', j']\!]$ *and* $z := \texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P})[i']))$*:*

(1) *Let* $\{y_1, y_2, \dots y_{\#u}\} := \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(z)$ *such that* $y_k \lhd_{P.\sigma} y_l \Rightarrow k < l$*; then*

$$\texttt{BWT}(P)[i, j] = \delta[\texttt{Child}_{\texttt{ACT}(\overleftarrow{P})(y_1)}(z)]\delta[\texttt{Child}_{\texttt{ACT}(\overleftarrow{P})(y_2)}(z)] \dots \delta[\texttt{Child}_{\texttt{ACT}(\overleftarrow{P})(y_{\#u})}(z)].$$

(2) *Let* $\{x_1, x_2, \dots, x_{\#(\texttt{BWT\_XBW}(u))}\} := \texttt{Children}_{\texttt{ACT}(\overleftarrow{P})}(z)$ *such that* $x_k \lhd_{P.\sigma} x_l \Rightarrow k < l$*; then*

$$\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))[i', j'] = \delta[x_1]\delta[x_2] \dots \delta[x_{\#(\texttt{BWT\_XBW}(u))}].$$

Theorem 5 provides us with a strong link between $\texttt{BWT}(P)$ and $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$, which allows transforming one structure into the other. This leads to the following corollary.

▶ **Corollary 6.** *Let* P *be an ordered set of strings.*

(1) *Using tables* $\texttt{BWT}(P)$, $\texttt{LCP}(P)$ *and* $\texttt{LRS}(P)$ *of an ordered set of strings* P*, we can build the tables* $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ *and* $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$ *in linear time of* $\|P.S\| \times \#\Sigma$*.*

(2) *Using tables* $\texttt{XBWT}(\texttt{ACT}(S))$ *and* $\texttt{XBWL}(\texttt{ACT}(S))$ *of a set of strings* S*, we can build the tables* $\texttt{BWT}(\overleftarrow{P})$, $\texttt{LCP}(\overleftarrow{P})$ *and* $\texttt{LRS}(\overleftarrow{P})$ *in linear time of* $\|S\| \times \#\Sigma$ *where* P *is an ordered set of strings such that* $P.S = S$*.*

The idea behind the algorithms is to exploit the link of Theorem 5 to compute each substring of the BWT or of the XBWT associated to each element of $\texttt{Decomp\_BWT}$ or of $\texttt{Decomp\_XBW}$.

4 Optimal ordering of strings for maximising compression

In this section, we show how to use Theorem 5 to compute the permutation that leads to a BWT and a XBWT having the minimum Run Length Encoding.

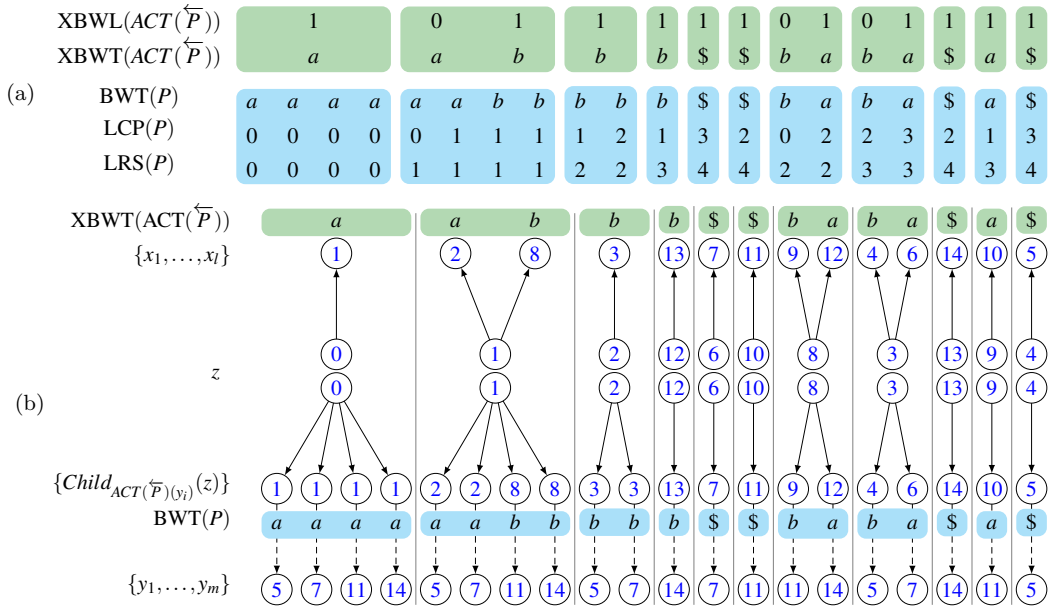


Figure 5 (a) Link between `Decomp_BWT`(P) (in blue) and `Decomp_XBW`$(\texttt{ACT}(\overleftarrow{P}))$ (in green). (b) Illustration of Theorem 5 for the running example.

4.1 Minimum permutation problem for BWT and XBWT

Run-Length Encoding [24] is a widely used method to compress strings. For a string w, the Run-Length Encoding splits w into the minimum number of substrings containing a single symbol. The size of the Run-Length Encoding of w is the cardinality of the minimum decomposition. For example for $abbaaaccabbb = a^1b^2a^3c^2a^1b^3$ (where the power notation α^n means n copies of symbol α), the size of the Run-Length Encoding is 6 (for the decomposition has 6 blocks).

We define Run-Length measures: d_B for a `BWT`, and d_X for a `XBW` (similar to those of [15]). For P an ordered set of strings, let $d_B(P)$ be the cardinality of the set $\{i \in [\![1, \#\texttt{BWT}(P) - 1]\!] \mid \texttt{BWT}(P)[i] \neq \texttt{BWT}(P)[i+1]\}$. Similarly, let $d_X(P)$ be the cardinality of the set $\{i \in [\![1, \#\texttt{XBWT}(\texttt{ACT}(P)) - 1]\!] \mid \texttt{XBWT}(\texttt{ACT}(P))[i] \neq \texttt{XBWT}(\texttt{ACT}(P))[i+1]\}$.

Given two ordered sets of strings, P_1 and P_2 such that $P_1.S = P_2.S$ (i.e., they contain the same set of strings), Theorem 5 implies that their `BWT` may differ, and thus $d_B(P_1)$ and $d_B(P_2)$ may also differ. We define the following minimisation problems. As the Run-Length Encoding of $\texttt{BWT}(P)$ has size $d_B(P)+1$, finding an optimal solution of `Min-Permutation-BWT` can help compressing $\texttt{BWT}(P)$.

▶ **Definition 7** (`Min-Permutation-BWT` and `Min-Permutation-XBWT`). *Let S be a set of strings. The problem* `Min-Permutation-BWT` *asks for an ordered set of strings P that minimises $d_B(P)$ and such that $P.S = S$. The problem* `Min-Permutation-XBWT` *asks for an ordered set of strings P that minimises $d_X(P)$ and such that $P.S = S$.*

To simplify `Min-Permutation-BWT`, we consider specific ordered sets of strings. Let P be an ordered set of strings and let $\perp$ denote the root of $\texttt{ACT}(\overleftarrow{P})$. We say that P is *topologically planar* if for each node u in $\texttt{ACT}(\overleftarrow{P})$ and $v \in \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(\perp) \setminus \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$, there does not exists u_1 and u_2 in $\texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$ such that $u_1 \lhd_{P.\sigma} v \lhd_{P.\sigma} u_2$. In other words, P is *topologically planar* if we can draw the tree $\texttt{ACT}(\overleftarrow{P})$ by ordering the leaves with $\lhd_{P.\sigma}$ without arcs crossing each other.

Let P be an ordered set of strings, which is not necessarily topologically planar. We denote by P_{tp} the ordered set of strings such that $P_{tp}.S = P.S$, and such that $P_{tp}.\sigma$ satisfies for any u in $\texttt{ACT}(\overleftarrow{P})$, any $v \in \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(\bot) \setminus \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$, and any u_1, u_2 in $\texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$ such that $u_1 \lhd_{P.\sigma} v \lhd_{P.\sigma} u_2$, we have $u_1 \lhd_{P_{tp}.\sigma} u_2 \lhd_{P_{tp}.\sigma} v$. As we have a bijection between the set of circular permutations of $P.S$ and the set of leaves of $\texttt{ACT}(\overleftarrow{P})$, we can unambiguously define the ordered set of strings P_{tp} that is topologically planar.

▶ **Proposition 8.** *Let P be an ordered set of strings. We have $d_B(P_{tp}) \leq d_B(P)$ and $d_B(P_{tp}) = d_X(\overleftarrow{P_{tp}})$.*

Proof. For the inequality, let us prove that any modification of the order used to create P_{tp} decreases the value of d_B. Let P be an ordered set of strings that is not topologically planar. Let u in $\texttt{ACT}(\overleftarrow{P})$, $v \in \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(\bot) \setminus \texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$, and let u_1 and u_2 in $\texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(u)$ such that $u_1 \lhd_{P.\sigma} v \lhd_{P.\sigma} u_2$. Let P' be the copy of P where the only difference is $u_1 \lhd_{P_{tp}.\sigma} u_2 \lhd_{P_{tp}.\sigma} v$. Let $x \in \texttt{Decomp\_BWT}(P)$ such that $\texttt{Dec\_Pre}[x] = u$. By Theorem 5, for all $y \in \texttt{Decomp\_BWT}(P) \setminus \{x\}$, we have $\texttt{BWT}(P)[y] = \texttt{BWT}(P')[y]$, $\texttt{BWT}(P)[x] = \ldots\delta[u_1]\delta[v_1]\delta[u_2]\ldots$ and $\texttt{BWT}(P')[x] = \ldots\delta[u_1]\delta[u_2]\ldots\delta[v_1]\ldots$ with $v_1 = \texttt{Child}_{\texttt{ACT}(\overleftarrow{P})(v)}(u)$. As $\delta[u_1] = \delta[u_2]$ and $\delta[u_1] \neq \delta[v_1]$, we have $d_B(P') \leq d_B(P)$.

For the equality, it is enough to see that for any element u in $\texttt{Decomp\_BWT}(P_{tp})$, the numbers of distinct successive symbols are identical in $\texttt{BWT}(P_{tp})[u]$ and in $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P_{tp}}))[\texttt{BWT\_XBW}[u]]$. Thus, for two successive elements $u = [\![i_1, j_1]\!]$ and $v = [\![i_2, j_2]\!]$ of $\texttt{Decomp\_BWT}(P_{tp})$, we obtain an equivalence between $\texttt{BWT\_XBW}[u] = [\![i'_1, j'_1]\!]$ and $\texttt{BWT\_XBW}[v] = [\![i'_2, j'_2]\!]$:

$$\texttt{BWT}(P_{tp})[j_1] = \texttt{BWT}(P_{tp})[i_2] \quad \text{iff} \quad \texttt{BWT}(\texttt{ACT}(\overleftarrow{P_{tp}}))[j'_1] = \texttt{BWT}(\texttt{ACT}(\overleftarrow{P_{tp}}))[i'_2].$$ ◀

Thanks to Proposition 8, we can restrict the search to ordered sets of strings that are topologically planar when solving Min-Permutation-BWT or Min-Permutation-XBWT. Furthermore, an optimal solution of Min-Permutation-BWT for S is also an optimal solution of Min-Permutation-XBWT for $\overleftarrow{S}$, and vice versa. This yields the following theorem.

▶ **Theorem 9.** *Let S be a set of strings. We can find an optimal solution for* Min-Permutation-BWT *and for* Min-Permutation-XBWT *in $O(\|S\| \times \#\Sigma)$ time.*

4.2 Proof of Theorem 9

As a reminder, Proposition 8 states that an optimal solution of Min-Permutation-BWT is also an optimal solution of Min-Permutation-XBWT, and vice versa. In the following of this proof, we only prove the result regarding Min-Permutation-XBWT.

To start, let us give an overview of algorithm:

1. we take a random permutation σ of S and define P such that $P.S = S$ and $P.\sigma = \sigma$,
2. we build $\texttt{ACT}(P)$, $\texttt{XBWT}(\texttt{ACT}(P))$ and $\texttt{Decomp\_XBW}(\texttt{ACT}(P))$,
3. we find P' which is an optimal solution of Min-Permutation-XBWT.

In the following, we define the problem Min-Permutation-Table and explicit its link to Min-Permutation-XBWT (Lemma 10). Lemma 10 gives us a linear time algorithm for finding an optimal solution of Min-Permutation-Table, and thus we can apply this algorithm to obtain an optimal solution for Min-Permutation-XBWT.

Given A an array of symbols of Σ, we define $\texttt{Char}(A)$ as the set of (different) symbols in A. Given T an array of n symbols of Σ and D an integer interval partition of $[\![1, n]\!]$ such for each interval $[\![i, j]\!]$ of D, $\#(\texttt{Char}(T[i, j])) = j - i + 1$ (i.e., all the symbols of

$T[i,j]$ are different), the problem `Min-Permutation-Table` is to find a T' such that for all $[\![i,j]\!] \in D$, $\#(\texttt{Char}(T[i,j])) = \#(\texttt{Char}(T'[i,j]))$ and which minimises $d_A(T', D) := \#\{i \in [\![1, n-1]\!] \mid T'[i] \neq T'[i+1]\}$. A proof of the following lemma is given in Appendix.

▶ **Lemma 10.** *Let S be a set of strings and let P be an ordered set of strings such that $P.S = S$. For an optimal solution T' of* `Min-Permutation-Table` *for* $\texttt{XBWT}(\texttt{ACT}(P))$ *and for* $\texttt{Decomp\_XBW}(\texttt{ACT}(P))$*, there exists an optimal solution P' of* `Min-Permutation-XBWT` *for S such that* $\texttt{XBWT}(\texttt{ACT}(P')) = T'$.

Let T be an array of n symbols of Σ and let D be an integer interval partition of $[\![1,n]\!]$ such for each interval $[\![i,j]\!]$ of D, $\#(\texttt{Char}(T[i,j])) = j - i + 1$. Let $A(D)$ be the array of all intervals in D in the order $<$ and $B(T,D)$ the array of size $\#A(D) - 1$ such that the position i of $B(T,D)$ is $B(T,D)[i] = \texttt{Char}(A(D)[i]) \cap \texttt{Char}(A(D)[i+1])$. For any set of symbols $C := \{c_1, \ldots, c_m\}$ where $c_1 < \ldots < c_m$, we define $\texttt{word}(C)$ as the string $c_1 \ldots c_m$.

▶ **Lemma 11.** *Let T be an array of n symbols of Σ and let D be an integer interval partition of $[\![1,n]\!]$ such for any interval $[\![i,j]\!]$ of D, $\#(\texttt{Char}(T[i,j])) = j - i + 1$.*

- *If there exists i in $[\![1,n]\!]$ such that $[\![i,i]\!] \in D$, then $T'_1[1, i-1]T'_2$ is an optimal solution of* `Min-Permutation-Table` *for T and for D, where T'_1 is an optimal solution of* `Min-Permutation-Table` *for $T[1,i]$ and for $\{[\![i',j']\!] \in D \mid j' \leq i\}$, and T'_2 is an optimal solution of* `Min-Permutation-Table` *for $T[i,n]$ and for $\{[\![i',j']\!] \in D \mid i' \geq i\}$.*
- *If there exists i in $[\![1, \#B(T,D)]\!]$ such that $\#(B(T,D)[i]) = 0$, then $T'_1T'_2$ is an optimal solution of* `Min-Permutation-Table` *for T and for D, where T'_1 is an optimal solution of* `Min-Permutation-Table` *for $T[1, A(D)[i][1]]$ and for $\{[\![i',j']\!] \in D \mid j' \leq A(D)[i][1]\}$, and T'_2 is an optimal solution of* `Min-Permutation-Table` *for $T[A(D)[i+1][0], n]$ and for $\{[\![i',j']\!] \in D \mid i' \geq A(D)[i+1][0]\}$.*
- *If there exists i in $[\![1, \#B(T,D)]\!]$ such that $\#(B(T,D)[i]) = 1$, then $T'_1aaT'_2$ is an optimal solution of* `Min-Permutation-Table` *for T and for D, where $B(T,D)[i] = \{a\}$, T'_1 is an optimal solution of* `Min-Permutation-Table` *for $T[1, A(D)[i-1][1]]\texttt{word}(\texttt{Char}(A(D)[i] \setminus \{a\})$ and for $\{[\![i',j']\!] \in D \mid j' < A(D)[i][1]\} \cup \{[\![A(D)[i][0], A(D)[i][1]-1]\!]\}$, and T'_2 is an optimal solution of* `Min-Permutation-Table` *for $\texttt{word}(\texttt{Char}(A(D)[i+1] \setminus \{a\})T[A(D)[i+2][0], n]$ and for $\{[\![i',j']\!] \in D \mid i' > A(D)[i+1][0]\} \cup \{[\![A(D)[i+1][0]+1, A(D)[i][1]]\!]\}$.*

Proof. All the proofs are derived from the equality $d_A(T[1,n], D) = d_A(T[1,i], \{[\![i',j']\!] \in D \mid j' \leq i\}) + d_A(T[i,n], \{[\![i',j']\!] \in D \mid i' \geq i\})$ for all $i \in [\![1,n]\!]$. ◀

▶ **Lemma 12.** *Let T be an array of n symbols of Σ and let D be an integer interval partition of $[\![1,n]\!]$ such that for any interval $[\![i,j]\!]$ of D, $\#(\texttt{Char}(T[i,j])) = j - i + 1$. Whenever for any $i \in [\![1, \#B(T,D)]\!]$ one has $\#(B(T,D)[i]) \geq 2$, Algorithm 1 gives an optimal solution for* `Min-Permutation-Table` *in $O(\#T \times \#\Sigma)$ time.*

Proof.

Complexity. To build the tables $A(D)$ and $B(T,D)$, we need $O(\#T \times \#\Sigma)$ time. As both tables are smaller than T, the **for** loop of Algorithm 1 also takes $O(\#T \times \#\Sigma)$ time.

Optimality. As for any i in $[\![1, \#B(T,D)]\!]$, one has $\#(B(T,D)[i]) \geq 2$, we get $B(T,D)[i] \setminus \{last\} \neq \emptyset$ (within the **for** loop of Algorithm 1). Let T^* be a string such that for all $[\![i,j]\!] \in D$, $\texttt{Char}(T[i,j]) = \texttt{Char}(T^*[i,j])$. The size of T^* is $\#T$ and the number of intervals of D is $\#D$, i.e., the maximum number of positions where two consecutive letters can be identical. Hence, we have $d_A(T^*, D) \leq \#T - \#D + 1$. Let T' be the string given by Algorithm 1. We have $d_A(T', D) = \sum_{u \in D}(\#u - 1) + 1 = \#T - \#D + 1$. This concludes the proof. ◀

Algorithm 1: Computation of an array T' of symbols satisfying for any $[\![i,j]\!] \in D$, $\texttt{Char}(T[i,j]) = \texttt{Char}(T'[i,j])$.

Input : An instance of `Min-Permutation-Table` T and D
Output : A string T'
$last \leftarrow \$$ such that $\$ \notin \Sigma$;
$T' \leftarrow$ empty string;
for $i \in [\![1, \#B(T,D)]\!]$ **do**
  $lettre \leftarrow$ random element of $B(T,D)[i] \setminus \{last\}$;
  $T' \leftarrow T'\ \texttt{word}(\texttt{Char}(A(D)[i]) \setminus \{lettre, last\})$;
  $T' \leftarrow T'\ lettre\ lettre$;
  $last \leftarrow lettre$;
$T' \leftarrow T'\ \texttt{word}(\texttt{Char}(A(D)[\#A(D)]) \setminus \{last\})$;
return T';

▶ **Lemma 13.** *Let T be an array of n symbols of Σ and let D be an integer interval partition of $[\![1,n]\!]$ such that for each interval $[\![i,j]\!]$ of D, $\#(\texttt{Char}(T[i,j])) = j - i + 1$. The problem* `Min-Permutation-Table` *can be solved in $O(\#T \times \#\Sigma)$ time.*

Proof. By Lemma 12 and Lemma 11, we can compute an optimal solution of `Min-Permutation-Table` by splitting the interval, applying Algorithm 1 on each part, and then merging the strings output by Algorithm 1. ◀

5 Conclusion and Perspectives

Here, we present a new view of the Burrows-Wheeler Transform: as the text representation of an Aho-Corasick automaton that depends on the concatenation order. This induces a link between the Burrows-Wheeler Transform and the eXtended Burrows-Wheeler Transform, via the Aho-Corasick automaton. This link allows one to transform one structure into the other (for which we provide algorithms). We also exploit this link to find in linear time an ordering of input strings that optimises the compression of the concatenated strings.

Of course, it would be interesting to evaluate even empirically the gain of this compression on real life data. In bioinformatics, one wishes to index a collection of genomes from indidual of the same species. For instance, it can be all variants of a virus genome within a host (*e.g.*, HIV or Ebola virus in human); their number and relative frequency may change along time [2]. An application is then to compare the sequencing reads obtained from any new infected individual with this index to determine whether some variants are becoming more frequent or resistant to some treatment. Some viruses evolve rapidly to circumvent immune response, the number of potential variants can be large. Hence, it is practically relevant to reduce the index space by finding an optimal permutation of the genomes that would maximised the Run-Length Encoding of the index.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

2 Jasmijn A. Baaijens, Amal Zine El Aabidine, Eric Rivals, and Alexander Schönhuth. De novo assembly of viral quasispecies using overlap graphs. *Genome Research*, 27(5):835–848, 2017.

3 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

4 Djamal Belazzougui. Succinct Dictionary Matching with No Slowdown. In *Combinatorial Pattern Matching*, pages 88–100, 2010.

5 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

6 Bastien Cazaux and Eric Rivals. Strong link between BWT and XBW via Aho-Corasick automaton and applications to Run-Length Encoding. *CoRR*, abs/1805.10070, 2018. `arXiv:1805.10070`.

7 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler Transform. *Bioinformatics*, 28(11):1415–1419, 2012.

8 Lavinia Egidi and Giovanni Manzini. Lightweight BWT and LCP Merging via the Gap Algorithm. In *String Processing and Information Retrieval*, pages 176–190, 2017.

9 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):1–33, 2009.

10 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

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 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017.

13 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symp. on Discrete Algorithms*, pages 841–850, 2003.

14 James Holt and Leonard McMillan. Constructing Burrows-Wheeler Transforms of large string collections via merging. In *ACM Conf. on Bioinformatics, Computational Biology, and Health Informatics*, pages 464–471, 2014.

15 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014.

16 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, 475:113–119, 2013.

17 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

18 Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

19 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and Retrieval of Individual Genomes. In *RECOMB, Tucson, AZ, USA*, pages 121–137, 2009.

20 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

21 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An Extension of the Burrows Wheeler Transform and Applications to Sequence Comparison and Data Compression. In *Combinatorial Pattern Matching*, pages 178–189, 2005.

22 Giovanni Manzini. XBWT tricks. In *String Processing and Information Retrieval*, pages 80–92, 2016.

23 Jouni Sirén. Burrows-Wheeler Transform for Terabases. In *Data Compression Conf.*, pages 211–220, 2016.

24 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections. In *String Processing and Information Retrieval*, pages 164–175, 2009.

A Appendix

A.1 Linear time construction of tables LRS(P) and LCP(P)

To build in linear time the tables $\texttt{LRS}(P)$ and $\texttt{LCP}(P)$, we use Algorithm 2, which is proved in Lemma 14. We summarise this property in Proposition 15. The tables for the running example are illustrated in Figure 2.

▶ **Lemma 14.** *Let $i \in [\![1, |m_P|]\!]$.*

$$\texttt{LRS}(P)[i] \quad = \quad \begin{cases} 0 & \text{if } i \in [\![1, \#(P.S)]\!], \\ \texttt{LRS}(P)[\texttt{LF}(P)[i]] - 1 & \text{otherwise.} \end{cases}$$

Proof. For any $i \in [\![1, |w|]\!]$, we know that $\texttt{SA}(w)[\texttt{LF}(w)[i]] = \texttt{SA}(w)[i] - 1$ if $\texttt{SA}(w)[i] \geq 2$ and $\texttt{SA}(w)[\texttt{LF}(w)[i]] = |w|$ otherwise; hence, we get

- if $\texttt{SA}(w)[i] \geq 2$,

$$\begin{aligned} \texttt{LRS}(P)[\texttt{LF}(P)[i]] &= \texttt{select}_{\$}(m_P[\texttt{SA}(m_P)[\texttt{LF}(P)[i]] : |m_P|], 1) - 1 \\ &= \texttt{select}_{\$}(m_P[\texttt{SA}(w)[i] - 1 : |m_P|], 1) - 1 \\ &= \begin{cases} 0 & \text{if } m_P[\texttt{SA}(w)[i] - 1] = \$, \\ \texttt{LRS}(P)[i] + 1 & \text{otherwise.} \end{cases} \end{aligned}$$

- If $\texttt{SA}(w)[i] = 1$,

$$\begin{aligned} \texttt{LRS}(P)[\texttt{LF}(P)[i]] &= \texttt{select}_{\$}(m_P[\texttt{SA}(m_P)[\texttt{LF}(P)[i]] : |m_P|], 1) - 1 \\ &= \texttt{select}_{\$}(m_P[|m_P| : |m_P|], 1) - 1 \\ &= 0. \end{aligned}$$

We define the function Letter from $[\![1, |m_P|]\!]$ to Σ such that $C[\texttt{Letter}[i]] < i \leq C[\texttt{Letter}[i+1]]$ (see the definition of LF on page 3) where $\Sigma = \{c_1 \ldots, c_{\#\Sigma}\}$ with $c_1 < \ldots < c_{\#\Sigma}$. Thus, we define $\texttt{RLF}(P)$ from $[\![1, |m_P|]\!]$ to $[\![1, |m_P|]\!]$ such that

$$\texttt{RLF}(P)[i] = \texttt{select}_{\texttt{Letter}[i]}(\texttt{BWT}(P), i - C[\texttt{Letter}[i]]).$$

Let i be an integer between 1 and $|m_P|$. We have

$$\begin{aligned} \texttt{RLF}(P)[\texttt{LF}(P)[i]] &= \texttt{select}_{\texttt{Letter}[\texttt{LF}(P)[i]]}(\texttt{BWT}(P), \texttt{LF}(P)[i] - C[\texttt{Letter}[\texttt{LF}(P)[i]]]) \\ &= \texttt{select}_{\texttt{BWT}(P)[i]}(\texttt{BWT}(P), \texttt{LF}(P)[i] - C[\texttt{BWT}(P)[i]]) \\ &= \texttt{select}_{\texttt{BWT}(P)[i]}(\texttt{BWT}(P), \texttt{rank}_{\texttt{BWT}(w)[i]}(\texttt{BWT}(w), i)) \\ &= i. \end{aligned}$$

Hence, the function $\texttt{RLF}(P)$ is the reverse bijection of $\texttt{LF}(P)$, and as $m_P[\texttt{SA}(m_P)[i] - 1] = \texttt{BWT}(P)[i]$, one gets

$$\begin{aligned} \texttt{LRS}(P)[i] = 0 &\Leftrightarrow \texttt{BWT}(P)[\texttt{RLF}(P)[i]] = \$ \\ &\Leftrightarrow \texttt{BWT}(P)[\texttt{select}_{\texttt{Letter}[i]}(\texttt{BWT}(P), i - C[\texttt{Letter}[i]])] = \$ \\ &\Leftrightarrow \texttt{Letter}[i] = \$ \\ &\Leftrightarrow i \in [\![1, \#(P.S)]\!]. \end{aligned}$$

Therefore, we derive the desired equality, which concludes the proof. ◀

▶ **Proposition 15.** *Let P be an ordered set of strings. Using tables* $\texttt{BWT}(P)$ *and* $\texttt{LCP}(m_P)$*, we can compute the tables* $\texttt{LRS}(P)$ *and* $\texttt{LCP}(P)$ *in a time that is linear in* $|m_P|$*.*

Proof. As the value of $\texttt{LCP}(P)$ at each position corresponds to the minimum between the values at same positions in $\texttt{LCP}(m_P)$ and in $\texttt{LRS}(P)$, we only need to proove that the table $\texttt{LRS}(P)$ can be computed from $\texttt{BWT}(P)$ in linear time.

Using Algorithm 2 and Lemma 14, we can compute table $\texttt{LRS}(P)$ in $O(|m_P|)$ time.

Algorithm 2: Computation of table $\texttt{LRS}(P)$.

Input : The ordered set of strings P
Output : The table $\texttt{LRS}(P)$
Compute the table $\texttt{BWT}(P)$;
Initialise $\texttt{LRS}$ as an empty table of length $|m_P|$;
for $i \in [\![1, \#(P.S)]\!]$ **do**
  $position \leftarrow i$;
  $nb \leftarrow 0$;
  $\texttt{LRS}[position] \leftarrow nb$;
  while $\texttt{BWT}(P)[position] \neq \$$ **do**
    $\texttt{LRS}[position] \leftarrow nb$;
    $nb \leftarrow nb + 1$;
    $position \leftarrow \texttt{LF}[position]$;
return $\texttt{LRS}$;

◀

A.2 Proof of Theorem 4

The next propositions exhibit the links between $\texttt{Decomp\_BWT}(P)$ and first, the arcs of $\texttt{ACT}(\overleftarrow{P.S})$ (Proposition 16), and second, the arcs of $\texttt{ACFL}(\overleftarrow{P.S})$ (Proposition 17).

▶ **Proposition 16** (See Figure 4)**.** *The graph* $G_T(P) = (\texttt{Decomp\_BWT}(P), A_T(P))$ *is isomorphic to the tree* $\texttt{ACT}(\overleftarrow{P.S})$*, where*

$$A_T(P) := \{(u, v) \in \texttt{Decomp\_BWT}(P)^2 \mid \exists x \in u \text{ such that } \texttt{LF}(P)[x] \neq 0 \text{ and } \texttt{LF}(P)[x] \in v\}.$$

Proof. First, we show that there exists a bijection between the node set of $\texttt{ACT}(\overleftarrow{P.S})$ and that of $G_T(P)$. We reuse the bijection $\texttt{Dec\_Pre}$, which served in Proposition 2. Let us show that for each arc (u, v) of $A_T(P)$, $(\texttt{Dec\_Pre}[u], \texttt{Dec\_Pre}[v])$ is an arc of $\texttt{ACT}(\overleftarrow{P.S})$, and vice versa. Let $(u, v) \in A_T(P)$, i.e., $(u, v) \in \texttt{Decomp\_BWT}(P)^2$ such that there exists $x \in u$ satisfying $\texttt{LF}(P)[x] \in v$.

According to [5], we know that $\texttt{SA}(m_P)[\texttt{LF}(m_P)[i]] = \texttt{SA}(m_P)[i] - 1$. By Lemma 14, we have $\texttt{LRS}(P)[\texttt{LF}(P)[x]] = \texttt{LRS}(P)[x] + 1$ for all x such that $\texttt{LF}(P)[x] \neq 0$. With both equalities and Lemma 3, we obtain

$$\begin{aligned}
\texttt{Dec_Pre}[v] &= \overleftarrow{m_P[\texttt{SA}(m_P)[\texttt{LF}(P)[x]] : \texttt{SA}(m_P)[\texttt{LF}(P)[x]] + \texttt{LRS}(P)[\texttt{LF}(P)[x]] - 1]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[x] - 1 : \texttt{SA}(m_P)[x] - 1 + \texttt{LRS}(P)[\texttt{LF}(P)[x]] - 1]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[x] - 1 : \texttt{SA}(m_P)[x] - 1 + \texttt{LRS}(P)[x]]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[x] : \texttt{SA}(m_P)[x] + \texttt{LRS}(P)[x] - 1]}\ m_P[\texttt{SA}(m_P)[x] - 1] \\
&= \texttt{Dec_Pre}[u]\ m_P[\texttt{SA}(m_P)[x] - 1].
\end{aligned}$$

The string $\texttt{Dec\_Pre}[u]$ is thus the longest prefix of $\texttt{Dec\_Pre}[v]$.

Let (x, y) be an arc of $\texttt{ACT}(\overleftarrow{P.S})$. We take z a leaf in the subtree of $\texttt{ACT}(\overleftarrow{P.S})$ in y. As x is the parent of y in $\texttt{ACT}(\overleftarrow{P.S})$, z is also a leaf in the subtree of $\texttt{ACT}(\overleftarrow{P.S})$ in x. We take $i \in [\![1, |m_P|]\!]$ such that $\overleftarrow{z}$ is a prefix of $m_P[i : |m_P|]$. Hence $\overleftarrow{y}$ is a prefix of $m_P[i + |z| - |y| : |m_P|]$ and $\overleftarrow{x}$ is a prefix of $m_P[i + |z| - |x| : |m_P|]$. As (x, y) is an arc of $\texttt{ACT}(\overleftarrow{P.S})$, $|y| - |x| = 1$. Thus, choosing k such that $\texttt{SA}(m_P)[k] = i + |z| - |y| + 1$, and u, v in $\texttt{Decomp\_BWT}(P)^2$ such that $k \in u$ and $\texttt{LF}(P)[k] \in v$, we get $\overleftarrow{x} = \texttt{Dec\_Pre}[u]$ and $\overleftarrow{y} = \texttt{Dec\_Pre}[v]$. This concludes the proof. ◀

▶ **Proposition 17** (See Figure 4). *The graph* $G_F(P) = (\texttt{Decomp\_BWT}(P), A_F(P))$ *is isomorphic to the tree* $\texttt{ACFL}(\overleftarrow{P.S})$*, where*

$$A_F(P) := \{(u, v) \in \texttt{Decomp\_BWT}(P)^2 \mid \big(\max_{\substack{k<i \\ \texttt{LRS}(P)[k]=\min_{k-1\le l\le i}(\texttt{LCP}(P)[l])}} (k)\big) \in v \text{ with } u = [\![i, j]\!]\}.$$

Proof. First, let us show the following equivalence. Let $u = [\![i, j]\!] \in \texttt{Decomp\_BWT}(P)$.

$$w \in \texttt{Decomp\_BWT}(P) \text{ such that } \exists k \in w \text{ with } k < i \text{ and } \texttt{LRS}(P)[k] = \min_{k-1\le l\le i}(\texttt{LCP}(P)[l])$$
$$\Leftrightarrow$$
$$\texttt{Dec\_Pre}[w] \text{ is a suffix of } \texttt{Dec\_Pre}[u].$$

Let $w \in \texttt{Decomp\_BWT}(P)$ such that $\exists k \in w$ with $k < i$ and $\texttt{LRS}(P)[k] = \min_{k-1\le l\le i}(\texttt{LCP}(P)[l])$. Hence, we have for all $l \in [\![k - 1, i]\!]$, $\texttt{LRS}(P)[k] \le \texttt{LCP}(P)[l]$, and thus

$$\begin{aligned}
\texttt{Dec_Pre}[u] &= \overleftarrow{m_P[\texttt{SA}(m_P)[i] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[i] + \texttt{LRS}(P)[k] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]} \\
&\quad \overleftarrow{m_P[\texttt{SA}(m_P)[i] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[k] - 1]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[i] + \texttt{LRS}(P)[k] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]} \\
&\quad \overleftarrow{m_P[\texttt{SA}(m_P)[k] : \texttt{SA}(m_P)[k] + \texttt{LRS}(P)[k] - 1]} \\
&= \overleftarrow{m_P[\texttt{SA}(m_P)[i] + \texttt{LRS}(P)[k] : \texttt{SA}(m_P)[i] + \texttt{LRS}(P)[i] - 1]}\ \texttt{Dec_Pre}[w].
\end{aligned}$$

Let $w = [\![i_1, j_1]\!]$ and $u = [\![i_2, j_2]\!]$ be two elements of $\texttt{Decomp\_BWT}(P)$ such that $\texttt{Dec\_Pre}[w]$ is a suffix of $\texttt{Dec\_Pre}[u]$. Hence, we have that $m_P[\texttt{SA}(m_P)[i_1] : \texttt{SA}(m_P)[i_1] + \texttt{LRS}(P)[i_1] - 1]$ is a prefix of $m_P[\texttt{SA}(m_P)[i_2] : \texttt{SA}(m_P)[i_2] + \texttt{LRS}(P)[i_2] - 1]$. By the definition of $\texttt{BWT}(P)$, for all $l \in [\![i_1, i_2]\!]$, $\texttt{LRS}(P)[i_1] \le \texttt{LCP}(P)[l]$. This concludes the proof of the equivalence. By the equivalence, given u and v in $\texttt{Decomp\_BWT}(P)$ such that $\texttt{Dec\_Pre}[u]$ is a suffix of $\texttt{Dec\_Pre}[v]$, for all $k_1 \in u$ and $k_2 \in v$, we have $k_1 \le k_2$. Hence, by taking the largest w satisfying the first step of the inequality, we obtain the longest suffix and vice versa. ◀

A.3 Proof of Theorem 5

Proof. We define T_B (resp. T_X) as the array of intervals of $\texttt{Decomp\_BWT}(P)$ (resp. $\texttt{Decomp\_XBW}(\texttt{ACT}(\overleftarrow{P}))$) sorted in the interval order. Let us prove that T_B and T_X have the same length, and that at the same position i, $T_B[i]$ and $T_X[i]$ represent the same prefix of $\overleftarrow{P}$. By Proposition 2, the length of T_B is $\#\texttt{Prefix}(\overleftarrow{P.S})$. By the definition of $\texttt{Decomp\_XBW}$, the length of T_X is the number of 1 in $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$, i.e., the number of internal nodes of $\texttt{ACT}(\overleftarrow{P})$, and thus is equal to $\#\texttt{Prefix}(\overleftarrow{P.S})$. Hence, T_B and T_X have the same cardinalities.

Let $i \in [\![1, \#\texttt{Prefix}(\overleftarrow{P.S})]\!]$. By Proposition 2, $T_B[i]$ represents the i^{th} suffix of strings of $P.S$ in lexicographic order. By the definition of $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$, all the nodes $\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[k]$ for $k \in T_X[i]$ have the same parent z in $\texttt{ACT}(\overleftarrow{P})$, and z represents the i^{th} node in $\prec$ order, i.e., the i^{th} suffix of strings of $P.S$ in lexicographic order.

As for a given position i, $T_B[i]$ and $T_X[i]$ represent the same prefix of $\overleftarrow{P}$, we define the bijection $\texttt{BWT\_XBW}$ such that for any i in $[\![1, \#\texttt{Prefix}(\overleftarrow{P.S})]\!]$, one has $\texttt{BWT\_XBW}[T_B[i]] = T_X[i]$. As the tree $\texttt{ACT}(\overleftarrow{P})$ represents the Aho-Corasick tree of $\overleftarrow{P.S}^{\$}$, we have a bijection b_1 from the node set of $\texttt{ACT}(\overleftarrow{P})$ onto the set of prefixes of $\overleftarrow{P.S}^{\$}$. By the definition of functions π and δ (see page 4), for any node v of $\texttt{ACT}(\overleftarrow{P})$, we have $b_1(v) = \overleftarrow{\pi(v)}\ \delta(v)$.

By the definition of $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$, we have that for $T_X[i] = [\![i', j']\!]$

$$\begin{aligned} \#T_X[i] &= \#\{v \text{ node of } \texttt{ACT}(\overleftarrow{P}) \mid \pi(v) = \overleftarrow{\texttt{Dec\_Pre}[\texttt{BWT\_XBW}^{-1}[T_X[i]]]}\} \\ &= \#\{v \text{ node of } \texttt{ACT}(\overleftarrow{P}) \mid v \text{ is a child of } b_1^{-1}(\texttt{Dec\_Pre}[\texttt{BWT\_XBW}^{-1}[T_X[i]]])\}. \end{aligned}$$

As $i' \in T_X[i]$ and $b_1^{-1}(\texttt{Dec\_Pre}[\texttt{BWT\_XBW}^{-1}[T_X[i]]]) = \texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])$,

$$\begin{aligned} \#T_X[i] &= \#\{v \text{ node of } \texttt{ACT}(\overleftarrow{P}) \mid v \text{ is a child of } \texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])\} \\ &= \#\texttt{Children}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])). \end{aligned}$$

Let $\{x_1, x_2, \ldots, x_{\#T_X[i]}\}$ be the set of children of $\texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])$ sorted such that $x_1 \lhd_{P.\sigma} \ldots \lhd_{P.\sigma} x_{\#T_X[i]}$. By the definition of $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$, for any $k \in [\![1, \#T_X[i]]\!]$, we have $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))[i' + k - 1] = \delta[x_k]$. By the definition of $\texttt{BWT}(P)$, for $T_B[i] = [\![i'', j'']\!]$, we get

$$\begin{aligned} \#T_B[i] &= \#\{w \in \overleftarrow{P.S}^{\$} \mid \texttt{Dec\_Pre}[T_B[i]] \text{ is a prefix of } w\} \\ &= \#\{v \text{ leaf of } \texttt{ACT}(\overleftarrow{P}) \mid b_1^{-1}(\texttt{Dec\_Pre}[T_B[i]]) \text{ is an ancestor of } v \text{ in } \texttt{ACT}(\overleftarrow{P})\}. \end{aligned}$$

As $b_1^{-1}(\texttt{Dec\_Pre}[T_B[i]]) = b_1^{-1}(\texttt{Dec\_Pre}[\texttt{BWT\_XBW}^{-1}[T_X[i]]]) = \texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])$,

$$\begin{aligned} \#T_B[i] &= \#\{v \text{ a leaf of } \texttt{ACT}(\overleftarrow{P}) \mid \texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i']) \\ &\qquad \text{is an ancestor of } v \text{ in } \texttt{ACT}(\overleftarrow{P})\} \\ &= \#\texttt{Leaves}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])). \end{aligned}$$

Let $\{y_1, y_2, \ldots, y_{\#T_B[i]}\}$ be the set of leaves of the subtree of $\texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i'])$ in $\texttt{ACT}(\overleftarrow{P})$ sorted such that $y_1 \lhd_{P.\sigma} \ldots \lhd_{P.\sigma} y_{\#T_B[i]}$. Given $k \in [\![i'', j'']\!]$, we define $w_P[k]$ as the string $m_P[\texttt{select}_{\$}(m_P, \texttt{rank}_{\$}(m_P, \texttt{SA}(P)[k] - 1)) + 1 : \texttt{SA}(P)[k] + \texttt{LRS}(P)[k] - 1]$. For all $k \in [\![i'', j'']\!]$, the string $w_P[k]$ is a string of $P.S$. Moreover, the definition of $<_{P.\sigma}$ implies $w_P[i''] <_{P.\sigma} \ldots <_{P.\sigma} w_P[j'']$. As for $l \in [\![1, \#T_B[i]]\!]$, the string $\pi(y_l)$ is also a string of $P.S$, we get $\pi(y_l) = w_P[i'' + l - 1]$. Given x and y in $[\![i'', j'']\!] \in \texttt{Decomp\_BWT}(P)$, we obtain the following equivalences between orders:

$$x < y \Leftrightarrow w_P[x] <_{P.\sigma} w_P[y] \Leftrightarrow \pi(y_{x-i''+1}) <_{P.\sigma} \pi(y_{y-i''+1}) \Leftrightarrow y_{x-i''+1} \lhd_{P.\sigma} y_{y-i''+1}.$$

By the definition of $\texttt{BWT}(P)$, for all $k \in [\![1, \#T_B[i]]\!]$, we get that

$$\begin{aligned} \texttt{BWT}(P)[i'' + k - 1] &= m_P[\texttt{SA}(m_P)[i'' + k - 1] - 1] \\ &= w_P[i'' + k - 1][|w_P[i'' + k - 1]| - \texttt{LRS}[i'' + k - 1] + 1] \\ &= \pi(y_k)[|w_P[i'' + k - 1]| - \texttt{LRS}[i'' + k - 1] + 1] \\ &= \delta[\texttt{Child}_{\texttt{ACT}(\overleftarrow{P})(y_k)}(\texttt{Parent}_{\texttt{ACT}(\overleftarrow{P})}(\texttt{PA}(\texttt{ACT}(\overleftarrow{P}))[i']))]. \end{aligned}$$

◀

A.4 Proof of Corollary 6

Let P be an ordered set of strings. To compute tables $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ and $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$ using only $\texttt{BWT}(P)$, $\texttt{LCP}(P)$ and $\texttt{LRS}(P)$, we first define a new table $\texttt{BWD}(P)$.

The *Burrows-Wheeler Decomposition* of P, denoted by $\texttt{BWD}(P)$, is the array of length $\#\texttt{Decomp\_BWT}(P)$ such that for each position i, $\texttt{BWD}(P)[i]$ is the cardinality of the i^{th} element of $\texttt{Decomp\_BWT}(P)$ in interval order.

▶ **Lemma 18.** *Using tables* $\texttt{LCP}(P)$ *and* $\texttt{LRS}(P)$*, Algorithm 3 computes* $\texttt{BWD}(P)$ *in linear time in* $\|P.S\|$ *and the table* $\texttt{BWD}(P)$ *can be stored with* $\|P.S\| \times log(\#(P.S))$ *bits.*

Proof of Lemma 18. For each i in $[\![1, |m_P|]\!]$, at the begining of the loop **for**, we have that $\texttt{LCP}(P)[i-j] \neq \texttt{LRS}(P)[i-j]$ and for all $k \in [\![i-j+1, i-1]\!]$, $\texttt{LCP}(P)[k] \neq \texttt{LRS}(P)[k]$. Hence, if $\texttt{LCP}(P)[i] \neq \texttt{LRS}(P)[i]$, the interval $[\![i-j, i-1]\!]$ is an element of $\texttt{Decomp\_BWT}(P)$ and the cardinality of $[\![i-j, i-1]\!]$ is j. Otherwise, we increase j by 1 because the position i does not correspond to a new interval of $\texttt{Decomp\_BWT}(P)$. For the complexity, as each step of the loop can be computed in constant time, Algorithm 3 computes $\texttt{BWD}(P)$ in linear time in $\|P.S\|$. As for each position i of $\texttt{BWD}(P)$, $\texttt{BWD}(P)[i]$ represents the number of strings of $P.S$ having as suffix $\texttt{Dec\_Pre}[u]$, where u is the i^{th} element of $\texttt{Decomp\_BWT}(P)$ sorted in interval order, it follows that $\texttt{BWD}(P)[i] \leq \#(P.S)$. This concludes the proof. ◀

Algorithm 3: Computation of table $\texttt{BWD}(P)$.

Input : The tables $\texttt{LCP}(P)$ and $\texttt{LRS}(P)$
Output : The table $\texttt{BWD}(P)$
$\texttt{BWD} \leftarrow$ empty list;
$j \leftarrow 0$;
for $i \in [\![1, |m_P|]\!]$ **do**
  if $\texttt{LCP}(P)[i] \neq \texttt{LRS}(P)[i]$ **then**
    insert j at the end of BWD;
    $j \leftarrow 1$;
  else $j \leftarrow j + 1$;
return BWD;

▶ **Lemma 19.** *Using tables* $\texttt{BWT}(P)$ *and* $\texttt{BWD}(P)$*, Algorithm 4 computes the tables* $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ *and* $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$ *in linear time of* $\|P.S\| \times \#\Sigma$.

Proof of Lemma 19. Algorithm 4 is an application of Theorem 5. ◀

Using Algorithm 3 to build $\texttt{BWD}(P)$ and Algorithm 4, we can compute tables $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ and $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$ in linear time of $\|P.S\| \times \#\Sigma$.
We define the equivalent of BWD for $\texttt{Decomp\_XBW}(P)$. The *eXtended Burrows Wheeler Decomposition* (XBWD) of a tree $\mathcal{T}$ is the array of length $\#\texttt{XBWL}(\mathcal{T})$ such that for any position i, $\texttt{XBWD}(P)[i]$ equals the cardinality of the i^{th} element of $\texttt{Decomp\_XBW}(\mathcal{T})$ sorted in interval order.

▶ **Lemma 20.** *From the table* $\texttt{XBWL}(\texttt{ACT}(S))$*, Algorithm 5 computes* $\texttt{XBWD}(\texttt{ACT}(S))$ *in linear time in* $\|P.S\|$ *and the table* $\texttt{XBWD}(\texttt{ACT}(S))$ *can be stored in* $\|P.S\| \times log(\#\Sigma)$ *bits.*

Algorithm 4: Computation of tables $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ and $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$.

Input : The tables $\texttt{BWT}(P)$ and $\texttt{BWD}(P)$
Output : The tables $\texttt{XBWT}(\texttt{ACT}(\overleftarrow{P}))$ and $\texttt{XBWL}(\texttt{ACT}(\overleftarrow{P}))$
Initialise both XBWT and XBWL as empty lists;
$nb \leftarrow 1$;
for $i \in [\![1, \#\texttt{BWD}(P)]\!]$ **do**
  $D \leftarrow$ Dictionnary such that for all $l \in \Sigma$, $D[l] \leftarrow true$;
  $begin \leftarrow nb$;
  $last \leftarrow begin + \texttt{BWD}(P)[i] - 1$;
  $nb \leftarrow last + 1$;
  for $j \in [\![begin, last]\!]$ **do**
    if $D[\texttt{BWT}(P)[j]]$ **then**
      insert $\texttt{BWT}(P)[j]$ at the end of XBWT ;
      insert 0 at the end of XBWL ;
      $D[\texttt{BWT}(P)[j]] \leftarrow false$;
  $\texttt{XBWL}[\#\texttt{XBWL}] \leftarrow 1$;
return XBWT and XBWL;

Proof. The proof of Lemma 20 is similar to that of Lemma 18. As for each position i of $\texttt{XBWD}(\texttt{ACT}(S))$, $\texttt{XBWD}(\texttt{ACT}(S))[i]$ represents the number of the right extensions of the strings $\texttt{PA}(\texttt{ACT}(S))[i]$ in S (i.e., the number of different strings $\texttt{PA}(\texttt{ACT}(S))[i]$ a which are substrings of a string of S for some $a \in \Sigma$), we have $\texttt{XBWD}(\texttt{ACT}(S))[i] \leq \#\Sigma$. ◀

Algorithm 5: Computation of table $\texttt{XBWD}(\texttt{ACT}(S))$.

Input : The table $\texttt{XBWL}(\texttt{ACT}(S))$; **Output**: The table $\texttt{XBWD}(\texttt{ACT}(S))$
$\texttt{XBWD} \leftarrow$ empty list;
$j \leftarrow 1$;
for $i \in [\![1, \#(\texttt{XBWL}(\texttt{ACT}(S)))]\!]$ **do**
  if $\texttt{XBWL}(\texttt{ACT}(S)) = 1$ **then**
    insert j at the end of XBWD ;
    $j \leftarrow 1$;
  else $j \leftarrow j + 1$;
return XBWD;

▶ **Lemma 21.** *Using tables* $\texttt{XBWT}(\texttt{ACT}(S))$ *and* $\texttt{XBWD}(\texttt{ACT}(S))$*, we can build the tables* $\texttt{BWT}(\overleftarrow{P})$*,* $\texttt{LCP}(\overleftarrow{P})$ *and* $\texttt{LRS}(\overleftarrow{P})$ *in linear time of* $\|P.S\| \times \#\Sigma$*, where* P *is a topologically planar, ordered set of strings such that* $P.S = S$*.*

Proof. In [9], Ferragina *et al.* prove that with both tables $\texttt{XBWT}(\texttt{ACT}(S))$ and $\texttt{XBWD}(\texttt{ACT}(S))$ one can access in constant time the children and the parents in $\texttt{ACT}(S)$. Hence, we can compute in linear time in $\|P.S\|$, the table $\texttt{TL}(\texttt{ACT}(S))$, where in each position of i we store the number of leaves in the subtree of the node $\texttt{PA}(\texttt{ACT}(S))[i]$. We finish the proof using the results of Theorem 5 and an algoritm similar to Algorithm 4. ◀

A.5 Proof of Lemma 10

Proof. Let T' be an optimal solution of `Min-Permutation-Table` for $\texttt{XBWT}(\texttt{ACT}(P))$ and for $\texttt{Decomp\_XBW}(\texttt{ACT}(P))$. By Theorem 5, for each $[\![i,j]\!] \in \texttt{Decomp\_XBW}(\texttt{ACT}(P))$, the order of the symbols in $\texttt{XBWT}(\texttt{ACT}(P))[i,j]$ depends on the order on the children of the parent of $\texttt{PA}(\texttt{ACT}(P))[i]$. Hence, the choice of T' corresponds to the choice of an order for each internal node of $\texttt{ACT}(P)$ over all its children. As we can extend this order to a total order on the leaves of $\texttt{ACT}(P)$, we can build P' the ordered set of strings satisfying $P'.S = S^{\$}$ and $P'.\sigma(\pi(f_i)) = s_i$, where the order on the leaves of $\texttt{ACT}(P)$ is $f_1 < \ldots < f_{\#S}$ and $s_1 < \ldots < s_{\#S}$ are the strings of S in lexicographic order. ◀

Quasi-Linear-Time Algorithm for Longest Common Circular Factor

Mai Alzamel
Department of Informatics, King's College London, UK
Department of Computer Science, King Saud University, Riyadh, Saudi Arabia
https://nms.kcl.ac.uk/mai.alzamel/
mai.alzamel@kcl.ac.uk

Maxime Crochemore
Department of Informatics, King's College London, UK
Laboratoire d'Informatique Gaspard-Monge, Université Paris-Est, Marne-la-Vallée, France
http://www-igm.univ-mlv.fr/~mac/
maxime.crochemore@kcl.ac.uk

Costas S. Iliopoulos
Department of Informatics, King's College London, UK
https://nms.kcl.ac.uk/costas.iliopoulos/
costas.iliopoulos@kcl.ac.uk

Tomasz Kociumaka
Institute of Informatics, University of Warsaw, Poland
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
https://www.mimuw.edu.pl/~kociumaka/
kociumaka@mimuw.edu.pl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland
https://www.mimuw.edu.pl/~jrad
jrad@mimuw.edu.pl

Wojciech Rytter
Institute of Informatics, University of Warsaw, Poland
https://www.mimuw.edu.pl/~rytter
rytter@mimuw.edu.pl

Juliusz Straszyński
Institute of Informatics, University of Warsaw, Poland
j.straszynski@mimuw.edu.pl

Tomasz Waleń
Institute of Informatics, University of Warsaw, Poland
https://www.mimuw.edu.pl/~walen
walen@mimuw.edu.pl

Wiktor Zuba
Institute of Informatics, University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

We introduce the Longest Common Circular Factor (LCCF) problem in which, given strings S and T of length at most n, we are to compute the longest factor of S whose cyclic shift occurs as a factor of T. It is a new similarity measure, an extension of the classic Longest Common Factor. We show how to solve the LCCF problem in $\mathcal{O}(n \log^4 n)$ time using $\mathcal{O}(n \log^2 n)$ space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases longest common factor, circular pattern matching, internal pattern matching, intersection of hyperrectangles

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis; Article No. 25; pp. 25:1–25:14

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

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.25

Related Version https://arxiv.org/abs/1901.11305

Funding *Tomasz Kociumaka*: Supported by ISF grants no. 824/17 and 1278/16 and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Jakub Radoszewski: 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.
Juliusz Straszyński: 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.

1 Introduction

We introduce a new variant of the Longest Common Factor (LCF) Problem, called the Longest Common Circular Factor (LCCF) Problem. In the LCCF problem, given two strings S and T, both of length at most n, we seek for the longest factor of S whose cyclic shift occurs as a factor of T. The length of the LCCF is a new string similarity measure that is 2-approximated by the length of the LCF. We show that the exact value of LCCF can be computed efficiently.

A linear-time solution to the LCF problem is one of the best-known applications of the suffix tree [2]. Just as the LCF problem was an extension of the classical pattern matching, the LCCF can problem be seen as an extension of the circular pattern matching. The latter can still be solved in linear time using the suffix tree and admits a number of efficient solutions based on practical approaches [4, 10, 17, 21, 28, 32], also in the approximate variant [6, 7, 18, 20], as well as an indexing variant [3, 21, 22], and the problem of detecting various circular patterns [29]. The LCCF problem is further related to the notion of unbalanced translocations [9, 11, 31, 33, 34].

One can formally state the problem in scope as follows.

Longest Common Circular Factor (LCCF)
Input: Two strings S and T of length at most n each.
Output: A pair of longest factors, F of S and F' of T, for which there exist strings U and V such that $F = UV$ and $F' = VU$; we denote $\mathsf{LCCF}(S, T) = (F, F')$.

This problem can be solved in a straightforward way in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space using *period queries* [26, 27, 24]; see Section 2.1. Our main result is the following.

▶ **Theorem 1** (Main Result). *The Longest Common Circular Factor problem on two strings of length at most n can be solved in $\mathcal{O}(n \log^4 n)$ time and $\mathcal{O}(n \log^2 n)$ space.*

Henceforth, we assume for simplicity that $|S| = |T| = n$; otherwise, the shorter string can be padded with a special character # that does not occur in either of the strings.

Our approach. We apply local consistency techniques from the area of internal pattern matching (in case U and V are not highly periodic; Section 3) and Lyndon roots (otherwise; Section 4). The LCCF problem is reduced to finding configurations satisfying conjunction of four conditions of type $i \in \mathsf{Occ}(X)$, where $\mathsf{Occ}(X)$ is the set of occurrences of a factor X.

Each configuration can be decomposed into two subconfigurations (pairs of consecutive fragments), one in S and one in T. We guarantee that the number of subconfigurations is

nearly linear so that we can compute them all for both S and T. Then, the task reduces to finding two subconfigurations which agree (produce a full configuration) and constitute an optimal solution. This is done using geometric techniques in Section 6. Each condition $i \in \mathsf{Occ}(X)$ can be seen as membership of a point in a range since $\mathsf{Occ}(X)$ forms an interval in the suffix array. This gives a reduction of the LCCF problem to an intersection problem for 4D-rectangles. The latter task is solved efficiently using a sweep line algorithm.

2 Preliminaries

We consider strings over an integer alphabet Σ. If W is a string, then by $|W|$ we denote its length and by $W[1], \ldots, W[|W|]$ its characters. By $x = W[i\,..\,j]$ we denote a *fragment* of W between the ith and jth character, inclusively. We also denote this fragment x by $W[i\,..\,j+1)$, and we define $\mathsf{first}(x) = i$ as well as $\mathsf{last}(x) = j$. If $\mathsf{first}(x) = 1$, then x is a prefix, and if $\mathsf{last}(x) = |W|$, it is a suffix of W. Fragments x and y are *consecutive* if $\mathsf{last}(x) + 1 = \mathsf{first}(y)$; we then also say that y *follows* x.

The string $W[i] \cdots W[j]$ that corresponds to the fragment x is a *factor* of W. We say that two fragments *match* if the corresponding factors are the same. Let us note that a fragment can be represented by its endpoints in $\mathcal{O}(1)$ space; this representation can also be used to specify the corresponding factor.

By W^R we denote the reversal of a string W. We say that a positive integer p is a *period* of a string W if $W[i] = W[i+p]$ for all $i = 1, \ldots, |W| - p$. By $\mathsf{per}(W)$ we denote the shortest period of W. A string W is called *(weakly) periodic* if its shortest period satisfies $2\mathsf{per}(W) \le |W|$. Fine and Wilf's Periodicity Lemma [16] asserts that if a string W has periods p and q such that $p + q \le |W|$, then $\gcd(p, q)$ is also a period of W.

2.1 $\mathcal{O}(n^2 \log n)$-Time and $\mathcal{O}(n)$-Space Algorithm

A *period query* [26] is an internal query that is defined on a text W as follows: Given a fragment x of the text, report all periods of x (represented as several arithmetic progressions). In particular, the answer gives the shortest period $p = \mathsf{per}(x)$ and the longest *border* $U = x[1\,..\,|x| - p] = x[1 + p\,..\,|x|]$. Period queries can be answered in $\mathcal{O}(\log n)$ time using a data structure of size $\mathcal{O}(n)$. A randomized $\mathcal{O}(n)$-time construction of this data structure was presented in [27], whereas a deterministic variant appeared in [24, Theorem 1.1.12].

▶ **Proposition 2.** *The Longest Common Circular Factor problem on two strings of length n can be solved in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space.*

Proof. Let us set $W = S\#T\#S$, where $\#$ is a special character that occurs neither in S nor in T. For every pair of positions $i, j \in [1\,..\,n]$, we ask a period query for $x = W[i\,..\,n+j] = S[i\,..\,n]\#T[1\,..\,j)$ and $y = W[n+j+1\,..\,2n+1+i] = T[j\,..\,n]\#S[1\,..\,i)$. This lets us recover the longest borders U of x and V of y so that (UV, VU) is a common circular factor of S and T. The longest of these factors over all pairs of positions (i, j) corresponds to the LCCF. ◀

2.2 Synchronizing Sets

In this section, we present the notion of *synchronizing sets* recently introduced by Kociumaka and Kempa [23] for BWT construction and answering LCE queries. Intuitively, a τ-synchronizing set P of a string W is a subset of position of W such that:

- the choice whether $i \in \mathsf{P}$ is made based on a context of 2τ subsequent characters,
- P contains at least one in every τ positions of each region of W whose period exceeds $\frac{1}{3}\tau$.

The underlying idea of making a locally consistent selection based on fixed-length contexts originates from internal pattern matching [27]. Later, the construction has been used for LCE queries [8] and derandomized [24]. We formalize our results using synchronizing sets as they provide a cleaner interface compared to that of the synchronizing functions of [24].

▶ **Definition 3** (Kempa and Kociumaka [23, Definition 3.1]). *Let W be a string of length n and let $\tau \leq \frac{1}{2}n$ be a positive integer. We say that a set $\mathsf{P} \subseteq [1\,..\,n-2\tau+1]$ is a τ-*synchronizing set *of W if it satisfies the following conditions (see Figure 1):*

- *if $W[i\,..\,i+2\tau) = W[i'\,..\,i'+2\tau)$, then $i \in \mathsf{P}$ if and only if $i' \in \mathsf{P}$ (for $i, i' \in [1\,..\,n-2\tau+1]$),*
- *$\mathsf{P} \cap [i\,..\,i+\tau) = \emptyset$ if and only if $\mathsf{per}(W[i\,..\,i+3\tau-2]) \leq \frac{1}{3}\tau$ (for $i \in [1\,..\,n-3\tau+2]$).*

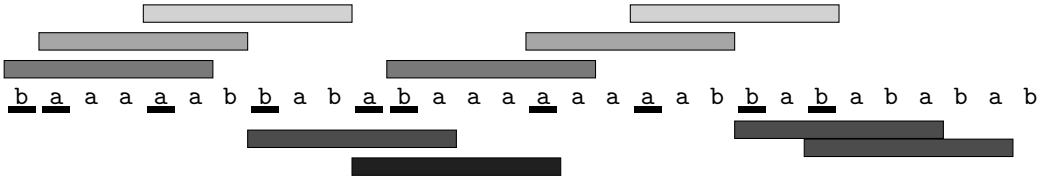


Figure 1 A 3-synchronizing set $\mathsf{P} = \{1, 2, 5, 8, 11, 12, 16, 19, 22, 24\}$ of a string W of length 30. The 10 positions in P are the starting positions of the occurrences of length-$(2 \cdot 3)$ factors `aabbab`, `aaaaab`, `baaaaa`, `bababa`, and `abaaaa` (marked as rectangles, listed top-down). Out of each three consecutive positions in $[1\,..\,25]$, at least one belongs to P. The only exception is $[13\,..\,16)$ due to the fact that $\mathsf{per}(W[13\,..\,20]) = 1 \leq \frac{1}{3} \cdot 3$.

A key technical result is a deterministic linear-time construction of a synchronizing set of optimal size $\mathcal{O}(\frac{n}{\tau})$. The linear running time is improved in [24, 23] to $\mathcal{O}(\frac{n}{\log_\sigma n})$ for small alphabet size σ and parameter $\tau = \Theta(\log_\sigma n)$, but that is not relevant for this paper.

▶ **Lemma 4** ([23, Proposition 8.10], [24, Lemma 4.4.9]). *Given a string W of length n and a positive integer $\tau \leq \frac{1}{2}n$, in $\mathcal{O}(n)$ time one can construct a τ-synchronizing set of size $\mathcal{O}(\frac{n}{\tau})$.*

The central property of a synchronizing set P is that if a factor X is sufficiently long and not highly periodic, then most of the positions of P contained in the occurrences of X are located consistently. In our applications, we actually use the leftmost of these positions only. Hence, for an integer i and a set P, we define $\mathsf{succ}_P(i) = \min\{p \in P : p \geq i\}$ to be the *successor* of i in P. We assume that $\min \emptyset = \infty$ so that $\mathsf{succ}_P(i) = \infty$ if $i > \max P$.

▶ **Lemma 5.** *Let P be a τ-synchronizing set in a string W. If $W[i\,..\,j] = X = W[i'\,..\,j']$, where $|X| \geq 3\tau - 1$ and $\mathsf{per}(X) > \frac{1}{3}\tau$, then $\mathsf{succ}_\mathsf{P}(i) - i = \mathsf{succ}_\mathsf{P}(i') - i' \leq |X| - 2\tau$.*

Proof. First, suppose for a proof by contradiction that $\mathsf{succ}_\mathsf{P}(i) - i > |X| - 2\tau = j - i + 1 - 2\tau$. Consequently, $\mathsf{P} \cap [i\,..\,j-2\tau+2) = \emptyset$, which yields $\mathsf{P} \cap [p\,..\,p+\tau) = \emptyset$ for $p \in [i\,..\,j-3\tau+2]$. Hence, Definition 3 implies $\mathsf{per}(W[p\,..\,p+3\tau-2]) \leq \frac{1}{3}\tau$ for $p \in [i\,..\,j-3\tau+2]$. Due to $j - i + 1 = |X| \geq 3\tau - 1$, this range is non-empty, so the Periodicity Lemma yields $\frac{1}{3}\tau < \mathsf{per}(X) = \mathsf{per}(W[i\,..\,j]) \leq \frac{1}{3}\tau$, contradicting our assumption that $\mathsf{succ}_\mathsf{P}(i) - i > |X| - 2\tau$.

Due to $\mathsf{succ}_\mathsf{P}(i) - i \leq |X| - 2\tau$, we now conclude that $W[\mathsf{succ}_\mathsf{P}(i)\,..\,\mathsf{succ}_\mathsf{P}(i) + 2\tau) = W[i' - i + \mathsf{succ}_\mathsf{P}(i)\,..\,i' - i + \mathsf{succ}_\mathsf{P}(i) + 2\tau)$. Therefore, $\mathsf{succ}_\mathsf{P}(i) \in \mathsf{P}$ implies $i' - i + \mathsf{succ}_\mathsf{P}(i) \in \mathsf{P}$ in the light of Definition 3. Consequently, $\mathsf{succ}_\mathsf{P}(i') - i' \leq \mathsf{succ}_\mathsf{P}(i) - i$. A symmetric argument shows that $\mathsf{succ}_\mathsf{P}(i) - i \leq \mathsf{succ}_\mathsf{P}(i') - i'$, which completes the proof. ◀

2.3 Types of Factors

We define the *type* of a (non-empty) string W as $\mathsf{type}(W) = \lfloor \log(|W|+1) - 1 \rfloor$. We denote by $\mathsf{LCCF}_{a,b}(S,T)$ the longest common circular factor (UV, VU) of S and T such that $\mathsf{type}(U) = a$ and $\mathsf{type}(V) = b$. We also say that it is the type-(a,b) LCCF. Moreover, we denote by $\mathsf{LCF}(S,T)$ the (ordinary) longest common factor of S and T (corresponding to $U = \varepsilon$ or $V = \varepsilon$). Our basic strategy is to compute $\mathsf{LCCF}_{a,b}(S,T)$ independently for every pair (a,b) and report the longest alternative among the obtained common circular factors, including (F,F) for $F = \mathsf{LCF}(S,T)$. However, we observe that if $\mathsf{LCCF}(S,T) = \mathsf{LCCF}_{a,b}(S,T) = (UV,VU)$, then $\frac{1}{2}|F| \le \frac{1}{2}|UV| \le \max(|U|,|V|) \le |F|$, and therefore $\mathsf{type}(F) - 1 \le \max(a,b) \le \mathsf{type}(F)$. Consequently, it suffices to iterate over $\mathcal{O}(\log |F|)$ pairs (a,b) satisfying the latter condition.

For each type a, we introduce a synchronizing set P_a of the concatenation ST. For $a = 0$, we set $\mathsf{P}_0 = [1\,..\,|ST|]$, while for $1 \le a \le \mathsf{type}(ST)$, let us define P_a as a 2^{a-1}-synchronizing set of W. Using Lemma 4, we make sure that $|\mathsf{P}_a| = \mathcal{O}(\frac{n}{2^a})$ and the set P_a can be constructed in $\mathcal{O}(n)$ time, which sums up to $\mathcal{O}(n \log n)$ across all types a.

Moreover, let us define

$$\mathsf{P}_a(S) = \{p \in \mathsf{P}_a \,:\, p \le |S|\} \qquad \text{and} \qquad \mathsf{P}_a(T) = \{p - |S| \,:\, p \in \mathsf{P}_a,\, p > |S|\}.$$

Intuitively, $\mathsf{P}_a(S)$ and $\mathsf{P}_a(T)$ represent the subsets of P_a corresponding to S and T, respectively. The following result relates these notions to the common factors of S and T.

▶ **Corollary 6.** *If $S[i\,..\,j] = F = T[i'\,..\,j']$, where F is a type-a string satisfying $\mathsf{per}(F) > \frac{1}{6}2^a$, then $\mathsf{succ}_{\mathsf{P}_a(S)}(i) - i = \mathsf{succ}_{\mathsf{P}_a(T)}(i') - i' < |F|$.*

Proof. The claim is trivial for $a = 0$ due to $\mathsf{succ}_{\mathsf{P}_0(S)}(i) - i = \mathsf{succ}_{\mathsf{P}_0(T)}(i') - i' = 0$. Otherwise, we have $|F| \ge 2^{a+1} - 1 > 3 \cdot 2^{a-1} - 1$ and $\mathsf{per}(F) \ge \frac{1}{6}2^a = \frac{1}{3}2^{a-1}$, so Lemma 5 yields $\mathsf{succ}_{\mathsf{P}_a(S)}(i) - i = \mathsf{succ}_{\mathsf{P}_a(T)}(i') - i' \le |F| - 2 \cdot 2^{a-1} < |F|$. ◀

3 Nonperiodic Case

We say that a string U of type a is *highly periodic* if $\mathsf{per}(U) \le \frac{1}{6}2^a$. We consider now $\mathsf{LCCF}_{a,b}(S,T) = (F,F')$ such that $F = UV$, $F' = VU$, U is of type a, V is of type b, and neither U nor V is highly periodic. We call it the *nonperiodic* case.

For a pair of fragments (u,v), by $\Gamma_{u,v}$ we denote a condition which states that u is followed by a fragment that matches v and by $\Delta_{u,v}$ we denote a condition which states that v follows a fragment that matches u. We say that two pairs of consecutive fragments, (x,y) in S and (z,t) in T, *agree* if and only if

$$\Gamma_{y,z} \text{ and } \Delta_{y,z} \text{ and } \Gamma_{t,x} \text{ and } \Delta_{t,x}.$$

We reduce the LCCF problem in this case to the following abstract problem; see Figure 2.

Fragment-Families-Problem
Input: Two collections $\mathcal{F}_1$ and $\mathcal{F}_2$ of pairs of consecutive fragments of a string W of length n, with $m = \lvert\mathcal{F}_1\rvert + \lvert\mathcal{F}_2\rvert$
Output: $(x,y) \in \mathcal{F}_1$ and $(z,t) \in \mathcal{F}_2$ that agree and maximize $\lvert x\rvert + \lvert y\rvert + \lvert z\rvert + \lvert t\rvert$

For a string $W \in \{S,T\}$ and a type a, we introduce the following set of *synchronizers*:

$$\mathsf{LeftSync}_a(W,i) = \mathsf{P}_a(W) \cap [i - 2^{a+2} + 2\,..\,i - 1],$$
$$\mathsf{RightSync}_a(W,i) = \mathsf{P}_a(W) \cap [i\,..\,i + 2^{a+2} - 3].$$

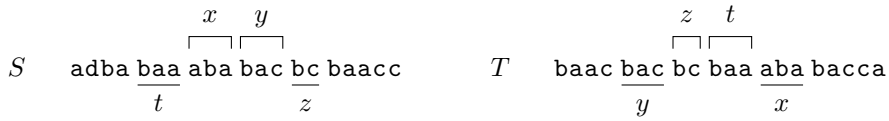


Figure 2 Pairs (x, y) and (z, t) agree; $txyz$ and $yztx$ form a common circular factor of S and T.

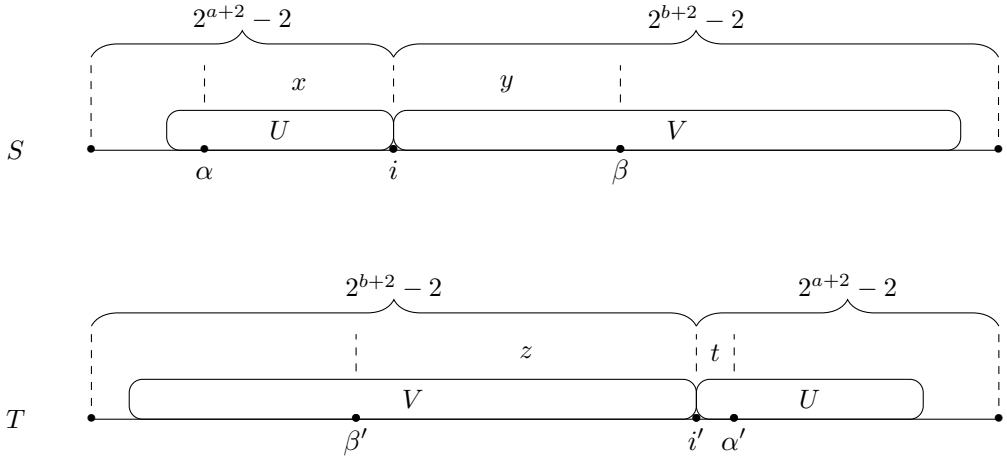


Figure 3 Assume that $\alpha \in \mathsf{LeftSync}_a(S, i)$, $\beta \in \mathsf{RightSync}'_b(S, i)$, $\beta' \in \mathsf{LeftSync}_a(T, i')$, $\beta' \in \mathsf{RightSync}'_b(T, i')$. Then $\Psi_S(\alpha, i, \beta) = (x, y)$ agrees with $\Psi_T(\beta', i', \alpha') = (z, t)$ if and only if there is a common circular factor of S and T: $F = UV$, $F' = VU$, where $U = tx$ and $V = yz$.

By $\mathsf{RightSync}'_a(W, i)$ we denote the singleton of the leftmost position in $\mathsf{RightSync}_a(W, i)$ or an empty set if there is no such position. For positions $\alpha \le i \le \beta$ in W, by

$$\Psi_W(\alpha, i, \beta) = (W[\alpha\,..\,i), W[i\,..\,\beta))$$

we denote a pair of consecutive fragments of W that are delimited by these positions. We then define the set of *candidates* (see Figure 3):

$$\mathsf{CAND}_{a,b}(W) = \{\Psi_W(\alpha, i, \beta) \ : \ \alpha \in \mathsf{LeftSync}_a(W, i),\ \beta \in \mathsf{RightSync}'_b(W, i),\ i \in [1\,..\,|W|]\}.$$

Using this terminology, an informal scheme of a general algorithm is as follows:

Algorithm 1: *Compute*-$\mathsf{LCCF}_{a,b}(S, T)$.

1 Compute the sets $\mathsf{CAND}_{a,b}(S)$, $\mathsf{CAND}_{b,a}(T)$
2 Find two pairs $(x, y) \in \mathsf{CAND}_{a,b}(S)$, $(z, t) \in \mathsf{CAND}_{b,a}(T)$ which agree and have maximum $|t| + |x| + |y| + |z|$
3 **return** $txyz$

▶ **Lemma 7** (Correctness for Nonperiodic Case). *The* $\mathsf{LCCF}_{a,b}$ *problem in the nonperiodic case can be reduced to the* Fragment-Families-Problem $(\mathcal{F}_S, \mathcal{F}_T)$ *for* $\mathcal{F}_S = \mathsf{CAND}_{a,b}(S)$ *and* $\mathcal{F}_T = \mathsf{CAND}_{b,a}(T)$.

Proof. Take a pair of fragments f_S of S and f_T of T such that f_S is an occurrence of a factor $F = UV$ and f_T is an occurrence of a factor $F' = VU$ such that U is of type a, V is of type b, and none of them is highly periodic. Denote by u_S and v_S the consecutive fragments of f_S corresponding to U and V, and similarly by v_T and u_T the consecutive fragments of f_T corresponding to V and U, and let $i = \mathsf{first}(v_S)$ and $j = \mathsf{first}(u_T)$. Moreover, consider $\alpha = \mathsf{succ}_{\mathsf{P}_a(S)}(\mathsf{first}(u_S))$ and $\alpha' = \mathsf{succ}_{\mathsf{P}_a(T)}(\mathsf{first}(u_T))$. By Corollary 6, $\alpha - \mathsf{first}(u_S) = \alpha' - \mathsf{first}(u_T) \leq |U|$. Consequently, $\alpha \in \mathsf{LeftSync}_a(S, i)$ and $\alpha' \in \mathsf{RightSync}'_a(T, j)$. Moreover, the relative position of α within u_S coincides with the relative position of α' within u_T. Symmetrically, $\beta = \mathsf{succ}_{\mathsf{P}_b(S)}(\mathsf{first}(v_S)) \in \mathsf{RightSync}'_b(S, i)$ and $\beta' = \mathsf{succ}_{\mathsf{P}_b(T)}(\mathsf{first}(v_T)) \in \mathsf{LeftSync}_b(T, j)$. Moreover, the relative position of β within v_S coincides with the relative position of β' within v_T. This means that there exists a pair $(x, y) \in \mathsf{CAND}_{a,b}(S)$ such that $x = S[\alpha \,..\, i) = T[\alpha' \,..\, i' + |U|)$ and $y = S[i \,..\, \beta) = T[i' - |V| \,..\, \beta')$, and a pair $(z, t) \in \mathsf{CAND}_{b,a}(T)$ such that $z = T[\beta' \,..\, i') = S[\beta \,..\, i + |V|)$ and $t = T[i' \,..\, \alpha') = S[i - |U| \,..\, \alpha)$. The equalities listed above imply that the two pairs agree.

Conversely, for every two pairs $(x, y) \in \mathsf{CAND}_{a,b}(S), (z, t) \in \mathsf{CAND}_{b,a}(T)$ that agree, there exists a factor F in string S matching $txyz$ and a factor F' matching $yztx$ in T. Thus, there is a one-to-one correspondence between pairs that agree and fragments of strings of right type that are cyclic shifts. Hence, by finding two pairs that agree and maximize $|x| + |y| + |z| + |t|$, we construct a solution to the $\mathsf{LCCF}_{a,b}$ problem. ◀

▶ **Lemma 8** (Complexity for Nonperiodic Case). *In the nonperiodic case, the LCCF problem can be reduced in $\mathcal{O}(n \log n)$ time to $\mathcal{O}(\log n)$ instances of the* Fragment-Families-Problem *with $m = \mathcal{O}(n)$.*

Proof. For each type $a \in [0 \,..\, \mathsf{type}(\mathsf{LCF}(S, T))]$, we compute the synchronizing sets $\mathsf{P}_a(S)$ and $\mathsf{P}_a(T)$ in $\mathcal{O}(n)$ time using Lemma 4 as described in Section 2.3. Observe that each position $p \in \mathsf{P}_a(W)$ may belong to just $\mathcal{O}(2^a)$ sets $\mathsf{LeftSync}_a(W, i)$. Consequently, the total size of the sets $\mathsf{LeftSync}_a(W, i)$ (for a fixed type a) is $\mathcal{O}(n)$, and we can compute them in $\mathcal{O}(n)$ time using a sliding window. The running time across all types a is $\mathcal{O}(n \log n)$.

The family $\mathsf{CAND}_{a,b}(W)$ is constructed straight from the definition based on the sets $\mathsf{LeftSync}_a(W, i)$ and the synchronizing set $\mathsf{P}_b(W)$. As $|\mathsf{CAND}_{a,b}(W)| \leq \sum_i |\mathsf{LeftSync}_a(W, i)|$, the size of this family is $\mathcal{O}(n)$, and the construction time is also linear. Across all $\mathcal{O}(\log n)$ pairs $a, b \geq 0$ with $\mathsf{type}(\mathsf{LCF}(S, T)) - 1 \leq \max(a, b) \leq \mathsf{type}(\mathsf{LCF}(S, T))$, the overall time complexity is $\mathcal{O}(n \log n)$. ◀

4 Periodic Case

We consider now $\mathsf{LCCF}_{a,b}(S, T) = (F, F')$ such that $F = UV$, $F' = VU$, U is of type a, V is of type b, and both U and V are highly periodic.

Recall that a *Lyndon string* is a string that is lexicographically smaller than all its non-trivial cyclic shifts. If W is a weakly periodic string with the shortest period p, then its *Lyndon root* λ is the Lyndon string that is a cyclic shift of $W[1 \,..\, p]$. A Lyndon representation of W is then (c, e, d) such that $W = \lambda' \lambda^e \lambda''$ where $|\lambda'| = c < |\lambda|$ and $|\lambda''| = d < |\lambda|$; see [13]. Lyndon strings have the following synchronization property that follows from the periodicity lemma: if λ is a Lyndon string, then it has exactly two occurrences in λ^2; see [12].

For a string W, by $\mathsf{HPerPref}_a(W)$ and $\mathsf{HPerSuf}_a(W)$ we denote the longest highly periodic type-a prefix and type-a suffix of W, respectively (or the empty string if there is no appropriate prefix or suffix). Let us start with the following simple observation; see Figure 4.

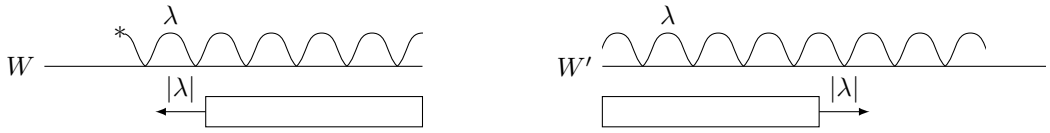


Figure 4 Illustration of Observation 9. A highly periodic suffix of W that is also a prefix of W' of length at most $\min(|X|, |Y|) - |\lambda|$ can be extended by $|\lambda|$ characters.

▶ **Observation 9.** *Let W and W' be two strings for which the strings $X = \mathsf{HPerSuf}_a(W)$ and $Y = \mathsf{HPerPref}_a(W')$ have the same Lyndon root λ. Then the longest suffix of W that is also a prefix of W' has length greater than* $\min(|X|, |Y|) - |\lambda|$.

For a position i in a string W and a type a, we denote by $\mathsf{LeftLyn}_a(W, i)$ the set of positions where the first, second, and last occurrence of the Lyndon root start in $\mathsf{HPerSuf}_a(W[1\,..\,i))$. If the latter string is empty, we assume that $\mathsf{LeftLyn}_a(W, i)$ is also empty. Similarly, we define $\mathsf{RightLyn}_a(W, i)$ as the set of positions where the first, second to last, and last occurrence of the Lyndon root start in $\mathsf{HPerPref}_a(W[i\,..\,|W|])$. We can redefine the set of candidates as follows (see Figure 5)

$$\mathsf{CAND}_{a,b}(W) = \{\Psi_W(\alpha, i, \beta) \ : \ \alpha \in \mathsf{LeftLyn}_a(W, i),\ \beta \in \mathsf{RightLyn}_b(W, i),\ i \in [1\,..\,|W|]\}.$$

The following lemma implies the correctness of our algorithm in this case.

▶ **Lemma 10** (Correctness for Periodic Case)**.** *The $\mathsf{LCCF}_{a,b}$ problem in the periodic case can be reduced to the* Fragment-Families-Problem *$(\mathcal{F}_S, \mathcal{F}_T)$ for the redefined sets $\mathcal{F}_S = \mathsf{CAND}_{a,b}(S)$ and $\mathcal{F}_T = \mathsf{CAND}_{b,a}(T)$.*

Proof. Take a pair of fragments f_S of S and f_T of T such that f_S is an occurrence of a factor $F = UV$ and f_T is an occurrence of a factor $F' = VU$ ($(F, F') = \mathsf{LCCF}_{a,b}(S, T)$) such that U is of type a, V is of type b, and both U and V are highly periodic. Denote by u_S and v_S the consecutive fragments of f_S corresponding to U and V, and similarly by v_T and u_T the consecutive fragments of f_T corresponding to V and U, and let $i = \mathsf{first}(v_S)$ and $j = \mathsf{first}(u_T)$.

Let $X = \mathsf{HPerSuf}_a(S[1\,..\,i))$ and $Y = \mathsf{HPerPref}_a(T[j\,..\,n])$. Note that u_S is a highly periodic suffix of X and that X has the same period as U (a different period would contradict the periodicity lemma). Symmetrically, u_T is a highly periodic prefix of Y and Y has the same period as U. Let λ be the Lyndon root of U, and observe that λ is also the Lyndon root of X and Y. By Observation 9, we have

$$|X| - |U| < |\lambda| \text{ or } |Y| - |U| < |\lambda|,$$

as otherwise we would be able to find a Common Circular Factor of type (a, b) that is longer by $|\lambda|$, thus contradicting our choice of f_S ad f_T.

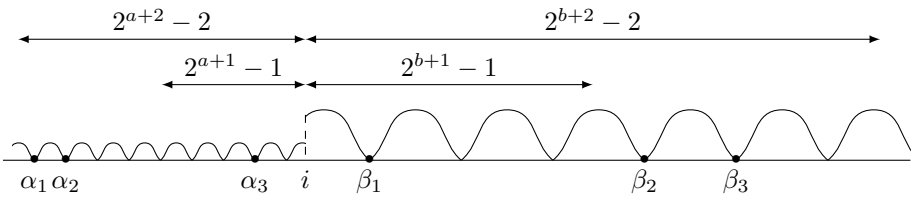


Figure 5 In this case, $\mathsf{CAND}_{a,b}(W)$ contains $\Psi_W(\alpha_p, i, \beta_q)$ for $p, q \in \{1, 2, 3\}$.

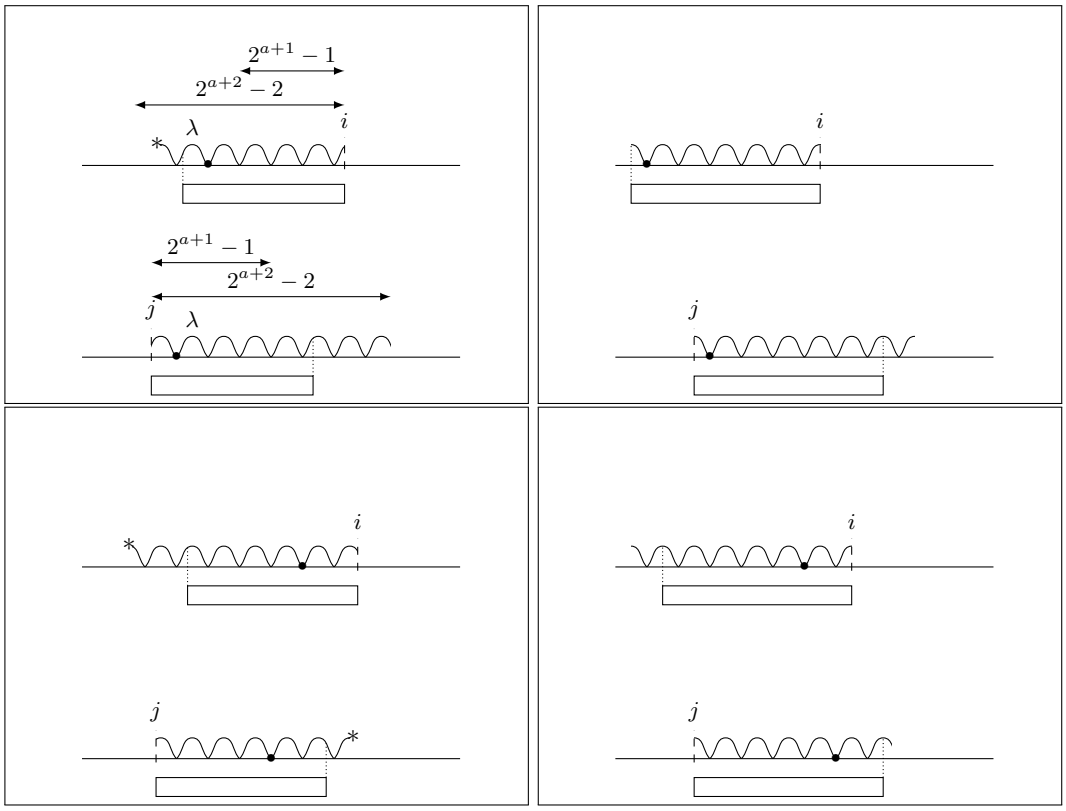


Figure 6 Four cases from the proof of Lemma 10.

If $|X|-|U|<|\lambda|$, then the first occurrence of λ in u_S is also the first or second occurrence of Lyndon root in X. This is due to the synchronization property of Lyndon strings. Moreover, the first λ in u_T is also the first occurrence of λ in Y. On the other hand, if $|Y|-|U|<|\lambda|$, then the last occurrence of λ in u_T is the last or the second to last occurrence of λ in u_T, whereas the last occurrence of λ in u_S is also the last occurrence of λ in X. In either case, u_S and u_T contain a pair of corresponding occurrences of λ whose starting positions belong to $\mathsf{LeftLyn}_a(S,i)$ and $\mathsf{RightLyn}_b(T,j)$, respectively; see Figure 6.

As the same reasoning can be applied to v_S and v_T, there exist pairs $(x,y)\in\mathsf{CAND}_{a,b}(S)$ and $(z,t)\in\mathsf{CAND}_{b,a}(T)$ which correspond to our choice of occurrences of the Lyndon roots. These pairs agree and $|x|+|y|+|z|+|t|=|F|$; thus, Fragment-Families-Problem $(\mathcal{F}_S,\mathcal{F}_T)$ will find a solution at least that good.

The converse direction is identical to the one from the proof of Lemma 7. ◀

We proceed with an efficient implementation. A *run* in string W is a maximal weakly periodic fragment $W[i\,..\,j]$ with a given period p. We use *2-period queries* which, given a weakly periodic fragment u of a string, compute its shortest period and the run of the same period it belongs to. Such queries can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing [27, 24] (for a simplified solution, see [5]). Let us also recall that the Lyndon representation of a run can be computed in constant time after linear-time preprocessing [13].

▶ **Lemma 11** (Complexity for Periodic Case). *In the periodic case, the LCCF problem can be reduced in $\mathcal{O}(n\log n)$ time to $\mathcal{O}(\log n)$ instances of the* Fragment-Families-Problem *with $m=\mathcal{O}(n)$.*

Proof. First, we spend $\mathcal{O}(n \log n)$ time in total to construct the sets $\mathsf{LeftLyn}_a(W,i)$ and $\mathsf{RightLyn}_a(W,i)$ for each $W \in \{S,T\}$ and $a \leq \mathsf{type}(W)$. For this, we use the following result:

▷ Claim 12. After $\mathcal{O}(n)$-time preprocessing of a string W, each set $\mathsf{LeftLyn}_a(W,i)$ and $\mathsf{RightLyn}_a(W,i)$ can be constructed in $\mathcal{O}(1)$ time.

Proof. To compute $\mathsf{HPerPref}_a(u)$ for a fragment u of W, it suffices to ask a 2-period query for $u[1\,..\,2^{a+1}-1]$ (see [27, 5]), determine the Lyndon representation of the resulting run (see [13, 5]), and then trim its Lyndon representation to u. A symmetric solution works for $\mathsf{HPerSuf}_a(u)$. This allows us to construct the sets $\mathsf{LeftLyn}_a$ and $\mathsf{RightLyn}_a$. ◁

Now, the family $\mathsf{CAND}_{a,b}(W)$, which is of size at most $9n$, can be computed in $\mathcal{O}(n)$ based on the sets $\mathsf{LeftLyn}_a(W,i)$ and $\mathsf{RightLyn}_b(W,i)$. Our reduction to the Fragment-Families-Problem problem relies on $\mathcal{O}(\log n)$ such families constructed for pairs $a,b \geq 0$ such that $\mathsf{type}(\mathsf{LCF}(S,T)) - 1 \leq \max(a,b) \leq \mathsf{type}(\mathsf{LCF}(S,T))$; see Section 2.3 and Lemma 8. ◀

5 General Case

Finally, we consider the general problem of computing $\mathsf{LCCF}_{a,b}(S,T) = (F,F')$. It can be reduced to several instances of the Fragment-Families-Problem directly by combining the techniques of the previous two sections.

▶ **Lemma 13** (Correctness for General Case). *The* $\mathsf{LCCF}_{a,b}$ *problem can be reduced to the* Fragment-Families-Problem $(\mathcal{F}_S, \mathcal{F}_T)$.

Proof. In the proofs of Lemmas 7 and 10 the U and V parts of the factors were considered separately. Hence, it is enough to define $\mathsf{CAND}_{a,b}(W)$ as

$$\mathsf{CAND}_{a,b}(W) = \{\Psi_W(\alpha,i,\beta) \;:\; \alpha \in \mathsf{LeftSync}_a(W,i) \cup \mathsf{LeftLyn}_a(W,i),\ \beta \in \mathsf{RightSync}'_b(W,i) \cup \mathsf{RightLyn}_b(W,i),\ i \in [1\,..\,|W|]\}.$$

Depending on whether U or V is highly periodic or not, the existence of an agreeing pair $(x,y) \in \mathcal{F}_S$ and $(z,t) \in \mathcal{F}_T$ can be shown be repeating the arguments in the proofs of Lemma 7 or Lemma 10, respectively. ◀

▶ **Lemma 14** (Complexity for General Case). *The LCCF problem can be reduced in* $\mathcal{O}(n \log n)$ *time to* $\mathcal{O}(\log n)$ *instances of the* Fragment-Families-Problem *with* $m = \mathcal{O}(n)$.

Proof. The families can be computed combining the methods from Lemmas 8 and 11, obtaining the desired complexities and sizes. ◀

6 Solution to Fragment-Families-Problem

In this section we show how to solve the Fragment-Families-Problem for a string W of length n by a reduction to intersecting special 4-dimensional rectangles.

First, we give a geometric interpretation of two predicates:

- a factor U has an occurrence in W starting at position q (is a prefix of the suffix starting at position q), and
- U has an occurrence ending at position q (is a suffix of the prefix ending at position q)

relating them to the membership of q in a corresponding subinterval of $[1\,..\,n]$.

Let us recall that the suffix array [30] of a string W, SA_W, is a permutation of $[1\,..\,n]$ such that $W[\mathsf{SA}_W[i]\,..\,n] < W[\mathsf{SA}_W[i+1]\,..\,n]$ for every $i \in [1\,..\,n-1]$. By $\mathsf{FirstPos}(U)$ let us denote the set of starting positions of occurrences of U in W. Our geometric interpretation is possible due to the following well known fact (see [12]).

▶ **Observation 15.** *The set* $\mathsf{FirstPos}(U)$ *consists of consecutive elements in* SA_W.

Let $\mathsf{LastPos}(U)$ be the set of ending positions of occurrences of U in W. We also use the FirstPos, LastPos notation for fragments which means operations on corresponding factors.

▶ **Observation 16.**

1. *A fragment u is a prefix/suffix of the suffix starting (prefix ending) at position q if and only if $q \in \mathsf{FirstPos}(u)$, $q \in \mathsf{LastPos}(u)$, respectively.*
2. $\Gamma_{u,v} \equiv ((\mathsf{last}(u)+1) \in \mathsf{FirstPos}(v))$ *and* $\Delta_{u,v} \equiv ((\mathsf{first}(v)-1) \in \mathsf{LastPos}(u))$.

We define a *d-rectangle* ($d \geq 2$) as a Cartesian product of d closed intervals, such that at least $d-2$ of them are singletons. E.g., $\{3\} \times [2\,..\,5] \times [1\,..\,7] \times \{0\}$ is a 4-rectangle. In other words, a d-rectangle is an isothetic hyperrectangle of dimension at most 2.

By $\mathcal{I}(U)$ and $\mathcal{J}(U)$ we denote the subintervals of $[1\,..\,n]$ that correspond to the intervals of $\mathsf{FirstPos}(U)$ in the suffix array SA_W and of $\mathsf{LastPos}(U)$ in the (analogously defined) prefix array of W, PA_W, respectively, as stated in Observation 15. (PA_W is a permutation of $[1\,..\,n]$ such that $W[1\,..\,\mathsf{PA}_W[i]]^R < W[1\,..\,\mathsf{PA}_W[i+1]]^R$ for every $i \in [1\,..\,n-1]$.) For pairs (x,y) and (z,t) of consecutive fragments, we denote:

$$\begin{aligned}\mathsf{RECT}(x,y) &= \mathcal{I}(x) \times \mathcal{J}(y) \times \{\mathsf{SA}_W^{-1}[\mathsf{last}(y)+1]\} \times \{\mathsf{PA}_W^{-1}[\mathsf{first}(x)-1]\},\\ \mathsf{RECT}'(z,t) &= \{\mathsf{SA}_W^{-1}[\mathsf{last}(t)+1]\} \times \{\mathsf{PA}_W^{-1}[\mathsf{first}(z)-1]\} \times \mathcal{I}(z) \times \mathcal{J}(t).\end{aligned}$$

Observation 16.2 now implies the following.

▶ **Observation 17.** *Two pairs of consecutive fragments (x,y), (z,t) agree if and only if* $\mathsf{RECT}(x,y) \cap \mathsf{RECT}'(z,t) \neq \emptyset$.

Two d-rectangles $[a_1\,..\,b_1] \times \cdots \times [a_d\,..\,b_d]$ and $[a'_1\,..\,b'_1] \times \cdots \times [a'_d\,..\,b'_d]$ are called *compatible* if, for each $i \in \{1,\ldots,d\}$, $[a_i\,..\,b_i]$ or $[a'_i\,..\,b'_i]$ is a singleton. Let us note that the 4-rectangles in the above observation are compatible.

6.1 Intersecting 4D Rectangles

We consider two families of 4-rectangles with weights and wish to find a pair of intersecting rectangles, one per family, with maximum total weight. The general problem of finding such an intersection of two families of m weighted hyperrectangles in d dimensions can be solved in $\mathcal{O}(m \log^{2d} m)$ time by an adaptation of a classic approach [14]. Below, we consider a special variant of the problem that has a much more efficient solution.

Max-Weight Intersection of Compatible Rectangles in 4D
Input: Two families $\mathcal{R}_1$ and $\mathcal{R}_2$ of 4-rectangles in $\mathbb{Z}^4$ with integer weights containing m rectangles in total, such that each $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$ are compatible
Output: Check if there is an intersecting pair of 4-rectangles $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$ and, if so, compute the maximum total weight of such a pair

A very similar problem was considered as Problem 3 in [19] for an arbitrary d. The sole difference is that the weight of an intersection of two d-rectangles $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$ in that problem was the maximum ℓ_1-norm of a point in $R_1 \cap R_2$. A solution to Problem 3 for $d = 4$ in the case that the 4-rectangles are compatible working in $\mathcal{O}(m \log^3 m)$ time and $\mathcal{O}(m \log^2 m)$ space was given as [19, Lemma 5.8]. The algorithm presented in that lemma actually solves the Max-Weight Intersection of Compatible Rectangles in 4D problem and applies it for specific weight assignment of the 4-rectangles on the input. It uses hyperplane sweep and a variant of an interval stabbing problem. Henceforth, we will use the following result.

▶ **Fact 18** (see [19, Lemma 5.8]). MAX-WEIGHT INTERSECTION OF COMPATIBLE RECTANGLES IN 4D *can be solved in* $\mathcal{O}(m \log^3 m)$ *time and* $\mathcal{O}(m \log^2 m)$ *space.*

6.2 Algorithm for Fragment-Families-Problem

Let us recall that the suffix tree [35] of a string W, ST_W, is a compacted trie of all the suffixes of W. It can be computed in $\mathcal{O}(n)$ time (see [15]) and reading the suffixes of W in its preorder traversal yields the suffix array of W. An efficient implementation of Observation 15 is known; see [1, 25].

▶ **Lemma 19.** *The sets* $\mathcal{I}(u)$ *and* $\mathcal{J}(u)$ *can be computed in* $\mathcal{O}(n+m)$ *total time for a batch of* m *fragments* u *of a length-*n *string* W*.*

Proof. Without loss of generality, it suffices to show how to compute $\mathcal{I}(u)$. For every explicit node of ST_W, we can compute the interval of elements of SA_W that are located in its subtree. This can be done in a bottom-up order in $\mathcal{O}(n)$ time.

A weighted ancestor query in ST_W, given a terminal node w and positive integer d, returns the ancestor of w located at depth d (being an explicit or implicit node). A batch of m such queries (for any tree of n nodes with positive integer weights of edges) can be answered in $\mathcal{O}(n+m)$ time; see [25, Section 7.1].

A weighted ancestor query can be used to compute, given a fragment u of W, the corresponding (explicit or implicit) node w of ST_W. The interval stored in the nearest explicit descendant of w equals $\mathcal{I}(u)$. ◀

We are now ready to show a solution to the FRAGMENT-FAMILIES-PROBLEM.

▶ **Lemma 20.** *The* FRAGMENT-FAMILIES-PROBLEM *can be solved in* $\mathcal{O}(n + m \log^3 m)$ *time and* $\mathcal{O}(n + m \log^2 m)$ *space.*

Proof. We construct families $\mathcal{R}_1$ and $\mathcal{R}_2$ of weighted 4-rectangles. For every $(x, y) \in \mathcal{F}_1$, we add $\mathsf{RECT}(x, y)$ to $\mathcal{R}_1$ with weight $|x| + |y|$. For every $(z, t) \in \mathcal{F}_2$, we add $\mathsf{RECT}'(z, t)$ to $\mathcal{R}_2$ with weight $|z| + |t|$. By Observation 17, the solution to MAX-WEIGHT INTERSECTION OF COMPATIBLE RECTANGLES IN 4D for $\mathcal{R}_1$ and $\mathcal{R}_2$ is the solution to FRAGMENT-FAMILIES-PROBLEM$(\mathcal{F}_1, \mathcal{F}_2)$.

Note that we have $|\mathcal{R}_1| = |\mathcal{F}_1|$ and $|\mathcal{R}_2| = |\mathcal{F}_2|$. Using Lemma 19 and a linear-time algorithm for constructing SA_W and PA_W (and SA_W^{-1} and PA_W^{-1}) [15], computation of 4-rectangles RECT, RECT' can be done in $\mathcal{O}(n+m)$ time in total. Finally, MAX-WEIGHT INTERSECTION OF COMPATIBLE RECTANGLES IN 4D can be solved in $\mathcal{O}(m \log^3 m)$ time and $\mathcal{O}(m \log^2 m)$ space. ◀

As a consequence of Lemmas 13 and 14 and the above lemma, we obtain the main result.

▶ **Theorem 1** (Main Result). *The Longest Common Circular Factor problem on two strings of length at most* n *can be solved in* $\mathcal{O}(n \log^4 n)$ *time and* $\mathcal{O}(n \log^2 n)$ *space.*

7 Conclusions

We have presented an $\mathcal{O}(n \log^4 n)$-time algorithm for computing the Longest Common Circular Factor (LCCF) of two strings of length n. Let us recall that the Longest Common Factor (LCF) of two strings can be computed in $\mathcal{O}(n)$ time. We leave an open question if the LCCF problem can also be solved in linear time.

References

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

2 Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. 40 years of suffix trees. *Communications of the ACM*, 59(4):66–73, 2016. doi:10.1145/2810036.

3 Tanver Athar, Carl Barton, Widmer Bland, Jia Gao, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Fast circular dictionary-matching algorithm. *Mathematical Structures in Computer Science*, 27(2):143–156, 2017. doi:10.1017/S0960129515000134.

4 Md. Aashikur Rahman Azim, Costas S. Iliopoulos, Mohammad Sohel Rahman, and M. Samiruzzaman. A fast and lightweight filter-based algorithm for circular pattern matching. In Pierre Baldi and Wei Wang, editors, *5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2014*, pages 621–622. ACM, 2014. doi:10.1145/2649387.2660804.

5 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

6 Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9:9, 2014. doi:10.1186/1748-7188-9-9.

7 Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Average-Case Optimal Approximate Circular String Matching. In Adrian-Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications, LATA 2015*, volume 8977 of *LNCS*, pages 85–96. Springer, 2015. doi:10.1007/978-3-319-15579-1_6.

8 Or Birenzwige, Shay Golan, and Ely Porat. Locally Consistent Parsing for Text Indexing in Small Space. ArXiv preprint, 2018. arXiv:1812.00359.

9 Domenico Cantone, Simone Faro, and Arianna Pavone. Sequence Searching Allowing for Non-Overlapping Adjacent Unbalanced Translocations. ArXiv preprint, 2018. arXiv:1812.00421.

10 Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *The Computer Journal*, 57(5):731–743, 2014. doi:10.1093/comjnl/bxt023.

11 Da-Jung Cho, Yo-Sub Han, and Hwee Kim. Alignment with non-overlapping inversions and translocations on two strings. *Theoretical Computer Science*, 575:90–101, 2015. doi:10.1016/j.tcs.2014.10.036.

12 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.

13 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

14 Herbert Edelsbrunner. A new approach to rectangle intersections, Part I. *International Journal of Computer Mathematics*, 13(3–4):209–219, 1983. doi:10.1080/00207168308803364.

15 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

16 Nathan J. Fine and Herbert S. Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965. doi:10.1090/S0002-9939-1965-0174934-9.

17 Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009. doi:10.1016/j.jda.2008.09.001.

18 Kimmo Fredriksson and Gonzalo Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9:1.4:1–1.4:47, 2004. doi:10.1145/1005813.1041513.

19 Szymon Grabowski, Tomasz Kociumaka, and Jakub Radoszewski. On Abelian Longest Common Factor with and without RLE. *Fundamenta Informaticae*, 163(3):225–244, 2018. doi:10.3233/FI-2018-1740.

20 Tommi Hirvola and Jorma Tarhio. Bit-Parallel Approximate Matching of Circular Strings with k Mismatches. *ACM Journal of Experimental Algorithmics*, 22:1.5:1–1.5:22, 2017. doi:10.1145/3129536.

21 Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and Indexing Circular Patterns. In Mourad Elloumi, editor, *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.

22 Costas S. Iliopoulos and M. Sohel Rahman. Indexing Circular Patterns. In Shin-Ichi Nakano and Md. Saidur Rahman, editors, *Algorithms and Computation, WALCOM 2008*, volume 4921 of *LNCS*, pages 46–57. Springer, 2008. doi:10.1007/978-3-540-77891-2_5.

23 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: Sublinear-time BWT construction and optimal LCE data structure. In Edith Cohen, editor, *51st Annual ACM Symposium on Theory of Computing, STOC 2019*. ACM, 2019. doi:10.1145/3313276.3316368.

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

25 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation. ArXiv preprint, 2019. arXiv:1107.2422v2.

26 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient Data Structures for the Factor Periodicity Problem. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval, SPIRE 2012*, volume 7608 of *LNCS*, pages 284–294. Springer, 2012. doi:10.1007/978-3-642-34109-0_30.

27 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal Pattern Matching Queries in a Text and Applications. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

28 Jie Lin and Donald A. Adjeroh. All-Against-All Circular Pattern Matching. *The Computer Journal*, 55(7):897–906, 2012. doi:10.1093/comjnl/bxr126.

29 Jie Lin, Yue Jiang, and Don Adjeroh. Circular Pattern Discovery. *The Computer Journal*, 58(5):1061–1073, 2015. doi:10.1093/comjnl/bxu009.

30 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

31 Hideaki Ogiwara, Takashi Kohno, Hirofumi Nakanishi, Kazuhiro Nagayama, Masanori Sato, and Jun Yokota. Unbalanced translocation, a major chromosome alteration causing loss of heterozygosity in human lung cancer. *Oncogene*, 27(35):4788–4797, 2008. doi:10.1038/onc.2008.113.

32 Robert Susik, Szymon Grabowski, and Sebastian Deorowicz. Fast and Simple Circular Pattern Matching. In Aleksandra Gruca, Tadeusz Czachórski, and Stanisław Kozielski, editors, *Man-Machine Interactions, ICMMI 2013*, volume 242 of *Advances in Intelligent Systems and Computing*, pages 537–544. Springer, 2013. doi:10.1007/978-3-319-02309-0_59.

33 Dorothy Warburton. De novo balanced chromosome rearrangements and extra marker chromosomes identified at prenatal diagnosis: clinical significance and distribution of breakpoints. *The American Journal of Human Genetics*, 49(5):995–1013, 1991. PMID:1928105.

34 Brooke Weckselblatt, Karen E. Hermetz, and M. Katharine Rudd. Unbalanced translocations arise from diverse mutational mechanisms including chromothripsis. *Genome Research*, 25(7):937–947, 2015. doi:10.1101/gr.191247.115.

35 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT 1973*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society. doi:10.1109/SWAT.1973.13.

Simulating the DNA Overlap Graph in Succinct Space

Diego Díaz-Domínguez[1]

CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Chile
Department of Computer Science, University of Chile, Chile
diediaz@dcc.uchile.cl

Travis Gagie

School of Computer Science and Telecommunications, Diego Portales University, Chile
CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Chile
travis.gagie@gmail.com

Gonzalo Navarro

CeBiB – Center for Biotechnology and Bioengineering, University of Chile, Chile
Department of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract

Converting a set of sequencing reads into a lossless compact data structure that encodes all the relevant biological information is a major challenge. The classical approaches are to build the string graph or the *de Bruijn* graph (dBG) of some order k. Each has advantages over the other depending on the application. Still, the ideal setting would be to have an index of the reads that is easy to build and can be adapted to any type of biological analysis. In this paper we propose *rBOSS*, a new data structure based on the Burrows-Wheeler Transform (BWT), which gets close to that ideal. Our *rBOSS* simultaneously encodes all the dBGs of a set of sequencing reads up to some order k, and for any dBG node v, it can compute in $\mathcal{O}(k)$ time all the other nodes whose labels have an overlap of at least m characters with the label of v, with m being a parameter. If we choose the parameter k equal to the size of the reads (assuming that all have equal length), then we can simulate the overlap graph of the read set. Instead of storing the edges of this graph explicitly, *rBOSS* computes them on the fly as we traverse the graph. As most BWT-based structures, *rBOSS* is unidirectional, meaning that we can retrieve only the suffix overlaps of the nodes. However, we exploit the property of the DNA reverse complements to simulate bi-directionality. We implemented a genome assembler on top of *rBOSS* to demonstrate its usefulness. The experimental results show that, using $k = 100$, our *rBOSS*-based assembler can process ~500K reads of 150 characters long each (a `FASTQ` file of 185 MB) in less than 15 minutes and using 110 MB in total. It produces contigs of mean sizes over 10,000, which is twice the size obtained by using a pure de Bruijn graph of fixed length k.

2012 ACM Subject Classification Applied computing → Computational biology; Information systems → Data compression

Keywords and phrases Overlap graph, de Bruijn graph, DNA sequencing, Succinct ordinal trees

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.26

Funding Partially supported by Basal Funds FB0001, Conicyt, Chile; by a Conicyt Ph.D. Scholarship; by Fondecyt Grants 1-171058 and 1-170048, Chile; and by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement No 690941].

Acknowledgements We thank the reviewers for their helpful comments.

[1] Corresponding author

1 Introduction

Obtaining and extracting the relevant information from a collection of DNA sequencing *reads*[2], for assembly and other analysis purposes, usually requires a lot of time and space. The techniques for compressed indexing developed in recent years (see [27] for a full review) have significantly contributed to reduce the computational costs. There is still no technique, however, that can preprocess the reads and represent all the relevant information in succinct space so that it can be used effectively.

The classical plain, and lossless, data structure to analyze reads is the *overlap graph*. In this model, each node represents a particular read, and two nodes v and v' are connected by an edge with weight $o \geq m$ if o is the maximum length of a suffix of v that matches a prefix of v', where m is a parameter to filter out spurious overlaps. Computing the overlap graph from a set of reads is not difficult: it can be built from the suffix tree of the set or even from its *Burrows-Wheeler Transform (BWT)* [8, 33] or the Longest Common Prefix array (LCP) [4]. Representing the graph, however, is expensive: a quadratic number of edges may have to be stored. A popular solution is to perform *structural compression* over the graph by removing the transitive edges. The resulting graph is usually called the *string graph* [25, 34] or the *irreductible overlap graph* [24]. This approach, however, limits the applications of the data strucutre, because it removes information from the graph that can be useful.

Historically, string graphs have been used mainly in the context of *genome assembly* [12, 25, 36], but as sequencing datasets have grown over the years, they have been discarded in favor of other lossy, but more succinct, representations. The most famous of these representations is the *de Bruijn* graph (dBG). This data structure encodes the relationship between all the substrings of length k in the set. A dBG is easy to construct and it can be represented succinctly [6]. Besides, it encodes the context of the substrings of length $k-1$ in its topology. Thus, for instance, if a substring of length $k-1$ appears in several contexts of the set, then its dBG node will have several edges. As for the string graph, the first application of the dBG was the assembly of genomes [2, 21, 31, 35], but through the years its use has been extended to other kind of analyses [7, 16].

The disadvantage of dBGs, however, is that they are lossy, because the information we can retrieve from them is limited by k. A way to overcome some of the restrictions imposed by k is to add variable-order functionality to the dBG, that is, to encode several dBGs with different values for k at the same time. The contexts of the graph can then be shortened or lengthened on demand, depending on the need to have more or less edges from a node. Some succinct dBG representations supporting variable-order functionality up to some maximum order k have been proposed [5], but they increase the space requirements by a $\log k$ factor. Besides, even using variable-order functionality, the data structure remains lossy, because the order of the graph can be lengthened only up to k. By choosing k equal to the size of the reads, the variable-order dBG becomes lossless, and equivalent to the overlap graph, but the $\log k$ factor becomes significant for the typical read sizes.

Almost every analysis over DNA sequencing data can be reduced to looking for suffix-prefix overlaps between the reads, and in this regard the dBG has adapted well to many bioinformatic applications because it is a lightweight (lossy) representation of the overlaps. Still, searching for biological signals in a dBG requires the detection of complex graph substructures such as bubbles, super bubbles, tips, and so on, and those can be expensive to find. The overlap graph, on the other hand, is a much simpler model. It can be adapted

[2] A string that represents the inferred sequence of base pairs in a segment of a DNA molecule.

to different applications other than assembly in the same way as the dBG, but it has the advantage of being lossless and not requiring too much preprocessing after its construction. The problem, as stated before, is the space to encode the edges of the graph. A better approach would be to have a compact data structure that can quickly compute the overlaps (i.e., the edges) on the fly instead of storing them explicitly. Such a solution would require a moderate preprocessing of the read set, would retain all the information, and would use a reasonable amount of space.

Our contribution. We address the problem of succinctly representing and analyzing a collection $\mathcal{R}$ of sequencing reads. To this end, we define a new compact data structure we call *rBOSS*. It is an intermediate structure between a dBG and an overlap graph, because it can compute the context of the sequences in the same way a dBG does, but it can also compute on the fly the overlaps between different substrings of $\mathcal{R}$. The *rBOSS* index is based on *BOSS* [6], a BWT-based representation of dBGs, which is augmented with a tree we call the *overlap tree*. This tree increases the size of the data structure by $4n + o(n)$ bits, where n is the number of nodes in the dBG encoded by *BOSS*. By choosing k equal to the length of the reads (which we assume to have all the same length), we can simulate in compressed space an overlap graph whose edges have a weight $o \geq m$, where m is given as a parameter. The simulation of the graph builds on the basic primitives `nextcontained` and `buildL`. Our overlap tree reduces their time complexity from $\mathcal{O}(k^2)$ to $\mathcal{O}(1)$ and $\mathcal{O}(k)$, respectively.

In addition to *rBOSS*, we also formalize the idea of weighting the overlap graph edges according to transitive connections, and explain how this new weighting scheme can be used to solve biological problems other than assembly. To our knowledge, this is the first time a measure of this kind is proposed for overlap graphs. Finally, we demonstrate the usefulness of *rBOSS* by implementing a genome assembler on top of it. Our experimental results show that, by using $k = 100$, the assembler can process ~500K reads of 150 characters long each in less than 15 minutes and using 110 MB in total. It produces contigs of mean sizes over 10,000, which is twice the size obtained by using a pure dBG of fixed length k.

2 Preliminaries

DNA strings. A DNA sequence R is a string over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ (which we map to $[2..\sigma]$), where every symbol represents a particular nucleotide in a DNA molecule. The DNA *complement* is a permutation $\pi[2..\sigma]$ that reorders the symbols in Σ exchanging `a` with `t` and `c` with `g`. The *reverse complement* of R, denoted R^{rc}, is a string transformation that reverses R and then replaces every symbol $R[i]$ by its complement $\pi(R[i])$. For technical convenience we add to Σ the so-called *dummy* symbol \$, which is always mapped to 1.

De Bruijn graphs. A de Bruijn Graph (dBG) [11] of order k of a set of strings $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$, $DBG_{\mathcal{R},k}$, is a labeled directed graph $G = (V, E)$ where every node $v \in V$ is labeled by a distinct substring of $\mathcal{R}$ of length $k-1$, and every edge $(v, u) \in E$ represents the substring $S[1..k]$ of $\mathcal{R}$ such that v is labeled by the prefix $S[1..k-1]$, u is labeled by the suffix $S[2..k]$, and the label of (v, u) is the symbol $S[k]$. We identify a node with its label.

A *variable-order* dBG (vo-dBG) [5], $voDBG_{\mathcal{R},k}$, is formed by the union of all the graphs $DBG_{\mathcal{R},k'}$, with $1 \leq k' \leq k$. Each $DBG_{\mathcal{R},k}$ represents a *context* of $voDBG_{\mathcal{R},k}$. In addition to the (directed) edges of each $DBG_{\mathcal{R},k}$, two nodes $v \in DBG_{\mathcal{R},k'}$ and $v' \in DBG_{\mathcal{R},k''}$, with $k' > k''$, are connected by an undirected edge (v, v') if v' is a suffix of v. Following the edge (v, v') or (v', v) is called a *change of order*. We then identify node order with length.

BOSS representation for de Bruijn graphs. *BOSS* [6] is a succinct data structure, similar to the *FM-index* [13], for encoding dBGs. In *BOSS*, the nodes are represented as rows in a matrix of $k-1$ columns, and are sorted in reverse lexicographical order (i.e., reading the labels right to left). All the edge (one-symbol) labels of the graph are stored in a unique sequence E sorted by the *BOSS* order of the source nodes, so the symbols of the outgoing edges of each node fall in a contiguous range. A bitmap B of size $e = |E|$ marks the last outgoing symbol in E of every dBG node. Finally, an array $C[1..\sigma]$ stores in $C[i]$ the number of node labels that end with a symbol lexicographically smaller than i.

Prefixes in $\mathcal{R}$ of size $d < k$ are artificially represented in *BOSS* as strings of length k padded at the left with $k-d$ symbols \$. Equivalently, suffixes of size $d < k$ are represented as strings of length k padded at the right with $k-d$ symbols \$. For this work, however, suffixes of size $d < k$ are not necessary. Strings formed only by symbols \$ are also called dummy.

The complete index is thus composed of the vectors E, C, and B. It can be stored in $e(\mathcal{H}_0(E) + \mathcal{H}_0(B))(1 + o(1)) + \mathcal{O}(\sigma \log n)$ bits, where $\mathcal{H}_0$ is the zero-order empirical entropy [26, Sec 2.3]. This space is reached with a Huffman-shaped Wavelet Tree [23] for E, a compressed bitmap [32] for B (as it is usually very dense), and a plain array for C.

BOSS supports several navigational queries, most of them within $O(\log \sigma)$ time or less. The most relevant ones for us are:

- `outdegree`(v): number of outgoing edges of v.
- `forward`(v, a): node reached by following an edge from v labeled with symbol a.
- `indegree`(v): number of incoming edges of v.
- `backward`(v): list of the nodes with an outgoing edge to v.

Boucher et al. [5] noticed that by considering just the last $k' - 1$ columns in the *BOSS* matrix, with $k' \leq k$, the resulting nodes are the same as those in the dBG of order k'. To allow changing the order of the dBG in *BOSS*, they augmented the data structure with the *longest common suffix* (*LCS*) array. The *LCS* array stores, for every node of order k, the size of the longest suffix shared with its predecessor node in the *BOSS* matrix. They called this new index the *variable-order BOSS* (*VO-BOSS*), which supports the following additional operations:

- `shorter`$([i, j], k')$: range of the nodes suffixed by the last k' characters of v.
- `longer`$([i, j], k')$: list of the nodes of length k' that end with v.
- `maxlen`$([i, j], a)$: a node in the index suffixed by v, and that has an outgoing edge labeled a.

Where $[i, j]$ is the range of rows in the *BOSS* matrix suffixed by the label of a vo-dBG node v. By using a Wavelet Tree [15], the *LCS* can be stored in $n \log k + o(n \log k)$ bits, the function `shorter`$([i, j], k')$ can be answered in $\mathcal{O}(\log k)$ time and the function `longer`$([i, j], k')$ in $\mathcal{O}(|U| \log k)$ time, where U is the set of rows of the *BOSS* matrix contained within the range $[i-1..j]$, and whose *LCS* values are below k'. The function `maxlen`$([i, j], a)$ is implemented using the arrays B and E from *BOSS*, and hence it is answered in $\mathcal{O}(\log \sigma)$ time.

Succinct representation of ordinal trees. An ordinal tree T with n nodes can be stored succinctly as a sequence of *balanced parentheses* (*BP*) encoded as a bit vector $B[1..2n]$. Every node v in T is represented by a pair of parentheses `(..)` that contain the encoding of the subtree rooted at v. Every node of T can be identified by the position in B of its open parenthesis. Many navigational operations over T can be simulated over B in constant time, by using a structure that requires $o(n)$ bits on top of B [29].

3 rBOSS

Basic definitions. Let $\mathcal{R} = \{R_1...R_r\}$ be a collection of r reads (strings) of length z and let $\mathcal{R}^{rc}$ be a collection, also of r strings, every $R_i' \in \mathcal{R}^{rc}$ being the reverse complement of one $R_j \in \mathcal{R}$. Aditionally, we define the set $\mathcal{R}^* = \mathcal{R} \cup \mathcal{R}^{rc}$. Let us denote $G = DBG_{\mathcal{R}^*,k}$ and $G' = voDBG_{\mathcal{R}^*,k}$. A *traversal* P over G, or G', is a sequence $(v_0, e_0, v_1...v_{t-1}, e_t, v_t)$ where $v_0, v_1, ...v_{t-1}, v_t$ are nodes and $e_1..e_t$ are edges, e_i connecting v_{i-1} with v_i. P will be a *path* if all the nodes are different, except possibly the first and the last. In such case, P is said to be a *cycle*. P is *unary* if the nodes $(v_0..v_{t-1})$ have outdegree 1 and the nodes $(v_1..v_t)$ have indegree 1. P will be a *right* traversal over G or G' if all its edges e_i are directed from v_{i-1} to v_i, and a *left* traversal if all its edges e_i are directed from v_i to v_{i-1}. The string formed by the concatenation of the edge symbols of P is referred to as its label. P will be *safe* if it is a path or a cycle and its label appears in $\mathcal{R}^*$ as a substring of some read or if it can be generated by overlapping two or more $R_i..R_j \in \mathcal{R}^*$ in tandem and then taking the string that results from the union of those reads. The overlaps between the reads have to be of minimum size m.

Let $BOSS(G)$ and $VO\text{-}BOSS(G')$ be the *BOSS* and *VO-BOSS* indexes, respectively, for the graphs G and G'. In both cases, the matrix with the $(k-1)$-length node labels is referred to as $M_{\mathcal{R}^*,k}$, or just M when the context is clear. In $VO\text{-}BOSS(G')$, the range of rows in M suffixed by v is denoted $\overset{\bullet\bullet}{v}$.

Rows of M representing substrings of size $k-1$ in $\mathcal{R}^*$ are called *solid* nodes and rows representing artificial $(k-1)$-length strings padded with dummy symbols from the left, and that represent prefixes in $\mathcal{R}^*$, are called *linker* nodes. For a linker node v, the function `llabel(`v`)` returns the non-\$ suffix of v. A solid node that appears as a suffix in $\mathcal{R}$ is called an *s-node* and a solid node that appears as a prefix in $\mathcal{R}^*$ is called a *p-node*. A linker node v is said to be *contained* within another node v' (solid or linker) if `llabel(`v`)` is a suffix of v'.

An overlap of size o between two solid nodes v and v', denoted $v \oplus^o v'$, occurs when the o-length suffix of v is equal to the o-length prefix of v'. Relative to v, $v \oplus^o v'$ is a *forward* overlap and $v' \oplus^o v$ is a *backward* overlap. An overlap $v \oplus^o v'$ is *valid* if (i) $m \leq o < k-2$, m being some parameter, and v' is a p-node, or (ii) $o = k-2$ and v' is a solid node of any kind. Notice that case (ii) is equivalent to the definition of two dBG nodes connected by an edge. The overlap $v \oplus^o v'$ is considered *transitive* if there is another solid node v'' with valid overlaps $v \oplus^{o'} v''$ and $v' \oplus^{o''} v''$, with $o' > o''$. If there is only one v'', then $v \oplus^{o'} v'$ is transitive and *unique*. If such v'' does not exist, then $v \oplus^o v'$ is *irreductible*. The string formed by the union of the solid nodes v and v' is denoted `label(`$v \oplus^o v'$`)`.

Link between variable-order and overlaps. Overlaps between reads in $\mathcal{R}^*$ can be computed using $VO\text{-}BOSS(G')$ as follows: extend a unary path using solid nodes as much as possible, and if a solid node v without outgoing edges is reached, then decrease its order with `shorter` to retrieve the vo-dBG nodes that represent both a suffix of v and a prefix of some read in $\mathcal{R}$. From these nodes, retrieve the overlapping solid nodes of v by using `forward` and continue the graph right traversal from one of them.

In *VO-BOSS*, however, `shorter` does not ensure that the label of the output node appears as a prefix in $\mathcal{R}^*$. The next lemma precises the condition that must hold to ensure this.

▶ **Lemma 1.** *In VO-BOSS, applying the operation* `shorter` *to a node v of order $k' \leq k$ will return a node u of order $k'' < k'$ that encodes a forward overlap for v iff $\overset{\bullet\bullet}{u}[1]$ is a linker node contained by v.*

Proof. If all the left contexts in $\overleftarrow{u}$ are non-dummy strings, then u does not appear as a prefix in $\mathcal{R}^*$, and hence, following none of its edges will lead to a valid overlap of v. On the other hand, if a suffix u of v appears as a prefix in $\mathcal{R}^*$, then there is a node v' in G' at order k whose label is formed by the concatenation of a dummy string and u, and that by definition is a linker node contained by v. Since elements in $\overleftarrow{u}$ are sorted by the left contexts, v' is placed in $\overleftarrow{u}[1]$, because the dummy string is always the lexicographically smallest. ◀

Lemma 1 allows us to find overlapping nodes that are not directly linked via edges in the dBG, by looking in smaller dBG orders. We formalize this operation as follows, where we look for the longest valid suffix, that is, the one with maximum lexicographic index.

- `nextcontained`(v): returns the greatest linker node v', in lexicographical order, whose `llabel` represents both a suffix of v and a prefix of some other node in G'.

▶ **Theorem 2.** *There is an algorithm that solves* `nextcontained` *in* $\mathcal{O}(k^2 \log \sigma)$ *time.*

Proof. Incrementally decrease the order of v by one until reaching a node u with $\overleftarrow{u} \supset \overleftarrow{v}$, and that satisfies Lemma 1. If such u exists, return it. If the order of v decreases below m before finding u, then v does not contain any linker node v' with $|\texttt{llabel}(v')| \geq m$. In such case, a dummy vo-dBG node is returned. The function reduces the order up to $k-2$ times. In each iteration, the operations `shorter` and `llabel` (to check Lemma 1) are used, which take $\mathcal{O}(\log k)$ and $\mathcal{O}(k \log \sigma)$ time, respectively. Thus, the total time is $\mathcal{O}(k^2 \log \sigma)$. ◀

Notice, however, that a vo-dBG node in G' might have more than one contained linker node, and those linkers whose `llabel` is of length $\geq m$ represent edges in the overlap graph. A useful operation is to build a set L with all those relevant linkers. We can then follow the outgoing edges of every $l \in L$ to infer the solid nodes that overlap v by at least m symbols.

- `buildL`(v,m): the set of all the linker nodes contained by v that represent a suffix of v of length $\geq m$.

Function `buildL` applies `nextcontained` iteratively until reaching a node u that contains a linker node whose `llabel` has length below m. The rationale is that if v contains v', and in turn v' contains v'', then v also contains v''. Algorithm 1 (in Appendix C) shows the details.

Note that, if we chose $k = z + 1$ to build *VO-BOSS*(G'), then we are simulating the full overlap graph in compressed space. The edges are not stored explicitly, but computed on the fly by first obtaining $L = \texttt{buildL}(v, m)$, and then following the dBG outgoing edges of every $l \in L$. Still, the complexities of the involved operations `nextcontained` makes *VO-BOSS* slow for exhaustive traversals, which is our main interest. We design a faster scheme in which follows; Figure 1 exemplifies the various concepts.

A compact data structure to compute overlaps. The function `buildL` can be regarded as a bottom-up traversal of the trie T induced by the $(k-1)$-length labels of M read in reverse. Every trie node t corresponds to a vo-dBG node v whose order is the string depth of t. The traversal starts in the trie leaf t corresponding to the vo-dBG node v given to `buildL`, and continues upward until finding the last ancestor t' of t with string depth $\geq m$. The movement from t to t' can be regarded as a sequence of applications of `nextcontained`. In each such application, we move from a node t to its nearest ancestor t' that is maximal (i.e., has more than one child) and whose leftmost child edge is labeled by a \$.

Since non-maximal nodes in T are not relevant for building L, the function `nextcontained` can be reimplemented using the topology of the *compact* trie T (i.e., collapsing unary paths) represented with BP (see Section 2) instead of using `shorter`. In this way, we can get rid of the LCS structure of *VO-BOSS*.

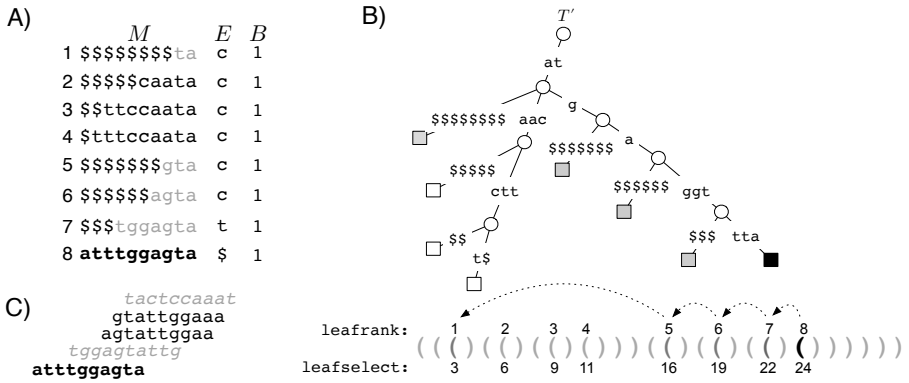


Figure 1 A segment of *rBOSS* for the set $\mathcal{R}$={atttggagta,gtattggaaa,agtattggaa,caatactcca}, with parameters $k = 11$ and $m = 2$. The computation of the forward overlaps for $R_1 = \mathcal{R}[1]$ are also shown. A) range of M that includes the solid node v that represents sequence R_1 (sequence in bold in row 8) and its contained nodes (grayed rows). B) subtree of T' that maps the range of M in A). The black leaf maps v in T' and the gray leaves map the linker nodes contained by v. The array of parentheses below represents the same subtree in BP. The rank of every leaf (`leafrank`) is shown above its relative position in the BP array (`leafselect`). Each dashed arrow represents a call of the function `nextcontained` and all the arrows together stand for the function `buildL`. C) Forward overlaps of v. Gray sequences in italic are those reads where the reverse complement matches with v.

The resulting *rBOSS* index can be built in linear time, as detailed in Appendix D.

Replacing the LCS with the topology of T in BP poses two problems, though. First, it is not possible to define a minimum dBG order m from which overlaps are not allowed, and second, Lemma 1 cannot be checked. Both problems arise because, unlike the LCS, the BP data structure does not encode the string depths of the tree nodes (and thus, the represented node lengths). Still, the topology of T can be reduced to precisely the nodes of interest for `nextcontained`, and thus avoid any check. We call this structurally-compressed version of T the *overlap tree*.

▶ **Theorem 3.** *There is a structure using* $4 + o(1)$ *bits per dBG node that implements the function* `nextcontained` *in* $\mathcal{O}(1)$ *time and the function* `buildL` *in time* $\mathcal{O}(k)$.

Proof. The structure is the BP encoding of a tree T' that is obtained by removing some nodes from the compact trie T (T has one leaf per dBG node, and less than 2 nodes per dBG node because it is compact). First, all the internal nodes of T with string depth below m are discarded, and the subtrees left are connected to the root of T'. Second, every internal node $t' \in T$ whose leftmost-child edge is not labeled by a dummy string is also discarded, and its children are recursively connected to the parent of t'. Note that all the leaves of T are in T'.

Therefore, T' has precisely the nodes of interest for operation `nextcontained`(v). We simply find the ith left-to-right leaf t in T', where i is the row of M corresponding to v (note that rows of M and leaves of T and T' are in the same order). Then, we move to the parent t' of t and return its leftmost child. An exception occurs if the leftmost child of t' is precisely t, which means that v is a linker node and thus its next contained node is the leftmost child of the parent of t'. Finally, we return the rank of the desired leftmost leaf.

Algorithm 2 (in Appendix C) shows the details. All its operations are implemented in constant time in BP, and thus `nextcontained`(v) is implemented in $\mathcal{O}(1)$ time. The function `buildL` stays the same, and its cost is dominated by the (at most) $k - 2$ calls to `nextcontained`. ◀

Once the set L with the contained nodes of v is built, we can compute the valid forward overlaps of v by following the edges of every $l \in L$ until finding a p-node. We then define:

- `foverlaps(`v`)`: the set of p-nodes whose prefixes overlap a suffix of v of length $\geq m$.

Computing the forward overlaps of v by following the edges of every $l \in L$ can be exponential. We devise a more efficient approach that uses T' and the reverse complements of the node labels. We need to define first the idea of bi-directionality in *rBOSS*.

Simulating bi-directionality. When building *rBOSS* on $\mathcal{R}^*$, the reverse complement R_i^{rc} of every read $R_i \in \mathcal{R}$ is also included, because there are several combinations in which two reads, R_i and R_j, can have a valid suffix-prefix overlap: (R_i, R_j), (R_i, R_j^{rc}), (R_i^{rc}, R_j), or (R_i^{rc}, R_j^{rc}), and all must be encoded in T'. An interesting consequence of including the reverse complements is that the topology of the dBG becomes symmetric.

▶ **Lemma 4.** *The incoming symbols of a node v are the DNA complements of the outgoing symbols of the node v^{rc} that represents the reverse complement of v. Further, the outgoing nodes of v^{rc} are the same as the DNA complements of the incoming nodes of v.*

Proof. Consider the $(k-1)$-length substring bXc of $\mathcal{R}$, and a symbol a that appears at the left of some occurrences of bXc. For building *rBOSS*, both substrings $abXc$ and its reverse complement $(abXc)^{rc} = c^c X^{rc} b^c a^c$ are considered. As a result, the dBG node v labeled bXc will have and incoming symbol a, and the dBG node v^{rc} labeled $(bXc)^{rc} = c^c X^{rc} b^c$ will have an outgoing symbol a^c. Thus, the label of node `forward(`v^{rc}`,` a^c`)` will be $X^{rc} b^c a^c$, which is the reverse complement of string abX, the label of node `backward(`v`,`a`)`. ◀

As a result of including the reverse complements of the reads, the cost of computing the incoming symbols of node v becomes proportional to the cost of computing the position of v^{rc} in the *BOSS* matrix.

▶ **Theorem 5.** *Computing the position in M of v^{rc} takes $\mathcal{O}(k \log \sigma)$ time. By augmenting rBOSS with $s \log s$ extra bits, s being the number of solid nodes, the time decreases to $\mathcal{O}(1)$.*

Proof. First, extract the label lab of v, then compute its reverse complement lab^{rc}, and finally, perform `backwardsearch(`lab^{rc}`)`. The label of v is extracted in time $\mathcal{O}(k \log \sigma)$ with the FM-index, and computing its reverse complement takes $\mathcal{O}(k)$ time. The function `backwardsearch`, also defined on the FM-index, returns the range of $(k-1)$-length strings in M suffixed by lab^{rc}, and it also takes $O(k \log \sigma)$ time. Therefore, computing the position of v^{rc} in M takes $\mathcal{O}(k \log \sigma)$ time. Alternatively, we can store an explicit permutation on the s solid nodes, so that using $s \log s$ bits we find the position of v^{rc} in M in constant time. ◀

Theorem 5 allows us to compute the forward overlaps of v in time proportional to the size of the label of v.

▶ **Theorem 6.** *The function `foverlaps` can be computed in $\mathcal{O}(k \log \sigma)$ time.*

Proof. First, create L_v = `buildL(`v`)`, and then obtain the reverse complement of the linker node $l = L_v[|L_v|]$, that is, the one representing the smallest suffix of v. Second, compute l^{rc} and search for the range $[i..j]$ in M of the $(k-1)$-length strings suffixed by l^{rc}. From the edge symbols in $[i..j]$ follow the dBG path $p_{v^{rc}}$ that spells the label of v^{rc}. Finally, every time a solid node v' is reached during the traversal of $p_{v^{rc}}$, report its reverse complement as a forward overlap for v. Computing L takes $\mathcal{O}(k)$ time. Both searching for $[i..i]$ and traversing $p_{v^{rc}}$ take $\mathcal{O}(k \log \sigma)$ time. All the shifts between reverse complements take $\mathcal{O}(1)$ time if we use permutations. ◀

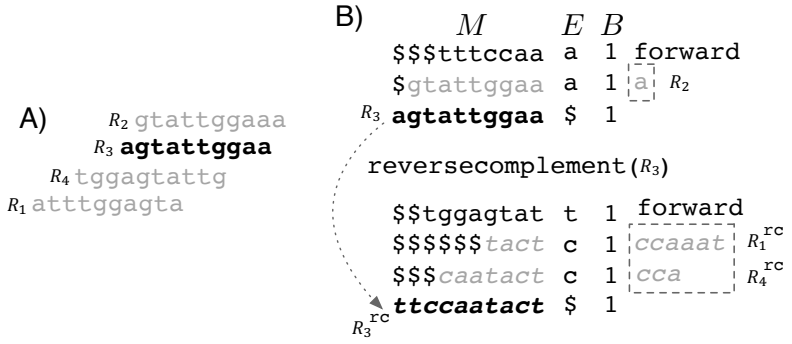


Figure 2 Example of the computation of the forward and backward overlaps for sequence $R_3 = \mathcal{R}[3]$ of the example of Figure 1. A) Sequence R_2 is a forward overlap for R_3 and sequences R_4 and R_1 are backward overlaps. B) The upper matrix represents the range in M that includes the solid node that represents R_3 (row in bold), and its contained nodes (grayed row) and the lower matrix is the range of M that includes the solid node of the reverse complement of R_3, R_3^{rc} (row in bold and italic), and its contained nodes (gray rows in italic). Gray symbols to the right of every matrix are the outgoing symbols retrieved from applying `foward` from every contained node until reaching the next solid node. For the case of R_3^{rc}, these solid nodes are R_4^{rc} and R_1^{rc}, the reverse complements of R_4 and R_1, respectively.

Figure 2 exemplifies the overlap function. Note that backward overlaps can be obtained by computing `foverlaps` for the reverse complement of v. The complexity of `foverlaps` is the same obtained by Simpson and Durbin [34].

By using T' and Lemma 4 we can access the topology of the overlaps, and to retrieve extra information from the data that irreductible overlap graphs or dBGs do not have. We formalize this idea as *weighted irreductible overlaps*.

Weighting irreductible overlaps. Given an irreductible overlap $v \oplus^o v$ between solid nodes v and v', we can use the number of unique transitive overlaps between them as a measure of confidence, $\texttt{weight}(v \oplus^o v') < o - m$, for $\texttt{label}(v \oplus^o v')$. In Figure A.1 we show different examples in which $\texttt{weight}(v \oplus^o v')$ can be helpful to detect patterns in the data. The function $\texttt{foverlaps}(v)$ in our scheme can be modified to return the list of irreductible overlaps for v, with their weights included. The idea is as follows: once the range $[i..j]$ is obtained, we form an array Y with the dBG nodes in $[i..j]$ that are not contained by any other node within the same range. The set Y will represent the possible irreductible overlaps of v. Y is built in one scan over $[i..j]$ by checking which T' leaves are not the leftmost children of their parent. The weight of every $y \in Y$ is computed as its depth minus the depth of its closest ancestor in T' with more than two children. We do the subtraction because only unique transitive connections count as weights. Every $y \in Y$ and its weighting nodes form a subrange $\overset{\bullet\bullet}{y}$ in $[i, j]$. We perform a right traversal starting from the outgoing edges of $\overset{\bullet\bullet}{y}$ to retrieve $p_{v^{rc}}$ as before. In the process, however, one or more elements of Y can be discarded or their weights decreased if they do have a branch spelling the reverse complement of some $l \in L_v$. Figure 3 exemplifies the process.

4 Experiments

We implemented *rBOSS* as a `C++` library, using the `SDSL` library [14] as a base. In Section 2 we stated that vector E can be represented using a Huffman-shaped Wavelet Tree, but our implementation uses run-length encoding [23] to exploit repetitions in the

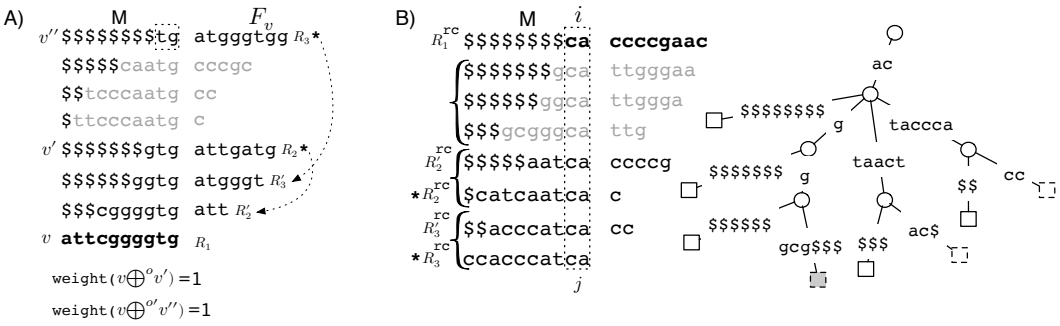


Figure 3 Computation of weighted forward overlaps as described in Section 3. A) Sequence R_1, represented by dBG node v (bold row in M), and its irreductible overlaps, R_2 and R_3 (asterisks), represented by dBG nodes v' and v'' respectively. The transitive overlaps that weight every irreductible overlap are shown with arrows to the right of M. The dashed box in the upper left corner of M is lab = `llabel`$(L_v[|L_v|])$. The left side of B) is the range $[i..j]$ in M resulting from searching the range of $(k-1)$-length strings suffixed by lab^{rc}. The right side of B) is the subtree in T' induced by the dBG nodes in $[i..j]$. Dashed leaves in the subtree are those that are (probably) irreductible overlaps of v (elements in Y). The gray dashed leaf of the subtree corresponds to an element y' that was originally added to Y but then discarded because none of the branches starting in its outgoing edges spell the reverse complement of some $l \in L_v$. Gray leaves are the ones that contribute to the weight of y'. The element y' and its weighting nodes are also represented in the range $[i..j]$ as gray rows. Subranges of $[i..j]$ whose outgoing branches lead to weighted irreductible overlaps of v are shown in curly brackets.

reads. We also include an extra bitmap $S[1..n]$ that marks the position of every solid node in M, which speeds up iterating over the solid nodes. We did not include the permutation to compute the reverse complements of the dBG nodes in constant time. Instead, we use `backwardsearch` as stated in Theorem 5. Additionally, we implemented the *VO-BOSS* data structure by modifying our *rBOSS* implementation and merging it with segments of the code[3] from Boucher et al. [5]. Our complete code is available at `https://bitbucket.org/DiegoDiazDominguez/eboss-dt/src/master/`. The compilation flags we used were `-msse4.2 -O3 -funroll-loops -fomit-frame-pointer -ffast-math`.

We used `wgsim` [22] to simulate a sequencing dataset (in `FASTQ` format) from the E.coli genome with 15x coverage. A total of 549,845 reads were generated, each 150 bases long, yielding a dataset of 185 MB. The input parameters for building *rBOSS* are k and m. We used a minimum value of 50 for k, and increased it up to 110 in intervals of 5. For every k, we used 6 values of m, from 15 to 40, also in intervals of 5. This makes up 72 indexes. We also built equivalent *VO-BOSS* instances using the same values for k.

Space and construction time. The sizes of the resulting *rBOSS* indexes are shown in Figure 4.A, which grow fairly linearly with k, at $0.29 + 0.036k$ bits per input symbol (i.e., 50–100 MB for our dataset), and do not depend much on m. Figure A.2 shows elapsed times and memory peaks during construction. These are also linear in k; for example with $m = 20$ (the most demanding value) the *rBOSS* index for our dataset is built in 4–6 minutes with a memory peak around 2.5 GB. Figure 4.B compares the sizes of *VO-BOSS* and *rBOSS*, showing that *rBOSS* is more than 20% smaller on average.

[3] `https://github.com/cosmo-team/cosmo`

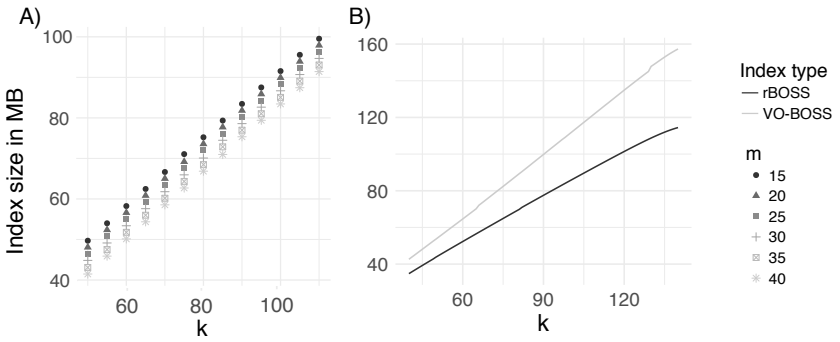


Figure 4 Index size statistics. The x-axis gives the values of k and the y-axis is the size of the index in MB. A) Sizes of the *rBOSS* indexes. Shapes denote the different values of m. B) Index size comparison between *rBOSS* and *VO-BOSS*, building the *rBOSS* indexes with $m = 30$.

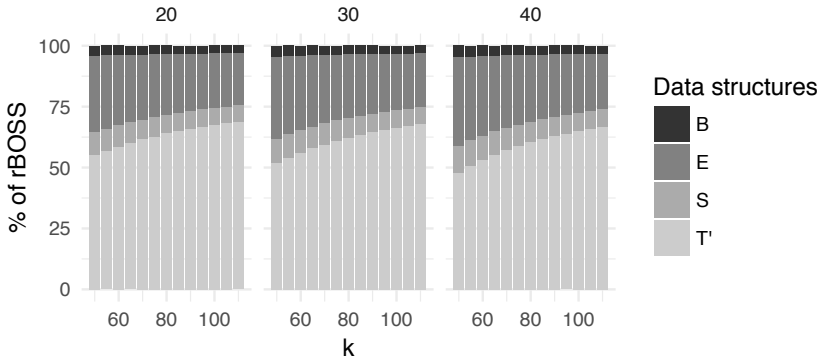


Figure 5 Stacked barplot with the percentage that each substructure uses in *rBOSS*. The x-axis shows the value of k and the numbers on top of the plot are the m values.

The space breakdown of our index is given in Figure 5, and further statistics in Table A1. The most expensive data structure in terms of space (50%–65%) is the BP representation of T'. The sequence E uses 20%–35%, and the rest are the bitmaps B and S.

Time for the primitives. For every index, we took 1000 solid nodes at random and computed the mean elapsed time for functions `nextcontained`, `buildL`, `foverlaps`. For the *rBOSS* indexes, we also measured the mean elapsed time for `reversecomplement`. Table 1 shows the results. Within the *rBOSS* implementation, `nextcontained` is the fastest operation, with a stable time around 1.5 μsec across different values of m and k. Operation `buildL` becomes slower as we increase k, but faster as we increase m. This is expected because the larger k, the longer the traversal through T', but if m grows the traversal shortens as well. In all cases, `buildL` takes under 10 μsec. The cost of operation `foverlaps` grows linearly with k, but also decreases as we increase m, reaching the millisecond. This is much slower than previous operations, dominated by the time to find the reverse complement of the shortest linker node with backward search. Finally, the time of `reversecomplement` is also a few milliseconds, growing steadily with k regardless of m.

Table 1 also compares *rBOSS* with the *VO-BOSS* implementation. All the functions are clearly slower in *VO-BOSS*, by two orders of magnitude for `next-contained` and `buildL`, and by a factor around 2 for `foverlaps`.

Table 1 Mean elapsed time, in μseconds, for the functions proposed in this article for both *rBOSS* and *VO-BOSS*.

k	m	*rBOSS*				*VO-BOSS*		
		`next-contained`	`buildL`	`foverlaps`	`reverse-complement`	`next-contained`	`buildL`	`foverlaps`
50	20	1.49	5.09	389.42	1226.53	225.93	804.81	825.11
50	30	1.53	4.22	352.41	1209.02	216.47	581.31	802.23
50	40	2.00	3.38	255.02	1226.56	191.62	337.95	770.70
70	20	1.55	6.46	601.94	1620.22	311.46	1614.49	1155.22
70	30	1.57	5.82	546.53	1620.78	310.74	1382.25	1115.33
70	40	1.54	5.26	517.43	1621.98	297.23	1083.36	1080.17
90	20	1.73	8.11	828.12	2013.00	374.09	2441.96	1495.37
90	30	1.58	7.35	768.83	2012.36	368.71	2211.05	1444.93
90	40	1.56	6.67	714.42	2016.41	372.76	1871.19	1398.07
110	20	1.67	9.25	1088.41	2411.10	429.86	3491.07	1865.60
110	30	1.77	8.64	1014.32	2410.03	428.17	3226.45	1801.85
110	40	1.64	8.10	942.17	2414.11	436.15	2965.48	1745.31

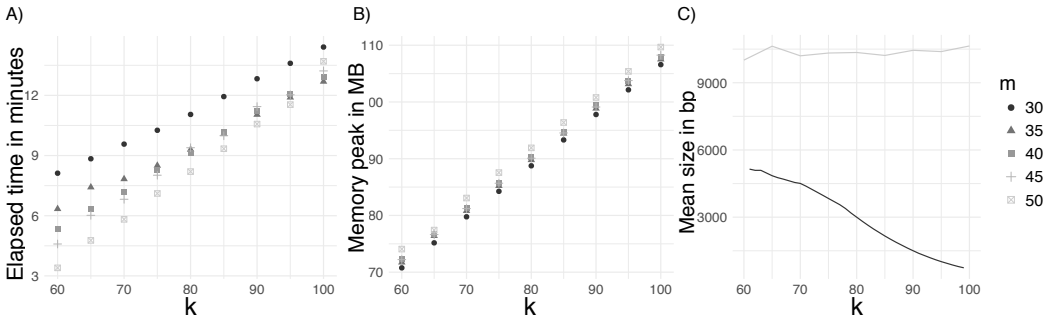


Figure 6 Results of genome assembly experiments. A) The elapsed time for the assembly. B) The memory peak achieved during the assembly. C) Comparison of the mean contig size in *rBOSS* (gray line) versus the mean unitig size of a fixed dBG (black line) using the same k, both with $m = 50$.

Genome assembly. We implemented a genome assembler on top of *rBOSS* to test the usefulness of the data structure. The algorithm is described in Appendix E. We used the same E. coli dataset as before, with a minimum value for k of 60, increasing it up to 100 in intervals of 5 for building the indexes. For each k, we selected 5 values for m, from 30 to 50, also in intervals of 5. The results are shown in Figure 6; time and space are again linear in k. Using $m = 30$, our assembler generates contigs in 7–14 minutes and has a memory peak of 70–105 MB, just 18–21 MB on top of the index itself.

Figure 6.C compares the quality of the assembly using variable k and *rBOSS*, with $m = 30$, versus the corresponding assembly generated with a fixed dBG that uses the same k. The dBG indexes were built using the `bcalm` tool [9]. It is clear that the ability to vary the value of k to compute overlapping sequences as we spell the contigs, also called *maximal paths* (MP) in our algorithm, yields an assembly of much higher quality. Figure A.3 gives further data on the assembly.

5 Conclusions and Further Work

We have introduced *rBOSS*, a succinct representation for vo-dBGs (of degree up to k) that avoids the $O(\log k)$-bit penalty factor of previous representations thanks to the use of a new structure we call the *overlap tree*. This enables the use of k values sufficiently large so as to simulate the full overlap graph, which is an essential tool for genome assembly and other bioinformatic analyses. Our index, for example, can assemble the contigs of 185 MB of 150-base reads, with $k = 100$, in less than 15 minutes and within 105 MB.

Our index builds fast, yet using significant space (in our experiment, 6 minutes and 2.5 GB). Future work includes reducing the construction space, even at some increase in construction time. We also aim to reduce the space of T', the most space-demanding component of our index. Preliminary experiments show that the topology of T' is highly repetitive, and that it can be about halved with a grammar-compressed representation [28].

The *rBOSS* index can be used for different bioinformatic analyses, not just genome assembly. An example is the detection of single nucleotide polymorphisms. Polymorphisms are usually inferred by first aligning a multiset[4] of reads to a reference genome and then looking for mismatches between the aligned reads and the genome. This approach is often expensive as it requires much preprocessing. As an alternative, we can build a *colored* version of the *rBOSS* index, that is, we color the reads according to the individual they were generated from, and then search for every read x that meets the following criteria: i) two or more overlaps with heavy weights, ii) two or more colors, and iii) the overlapping reads share one or more colors with x, but not among them. Reads meeting these criteria (and their overlapping sequences) are candidates to map polymorphic sites in the genome. We can then align them to the reference genome and check the sequencing quality of their characters to be sure. This idea for inferring SNPs is similar to the one described in [16].

Another possible application is sequencing error correction. In this case, we search for reads whose overlaps have small weights. If for a particular read x, all its forward and backward overlaps have very small weights, say < 2, then it is reasonable to assume that x contains errors, especially if the sequencing qualities of its characters are low.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

2 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

3 Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proc. 46th Annual Symposium on the Theory of Computing (STOC)*, pages 148–193, 2014.

4 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Constructing String Graphs In External Memory. In *Proc. 14th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 311–325, 2014.

5 Christina Boucher, Alexander Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-Order de Bruijn Graphs. In *Proc. 25th Data Compression Conference (DCC)*, pages 383–392, 2015.

[4] A sequencing data set generated from the DNA of several individuals from the same specie.

6 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn Graphs. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012.

7 Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*, 34(5):525, 2016.

8 M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

9 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):201–208, 2016.

10 David Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

11 Nicolaas Govert De Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

12 Gennady Denisov, Brian Walenz, Aaron L Halpern, Jason Miller, Nelson Axelrod, Samuel Levy, and Granger Sutton. Consensus generation and variant detection by Celera Assembler. *Bioinformatics*, 24(8):1035–1040, 2008.

13 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

14 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

15 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

16 Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226, 2012.

17 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear Work Suffix Array Construction. *Journal of the ACM*, 53(6):918–936, 2006.

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 *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

19 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

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

21 Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015.

22 Heng Li. wgsim — Read simulator for next generation sequencing. *Bioinformatics*, 28:593–594, 2012.

23 V. Mäkinen and G. Navarro. Succinct Suffix Arrays based on Run-Length Encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

24 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.

25 Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, 2005.

26 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

27 Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

28 Gonzalo Navarro and Alberto Ordóñez. Faster Compressed Suffix Trees for Repetitive Collections. *ACM Journal of Experimental Algorithmics*, 21(1):article 1.8, 2016.

29 Gonzalo Navarro and Kunihiko Sadakane. Fully-Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

30 Daisuke Okanohara and Kunihiko Sadakane. A Linear-Time Burrows-Wheeler Transform using Induced Sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 90–101, 2009.

31 Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA-a practical iterative de Bruijn graph de novo assembler. In *Proc. 14th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 426–440, 2010.

32 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):article 43, 2007.

33 Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):367–373, 2010.

34 Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

35 Daniel Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(3):821–829, 2008.

36 Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

A Figures

Figure A.1 Different cases in which the weights of the irreductible overlaps can detect patterns in the data. A) sequence R_1 (dBG node v) has only one irreductible overlap, with R_5 (dBG node v'), and there are 3 transitive overlaps between them (R_2,R_3 and R_4 in gray). In this case, there is enough evidence (transitive overlaps) to infer that the string formed by the union of R_1 and R_5 exists in the input DNA. B) sequence R_1 has 2 irreductible overlaps, with R_4 and R_6 (dBG nodes v' and v'' respectively), but the number of unique transitive overlaps between R_1 and R_4 is zero (weight($v \oplus^{o} v'$) = 0), so the most probable option is that R_4 contains a sequencing error (italic underlined symbol). C) R_1 has two irreductible overlaps, with R_5 and R_7, and both overlaps have a weight of 2. In this circumstance, it is more probable that R_1 belongs to a repeated region or it is next to a genetic variation, if the reads R_5 and R_7 come from different individuals.

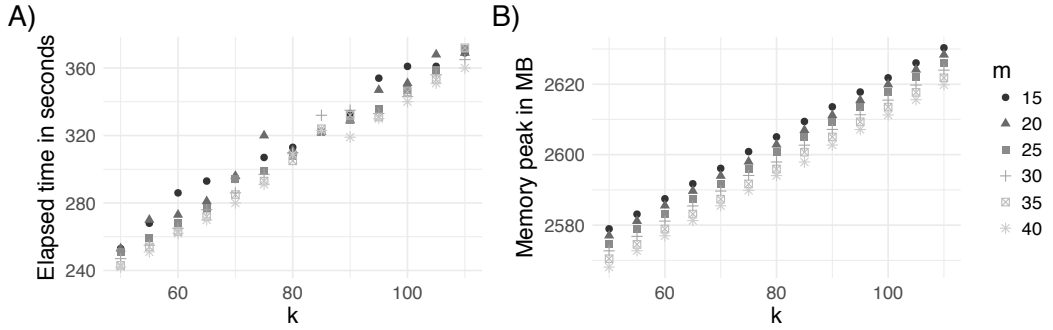


Figure A.2 Statistics about the construction of *rBOSS*. In all the plots, the x-axis represents the different values used for k, and every shape represents a particular value of m. The y-axis in A) is the mean elapsed time; in B) it is the memory peak achieved during the construction.

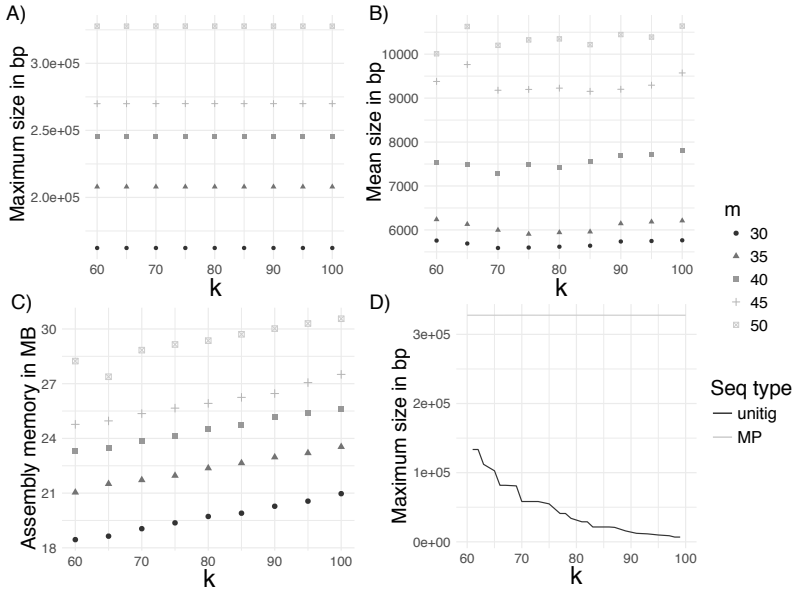


Figure A.3 Other results of the assembly experiments. In all the figures, the x-axis are the values for k. In figures A), B), and C), the shapes are the values of m. A) Maximum length achieved for a contig in each *rBOSS* index. B) Mean size for contigs in every *rBOSS* index. C) Difference between the memory peak and the size of the index, that is, the memory used exclusively for assembly. D) Comparison of the longest contig in *rBOSS* (gray line) versus the longest unitig of a fixed dBG (black line) using the same k, both with m = 50.

B Tables

Table A1 Statistics about the different instances of the dBG graphs generated in the experiments. Except for the first and second column, all the values are expressed in millions. Columns one and two are the values used for k and m, respectively, in the *rBOSS* index. Column three contains the total number of dBG nodes at order k. Column four show the number of solid nodes and column five the number of linker nodes. Column six is the total number of edges in the dBG (number of symbols in E). Column seven is the total number of nodes in the overlap tree and column eight is the number of internal nodes in the overlap tree.

k	m	dBG nodes	solid nodes	linker nodes	edges	tree nodes	tree int nodes
50	20	48.86	9.13	39.73	50.92	78.78	29.92
50	30	48.86	9.13	39.73	50.92	68.47	19.61
50	40	48.86	9.13	39.73	50.92	58.15	9.29
70	20	69.52	9.14	60.38	71.58	120.09	50.57
70	30	69.52	9.14	60.38	71.58	109.78	40.26
70	40	69.52	9.14	60.38	71.58	99.46	29.94
90	20	90.18	9.14	81.04	92.24	161.40	71.22
90	30	90.18	9.14	81.04	92.24	151.08	60.91
90	40	90.18	9.14	81.04	92.24	140.76	50.59
110	20	110.79	9.09	101.69	112.84	202.61	91.82
110	30	110.79	9.09	101.69	112.84	192.30	81.51
110	40	110.79	9.09	101.69	112.84	181.98	71.19

C Pseudocodes

Algorithm 1 Build set L with the linker nodes contained by v.

```
1:  procedure nextcontained(v, m)
2:      d ← v.order − 1
3:      while d ≥ m do
4:          u ← shorter(v, d)
5:          if u ⊃ v then
6:              if isLinker(v') and |llabel(v')| = d then
7:                  return u
8:              v ← u
9:          d ← d − 1
10:     return 0                                          ▷ dummy node
11: procedure buildL(v, m)      ▷ v is a vo-dBG node and m is the minimum suffix size
12:     L ← ∅
13:     c ← nextcontained(v, m)
14:     while c>0 do
15:         L ← L ∪ {c[1]}
16:         c ← nextcontained(c, m)
17:     return L
```

Algorithm 2 Function `nextcontained` implemented with the topology of T'.

```
1: procedure nextcontained(v)                    ▷ v is a vo-dBG node at order K − 1
2:     t ← leafselect(T', v)                     ▷ node in T' mapping v
3:     t' ← parent(T', t)
4:     if firstchild(T', t') = t then            ▷ t is already the leftmost sibling
5:         t' ← parent(T', t')
6:     l ← lchild(T', t')
7:     return leafrank(T', l)                    ▷ vo-dBG node mapping l
```

D Building *rBOSS*

To build our data structure, we first form the string $R = R_1\$R_1^{rc}..R_r\$R_r^{rc}\#$ over the alphabet $\Sigma \cup \{\$, \#\}$, with size $n' = |R|$, and that represents the concatenation of the reads in $\mathcal{R}^*$. In R, symbol $\#$ is the least in lexicographical order. Next, we build the SA, BWT and LCP arrays for $\overline{R}$, the reversal of R. We use $\overline{R}$ instead of R because the BWT (E in $BOSS$) contains the symbols to the left of every suffix (node labels in $BOSS$), but we actually need the symbol to the right when we call `forward`. After building these arrays, we modify LCP to simulate the padding of the dummy symbols: for every $LCP[i]$, we compute the distance d between $SA[i]$ and the position in $\overline{R}$ of the next occurrence of symbol \$ after $SA[i]$. If $d < k-1$ and $d < LCP[i]$, then we set $LCP[i] = k-1$. To compute d in constant time, we can generate a bitmap D, with *rank* and *select* support (see Section D.1), that marks in $\overline{R}$ the position of every \$. Thus, d is computed as $select_1(D, rank_1(D, SA[i]) + 1) - SA[i]$.

The next step is to build E. To this end, we traverse $BWT[i]$ for increasing i as long as $LCP[i] \geq k-1$, and in the process we mark in a bitmap $S[1..\sigma]$ the symbols seen so far. When $LCP[i] < k-1$, we append the marked symbols of S to E, and also append the same number of bits to B, all zeros except the last in each step. We then reset S and restart the traversal of BWT.

The final step is to build T'. We use an algorithm [3] to build the topology in BP of a tree, modified to discard on the fly the unnecessary nodes. We first compute the virtual suffix tree ST from LCP [1], and then traverse it in preorder. For each node v, we write an opening parenthesis if it satisfies the restrictions, then we recursively traverse its children, and finally write a closing parenthesis if v satisfied the conditions.

The SA for $\overline{R}$ can be built in linear time [17, 20, 19], and so can BWT, [30] LCP [18], and the virtual ST [1]. Our modifications are obviously linear-time, and therefore the *rBOSS* structure can be built in linear time as well.

D.1 Rank and select data structures

Rank and *select* dictionaries are fundamental in most succinct data structures. Given a sequence $B[1..n]$ of elements over the alphabet $\Sigma = [1..\sigma]$, $B.\texttt{rank}_b(i)$ with $i \in [1..n]$ and $b \in \Sigma$, returns the number of times the element b occurs in $B[1..i]$, while $B.\texttt{select}_b(i)$ returns the position of the ith occurrence of b in B. For binary alphabets, B can be represented in $n + o(n)$ bits so that `rank` and `select` are solved in constant time [10]. When B has $m \ll n$ 1s, a compressed representation using $m \lg \frac{n}{m} + O(m) + o(n)$ bits, still solving the operations in constant time, is of interest [32]. This space is $o(n)$ if $m = o(n)$.

E Genome assembly

In this section we briefly describe how to use *rBOSS* to assemble a genome. We define some concepts first.

▶ **Lemma 7.** *A solid node v is right-extensible (RE) (respectively left-extensible (LE)) if (i) it is a non-s-node with outdegree 1 (respectively, a non-p-node with indegree 1) or (ii) it is an s-node with outdegree ≤ 1 (respectively p-node with indegree ≤ 1) and following its outgoing edge (if outdegree is 1) and the outgoing edges of every $l \in L_v$ (respectively its incoming edge, if indegree is 1, and the incoming edges of its backward overlaps) leads to a unique solid node v' in at most $(k-1)-m$ forward operations.*

▶ **Theorem 8.** *Computing if a solid node v is RE has $\mathcal{O}(k + |L_v|(k-m)\log\sigma)$ worst case time complexity. Computing if v is LE also has $\mathcal{O}(k + |L_v|(k-m)\log\sigma)$ time complexity if we augment rBOSS with $s \log s$ bits, where s is the number of solid nodes.*

Proof. When v is not an s-node, testing if it is RE reduces to checking its outdegree. The other case, when v is a s-node, is harder. First compute L_v = `buildL`(v), and then perform a set of operations in batches over the elements of L_v, as follows. Regard L_v as a queue. If $L_v[1]$, the linker node that represents the greatest suffix of v, has outdegree 0, then remove it from L_v. After that, check that each $L_v[i]$, with $i \in [1..|L_v|]$, has outdegree 1 and that all the outgoing edges are labeled with the same symbol. If some $L_v[i]$ has outdegree > 1 or two or more different symbols are seen in the outgoing edges of L_v, then return `false`. If all outgoing edges in L_v have the same symbol a, then perform $L_v[i]$ = `forward`$(L_v[i], a)$ for every i. Repeat the process until L_v becomes empty, if that happens, then return `true`. Computing LE is exactly the same process, but first we have to compute v^{rc}, the reverse complement of v. If we use $s \log s$ extra bits to store the permutation with the reverse complements of the solid nodes, then we can compute v^{rc} in $\mathcal{O}(1)$. ◀

We also define the concept of right and left maximal paths.

▶ **Lemma 9.** *A right-maximal path (respectively, left-maximal) over rBOSS is a path where all the nodes are RE (respectively, LE) except the rightmost (respectively, leftmost) node. A maximal-path is the concatenation of a left-maximal and a right-maximal path.*

E.1 Marking non-extensible nodes

Computing whether a solid node v is extensible during a graph traversal can be expensive (Lemma 8), especially if the traversal is exhaustive. The amount of computation can be reduced, however, by computing beforehand which nodes are non-extensible and marking them in a bitmap N of size s. Notice that only a small fraction of the nodes will be non-extensible, so N is highly compressible.

There are four cases in which v is non-extensible; (i) it has outdegree > 1, (ii) there are two or more different outgoing symbols in L_v, (iii) the outgoing symbol in v differs from the symbol in L_v, or (iv) the computation of the forward overlaps of v yields two or more different irreductible overlaps. To detect non-extensible nodes, we use the topology of T' instead of directly calling the function `foverlaps`.

We descend on T' in DFS, and every time we reach a leaf t that is the leftmost child of its parent, we append its edge symbols into a sequence U and the leaf rank of t to an array I, one append per edge. We also keep track of the different symbols appended into U so far. If after consuming t there are two or more different symbols in U, we scan U from

right to left until finding the first position i such that $U[i] \neq U[i+1]$. Then, we mark as non-extensible all the solid nodes that contain any prefix of $\texttt{llabel}(I[i+1])$ that in turns contains any prefix of $\texttt{llabel}(I[i])$ of length $\geq m$. The rationale is that any solid node v containing $I[i+1]$ will also contain $I[i]$ (we now this fact because the DFS order). The problem, however, is that the outgoing symbols of $I[i]$ and $I[i+1]$ differ. Thus, v is a case (ii) non-extensible node. It can still happen that one of the prefixes of $I[i+1]$ is contained by a solid node v', and if it does, then it might happen that v' also contains a prefix of $I[i]$. In such case, v' is a case (iv) non-extensible node, because the elements of $L_{v'}$ will lead to $I[i]$ and $I[i+1]$, which are known to differ in their outgoing edges. To be sure, we must go backwards in $I[i+1]$ marking the solid nodes that contain prefixes of $I[i+1]$, and we stop when $I[i+1]$ does not contain any prefix of $I[i]$ of size $\geq m$.

When a solid node v is reached during the DFS traversal, we first have to check if it was already marked. If it is still unmarked, then we check if it has outdegree more than two (v is a case i), or if it has outdegree 1, but its outgoing symbol differ from the outgoing symbols in L_v (v is case iii).

E.2 Spelling maximal paths

Once *rBOSS* and the bitmap N are built, the process of spelling maximal paths can be implemented as an stream algorithm, which is very space-efficient. For every non-extensible node v compute the set $O_v = \texttt{overlaps}(v)$. We start a forward traversal from each $o_i \in O_v$ and continue until reaching a non-extensible node. During the process, append the edge symbols to a vector F. After finishing, compute o_i^{rc}, start a forward traversal from it and continue until reaching the next non-extensible node. As with o_i, also append the outgoing edges to a vector R. The final string spelled by the maximal path will be $R^{rc} \cdot \texttt{label}(o_i) \cdot F$. If in either of both traversals, forward or backward, an extensible solid node with outdegree 0 is reached, then call `nextcontained` and continue through its edges.

Faster Queries for Longest Substring Palindrome After Block Edit

Mitsuru Funakoshi
Department of Informatics, 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

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

Palindromes are important objects in strings which have been extensively studied from combinatorial, algorithmic, and bioinformatics points of views. Manacher [J. ACM 1975] proposed a seminal algorithm that computes the longest substring palindromes (LSPals) of a given string in $O(n)$ time, where n is the length of the string. 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 $O(\ell + \log\log n)$ time, after a substring in T is replaced by a string of arbitrary length ℓ. This outperforms the query algorithm proposed in our previous work [CPM 2018] that uses $O(\ell + \log n)$ time for each query.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases palindromes, string algorithm, periodicity

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.27

Related Version https://arxiv.org/abs/1901.10722

Funding *Yuto Nakashima*: Supported by JSPS KAKENHI Grant Number JP18K18002.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.
Hideo Bannai: Supported by JSPS KAKENHI Grant Number JP16H02783.
Masayuki Takeda: Supported by JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

Palindromes are strings that read the same forward and backward. Finding palindromic structures in strings has important applications in analysis of DNA, RNA, and protein sequences, and thus a variety of efficient algorithms for finding palindromic structures occurring in a given string have been proposed (e.g., see [3, 18, 12, 15, 19, 14, 10] and references therein).

In this paper, we consider the fundamental problem of finding the *longest substring palindrome* (*LSPal*) in a given string T. 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. Hence, in order to compute the LSPal of a given string T, it suffices to compute all maximal palindromes in T. Manacher [16] gave an elegant $O(n)$-time algorithm to find all maximal palindromes in a given string of length n. Manacher's algorithm utilizes symmetry of palindromes and character equality comparisons only, and therefore works in $O(n)$ time for any alphabet. There is an alternative suffix tree [21] based algorithm which works in $O(n)$ time in the case of an integer alphabet of polynomial size in n [13]. Finding the longest substring palindrome in the streaming model has also been considered [6, 11].

Now we consider the following question: what happens to those palindromes if the string T is edited? It seems natural to ask this kind of question since a typical biological sequence can contain some uncertainties such that there are multiple character possibilities at some positions in the sequence. In our recent work [9], we initiated this line of research and showed the following results. Let n be the length of the input string T and σ the alphabet size.

1-ELSPal. We can preprocess T in $O(n)$ time and space such that later, we can answer in $O(\log \min\{\sigma, \log n\})$ time the longest substring palindrome after a single character edit operation (insertion, deletion, or substitution).

ℓ-ELSPal. We can preprocess T in $O(n)$ time and space such that later, we can answer in $O(\ell + \log n)$ time the longest substring palindrome after a block-wise edit operation, where ℓ is the length of the new block that substitutes the substring in T.

In this paper, we further pursue the second variant of the problem (ℓ-ELSPal) where an existing substring is replaced with a new string (block) of length ℓ. We remark that the length ℓ of a new block is arbitrary. The main result of this paper is an $O(\ell + \log \log n)$-time query algorithm that answers the longest substring palindrome after a block-wise edit operation, with $O(n)$-time and space preprocessing.

Note that ℓ-ELSPal is a generalization of 1-ELSPal, where $\ell = 1$ for insertion and substitution and $\ell = 0$ for deletion. Therefore, the result of this paper achieves $O(\log \log n)$-time query algorithm for 1-ELSPal. This is as efficient as the $O(\log \min\{\sigma, \log n\})$-time query of [9] when the alphabet size σ is at least $O(\log n)$ (e.g., in the case of an integer alphabet).

Related work

Amir et al. [1] proposed an algorithm to find the *longest common factor* (*LCF*) of two strings, after a single character edit operation is performed in one of the strings. Their data structure occupies $O(n \log^3 n)$ space and uses $O(\log^3 n)$ query time, where n is the length of the input strings. Their data structure can be constructed in $O(n \log^4 n)$ expected time. Urabe et al. [20] considered the problem of computing the longest *Lyndon word* after an edit operation. They showed $O(\log n)$-time queries for a single character edit operation and $O(\ell \log \sigma + \log n)$-time queries for a block-wise edit operation, both using $O(n)$ time and space for preprocessing. We note that in these results including ours in this current paper, edit operations are given as *queries* and thus the input string(s) remain static even after each query. This is due to the fact that changing the data structure dynamically can be too costly in many cases. It is noteworthy, however, that very recently Amir et al. [2] solved dynamic versions for the LCF problem and some of its variants. In particular, when n is the maximum length of the string that can be edited, they showed a data structure of $O(n \log n)$ space that can be dynamically maintained and can answer 1-ELSPal queries in $O(\sqrt{n} \log^{2.5} n)$ time, after $O(n \log^2 n)$ time preprocessing.

2 Preliminaries

Let Σ be the *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε 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 $1 \leq i \leq |T| - p$. If a string B is both a proper prefix and a proper suffix of another string T, then B is called a *border* of T.

For any string P, let P^R denote the reversed string of P. A string P is called a *palindrome* if $P = P^R$. 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 $j = |T|$. 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 $q = 1, 1.5, \ldots, n - 0.5, n$, 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 (including empty ones which occur at center $\frac{i+j}{2}$ when $T[i] \neq T[j]$).

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 $\mathsf{RightLCE}_T(i, j)$ and $\mathsf{LeftLCE}_T(i, j)$ 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 $\mathsf{OutLCE}_T(i, j)$ an outward LCE query for positions $i < j$ in the string T.

Manacher [16] 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 [21] for string $T\$T^R\#$ enhanced with a lowest common ancestor data structure [4], 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 [8, 13].

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 problem of finding an LSPal after a substring of T is replaced with another string. The problem is formally defined as follows:

▶ **Definition 1** (Longest substring palindrome query after block edit).
Preprocess: *A string T of length n.*
Query input: *An interval $[i, j] \subseteq [1, n]$ and a string X of length ℓ.*
Query output: *(The length of) a longest substring palindrome in the edited string $T' = T[1..i-1]XT[j+1..n]$.*

The query in the above problem is called an *ℓ-block edit longest substring palindrome* query (*ℓ-ELSPal* query in short). In the following section, we will propose an $O(n)$-time and space preprocessing scheme such that subsequent ℓ-ELSPal queries can be answered in $O(\ell + \log \log n)$ time. We remark that in this problem string edits are only given as *queries*, i.e., we do not explicitly rewrite the original string T into T' and T remains unchanged for further queries. We also remark that in our problem the length ℓ of a substring X that substitutes a given interval (substring) can be arbitrary.

Let ℓ' be the length of the substring to be replaced, i.e., $\ell' = j - i + 1$. Our block-wise edit operation generalizes character-wise substitution when $\ell' > 0$ and $\ell > 0$, character-wise insertion when $\ell' = 0$ and $\ell > 0$, and character-wise deletion when $\ell' > 0$ and $\ell = 0$.

The following properties of palindromes are useful in our algorithms.

▶ **Lemma 2.** *Any border B of a palindrome P is also a palindrome.*

Proof. Since P is a palindrome, for any $1 \leq m \leq |P|$, clearly $P[1..m] = (P[|P| - m + 1..|P|])^R$. Since B is a border of P, we have that $B = P[1..|B|] = (P[|P| - |B| + 1..|P|])^R = B^R$. ◀

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 $\mathbf{S}_i = s_1, \ldots, s_g$ be the sequence of lengths of maximal palindromes in $MaxPalEnd_T(i)$ sorted in increasing order, where $g = |MaxPalEnd_T(i)|$. Let d_j be the progression difference for s_j, i.e., $d_j = s_j - s_{j-1}$ for $2 \leq j \leq g$. For convenience, let $d_1 = 0$. We use the following lemma which is based on periodic properties of maximal palindromes ending at the same position.

▶ **Lemma 3** ([9])**.**

(i) *For any $1 \leq j < g$, $d_{j+1} \geq d_j$.*

(ii) *For any $1 < j < g$, if $d_{j+1} \neq d_j$, then $d_{j+1} \geq d_j + d_{j-1}$.*

(iii) *$\mathbf{S}_i$ can be represented by $O(\log i)$ arithmetic progressions, where each arithmetic progression is a tuple $\langle s, d, t \rangle$ representing the sequence $s, s + d, \ldots, s + (t - 1)d$ with common difference d.*

(iv) *If $t \geq 2$, then the common difference d is a period of every maximal palindrome which ends at position i in T and whose length belongs to the arithmetic progression $\langle s, d, t \rangle$.*

See also Fig. 3 in the next section. 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. For all $1 \leq i \leq n$ we can compute $MaxPalEnd_T(i)$ and $MaxPalBeg_T(i)$ 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 Algorithm for ℓ-ELSPal

Consider to substitute a string X of length ℓ 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 the edit.

In order to compute (the lengths of) maximal palindromes that are affected by the block-wise edit operation, we need to know the first (leftmost) mismatching position between T and T', and that between T^R and T'^R. Let h and l be the smallest integers such that $T[ht] \neq T'[ht]$ and $T^R[l] \neq T'^R[l]$, respectively. If such h does not exist, then let $h = \min\{|T|, |T'|\} + 1$. Similarly, if such l does not exist, then let $l = \min\{|T|, |T'|\} + 1$. Let $j_1 = lcp(T[i_b..n], XT[i_e..n])$, $j_2 = lcp((T[1..i_e])^R, (T[1..i_b]X)^R)$, $p_b = i_b + j_1$, and $p_e = i_e - j_2$. There are two cases: (1) If $j_1 = j_2 = 0$, then the first and last characters of $T[i_b..i_e]$ differ from those of X. In this case, we have $i_b = h$ and $i_e = n - l + 1$. We use these positions i_b and i_e to compute maximal palindromes after the block-wise edit. (2) Otherwise, we have $p_b = i_b + j_1 = h$ and $p_e = i_e - j_2 = n - l + 1$. We use these positions p_b and p_e to compute maximal palindromes after the block-wise edit. See Figure 1 for illustration.

In the next subsection, we describe our algorithm for Case (1). Case (2) can be treated similarly, by replacing i_b and i_e with p_b and p_e, respectively. Our algorithm can handle the case where $p_e < p_b$. Remark that p_b and p_e can be computed in $O(\ell)$ time by naïve character comparisons and a single LCE query each.

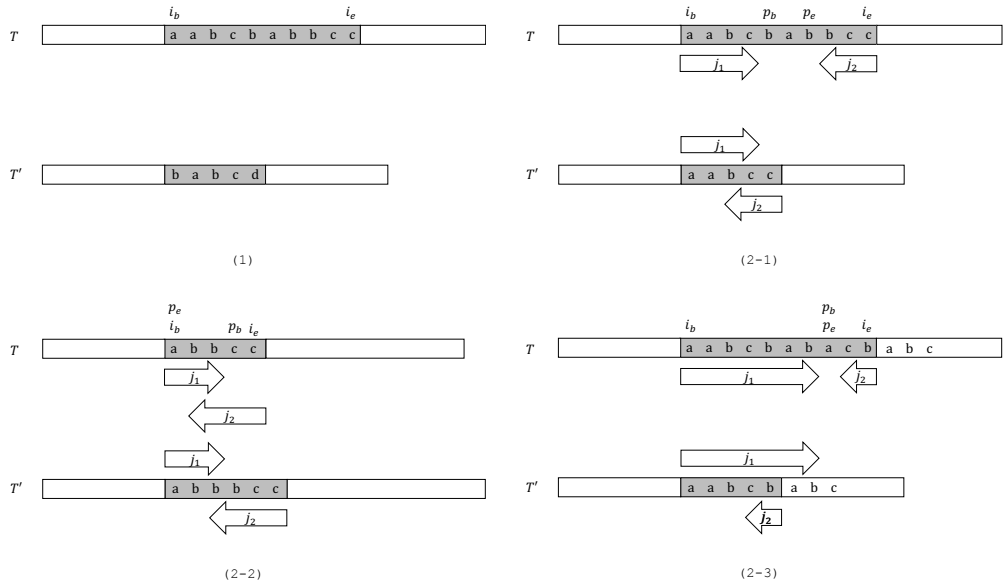


Figure 1 Illustration for the mismatching position between T and T', and that between T^R and T'^R. In particular, (2-2) is the sub-case of Case (2) with $p_e < p_b$, and (2-3) is the sub-case of Case (2) with $j_1 > \ell$.

We remark that the longest extension of the maximal palindromes in T which are unchanged or shortened after the block edit can be found $O(1)$ time upon query, after $O(n)$-time and space preprocessing, using similar techniques to the 1-ELSPal queries from our previous work [9]. Also, the longest maximal palindromes that have centers in the new block can be computed in $O(\ell)$ time after $O(n)$-time and space preprocessing, in a similar way to our previous algorithm for the 1-ELSPal [9]. Hence, in this paper we concentrate on the maximal palindromes of T which get extended after the block edit. The next observation describes such maximal palindromes.

▶ **Observation 4** ([9]). *For any $s \in MaxPalEnd_T(i_b - 1)$, the corresponding maximal palindrome $T[i_b - s..i_b - 1]$ centered at $\frac{2i_b - s - 1}{2}$ gets extended in T' iff $\mathsf{OutLCE}_{T'}(i_b - s - 1, i_b) \geq 1$. Similarly, for any $s \in MaxPalBeg_T(i_e + 1)$, the corresponding maximal palindrome $T[i_e + 1..i_e + s]$ centered at $\frac{2i_e + s + 1}{2}$ gets extended in T' iff $\mathsf{OutLCE}_{T'}(i_e, i_e + s + 1) \geq 1$.*

It follows from Observation 4 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'. The following lemma, which is a generalization of Lemma 21 from [9], shows how to efficiently compute the extensions of any given maximal palindromes that end at position $i_b - 1$. Those that begin at position $i_e + 1$ can be treated similarly.

▶ **Lemma 5.** *Let T be a string of length n over an integer alphabet of size polynomially bounded in n. We can preprocess T in $O(n)$ time and space so that later, given a list of any f maximal palindromes from $MaxPalEnd_T(i_b - 1)$, we can compute in $O(\ell + f)$ time the extensions of those f maximal palindromes in the edited string T', where ℓ is the length of a new block.*

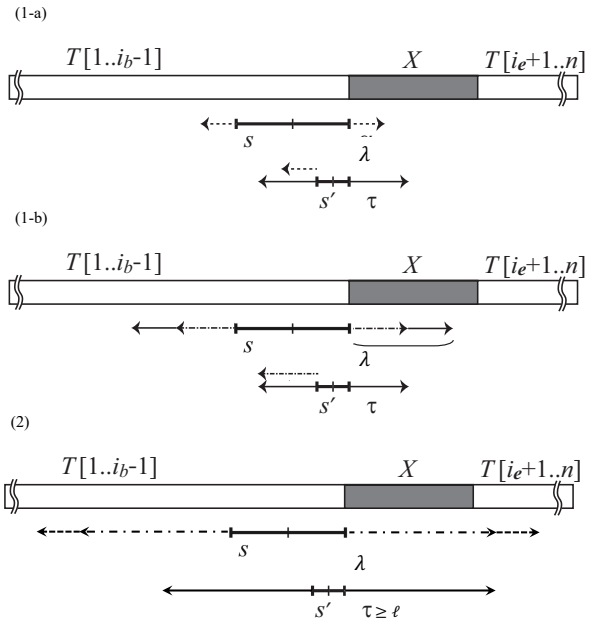


Figure 2 Illustration for Lemma 5, where solid arrows represent the matches obtained by naïve character comparisons, and broken arrows represent those obtained by LCE queries. This figure only shows the case where $s' < s$, but the other case where $s' > s$ can be treated similarly.

Proof. Let us remark that the maximal palindromes in the list can be given to our algorithm in any order. Firstly, we compute the extensions of given maximal palindromes from the list until finding the first maximal palindrome whose extension τ is at least one, and let s' be the length of this maximal palindrome. Namely, $s' + 2\tau$ is the length of the extended maximal palindrome for s', and the preceding maximal palindromes (if any) were not extended. Let s be the length of the next maximal palindrome from the list after s', and now we are to compute the extension λ for s. See also Figure 2. There are two cases: (1) If $0 < \tau < \ell$, then we first compute $\delta = \mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$. We have two sub-cases: (1-a) If $\delta < \tau$, then $\lambda = \delta$. (1-b) Otherwise ($\delta \geq \tau$), then we know that λ is at least as large as τ. We then compute the remainder of λ by naïve character comparisons. If the character comparison reaches the end of X, then the remainder of λ can be computed by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. Then we update τ with λ. (2) If $\tau \geq \ell$, then we can compute λ by $\mathsf{LeftLCE}_T(i_b - s - 1, i_b - s' - 1)$, and if this value is at least ℓ, then by $\mathsf{OutLCE}_T(i_b - s - \ell - 1, i_e + 1)$. The extensions of the following palindromes can also be computed similarly.

The following maximal palindromes from the list after s can be processed similarly. After processing all the f maximal palindromes in the given list, the total number of matching character comparisons is at most ℓ since each position of X is involved in at most one matching character comparison. Also, the total number of mismatching character comparisons is $O(f)$ since for each given maximal palindrome there is at most one mismatching character comparison. The total number of LCE queries on the original text T is $O(f)$, each of which can be answered in $O(1)$ time. Thus, it takes $O(\ell + f)$ time to compute the length of the f maximal palindromes of T' that are extended after the block edit. ◀

However, there can be $\Omega(n)$ maximal palindromes beginning or ending at each position of a string of length n. In what follows, we show how to reduce the number of maximal palindromes that need to be considered, by using periodic structures of maximal palindromes.

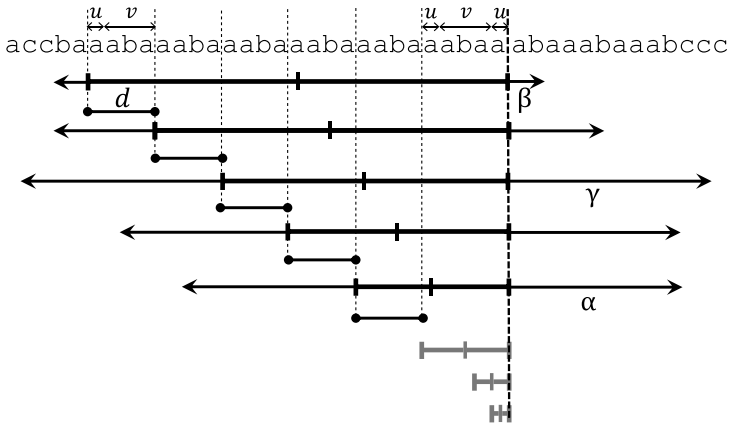


Figure 3 Example for Lemma 6, where Y = accbaaabaaabaaabaaabaaabaaabaaabaa and Z = abaaabaaabccc. Here u = a and v = aba. The first five maximal palindromes $(uv)^k u = (\texttt{aaba})^k\texttt{a}$ with $2 \leq k \leq 5$ belong to the same arithmetic progression (i.e. the same group) with common difference $|uv| = d = 4$. For this group of maximal palindromes, $\alpha = 10$, $\beta = 2$, and $\gamma = 12$. Notice that the sixth maximal palindrome uvu = aabaa belongs to another group since the length difference between it and the seventh one aa is 3.

Let $\langle s, d, t\rangle$ be an arithmetic progression representing a group of maximal palindromes ending at position i_b-1. For each $1 \leq j \leq t$, we will use the convention that $s(j) = s+(j-1)d$, namely s denotes the jth shortest element for $\langle s, d, t\rangle$. For simplicity, let $Y = T[1..i_b - 1]$ and $Z = XT[i_e + 1..n]$. Let $Ext(s(j))$ denote the length of the maximal palindrome that is obtained by extending $s(j)$ in YZ.

▶ **Lemma 6** ([9]). *For any $\langle s, d, t\rangle \subseteq MaxPalEnd_T(i_b - 1)$, there exist palindromes u, v and a non-negative integer p, such that $(uv)^{t+p-1}u$ (resp. $(uv)^p u$) is the longest (resp. shortest) maximal palindrome represented by $\langle s, d, t\rangle$ with $|uv| = d$. Let $\alpha = lcp((Y[1..|Y| - s_1])^R, Z)$ and $\beta = lcp((Y[1..|Y| - s_t])^R, Z)$. If there exists $s_h \in \langle s, d, t\rangle$ such that $s_h + \alpha = s_t + \beta$, then let $\gamma = lcp((Y[1..|Y| - s_h])^R, Z)$. Then, for any $s(j) \in \langle s, d, t\rangle \setminus \{s_h\}$, $Ext(s(j)) = s(j) + 2\min\{\alpha, \beta + (t - j)d\}$. Also, if s_h exists, then $Ext(s_h) = s_h + 2\gamma \geq Ext(s(j))$ for any $j \neq h$.*

See Figure 3 for a concrete example of Lemma 6. It follows from Lemma 6 that it suffices to consider only three maximal palindromes from each group (i.e. each arithmetic progression). Then using Lemma 5, one can compute the longest maximal palindrome that gets extended in $O(\ell + \log n)$ time, and this is exactly how the algorithm from our previous paper [9] works.

To further speed up computation, we take deeper insights into combinatorial properties of maximal palindromes in $MaxPalEnd_T(i_b - 1)$. Let $G_1, \ldots, G_m$ be the list of all groups for the maximal palindromes from $MaxPalEnd_T(i_b - 1)$, which are *sorted in increasing order of their common difference*. When $m = O(\log \log n)$, $O(\ell + \log \log n)$-time queries immediately follow from Lemmas 5 and 6. In what follows we consider the more difficult case where $m = \omega(\log \log n)$. Recall also that $m = O(\log n)$ always holds.

For each $G_r = \langle s_r, d_r, t_r\rangle$ with $1 \leq r \leq m$, let α_r, β_r, γ_r, u_r, and v_r be the corresponding variables used in Lemma 6. If there is only a single element in G_r, let β_r be the length of extension of the palindrome and $\alpha_r = \beta_{r-1}$. For each G_r, let S_r (resp. L_r) denote the shortest (resp. longest) maximal palindrome in G_r, namely, $|S_r| = s_r(1)$ and $|L_r| = s_r(t_r)$.

Each group G_r is said to be of *type-1* (resp. *type-2*) if $\alpha_r < d_r$ (resp. $\alpha_r \geq d_r$).

Let k $(1 \leq k \leq m)$ be the unique integer such that G_k is the type-2 group where d_k is the largest common difference among all the type-2 groups. Additionally, let $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$. Note that u_k belongs to one of $G_1, \ldots, G_{k-1}$, and $u_k v_k u_k$ belongs to either G_k or one of $G_1, \ldots, G_{k-1}$. In the special case where $\alpha_k = \beta_k + td_k$, the extensions of u_k and $u_k v_k u_k$ can be longer than the extension of the shortest maximal palindrome in G_k (see Figure 4 for a concrete example). Thus, it is convenient for us to treat $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$ as if it is a single group. We also remark that this set G'_k is defined only for this specific type-2 group G_k.

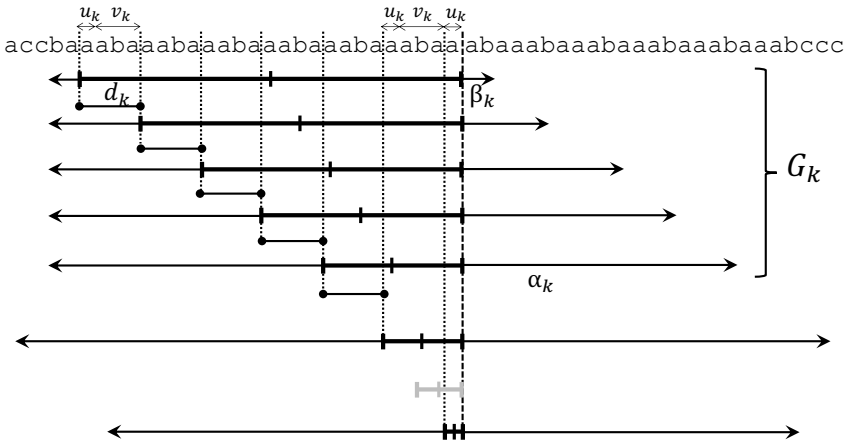


Figure 4 Example for $G'_k = G_k \cup \{u_k v_k u_k, u_k\}$, where the extensions of $u_k v_k u_k$ and u_k are longer than the extensions of any maximal palindromes in G_k.

▶ **Lemma 7.** *There is a longest substring palindrome in the edited string T' that is obtained by extending the maximal palindromes in G_m, G_{m-1}, or G'_k.*

Proof. The lemma holds if the two following claims are true:

▷ Claim (1). The extensions of the maximal palindromes in $G_1, \ldots, G_{k-1}$, except for $u_k v_k u_k$ and u_k, cannot be longer than the extension of the shortest maximal palindrome in G_k.

▷ Claim (2). Suppose both G_m and G_{m-1} are of type-1. Then, the extensions of the maximal palindromes from $G_{k+1}, \ldots, G_{m-2}$, which are also of type-1, cannot be longer than the extensions of the maximal palindromes from G_m or G_{m-1}.

Proof for Claim (1). Here we consider the case where the maximal palindrome $u_k v_k u_k$ does not belong to G_k, which implies that the shortest maximal palindrome S_k in G_k is $(u_k v_k)^2 u_k$ (The other case where $u_k v_k u_k$ belongs to G_k can be treated similarly). Now, $u_k v_k u_k$ belongs to one of $G_1, \ldots, G_{k-1}$. Consider the prefix $P = T[1..i_b - |u_k v_k u_k| - 1]$ of T that immediately precedes $u_k v_k u_k$. The extension of $u_k v_k u_k$ is obtained by $lcp(P^R, Z)$. Consider the prefix $P' = T[1..i_b - |(u_k v_k)^2 u_k|]$ of T that immediately precedes. It is clear that P is a concatenation of P' and $u_k v_k$. Similarly, the prefix $T[1..i_b - |u_k| - 1]$ of T that immediately precedes u_k is a concatenation of P and $u_k v_k$. From Lemma 6 and the definition of G'_k, it suffices for us to consider only the three maximal palindromes from G'_k. For any other maximal palindrome Q from $G_1, \ldots, G_{k-1}$, assume on the contrary that Q gets extended by at least d_k to the left and to the right. If $|u_k v_k| = d_k < |Q| < |u_k v_k u_k|$, then there is an internal occurrence of $u_k v_k$ inside the prefix $(u_k v_k)^2$ of $(u_k v_k)^2 u_k$. Otherwise ($|u_k| < |Q| < |u_k v_k| = d_k$ or $|Q| < |u_k|$), there is an internal occurrence of $u_k v_k$ inside $u_k v_k u_k$. Here we only consider the first case

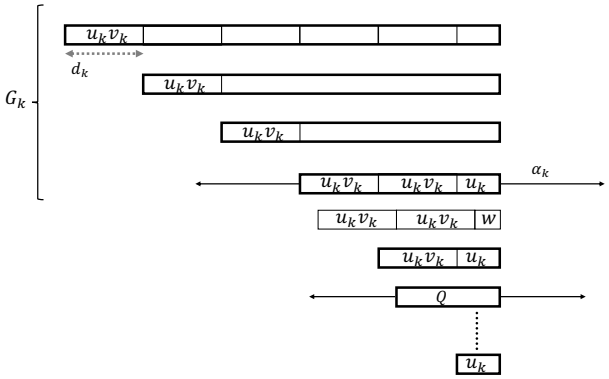


Figure 5 Illustration for the proof for Claim (1) of Lemma 7.

but other cases can be treated similarly. See also Figure 5. This internal occurrence of u_kv_k is immediately followed by u_kv_kw, where w is a proper prefix of u_k with $1 \leq |w| < |u_k|$. Namely, $(u_kv_k)^2w$ is a proper suffix of $(u_kv_k)^2u_k$. On the other hand, $(u_kv_k)^2w$ is also a proper prefix of $(u_kv_k)^2u_k$. Since $(u_kv_k)^2u$ is a palindrome, it now follows from Lemma 2 that $(u_kv_k)^2w$ is also a palindrome. Since $1 \leq |w| < |u_k|$, we have $|(u_kv_k)^2w| > |u_kv_ku_k|$ (note that this inequality holds also when v_k is the empty string). Then, $(u_kv_k)^2w$ is also immediately preceded by u_kv_k because of periodicity and is extended by at least d_k to the left and to the right. Since $T'[i_b] = T[i_b - |(u_kv_k)^2w| - 1]$ and $T'[i_b] \neq T[i_b]$, $(u_kv_k)^2w$ is a maximal palindrome. However this contradicts that $(u_kv_k)^2u_k$ belongs to G_k with common difference $d_k = |u_kv_k|$. Thus Q cannot be extended by d_k nor more to the left and to the right. Since G_k is of type-2, $\alpha_k \geq d_k$. Since $|Q| < |(u_kv_k)^2u_k|$, the extension of Q cannot be longer than the extension for $(u_kv_k)^2u_k$. This completes the proof for Claim (1). ◁

Proof for Claim (2). Consider each group $G_r = \langle s_r, d_r, t_r \rangle$ with $k+1 \leq r \leq m-2$. By Lemma 6, $s_r(t_r) + 2\beta_r$ and $s_r(t_r - 1) + 2\alpha_r$ are the candidates for the longest extensions of the maximal palindromes from G_r. Recall that both G_{m-1} and G_m are of type-1, and that if G_r is of type-1 then G_{r+1} is also of type-1. Now the following sub-claim holds:

▶ **Lemma 8.** *$\beta_r = \alpha_{r+1}$ for any $k+1 \leq r \leq m-2$.*

Proof. If G_{r+1} is a singleton, then by definition $\beta_r = \alpha_{r+1}$ holds. Now suppose $|G_{r+1}| \geq 2$. Since the shortest maximal palindrome S_{r+1} from G_{r+1} is either $(u_{r+1}v_{r+1})^2u_{r+1}$ or $u_{r+1}v_{r+1}u_{r+1}$, the longest maximal palindrome L_r from G_r is either $u_{r+1}v_{r+1}u_{r+1}$ or u_{r+1}. The prefix $T[1..i_b - |L_r| - 1]$ of T that immediately precedes L_r contains $u_{r+1}v_{r+1}$ as a suffix, which alternatively means $(u_{r+1}v_{r+1})^R$ is a prefix of $(T[1..i_b - |L_r| - 1])^R$. Moreover, it is clear that the prefix $T[1..i_b - |S_{r+1}| - 1]$ of T that immediately precedes S_{r+1} contains $u_{r+1}v_{r+1}$ as a suffix since $|G_{r+1}| \geq 2$. In addition, $\alpha_{r+1} < d_{r+1} = |u_{r+1}v_{r+1}|$ since G_{r+1} is of type-1. From the above arguments, we get $\beta_r = \alpha_{r+1}$. ◀

Since $\beta_r = \alpha_{r+1}$ and $\alpha_{r+1} < d_{r+1}$, we have $s_r(t_r) + 2\beta_r < s_r(t_r) + 2d_{r+1}$. In addition, $s_r(t_r - 1) + 2\alpha_r < s_r(t_r - 1) + 2d_r = s_r(t_r) + d_r$. It now follows from $d_r < d_{r+1}$ that $s_r(t_r) + d_r < s_r(t_r) + 2d_{r+1}$. Since the lengths of the maximal palindromes and their common differences are arranged in increasing order in the groups $G_{k+1}, \ldots G_{m-2}$, we have that the longest extension from $G_{k+1}, \ldots G_{m-2}$ is shorter than $s_{m-2}(t_{m-2}) + 2d_{m-1}$. Since $d_{m-1} < d_m$, we have

$$s_{m-2}(t_{m-2}) + 2d_{m-1} < s_{m-2}(t_{m-2}) + d_{m-1} + d_m \leq s_{m-1}(t_{m-1}) + d_m \leq s_m = s_m(1).$$

This means that the longest extended maximal palindrome from the type-1 groups G_{k+1}, ..., G_{m-2} cannot be longer than the original length of the maximal palindrome from G_m before the extension. This completes the proof for Claim (2). ◁

◀

It follows from Lemmas 5, 6 and 7 that given G_k, we can compute in $O(\ell)$ time the length of the LSPal of T' after the block edit. What remains is how to quickly find G_k, that has the largest common difference among all the type-2 groups. Note that a simple linear search from G_m or G_1 takes $O(\log n)$ time, which is prohibited when $\ell = o(\log n)$. In what follows, we show how to find G_k in $O(\ell + \log\log n)$ time.

Recall that $T[1..i_b - |L_{r-1}| - 1]$ which immediately precedes S_r contains $u_r v_r$ as a suffix. Thus, $(u_r v_r)^R$ is a prefix of $(T[1..i_b - |L_{r-1}| - 1])^R$. We have the following observation.

▶ **Observation 9.** *Let $W_1 = (T[1..i_b - 1])^R$, and $W_r = (T[1..i_b - |L_{r-1}| - 1])^R$ for $2 \le r \le m$. Let W be the string such that $lcp(W_r, Z)$ is the largest for all $1 \le r \le m$ (i.e. for all groups $G_1, \ldots, G_m$), namely, $W = \arg\max_{1 \le r \le m} lcp(W_r, Z)$. Then $G_k = G_x$ such that*

(a) *$(u_x v_x)^R$ is a prefix of W,*
(b) *$d_x \le lcp(W, Z)$, and*
(c) *d_x is the largest among all groups that satisfy Conditions (a) and (b).*

Due to Observation 9, the first task is to find W.

▶ **Lemma 10.** *W can be found in $O(\ell + \log\log n)$ time after $O(n)$-time and space preprocessing.*

Proof. We preprocess T as follows. For each $1 \le i \le n$, let $G_1, \ldots, G_m$ be the list of groups that represent the maximal palindromes ending at position i in T. Let $W_1 = (T[1..i])^R$ and $W_r = (T[1..i - |L_{r-1}|])^R$ for $2 \le r \le m$. Let $\mathcal{A}_i$ be the *sparse suffix array* of size $m = O(\log i)$ such that $\mathcal{A}_i[j]$ stores the jth lexicographically smallest string in $\{W_1, \ldots, W_m\}$. We build $\mathcal{A}_i$ with the LCP array $\mathcal{L}_i$. Since there are only $2n - 1$ maximal palindromes in T, $\mathcal{A}_i$ for all positions $1 \le i \le n$ can easily be constructed in a total of $O(n)$ time from the full suffix array of T. The LCP array $\mathcal{L}_i$ for all $1 \le i \le n$ can also be computed in $O(n)$ total time from the LCP array of T enhanced with a range minimum query (RMQ) data structure [4].

To find W, we binary search $\mathcal{A}_{i_b - 1}$ for $Z[1..\ell] = X$ in a similar way to pattern matching on the suffix array with the LCP array [17]. This gives us the range of $\mathcal{A}_{i_b - 1}$ such that the corresponding strings have the longest common prefix with X. Since $|\mathcal{A}_{i_b - 1}| = O(\log n)$, this range can be found in $O(\ell + \log\log n)$ time. If the longest prefix found above is shorter than ℓ, then this prefix is W. Otherwise, we perform another binary search on this range for $Z[\ell + 1..|Z|] = T[i_e + 1..n]$, and this gives us W. Here each comparison can be done in $O(1)$ time by an outward LCE query on T. Hence, the longest match for $Z[\ell + 1..|Z|]$ in this range can also be found in $O(\log\log n)$ time. Overall, W can be found in $O(\ell + \log\log n)$ time. ◀

▶ **Lemma 11.** *We can preprocess T in $O(n)$ time and space so that later, given W for a position in T, we can find G_k for that position in $O(\log\log n)$ time.*

Proof. Let $\mathcal{D}_i$ be an array of size $|\mathcal{A}_i|$ such that $\mathcal{D}_i[j]$ stores the value of $d_r = |u_r v_r|$, where W_r is the lexicographically jth smallest string in $\{W_1, \ldots, W_m\}$. Let $\mathcal{R}_i$ be an array of size $|\mathcal{A}_i|$ where $\mathcal{R}_i[j]$ stores a sorted list of common differences $d_r = |u_r v_r|$ of groups G_r, such that G_r stores maximal palindromes ending at position i and $(u_r v_r)^R$ is a prefix of the string $\mathcal{A}_i[j]$. Clearly, for any j, $\mathcal{D}_i[j] \subseteq \mathcal{R}_i$. See also Figure 6 for an example of $\mathcal{R}_i$.

Suppose that we have found W by Lemma 10, and let j be the entry of $\mathcal{A}_{i_b-1}$ where the binary search for X terminated. We then find the largest d_x that satisfies Condition (b) of Observation 9, by binary search on the sorted list of common differences stored at $\mathcal{R}_{i_b-1}[j]$. This takes $O(\log\log n)$ time since the list stored at each entry of $\mathcal{R}_{i_b-1}$ contains at most $|A_{i_b-1}| = O(\log n)$ elements.

We remark however that the total number of elements in $\mathcal{R}_i$ is $O(\log^2 i)$ since each entry $\mathcal{R}_i[j]$ can contain $O(\log i)$ elements. Thus, computing and storing $\mathcal{R}_i$ explicitly for all text positions $1 \leq i \leq n$ can take superlinear time and space.

Instead of explicitly storing $\mathcal{R}_i$, we use a tree representation of $\mathcal{R}_i$, defined as follows: The tree consists of exactly $m = |\mathcal{A}_i|$ leaves and exactly m non-leaf nodes. Each leaf corresponds to a distinct entry $j = 1, \ldots, m$, and each non-leaf node corresponds to a value from $\mathcal{D}_i$. Each leaf j is contained in a (sub)tree rooted at a node with $d \in \mathcal{D}_i$, iff there is a maximal interval $[j'..j'']$ such that $j' \leq j \leq j''$ and $\mathcal{L}_i[j+1] \geq \mathcal{D}_i[j]$. We associate each node with this maximal interval. Since we have assumed $d_1 = 0$, the root stores 0 and it has at most σ children. See Figure 6 that illustrates a concrete example for $T[1..i_b - 1] = dddF_7^4F_6^2F_5^2F_4F_3^3F_2^2F_1^3$ with $i_b = 3451$, where

$$
\begin{aligned}
F_1 &= a \\
F_2 &= {F_1^3}^R b \\
F_3 &= {F_1^3}^R {F_2^2}^R \\
F_4 &= {F_1^3}^R {F_2^2}^R {F_3^2}^R F_2 \\
F_5 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R c \\
F_6 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R {F_5}^R c F_4 F_3^2 \\
F_7 &= {F_1^3}^R {F_2^2}^R {F_3^3}^R {F_4}^R {F_5^2}^R {F_6}^R {F_3^2}^R {F_4}^R c F_5 F_4 F_3^3 F_2^2 F_1^2.
\end{aligned}
$$

We remark that $F_7^4F_6^2F_5^2F_4F_3^3F_2^2F_1^3$, $F_7^3F_6^2F_5^2F_4F_3^3F_2^2F_1^3$, $\ldots$, F_1 are suffix palindromes of $T[1..i_b - 1]$.

We can easily construct this tree in time linear in its size $m = |\mathcal{A}_i|$, in a bottom up manner. First, we create leaves for all entries $j = 1, \ldots, m$. Next, we build the tree in a bottom-up manner, by performing the following operations in decreasing order of $\mathcal{D}_i[j]$.

(1) Create a new node with $\mathcal{D}_i[j]$, and connect this node with the highest ancestor of leaf j.

(2) We check $j' < j$ in decreasing order, and connect the new node with the highest ancestor of leaf j' iff $\mathcal{L}_i[j'+1] \geq \mathcal{D}_i[j]$. We skip the interval corresponding to this ancestor, and perform the same procedure until we find j' that does not meet the above condition. We do the same for $j'' > j$.

Since each node is associated with its corresponding interval in the LCP array, it suffices for us to check the conditions $\mathcal{L}_i[j'+1] \geq \mathcal{D}_i[j]$ and $\mathcal{L}_i[j''] \geq \mathcal{D}_i[j]$ only at either end of the intervals that we encounter. Clearly, in the path from the root to leaf j, the values in $\mathcal{R}_j[j]$ appear in increasing order. Thus, we can find the largest d_x that satisfies Condition (b) of Observation 9, by a binary search on the corresponding path in the tree. We augment the tree with a level ancestor data structure [7, 5], so that each binary search takes logarithmic time in the tree height, namely $O(\log\log n)$ time. The size of the tree for position i is clearly bounded by the number of maximal palindromes ending at position i. Thus, the total size and construction time for the trees for all positions in T is $O(n)$. ◀

j	$W_{\mathcal{A}_i[1]}, \ldots, W_{\mathcal{A}_i[m]}$	$\mathcal{D}_i$	$\mathcal{L}_i$	$\mathcal{R}_i$
1	abaaabaaaaaabaaaba$\cdots$	689	-	0,1,689
2	aabaaabaaaaaabaaab$\cdots$	1	1	0,1
3	aaabaaabaaaaaabaaa$\cdots$	223	2	0,1,11,223
4	aaabaaabaaaaaabaaa$\cdots$	0	22	0,1,11
5	aaabaaabaaaaaabaaa$\cdots$	11	33	0,1,11
6	baaabaaaaaabaaabaa$\cdots$	4	0	0,4
7	baaaaaabaaabaaaaaa$\cdots$	37	4	0,4,37
8	caaabaaabaaaaaabaa$\cdots$	82	0	0,82

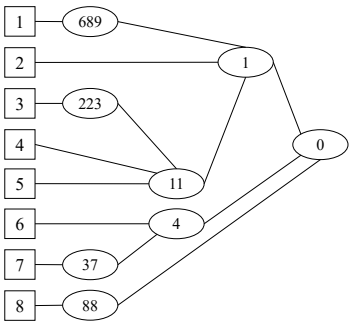


Figure 6 Example for $\mathcal{R}_i$ (left) and its corresponding tree (right). The remaining parts of the strings $W_{\mathcal{A}_i[1]}, \ldots, W_{\mathcal{A}_i[m]}$ are omitted due to lack of space.

By Lemma 5, 6, 7 and 11, we can compute in $O(\ell + \log\log n)$ time the length of the LSPal of T' that are extended after the block edit.

Consequently we obtain the following theorem:

▶ **Theorem 12.** *There is an $O(n)$-time and space preprocessing for the ℓ-ELSPal problem such that each query can be answered in $O(\ell + \log\log n)$ time, where ℓ denotes the length of the block after edit.*

Note that the time complexity for our algorithm is independent of the length of the original block to edit. Also, the length ℓ of a new block is arbitrary.

References

1 Amihood 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 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest Common Factor Made Fully Dynamic. *CoRR*, abs/1804.08731, 2018. `arXiv:1804.08731`.

3 Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141:163–173, 1995.

4 Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *LATIN 2000*, pages 88–94, 2000.

5 Michael A. Bender and Martin Farach-Colton. The Level Ancestor Problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

6 Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome Recognition In The Streaming Model. In *STACS 2014*, pages 149–161, 2014.

7 O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. System Sci.*, 48(2):214–230, 1994.

8 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

9 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest substring palindrome after edit. In *CPM 2018*, pages 12:1–12:14, 2018.

10 Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter Bounds and Optimal Algorithms for All Maximal α-gapped Repeats and Palindromes – Finding All Maximal α-gapped Repeats and Palindromes in Optimal Worst Case Time on Integer Alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018.

11 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams. In *CPM 2016*, pages 18:1–18:13, 2016.

12 Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010.

13 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

14 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.

15 Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Finding Distinct Subpalindromes Online. In *PSC 2013*, pages 63–69, 2013.

16 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.

17 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

18 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.

19 Alexandre H. L. Porto and Valmir C. Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 35:2581–2591, 2002.

20 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon Substring After Edit. In *CPM 2018*, pages 19:1–19:10, 2018.

21 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

A Rearrangement Distance for Fully-Labelled Trees

Giulia Bernardini
DISCo, Università degli Studi Milano – Bicocca, Italy
giulia.bernardini@unimib.it

Paola Bonizzoni
DISCo, Università degli Studi Milano – Bicocca, Italy
paola.bonizzoni@unimib.it

Gianluca Della Vedova
DISCo, Università degli Studi Milano – Bicocca, Italy
gianluca.dellavedova@unimib.it

Murray Patterson
DISCo, Università degli Studi Milano – Bicocca, Italy
murray.patterson@unimib.it

Abstract

The problem of comparing trees representing the evolutionary histories of cancerous tumors has turned out to be crucial, since there is a variety of different methods which typically infer multiple possible trees. A departure from the widely studied setting of classical phylogenetics, where trees are leaf-labelled, tumoral trees are fully labelled, *i.e.*, *every* vertex has a label.

In this paper we provide a rearrangement distance measure between two fully-labelled trees. This notion originates from two operations: one which modifies the topology of the tree, the other which permutes the labels of the vertices, hence leaving the topology unaffected. While we show that the distance between two trees in terms of each such operation alone can be decided in polynomial time, the more general notion of distance when both operations are allowed is NP-hard to decide. Despite this result, we show that it is fixed-parameter tractable, and we give a 4-approximation algorithm when one of the trees is binary.

2012 ACM Subject Classification Mathematics of computing → Trees; Mathematics of computing → Graph theory

Keywords and phrases Tree rearrangement distance, Cancer progression, Approximation algorithms, Computational complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.28

Acknowledgements The authors wish to thank Mauricio Soto Gomez for the inspiring discussions.

1 Introduction

Tree rearrangement concerns modifying the topology of a set of elements arranged in a tree and has a large literature [33] on phylogenies, that is, trees whose leaves, and only leaves, are labelled by *taxa*. This setting is convenient, since it is a faithful model of the actual problem that biologists want to solve: to find a plausible evolutionary history that can explain a set of extant species (or individuals) [13] – the internal nodes being the hypothetical ancestral taxa, which are extinct. The problem of inferring phylogenies is centuries old, and there is a rich literature on computational methods, which fall into major groups such as parsimony methods [12, 14], or maximum-likelihood methods [7, 13]. With the wealth of different methods for inferring phylogenetic trees on a given set of taxa, a notion of *rearrangement distance* between output trees can be useful to assess the reliability of methods or even the data itself in inferring such trees. This sparked a body of research on rearrangement distances

for phylogenetic trees, resulting in rearrangement distance measures such as *nearest neighbor interchange* (NNI) [9], *subtree prune and regraft* (SPR) [5] and *tree bisection and reconnection* (TBR) [2] – interestingly, all such rearrangement distances are NP-hard to decide. See [23] for a comprehensive survey of the above rearrangement distance measures, as well as other more general distance measures for phylogenetic trees, such as the classical Robinson-Foulds distance [30].

This paper focuses on a different, albeit related, biological field: the study of cancer progression. While the theory on cancer progression as an evolutionary process is several decades old [27], only in the past decade are data appearing that allow us to reconstruct in detail the evolutionary history of the progression of various cancers [20, 17] – providing insight into drug resistance and devising therapeutic strategies [26, 37]. In this setting, we have one or more tumor samples where the taxa are cancer *clones* [27, 17], or groups of cancer cells at various stages of mutation – all of which originate from a single *driver mutation*, and the goal is again to construct the most likely evolutionary history of these clones. The key difference in this setting is that – since the tumor is only months old – all of the clones, even the one representing only the driver mutation, are present in the samples, *i.e.*, the internal nodes are extant taxa. In this setting, the inferred evolutionary history is rather a *fully-labelled* tree, where a label represents a single mutation that has been acquired by a clone during evolution. It is quite common to assume that the evolution of mutations follows the *infinite sites assumption* [18, 17, 11] which implies that once a mutation is acquired in a node it is never lost, and thus it will be present in all the clones associated with the descendants of the node. The above assumption motivates the fact that we can label each node with a single mutation. More recently, this assumption has been challenged [24].

There is already a wealth of methods for inferring such fully-labelled cancer evolutionary trees, most of them leveraging bulk sequencing data [20, 17, 38, 11, 4, 8], however methods taking advantage of higher resolution Single Cell Sequencing (SCS) technologies [18, 31] – even some hybrid methods – are beginning to appear [32]. With the amount of data and methods becoming available for inferring cancer evolution, a main challenging problem turns out to be the comparison of the multiple trees that are produced by a single method or by different approaches, see, *e.g.*, [29]. The investigation of operations for defining the rearrangement distance between trees output by these methods is still in its infancy and require the comparison of trees over the same set of mutations, *i.e.*, labels. Indeed, most recent works, *e.g.*, [16] are mainly focused on defining a consensus tree on path, or on ancestor-based distance measures, rather than on transforming a tree into another considering also the topology of the trees and the ancestor-descendant relationship of the labels – since *all* nodes of the trees are now labelled.

While we concentrate in this work on rearrangement distance between fully-labelled trees, this move to fully-labelled trees opens up the discussion to the more general *edit distance* between fully-labelled trees, purportedly introduced in [36], and of which there is a sizeable literature, *e.g.*, [39, 25]. There is even a comprehensive survey on the topic in [3], while a recent implementation is reported in [28, 28]. Even in the context of cancer progression, a recent paper [22] provides a notion of edit distance for multi-labelled trees has been defined with the goal of reconciling two trees over distinct sets of labels into a common one.

In this work, we open the investigation of some notions of the rearrangement distance for two rooted trees which are fully labelled by the same set of labels. Following the existing literature [33, 35] on phylogeny rearrangement, we extend to several operations for rearranging a fully-labelled tree. The distance between a pair of trees is then the shortest sequence of these operations that transforms the first tree into the second tree. The first

operation we introduce is an adaptation of the SPR operation [5] to a fully-labelled tree. We introduce a second operation that consists of a permutation of the labels of the tree – notice that such an operation does not really make sense on leaf-labelled phylogenies. For both operations, we provide an algorithm for computing the shortest sequence of operations needed to transform an input tree into a second input tree. Then we extend this rearrangement measure by allowing both operations: we show that the new computational problem of finding a shortest sequence of operations is NP-hard, but we give a 4-approximation ratio and a fixed parameter algorithm.

2 Preliminaries

A *tree* is an undirected connected graph $T = (V_T, E_T)$ without cycles: its degree-one vertices are called *leaves*, while the remaining vertices are called *internal* vertices. Trees T_1 and T_2 are is *isomorphic*, and we write $T_1 \cong T_2$, if there is a bijective (or one-to-one) mapping $m : V_{T_1} \to V_{T_2}$ such that $(u, v) \in E_{T_1}$ iff $(m(u), m(v)) \in E_{T_2}$. Such a mapping is referred to as an *isomorphic mapping*, or an *isomorphism*.

A *rooted* tree has an edge between some vertex $w \in V_T$ and an extra *root* vertex $\lambda_T \notin V_T$ which has been added, implicitly directing the edges, *e.g.*, away from the root. We can hence define the *parent* and *child* relationships, $p_T : V_T \to V_T \cup \{\lambda_T\}$ and $c_T : V_T \cup \{\lambda_T\} \to 2^{V_T}$ respectively, where $p_T(v) = u$ (resp., $v \in c_T(u)$) if $(u, v) \in E_T \cup \{(\lambda_T, w)\}$ and u is on the path from λ_T to v. Note that since the children $c_T(u)$ of some vertex u is a *set*, hence they are *unordered*, unlike in some notions defined in [3] where an ordering can be specified. Moreover, $|c_T(u)|$ for any u is not of any fixed size, *i.e.*, vertices are of *unbounded degree* – the trees are not necessarily binary, for example. More generally, we say that a node u is an *ancestor* of a node v if u is on the path from λ_T to v, and conversely, v is a *descendant* of u. We extend the above notion of isomorphism to a pair T_1, T_2 of rooted trees by adding the condition that $m(\lambda_{T_1}) = \lambda_{T_2}$.

A tree T is *fully labelled* when its vertices V_T are in a one-to-one correspondence with some set $\mathcal{L}$ of labels, implying that $|V_T| = |\mathcal{L}|$. Since all trees in this paper are rooted and fully labelled, we will henceforth use the term *tree* to denote a rooted, fully labelled tree, and the expression *tree T labelled by $\mathcal{L}$* to denote a rooted tree T, fully labelled by the set $\mathcal{L}$ of labels. Trees T_1 and T_2 each labelled by $\mathcal{L}$ are *congruent* if T_1 and T_2 are isomorphic, and one of the isomorphisms m also has the property that for every $u \in V_{T_1}$, u and $m(u)$ have the same label. Note that if T_1 and T_2 are congruent, then the unique child of λ_{T_1} has the same label as the unique child of λ_{T_2}. Since all vertices of T_1 and T_2 are uniquely labelled and λ_{T_1}, λ_{T_2} are special, we refer to a vertex of a tree and its label interchangeably, when this has no effect, while clearly distinguishing them in contexts where it matters.

In this paper, we study some notions of distance between pairs T_1, T_2 of trees based on some *rearrangement operations* that transform T_1 into some T_1' that is congruent to T_2. For the sake of simplicity, from now on we will slightly abuse terminology in saying that a sequence of operations transforms T_1 into T_2. Given a tree T labelled by $\mathcal{L}$, the operations are:

- **link-and-cut operation**: given labels v, $p_T(v) = u$ and a third label w which is not a descendant of v, remove the edge (u, v) and add the edge (w, v), effectively switching the parent $p_T(v)$ of v from u to w. We denote this operation $v|u \to w$.
- **permutation operation**: apply some permutation $\pi : \mathcal{L} \to \mathcal{L}$ to the labels of V_T. Each label $v \in \mathcal{L}$ of T will have the new label $\pi(v)$ after this operation.

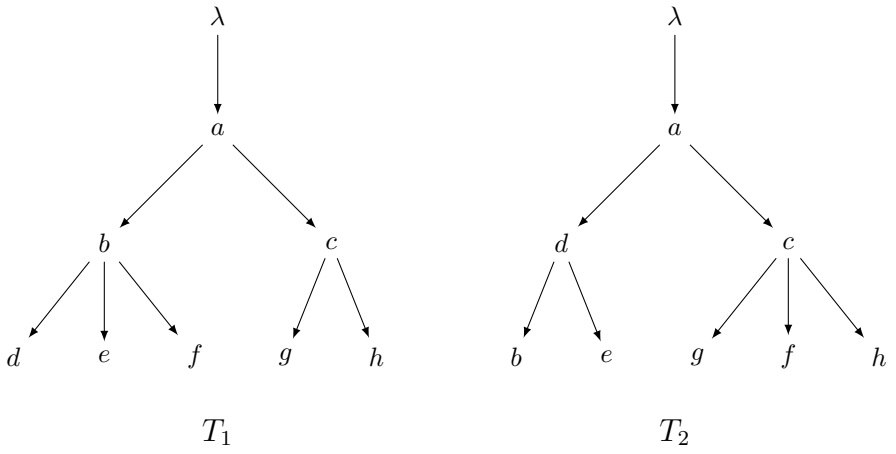


Figure 1 Trees T_1 and T_2 labelled by $\mathcal{L} = \{a, b, c, d, e, f, g, h\}$. The link-and-cut distance $d_\ell(T_1, T_2)$ between T_1 and T_2 is 4 – the sequence $d|b \to a$; $e|b \to d$; $f|b \to c$; $b|a \to d$ being an example of a smallest sequence of such operations transforming T_1 into T_2. Notice that the operation $b|a \to d$ only becomes valid after performing $d|b \to a$. The permutation distance is $d_\pi(T_1, T_2) = 6$, for example, $\pi = (b\ g\ d\ c)(e\ h)$. Finally, the rearrangement distance is $d(T_1, T_2) = 3$, for example, $(b\ d)$; $f|d \to c$.

Notice that the link-and-cut operation modifies the topology of the tree, while the permutation operation shuffles the labels without affecting the topology. The link-and-cut operation is so called, following the terminology of [34] – in this article they define the two operations separately, while ours is a certain combination of them. Our link-and-cut operation is quite similar to the *subtree moving* (*edit distance*) operation of [25], however the operation here has the constraint that the new parent w of child v must be within a certain distance in the tree from the original parent u – the only restriction on our link-and-cut operation is that w cannot be a descendant of v. Both of these operations are invertible: if an operation σ transforms T into T', then its inverse operation σ^{-1} transforms T' into T. For example, if T' was obtained by applying $v|u \to w$ to T, then applying $v|w \to u$ to T' results in T. Similarly, if T' was obtained by applying π to T, then applying π^{-1} to T' results in T. By induction, any sequence of the above operations is invertible.

Additionally, by the definition of permutation, a sequence $\mathcal{S} = \pi_1, \ldots, \pi_k$ of permutation operations can be expressed as the single $\pi = \pi_k \cdot \pi_{k-1} \cdots \pi_2 \cdot \pi_1$ – the composition of the permutations of sequence $\mathcal{S}$. Finally, link-and-cut and permutation operations are interchangeable in a sequence of operations, in the sense that the application of the link-and-cut operation $v|u \to w$ followed by a permutation π, has the same effect as π followed by the link-and-cut operation $\pi(v)|\pi(u) \to \pi(w)$ – inspect Figure 1 and Example 2. We have the following important property.

▶ **Lemma 1.** *Let T_1 and T_2 be each labelled by $\mathcal{L}$, and $\mathcal{S} = \sigma_1, \ldots, \sigma_k$ be a sequence of k operations that transforms T_1 into T_2. Then there exists a sequence $\mathcal{S}^* = \sigma_1^*, \ldots, \sigma_k^*$ of k operations in which all permutation operations precede all link-and-cut operations, and $\mathcal{S}^*$ transforms T_1 into T_2. We say that $\mathcal{S}^*$ is (sequence $\mathcal{S}$) in* canonical form.

Proof. We continue to swap consecutive pairs σ_i, σ_{i+1}, $1 \leq i < k$, of operations in sequence $\mathcal{S}$, where $\sigma_i = u|v \to w$ is a link-and-cut operation and $\sigma_{i+1} = \pi$ is a permutation operation, with the pair σ_{i+1}, $\sigma_i' = \pi(v)|\pi(u) \to \pi(w)$, until we obtain a sequence $\mathcal{S}^*$ in which all permutation operations precede all link-and-cut operations. By induction on this interchange, the resulting $\mathcal{S}^*$ is of length k and transforms T_1 into T_2. ◀

▶ **Example 2.** Consider T_1 and T_2 of Figure 1. The application of the link-and-cut operation $f|b \to c$ followed by the permutation $\pi = (b\ d)$, has the same effect as π followed by the link-and-cut operation $\pi(f)|\pi(b) \to \pi(c) = f|d \to c$.

We now give the following notions of *distance* between trees T_1 and T_2 labelled by $\mathcal{L}$.

▶ **Definition 3** (link-and-cut distance). *The* link-and-cut distance $d_\ell(T_1, T_2)$ *is the length of the shortest sequence of link-and-cut operations which transforms* T_1 *into* T_2.

The following Lemma ensures that the definition of link-and-cut distance is well posed. See also Figure 1.

▶ **Lemma 4.** *Given trees* T_1 *and* T_2 *each labelled by* $\mathcal{L}$*, there always exists a sequence of link-and-cut operations that transforms* T_1 *into* T_2.

Proof. For any node v, $p_{T_1}(v) = u$, such that $p_{T_2}(v) = w$ and w is a descendant of v in T_1 – and thus the operation $v|u \to w$ is not directly applicable – we prove that there exists a node z on the path from v to w in T_1 (including w) such that $p_{T_2}(z)$ is not a descendant of v in T_2 nor a descendant of z in T_1. This implies that after applying the valid operation $z|p_{T_1}(z) \to p_{T_2}(z)$, the operation $v|u \to w$ becomes valid too. There is always such a node z because, should it not exist, w would be a descendant of v also in T_2, giving rise to the cycle $(w \to v \to \cdots \to w)$ and thus contradicting the fact that T_2 is a tree. ◀

Note that, given a *permutation* π of some set S of elements, we denote its *size* $|\pi|$ as the number of elements perturbed by π, *i.e.*, the size of the set $\{s \in S\ :\ \pi(s) \neq s\}$.

▶ **Definition 5** (permutation distance). *The* permutation distance $d_\pi(T_1, T_2)$ *is the size* $|\pi|$ *of the smallest permutation* π *that transforms* T_1 *into* T_2.

Finally, we also define the *size* $|\mathcal{S}|$ of a sequence $\mathcal{S}$ of rearrangement operations as the size of the permutation obtained by composing the permutations of $\mathcal{S}^*$ plus the length of the sequence of link-and-cut operations of $\mathcal{S}^*$, where $\mathcal{S}^*$ is sequence $\mathcal{S}$ in canonical form.

▶ **Definition 6** (rearrangement distance). *The* rearrangement distance $d(T_1, T_2)$ *is the smallest size of any sequence of operations that transforms* T_1 *into* T_2.

Clearly, the permutation distance $d_\pi(T_1, T_2)$ is defined only when T_1 and T_2 are isomorphic, and it is evidently well posed. As a direct consequence of this and Lemma 4, the definition of rearrangement distance is also well posed. Moreover, since these operations are invertible, all the above distance measures are symmetric, and they satisfy by definition the triangle inequality: consider, *e.g.*, the rearrangement distance. Given T_1, T_2 and T_3 labelled by the same set of labels, let $\mathcal{S}_{12}$ be a sequence that transforms T_1 into T_2 such that $|\mathcal{S}_{12}| = d(T_1, T_2)$, $\mathcal{S}_{23}$ a sequence that transforms T_2 into T_3 with $|\mathcal{S}_{23}| = d(T_2, T_3)$, $\mathcal{S}_{13}$ a sequence that transforms T_1 into T_3 and $|\mathcal{S}_{13}| = d(T_1, T_3)$. It is evident that the concatenation $\mathcal{S}_{12}\mathcal{S}_{13}$ of $\mathcal{S}_{12}$ and $\mathcal{S}_{23}$ is a sequence that transforms T_1 into T_3, and by Definition 6 its size is larger or equal to $d(T_1, T_3)$: thus $d(T_1, T_2) + d(T_2, T_3) \geq |\mathcal{S}_{12}\mathcal{S}_{23}| \geq d(T_1, T_3)$. A similar argument shows that the triangular inequality also holds for the link-and-cut distance and the permutation distance.

We now have the important structures.

▶ **Definition 7** (active set). *Given trees* T_1 *and* T_2 *each labelled by* $\mathcal{L}$*, we call* active *the subset* $\mathcal{X} \subseteq \mathcal{L}$ *of labels which have different parents in* T_1 *and* T_2*, i.e.,* $v \in \mathcal{X}$ *iff* $p_{T_1}(v) \neq p_{T_2}(v)$.

Given trees T_1 and T_2 each labelled by $\mathcal{L}$, for each vertex v of the active set $\mathcal{X}$, we can associate with v the pair $(p_{T_1}(v), p_{T_2}(v))$ of the parents of v in the two trees. Let $\mathcal{P}_{(u,w)}$ be the set $\{v \ : \ p_{T_1}(v) = u, \ p_{T_2}(v) = w\}$ – since each vertex has exactly one parent in each tree, each vertex $v \in \mathcal{X}$ belongs to one and only one set $\mathcal{P}_{(u,w)}$. This fact is formalized in the following definition and illustrated in Example 9.

▶ **Definition 8** (family partition)**.** *Let trees T_1 and T_2 each be labelled by $\mathcal{L}$: for each vertex $v \in \mathcal{X}$ we denote the set $\mathcal{P}_{(u,w)} = \{v \ : \ p_{T_1}(v) = u, \ p_{T_2}(v) = w\}$. Then $\mathcal{P}$ is the partition of set $\mathcal{X}$ into the nonempty sets $\mathcal{P}_{(u,w)}$, $u, w \in V$. Partition $\mathcal{P}$ is called the* family partition *of the active set $\mathcal{X}$, and we denote its* size *$|\mathcal{P}|$ as the number of different (non-empty) subsets $\mathcal{P}_{(u,w)}$ it is composed of.*

▶ **Example 9.** Consider T_1 and T_2 of Figure 1. The active set is $\mathcal{X} = \{b, d, e, f\}$. The family partition is composed of the following sets: $\mathcal{P}_{(a,d)} = \{b\}$, $\mathcal{P}_{(b,a)} = \{d\}$, $\mathcal{P}_{(b,d)} = \{e\}$, $\mathcal{P}_{(b,c)} = \{f\}$.

Note that the family partition encodes the elements of any shortest sequence of link-and-cut operations for transforming T_1 into T_2: $v \in \mathcal{P}_{(u,w)}$ corresponds to operation $v|u \to w$. It is easy to see, from the proof of Lemma 4, that a shortest sequence of valid link-and-cut operations can be obtained from $\mathcal{P}$ by ordering the set of operations it encodes with respect to a *depth-first traversal* (DFT) of T_1: $u|p_{T_1}(u) \to p_{T_2}(u)$ precedes $v|p_{T_1}(v) \to p_{T_2}(v)$ if u precedes v in a DFT of T_1. Hence $d_\ell(T_1, T_2) = |\mathcal{X}|$, *i.e.*, the link-and-cut distance is equal to the cardinality of the active set, of which $\mathcal{P}$ is a partition.

3 Computational complexity

In this section we determine the complexity of computing the distance between two trees labelled by the same set of labels, in terms of the three distance measures defined in Section 2. More precisely, despite the fact that the link-and-cut and the permutation distances are polynomial-time computable, computing the rearrangement distance is NP-hard.

3.1 Link-and-cut distance

We first show that we can compute the link-and-cut distance between two trees in linear time by showing that the family partition can be built in linear time.

▶ **Lemma 10.** *The link-and-cut distance $d_\ell(T_1, T_2)$ between trees T_1 and T_2 each labelled by $\mathcal{L}$ can be computed in time $O(|\mathcal{L}|)$.*

Proof. Since the link-and-cut distance is $|\mathcal{X}|$, it suffices to demonstrate that the family partition $\mathcal{P}$ can be built in time $O(|\mathcal{L}|)$. The procedure is as follows: we first do a DFT of tree T_1, building an array $p_{T_1}(v)$ of the parents in T_1, indexed by the child v. We build the same array $p_{T_2}(v)$ for tree T_2. Then we go through the set $\mathcal{L}$ of labels, in some order: at each label v, should $p_{T_1}(v) = u \neq p_{T_2}(v) = w$, we add v to $\mathcal{P}_{(u,w)}$ of the family partition $\mathcal{P}$. Then we just sum up the sizes of the non-empty subsets $\mathcal{P}_{(u,w)}$ of $\mathcal{P}$ in order to obtain $|\mathcal{X}| = d_\ell(T_1, T_2)$. Clearly each tree traversal can be done in time $O(|\mathcal{L}|)$, because $|V_{T_1}| = |V_{T_2}| = |\mathcal{L}|$. In going through the labels, for each label v, we either add or do not add the single vertex v to $\mathcal{P}$, and so this procedure takes time $O(|\mathcal{L}|)$. ◀

3.2 Permutation distance

This subsection is dedicated to proving the following lemma, which shows that we can compute the permutation distance between two trees in cubic time.

▶ **Lemma 11.** *The permutation distance $d_\pi(T_1, T_2)$ between isomorphic trees T_1 and T_2 each labelled by $\mathcal{L}$ can be computed in time $O(|\mathcal{L}|^3)$.*

We need the following definitions and auxiliary lemmas. The *mismatch number* $\Delta(m)$ of some isomorphic mapping m from tree T_1 to tree T_2 is the number of vertices whose label is not conserved by m. More formally, $\Delta(m) = |\{u \in V_{T_1} : \ell_{T_1}(u) \neq \ell_{T_2}(m(u))\}|$, where $\ell_T : V_T \to \mathcal{L}$ is the the one-to-one correspondence between the vertices V_T of tree T and the set $\mathcal{L}$ of labels. Let $\mathcal{I}(T_1, T_2)$ be the set of isomorphic mappings from T_1 to T_2. Given isomorphic trees T_1 and T_2 each labelled by $\mathcal{L}$, let the *mismatch distance* $d_\Delta(T_1, T_2) = \min\{\Delta(m) : m \in \mathcal{I}(T_1, T_2)\}$ be the minimum mismatch number of an isomorphic mapping from T_1 to T_2. The following equality between permutation distance and mismatch distance holds.

▶ **Lemma 12.** *Given isomorphic trees T_1 and T_2 each labelled by $\mathcal{L}$, it follows that $d_\pi(T_1, T_2) = d_\Delta(T_1, T_2)$.*

Proof. Consider an isomorphic mapping m from T_1 to T_2 that has the minimum mismatch number $\Delta(m) = d_\Delta(T_1, T_2)$ and let $\mathcal{L}' \subseteq \mathcal{L}$ be the set of labels of the vertices involved in the set of mismatching vertices between V_{T_1} and V_{T_2} given by m.

Clearly, such labels are in a permutation π which rearranges the labels of tree T_1 to obtain T_2, while, by construction of m, all the other labels will not be perturbed by π. Then we need to show that such a permutation rearranges the minimum number of distinct labels, that is, its size $|\pi| = d_\pi(T_1, T_2)$ is the permutation distance. Indeed, assume to the contrary that the permutation distance $d_\pi(T_1, T_2) < |\pi|$. This implies the existence of a permutation π' that rearranges fewer labels than π, *i.e.*, $|\pi'| < |\pi|$. Then we show that there exists an isomorphic mapping m' that has mismatch number less than the one of m, contradicting the initial assumption.

Indeed, consider the permutation π' and define the mapping m' from T_1 to T_2 such that $m'(u) = v$ whenever $\pi'(v) = u$ The mapping m' is an isomorphism by the construction of π', since $m'(\pi'(u)) = u$ for all $u \in V_{T_1}$, and with the application of π', the two trees are congruent, hence congruency of the labels implies isomorphism of the two trees. This concludes the proof that m' is an isomorphism thus leading to a contraction. ◀

Now the task is to show how we can efficiently find an isomorphic mapping $m \in \mathcal{I}(T_1, T_2)$ from tree T_1 to tree T_2 such that the mismatch number $\Delta(m)$ is minimized – in other words, we need to compute $d_\Delta(T_1, T_2)$. For tree T, let L_T be its set of leaves, and for vertex $u \in V_T$, let $T|u$ be the subtree of T *rooted* at u – the connected component of T' containing u, where T' is the tree obtained from T by removing the edge $(p_T(u), u)$, and $p_T(u)$ is the *parent* of u in T. If u and v are both vertices, we slightly abuse notation using $\Delta(u, v)$ to mean the mismatch distance of (the only possible mapping between) u and v, that is $\Delta(u, v) = 0$ if $\ell_{T_1}(u) = \ell_{T_2}(v)$ – that is, the vertex u of T_1 and the vertex v of T_2 have the same label – and $\Delta(u, v) = 1$ otherwise. Recall that the *children* of u in T is the set $c_T(u) = \{v : (u, v) \in E_{T|u}\}$ – let $\mathcal{C}_{u,v}$ be the set of bijective mappings $m : c_{T_1}(u) \to c_{T_2}(v)$ from the children of $u \in V_{T_1}$, $v \in V_{T_2}$ – clearly any isomorphic mapping in $\mathcal{I}(T_1, T_2)$ is such a mapping when restricted to $c_{T_1}(u)$ and $c_{T_2}(v)$. We define the following (recursive) relationship $D(u, v)$, showing later how we can use it to compute the mismatch distance:

$$D(u,v) = \begin{cases} \Delta(u,v) & \text{if } u \in L_{T_1},\ v \in L_{T_2}, \\ \min\limits_{m \in \mathcal{C}_{u,v}} \sum_{z \in c_{T_1}(u)} D(x, m(z)) + \Delta(u,v) & \text{if } T_1|u \cong T_2|v, u \notin L_{T_1}, v \notin L_{T_2}, \\ \infty & \text{otherwise.} \end{cases}$$

We first show how to compute $T_1|u \cong T_2|v$ (true or false) for all $u \in V_{T_1}$, $v \in V_{T_2}$, since we need it for computing $D(u,v)$. We can build this relationship by extending the time $O(n \log n)$ algorithm of [6], where $n = |V_{T_1}| = |V_{T_2}|$, for determining if two rooted trees are isomorphic.[1] The idea of this algorithm is to first organize each tree in *levels*, where the level of a vertex is its distance from the root – this can be done with a simple DFT of each tree. Suppose each tree has k levels, otherwise they do not have the same number of levels, hence they cannot be isomorphic. Starting from level k in both trees, we move up level by level towards the root in both trees simultaneously. At each level i we perform the following steps: (1) for each vertex u of each tree T on level i, we store a representation of the topology of $T|u$, computing it recursively from the representations stored in the children of u on level $i+1$; then (2) sort in each tree the vertices at level i by representation[2]; and finally (3) compare the two resulting sorted orders – only when they are identical, may we proceed to the next level. If we make it all the way to the first level (the root), and we succeed with the 3 steps at this level, then the two trees are isomorphic, otherwise not. We can extend this algorithm with a fourth step: (4) for each pair $u \in V_{T_1}$, $v \in V_{T_2}$ of vertices on level i, if their representations are identical, then $T_1|u \cong T_2|v$ is true, and false otherwise. Clearly, only for pairs of vertices on the same level, can their subtrees be isomorphic, and so this is an exhaustive search for all such pairs. Since the number of vertices compared in step (4) over all of the levels is no more than n^2, this computation of $T_1|u \cong T_2|v$ for all pairs $u \in V_{T_1}$, $v \in V_{T_2}$ of vertices requires time $O(n^2)$. We are now ready to prove that $D(u,v) = d_\Delta(T_1|u, T_2|v)$ for each vertex u of T_1 and v of T_2.

▶ **Lemma 13.** *Let trees T_1 and T_2 each be labelled by $\mathcal{L}$. Then (1) $D(u,v) = d_\Delta(T_1|u, T_2|v)$ for all pairs $u \in V_{T_1}$, $v \in V_{T_2}$. Moreover, (2) $D(u,v)$ can be computed in time $O(|\mathcal{L}|^3)$.*

Proof. (1) It is essentially a proof by induction. If both u and v are leaves, then $T_1|u \cong T_2|v$ is trivially true and $D(u,v) = \Delta(u,v)$. When $T_1|u \cong T_2|v$ does not hold, then the mismatch distance is undefined.

Otherwise, if u and v are internal vertices, and $T_1|u \cong T_2|v$, then let m be a bijective mapping from the nodes of $T_1|u$ to the nodes of $T_2|v$ minimizing the mismatch distance. By the definition of d_Δ and the construction of m, $d_\Delta(T_1|u, T_2|v) = \sum_{z \in c_{T_1}(u)} d_\Delta(T_1|z, T_2|m(z)) + \Delta(u,v)$. By the inductive hypothesis $\sum_{z \in c_{T_1}(u)} d_\Delta(T_1|z, T_2|m(z)) = \sum_{z \in c_{T_1}(u)} D(z, m(z))$. Combining these two facts implies that $d_\Delta(T_1|u, T_2|v) = \sum_{z \in c_{T_1}(u)} D(z, m(z)) + \Delta(u,v)$. Since m is the bijective mapping minimizing the mismatch distance, no other mapping m' can achieve a smaller value of $\sum_{z \in c_{T_1}(u)} D(z, m(z)) + \Delta(u,v)$, hence $D(u,v) = d_\Delta(T_1|u, T_2|v)$.

(2) Computing $D(u,v)$ when u and v are leaves requires constant time. When u and v are internal nodes – assuming we have already computed $T_1|u \cong T_2|v$ – we compute a *minimum weight* matching in the weighted bipartite graph with $V = c_{T_1}(u) \cup c_{T_2}(v)$ and $E = \{(x,y) \ :\ T_1|x \cong T_2|y,\ x \in c_{T_1}(u),\ y \in c_{T_2}(v)\}$ with weight function $w : E \to \mathbb{N}$ such

[1] Note that there is a linear time algorithm in [1], but it assumes that $\log n$ is fixed, where n is the number of vertices – which is the likely the case for all practical instances.

[2] We assume that there is a total ordering on the representations of the topologies, they are of constant size, and can be compared in constant time – for details see [6, 1]

that $w(x,y) = D(x,y)$. Such a matching can be found in time $O(|V|\log|V| + |V||E|)$ using a Fibonacci heap [15]. If we sum over all of graphs in which we compute these matchings during the recursive computation of $D(u,v)$ for all $u \in V_{T_1}$, $v \in V_{T_2}$, the number of vertices and edges in each graph, these sums will be at most n (vertices) and n^2 (edges) respectively, since both trees have $n = |V_{T_1}| = |V_{T_2}| = |\mathcal{L}|$ vertices. This means that this matching procedure will take overall time $O(|\mathcal{L}|^3)$. Computing $T_1|u \cong T_2|v$ for each pair $u \in V_{T_1}$, $v \in V_{T_2}$ takes time $O(|\mathcal{L}|^2)$ overall, hence computing computing $D(u,v)$ for all $u \in V_{T_1}$, $v \in V_{T_2}$ has a total running time $O(|\mathcal{L}|^3)$. ◀

Lemma 11 then follows from Lemmas 12 and 13. Notice that, when computing $D(u,v)$ we can also maintain the set $C(u,v)$ of the labels that are *conserved* in the minimum weight matchings, that is, those that are not involved in a mismatch. More precisely, $C(u,v)$ is equal to the union of the $C(x,y)$ over all edges (x,y) of the optimal matching. To that set, we add the label $\ell_{T_1}(u)$ if $\Delta(u,v) = 0$. Once we have all sets $C(u,v)$, the permutation that we want to compute involves exactly the labels not in $C(\lambda_{T_1}, \lambda_{T_2})$. More precisely, the label $\ell_{T_1}(u)$ must be replaced with the label of the vertex $m(u)$, where the isomorphism m can be constructed from the perfect matchings of the optimal solution.

3.3 Rearrangement distance

Finally, we show that deciding the rearrangement distance between two trees is NP-hard. We show this by reduction from 3-dimensional matching, one of Karp's 21 NP-complete problems [21].

In 3-dimensional matching, we are given three disjoint sets A, B and C, along with a set $\mathcal{T}$ of triples (a,b,c), such that $a \in A$, $b \in B$ and $c \in C$ – essentially a 3-uniform hypergraph H. A *matching* is then a subset $\mathcal{M} \subseteq \mathcal{T}$ such that for every two triples $(a,b,c) \in \mathcal{M}$, $(a',b',c') \in \mathcal{M}$, it follows that $a \neq a'$, $b \neq b'$ and $c \neq c'$, that is, all triples of $\mathcal{M}$ are pairwise disjoint. It is then NP-hard to decide for a given k if there is a matching $\mathcal{M}$ of size k [21]. It has been proved that the problem remains NP-hard even in the case of 3-bounded 1-common 3-dimensional matching, which is a restriction of the problem where the number of occurrences of an element in the triples is at most 3, and each pair of triples has at most one element in common [19]. We also make use of the following structure in this proof.

▶ **Definition 14** (movements graph). *Given trees T_1 and T_2 each labelled by $\mathcal{L}$, the* movements graph *G has an edge for every element $\mathcal{P}_{(u,w)}$ of the family partition $\mathcal{P}$ of T_1 and T_2, that is, $E_G = \{(u,w) \ : \ \mathcal{P}_{(u,w)} \in \mathcal{P}\}$, while its vertex set is $V_G = \bigcup_{(u,w)\in E_G}\{u,w\}$.*

We now prove that computing the rearrangement distance is NP-hard.

▶ **Theorem 15.** *Given trees T_1 and T_2 each labelled by $\mathcal{L}$, and some integer k, it is NP-hard to decide if $d(T_1,T_2) \leq k$.*

Proof. Reduction from 3-bounded 1-common 3-dimensional matching. We are given an instance H of 3-dimensional matching consisting of a set $\mathcal{T}$ of m triples (a,b,c) over the disjoint sets A, B, C. We construct two trees T_1^H and T_2^H each with $|A|+|B|+|C|+6m+2$ vertices, for which the rearrangement distance $d(T_1^H, T_2^H) \leq 3n + 6(m-n)$ if and only if H has a 3-dimensional matching of size n.

Consider such an instance H of 3-dimensional matching as above. In the construction, the trees T_1^H and T_2^H each have a root vertex r, and a vertex for every element of A, B and C – each of which have r as the parent. To each $v \in \{a,b,c\}$ in T_1^H and triple t, we add a

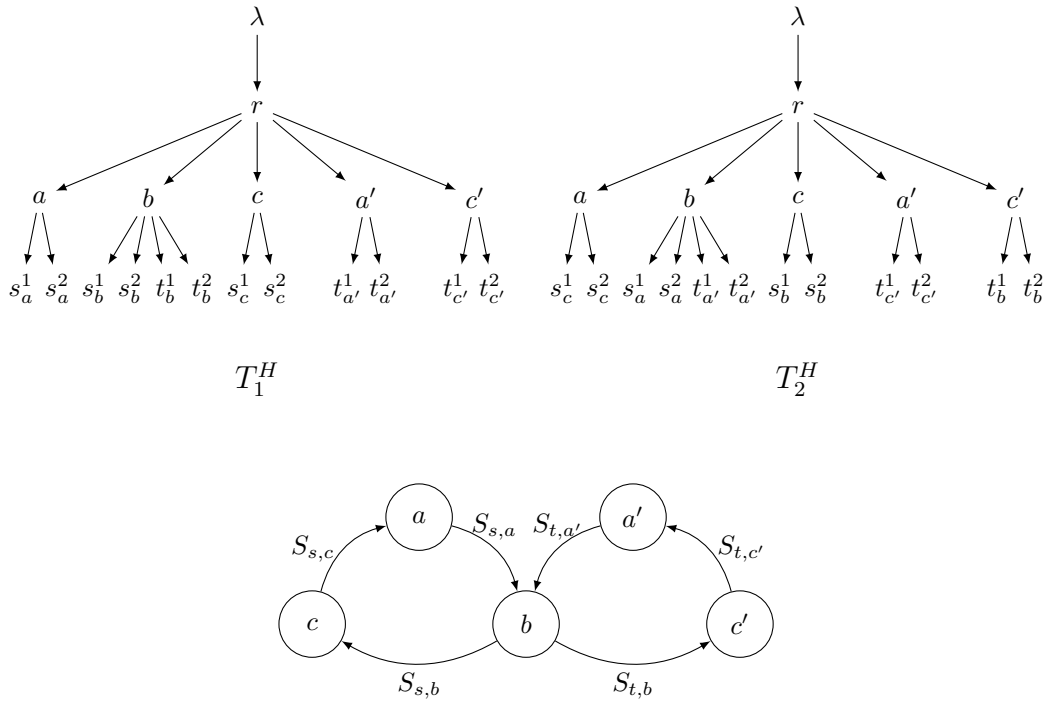


Figure 2 The trees T_1^H and T_2^H given instance H of 3-dimensional matching with $A = \{a, a'\}$, $B = \{b\}$ and $C = \{c, c'\}$ and $\mathcal{T} = \{s = \{a, b, c\},\ t = \{a', b, c'\}\}$ (top), and the corresponding movements graph for the trees T_1^H and T_2^H (bottom).

set $S_{t,v} = \{t_v^1, t_v^2\}$ of two (uniquely labelled) children. In T_2^H, we add the sets $S_{t,v}$ of two children to each of these three vertices, but cyclically shifted, with respect to T_1^H, *i.e.*, we add $S_{t,a}$ to b, $S_{t,b}$ to c, $S_{t,c}$ back to a again. Note that this induces in the movements graph G a cycle $\mathcal{C}_t = \{(a, b), (b, c), (c, a)\}$ – see Figure 2. Now, observe that the movements graph G will have cycles of length 3 corresponding to each triple and two cycles may share one common vertex v if the triples share element v. By Lemma 1, a sequence of operations of total size $d(T_1^H, T_2^H)$ will consist of a permutation followed by a sequence of link-and-cut operations. Observe that any permutation involves for each cycle $\mathcal{C}_t$ an edge, two edges or all three edges. Now, the rearrangement distance aims to solve cycles in the movements graph in the sense that after the operations, the movements graph has no edges. Observe that given a cycle $\mathcal{C}_t$ of the movements graph G, then the minimum cost rearrangement to solve $\mathcal{C}_t$ consists of applying a permutation of size 3 involving the three vertices of the cycle, thus of total cost 3. Observe that two cycles sharing a common vertex cannot both be solved by a permutation that is a cyclic shift of the vertices of the cycle, that is they cannot be both solved with cost 3. Moreover, permutations of vertices cannot solve more than one vertex of a cycle $\mathcal{C}_t$ if it is not a cyclic shift of the vertices of $\mathcal{C}_t$, as cycles do not share edges. In case of a cyclic shift of two vertices of $\mathcal{C}_t$, (1) a permutation of size 2 and then four link-and-cut operations are required. If instead (2) at most a single vertex of $\mathcal{C}_t$ is involved in a permutation, then six link-and-cut operations are required. We now detail how this implies that $d(T_1^H, T_2^H) \leq 3n + 6(m - n)$ if and only if H has a 3-dimensional matching of size n.

($\Rightarrow$) Assume that $d(T_1^H, T_2^H) \leq 3n + 6(m - n)$. By the above observation on how cycles of the movements graph G are solved by the sequence of permutations, cases (1) and (2) for solving cycles have the same cost equal to 6. Thus the only possible way to have a

rearrangement distance less than or equal to $3n + 6(m - n)$ is by taking n disjoint cycles solved by the permutation operation of cost 3. This implies a 3-dimensional matching of size n.

($\Leftarrow$) Now, suppose that H has a 3-dimensional matching $\mathcal{M} \subseteq \mathcal{T}$ of size n. This implies that there are n triples that are disjoint and thus the movements graph G has n disjoint cycles. By solving each cycle with a permutation of size 3, $m - n$ cycles are left in the movements graph. The remaining cycles, in the worst case, share common vertices with the cycles solved by the permutations, and thus they can be solved with a cost that is 6 in the worst case. Thus we obtain that $d(T_1^H, T_2^H) \leq 3n + 6(m - n)$, completing the proof. ◀

4 Bounds and approximation

We first give the following important lemma which states that when we apply a permutation to the labels of T_1 obtaining T_1', the size of the resulting family partition $\mathcal{P}'$ cannot increase or decrease too much with respect to the size of $\mathcal{P}$.

▶ **Lemma 16.** *Given trees T_1 and T_2 with corresponding active set $\mathcal{X}$ and family partition $\mathcal{P}$, if T_1' is the tree (isomorphic to T_1) resulting from the application of permutation π of the labels of T_1, and $\mathcal{X}'$ and $\mathcal{P}'$ are the active set and the family partition of T_1' and T_2, respectively, then $|\mathcal{P}| - 2|\pi| \leq |\mathcal{P}'| \leq |\mathcal{P}| + 2|\pi|$.*

Proof. Let a be some label of T_1 which has been perturbed by permutation π, *i.e.*, $\pi(a) = b \neq a$. The new family partition $\mathcal{P}'$ is obtained from $\mathcal{P}$ by means of deletions, insertions and substitutions of subsets. The crucial observation is that such an operation will only affect the *neighborhood* of a, namely the (possibly empty) set of its children $c_{T_1}(a)$ and its parent $p_{T_1}(a)$ in T_1. Let us consider each child $v \in c_{T_1}(a)$ first. We have the following cases.

- ($v \in \mathcal{P}_{(a,b)} \subseteq \mathcal{X}$): since π makes b the parent of v, which is exactly the parent of v in T_1, $v \notin \mathcal{X}'$ and $\mathcal{P}_{(a,b)}$ will be missing from $\mathcal{P}'$;
- ($v \in \mathcal{P}_{(a,c)} \subseteq \mathcal{X}$, $c \neq b$): after applying π, v will belong to set $\mathcal{P}'_{(b,c)}$, thus $\mathcal{P}_{(a,c)}$ will be replaced by $\mathcal{P}'_{(b,c)}$ in $\mathcal{P}'$;
- ($v \notin \mathcal{X}$): then $v \in \mathcal{P}'_{(b,a)}$ in $\mathcal{P}'$, thus $\mathcal{P}'$ might have an extra element with respect to $\mathcal{P}$.

Consider now the possible effects of π on $p_{T_1}(a)$. There are two possible scenarios:

- ($b \in \mathcal{P}_{(p_{T_1}(b), p_{T_2}(b))} \subseteq \mathcal{X}$): if $p_{T_2}(b) = p_{T_1}(a)$, then $b \notin \mathcal{X}'$ and if b was the only element of $\mathcal{P}_{(p_{T_1}(b), p_{T_2}(b))}$, the latter will be missing from $\mathcal{P}'$; else, $b \in \mathcal{P}'_{(p_{T_1}(a), p_{T_2}(b))}$, thus $\mathcal{P}_{(p_1(b), p_2(b))}$ will be replaced by $\mathcal{P}'_{(p_{T_1}(a), p_{T_2}(b))}$ in $\mathcal{P}'$;
- ($b \notin \mathcal{X}$): then $b \in \mathcal{P}'_{(p_{T_1}(a), p_{T_2}(b))}$ in $\mathcal{P}'$, thus $\mathcal{P}'$ might have an extra element with respect to $\mathcal{P}$.

In summary, $\mathcal{P}'$ is obtained from $\mathcal{P}$ with up to two deletions and two additions of sets in the family partition for each label involved in the permutation π, thus the result follows. ◀

In the special case where one of the trees, *e.g.*, T_1, is *binary*, *i.e.*, each node has up to two children, we have the following lemma connecting link-and-cut and rearrangement distance.

▶ **Lemma 17.** *Given T_1 a binary tree, T_2 any tree, we have that $d_\ell(T_1, T_2) \leq 4 \cdot d(T_1, T_2)$.*

Proof. Suppose that T_2 is optimally obtained from T_1 by applying a permutation π of the labels followed by a number of link-and-cut operations – something we can assume in virtue of Lemma 1. Let T_1' be the tree resulting from the application of permutation π of the labels of T_1, $\mathcal{X}'$ and $\mathcal{P}'$ the active set and family partition of T_1' and T_2, respectively. By the construction of the family partition, the optimal number of link-and-cut operations to obtain

T_2 from T_1' is at least $|\mathcal{P}'|$; we thus have that $d(T_1, T_2) = |\pi| + |\mathcal{X}'| \geq |\pi| + |\mathcal{P}'|$. Moreover, Lemma 16 says that $|\pi| \geq \frac{|\mathcal{P}|-|\mathcal{P}'|}{2}$, thus $d(T_1, T_2) \geq \frac{|\mathcal{P}|-|\mathcal{P}'|}{2} + |\mathcal{P}'| = \frac{|\mathcal{P}|}{2} + \frac{|\mathcal{P}'|}{2} \geq \frac{|\mathcal{P}|}{2}$. Now, since T_1 is binary, each set in the family partition $\mathcal{P}$ consists of up to two elements (the elements of $\mathcal{P}_{(x,y)}$ are the ones among the children of x in T_1 that becomes the children of y in T_2, thus they cannot be more than the number of children of x). It follows that $|\mathcal{X}| = d_\ell(T_1, T_2) \leq 2 \cdot |\mathcal{P}|$, hence $d(T_1, T_2) \geq \frac{d_\ell(T_1,T_2)}{4}$. ◀

Importantly, we note that Lemma 17 states that the link-and-cut distance algorithm provides a linear 4-approximation for the rearrangement distance when at least one of the trees involved is binary. We have the following corollary from Lemma 17 and Lemma 10.

▶ **Corollary 18.** *There exists a polynomial time 4-approximation algorithm for the rearrangement distance problem for binary trees.*

5 Fixed parameter tractability

This section is devoted to showing that computing the rearrangement distance between trees T_1 and T_2 is fixed-parameter tractable, essentially via the bounded search tree technique [10]. In this case, the instance also contains a parameter k: in time $O((4k)^{2k^2}n)$ we (1) determine if $d(T_1, T_2) \leq k$ and, if this is the case, (2) find the minimum sequence of operations transforming T_1 into T_2.

The main idea of our algorithm is that, in virtue of Lemma 1, we can reorder the sequence of operations that transforms T_1 into T_2 so that all permutations precede the link-and-cut operations. Let T^* be the tree obtained from T_1 using only permutations and such that we can optimally obtain T_2 from T^* using only link-and-cut operations. Then $d(T_1, T_2) = d_\pi(T_1, T^*) + d_\ell(T^*, T_2)$. Our algorithm consists of showing that $d_\pi(T_1, T^*)$ is related to the size of the family partition, and that we can compute $d_\ell(T^*, T_2)$ in linear time.

The main consequence of Lemma 1 here is that we can restrict our attention to permutations first (to obtain a tree T^*), and to link-and-cut operations afterwards. Finding such a tree T^* is easier when we want to determine if the rearrangement distance $d(T_1, T_2)$ is at most k.

In fact, a consequence of Lemma 16 is that $d(T_1, T_2) \geq d_\pi(T_1, T^*) \geq |\mathcal{P}|/2$, where $\mathcal{P}$ is the family partition associated with T_1 and T_2. Notice that any sequence of operations that transforms T_1 into T_2 also transforms $\mathcal{X}$ into the empty set – thus $\mathcal{P}$ into the empty partition.

Since $d(T_1, T_2) \geq |\mathcal{P}|/2$, the first step of our algorithm is to compute the family partition $\mathcal{P}$ of T_1 and T_2 and verify that $k \geq |\mathcal{P}|/2$. If that inequality is not satisfied, then, since as stated above $d_\pi(T_1, T^*) \geq |\mathcal{P}|/2$, it would follow that $d(T_1, T_2) > k$. Hence we can focus on the instances where $k \geq |\mathcal{P}|/2$, that is $|\mathcal{P}| \leq 2k$. Since the family partition is sufficiently small, we can compute all sequences of permutations of at most k labels of $\mathcal{X}$ in time $O((4k)^{2k^2})$. In fact, each of the permutations involves one of the 2^{2k} subsets of vertices of $\mathcal{X}$, and there can be at most $(2k)!$ permutations of a set of $2k$ elements. Overall there are at most $\left(2^{2k}(2k)!\right)^k$ such sequences: it is trivial to organize them in a search tree that can be generated and traversed in linear time, and some crude upper bound results in the desired time bound. Let $\mathcal{T}$ be the set of trees that are obtained by applying to T_1 the sequence of operations corresponding to a node of the search tree.

The second part of our algorithm is to compute $d_\ell(T, T_2)$ for each tree in $T \in \mathcal{T}$, which, by Lemma 10, requires $O(n)$ time for each tree, keeping track of the tree T^* minimizing $d_\pi(T_1, T^*) + d_\ell(T^*, T_2)$. The algorithm has therefore $O((4k)^{2k^2}n)$ time complexity.

6 Open problems

In this paper we provide a NP-hardness proof of the rearrangement distance for trees with vertices of unbounded degree. The computational complexity of the rearrangement distance in the case of bounded degree trees remains an open problem. Mainly it would be of theoretical interest to see if it is still NP-hard for binary trees. On the other hand, we provide a constant approximation algorithm for the rearrangement distance of binary trees. Extending this result to general trees is still open. Such a result could be of interest in developing practical algorithms for comparing tumor phylogenies.

References

1. A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*, pages 84–85. Addison-Wesley Publishing Co., 1974.
2. B.L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5:1–13, 2001.
3. P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1):217–239, 2005.
4. P. Bonizzoni, S. Ciccolella, G. Della Vedova, and M. Soto. Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 1–10. ACM, 2017.
5. M. Bordewich and C. Semple. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combintorics*, 8:409–423, 2005.
6. D.M. Campbell and D. Radford. Tree isomorphism algorithms: Speed vs. clarity. *Mathematics Magazine*, 64(4):252–261, 1991.
7. B. Chor and T. Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21(1):i97–i106, 2005.
8. S. Ciccolella, M. Soto Gomez, M. Patterson, G. Della Vedova, I. Hajirasouliha, and P. Bonizzoni. gpps: an ILP-based approach for inferring cancer progression with mutation losses from single cell data. In *8th IEEE International Conference on Computational Advances in Bio and Medical Sciences, ICCABS 2018, Las Vegas, NV, USA, October 18-20, 2018*, page 1. IEEE Computer Society, 2018.
9. B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang. On distances between phylogenetic trees. In *The 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 427–436, 1997.
10. R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Springer-Verlag, New York, USA, 1999.
11. M. El-Kebir, L. Oesper, H. Acheson-Field, and B.J. Raphael. Reconstruction of clonal trees and tumor composition from multi-sample sequencing data. *Bioinformatics*, 31(12):i62–i70, 2015.
12. J.S. Farris. Methods for computing Wagner trees. *Systematic Zoology*, 19:83–92, 1970.
13. J. Felsenstein. *Inferring Phylogenies*, volume 2. Sinauer Associates, 2004.
14. W.M. Fitch. Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, 20(4):406–416, 1971.
15. M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*, 34(3):596–615, July 1987.
16. K. Govek, C. Sikes, and L. Oesper. A Consensus Approach to Infer Tumor Evolutionary Histories. In *Proceedings of the 9th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB)*, pages 63–72, 2018.

17 I. Hajirasouliha, A. Mahmoody, and B.J. Raphael. A combinatorial approach for analyzing intra-tumor heterogeneity from high-throughput sequencing data. *Bioinformatics*, 30(12):i78–i86, June 2014.

18 K. Jahn, J. Kuipers, and N. Beerenwinkel. Tree inference for single-cell data. *Genome Biology*, 17(1):86, May 2016.

19 H. Jiang, J. Ma, J. Luan, and D. Zhu. Approximation and Nonapproximability for the One-Sided Scaffold Filling Problem. In *International Computing and Combinatorics Conference (COCOON 2015)*, pages 251–263, 2015.

20 W Jiao, S. Vembu, A.G. Deshwar, L. Stein, and Q. Morris. Inferring clonal evolution of tumors from single nucleotide somatic mutations. *BMC Bioinformatics*, 15(1):35, 2014.

21 R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

22 N. Karpov, S. Malikic, M.K. Rahman, and S.C. Sahinalp. A Multi-labeled Tree Edit Distance for Comparing "Clonal Trees" of Tumor Progression. In *18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland*, pages 22:1–22:19, 2018.

23 M.K. Kuhner and J. Yamato. Practical Performance of Tree Comparison Metrics. *Systematic Biology*, 64(2):205–214, December 2014.

24 J. Kuipers, K. Jahn, B.J. Raphael, and N. Beerenwinkel. Single-cell sequencing data reveal widespread recurrence and loss of mutational hits in the life histories of tumors. *Genome research*, 27(11):1885–1894, 2017.

25 M. McVicar, B. Sach, C. Mesnage, J. Lijffijt, E. Spyropoulou, and T. De Bie. SuMoTED: An intuitive edit distance between rooted unordered uniquely-labelled trees. *Pattern Recognition Letters*, 79:52–59, 2016.

26 A.S. Morrissy and L. Garzia et al. Divergent clonal selection dominates medulloblastoma at recurrence. *Nature*, 529:351 EP, January 2016.

27 P.C. Nowell. The clonal evolution of tumor cell populations. *Science*, 194:23–28, 1976.

28 M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems (TODS)*, 40(1), 2015.

29 V. Popic, R. Salari, I. Hajirasouliha, D. Kashef-Haghighi, R.B. West, and S. Batzoglou. Fast and scalable inference of multi-sample cancer lineages. *Genome Biology*, 16(1):91, 2015.

30 D.F. Robinson and L.R. Foulds. Comparison of weighted labeled trees. *Lecture Notes in Mathematics*, 748:119–126, 1979.

31 E.M. Ross and F. Markowetz. OncoNEM: inferring tumor evolution from single-cell sequencing data. *Genome Biology*, 17(1):69, April 2016.

32 S. Salehi, A. Steif, A. Roth, S. Aparicio, A. Bouchard-Côté, and S.P. Shah. ddClone: joint statistical inference of clonal populations from single cell and bulk tumour sequencing data. *Genome Biology*, 18(1):44, March 2017.

33 C. Semple and M. Steel. *Phylogenetics*. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press, USA, 2003.

34 D.D. Sleator and R.E. Tarjan. A Data Structure for Dynamic Trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, New York, NY, USA, 1981. ACM.

35 M. A. Steel. *Phylogeny: discrete and random processes in evolution*. Number 89 in CBMS-NSF regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 2016.

36 K.C. Tai. The Tree-to-Tree Correction Problem. *J. ACM*, 26(3):422–433, July 1979.

37 J. Wang, E. Cazzato, E. Ladewig, V. Frattini, D.I.S. Rosenbloom, S. Zairis, F. Abate, Z. Liu, O. Elliott, Y. Shi n, J. Lee, I. Lee, W. Park, M. Eoli, A.J. Blumberg, A. Lasorella, D. Nam, G. Finocchiaro, A. Iavarone, and R. Rabadan. Clonal evolution of glioblastoma under therapy. *Nature Genetics*, 48:768 EP, June 2016.

38 K. Yuan, T. Sakoparnig, F. Markowetz, and N. Beerenwinkel. BitPhylogeny: a probabilistic framework for reconstructing intra-tumor phylogenies. *Genome Biology*, 16(1):36, February 2015.

39 K.Z. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

On the Size of Overlapping Lempel-Ziv and Lyndon Factorizations

Yuki Urabe
Department of Informatics, 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

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

Lempel-Ziv (LZ) factorization and Lyndon factorization are well-known factorizations of strings. Recently, Kärkkäinen et al. studied the relation between the sizes of the two factorizations, and showed that the size of the Lyndon factorization is always smaller than twice the size of the non-overlapping LZ factorization [STACS 2017]. In this paper, we consider a similar problem for the overlapping version of the LZ factorization. Since the size of the overlapping LZ factorization is always smaller than the size of the non-overlapping LZ factorization and, in fact, can even be an $O(\log n)$ factor smaller, it is not immediately clear whether a similar bound as in previous work would hold. Nevertheless, in this paper, we prove that the size of the Lyndon factorization is always smaller than four times the size of the overlapping LZ factorization.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Lyndon factorization, Lyndon words, Lempel-Ziv factorization

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.29

Funding *Yuto Nakashima*: Supported by JSPS KAKENHI Grant Number JP18K18002.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.
Hideo Bannai: Supported by JSPS KAKENHI Grant Number JP16H02783.
Masayuki Takeda: Supported by JSPS KAKENHI Grant Number JP18H04098.

1 Introduction

A *factorization* of a string w is a sequence of non-empty substrings of w such that the concatenation of the substrings in the sequence is w. Various types of factorizations of strings have been proposed so far, and most, if not all, of them are categorized into two (not necessarily disjoint) categories. One is to factorize a given string w into *combinatorial objects* such as squares (square factorization [9, 18]), repetitions (repetition factorization [14]), palindromes (palindromic factorization [13, 10, 4, 2]), closed words (closed factorization [1]), and Lyndon words (Lyndon factorization [6]), while the other is to factorize a given string w as *efficient preprocessing* for text processing, in particular, text compression [21, 22, 20].

Amongst the variety of string factorizations, the Lyndon factorization [6] and the Lempel-Ziv (LZ for short) factorization [21] are probably those that are most well-known and extensively studied from the above categories, respectively, and this paper also deals with these factorizations.

As will be seen below, the definitions of LZ and Lyndon factorizations are rather different, and hence the results of these factorizations of the same string can also be very different. On the other hand, quite interestingly, both LZ and Lyndon factorizations have been used as efficient preprocessing for linear-time computation of *runs* or *maximal repetitions* in a given string [16, 5, 7, 3, 17, 11, 8]. Another connection between LZ and Lyndon factorizations is that both of the sizes of the LZ and Lyndon factorizations of a string w are lower bounds of the output size of any grammar compression for w [19, 12]. Here, by the size of a factorization we mean the number of factors in the factorization. Now, a natural question would be: How much the sizes of the LZ and Lyndon factorizations of the same string can differ?

This question was first considered by Kärkkäinen et al. [15] for the non-overlapping variant of LZ factorization. The non-overlapping LZ factorization of a string w is a sequence $p_1, \ldots, p_{z_{no}}$ of z_{no} factors such that each p_i is a single character if it is the first occurrence of the character in w, or p_i is the longest prefix of $p_i \cdots p_{z_{no}}$ that has an occurrence in $p_1 \cdots p_{i-1}$. A string ℓ is said to be a *Lyndon word*, if ℓ is lexicographically smaller than all of its non-empty proper suffixes. A factorization $f_1^{e_1}, \ldots, f_m^{e_m}$ is said to be the Lyndon factorization of a string w if f_i is a Lyndon word, $e_i \geq 1$, and f_i is lexicographically larger than f_{i+1} for all i. For many strings, the size m of Lyndon factorization is smaller than the size z_{no} of non-overlapping LZ factorization. However, they showed that there is a series of strings for which $m = z_{no} + \Theta(\sqrt{z_{no}})$ holds. In addition, they proved that the inequality $m < 2z_{no}$ holds for any string.

In this paper, we consider the relationship between the size of *overlapping variant of LZ factorization* and Lyndon factorization of the same string. The non-overlapping LZ factorization of a string w is a sequence $q_1, \ldots, q_z$ of z factors such that each q_i is a single character if it is the first occurrence of the character in w, or q_i is the longest prefix of $q_i \cdots q_{z_{no}}$ that has another occurrence in w beginning at a position within $q_1 \cdots q_{i-1}$. It is known that $z \leq z_{no}$ always holds, and there are cases where z is by a factor of $O(\log n)$ smaller than z_{no}: E.g., for a trivial string a^n, $z = 2$ while $z_{no} = \Theta(\log n)$. These facts make it more challenging to show an upper bound for m in terms of z. Still, in this paper, we prove that the inequality $m < 4z$ holds for any string. Our proof generally follows the scheme introduced by Kärkkäinen et al. [15], but our analysis leading to the inequality $m < 4z$ is original and seems to be interesting.

2 Preliminaries

2.1 Strings

Let Σ be an ordered *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0. Let Σ^+ 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. 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 $lcp(x, y)$ 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.

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 *Lyndon factorization* of a string w is the factorization $f_1^{e_1}, \ldots, f_m^{e_m}$ of w, such that each $f_i \in \Sigma^+$ is a Lyndon word, $e_i \geq 1$, and $f_i \succ f_{i+1}$ for all $1 \leq i < m$. We call m the size of the Lyndon factorization of w. We also refer to each f_i as a Lyndon factor and each $F_i = f_i^{e_i}$ as a Lyndon run of w.

2.3 Lempel-Ziv factorization of strings

The *overlapping Lempel-Ziv factorization* (LZ factorization for short) of a string w is the factorization $p_1, \ldots, p_z$ of w such that either p_i is a character which does not appear in $p_1 \cdots p_{i-1}$ or p_i is the longest prefix of $p_i \cdots p_z$ which has another occurrence to the left. We refer to each p_i as an *LZ phrase*. For any substring $w[i..j]$ $(1 \leq i \leq j \leq |w|)$ in w, $w[i..j]$ is said to contain an *LZ phrase boundary* if there exists an LZ phrase which begins in $[i, j]$.

3 Tools for non-overlapping LZ factorization

In this paper, we give the following result.

▶ **Theorem 1.** *Let m be the size of the Lyndon factorization of a string w and z the size of the (overlapping) LZ factorization of w. For any string w, $m < 4z$ holds.*

We prove Theorem 1 in Section 4. Our proof follows similar techniques for non-overlapping version which was introduced by Kärkkäinen et al. [15]. In this section, we explain their techniques which can be also applied for overlapping version.

3.1 Leftmost occurrence and factorizations

Each factorization catches the leftmost occurrences of particular substrings. Lemma 2 can be easily obtained by the definition of LZ factorization.

▶ **Lemma 2.** *If a substring $w[i..j]$ does not have any occurrence to the left, $w[i..j]$ contains an LZ phrase boundary.*

▶ **Lemma 3** (Lemma 4 of [15]). *Let $d \geq 1$ and $1 \leq i \leq m - d + 1$, and assume that $F_i \cdots F_{i+d-1}$ has an occurrence to the left of the trivial one in w. Then:*
1. *The leftmost occurrence of $F_i \cdots F_{i+d-1}$ is a prefix of f_j for some $j < i$;*
2. *$F_i \cdots F_{i+d-1}$ is a prefix of every f_k with $j \leq k < i$.*

3.2 Domains

Due to Lemma 3, each concatenation of several Lyndon runs has a range such that every Lyndon run in the range has the concatenation as a prefix.

▶ **Definition 4** (Definition 5 of [15]). *Let $d \geq 1$ and $1 \leq i \leq m - d + 1$. d-*domain *of a Lyndon run F_i, denoted by $\mathsf{dom}_d(F_i)$, is the substring $F_j \cdots F_{i-1}$ where F_j is the Lyndon run starting at the same position as the leftmost occurrence of $F_i \cdots F_{i+d-1}$ in w. Note that*

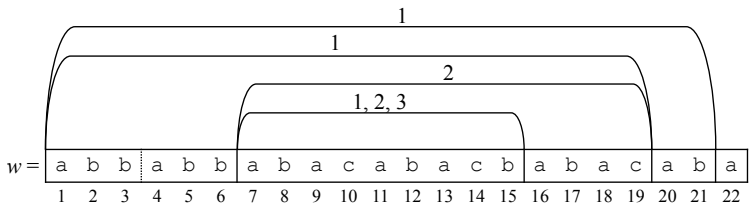


Figure 1 All non-empty domains in string $w = \texttt{abbabbabacabacbabacaba}$ are illustrated. Since the leftmost occurrence of $w[20..22]$ in w is $w[7..9]$, 2-domain of Lyndon run $w[20..21]$ is $w[7..19]$. Moreover, $w[7..9]$ is associated with this 2-domain. (This figure imitates Figure 1 of [15].)

if $F_i \cdots F_{i+d-1}$ *does not have any occurrence to the left of the trivial one then* $\mathsf{dom}_d(F_i) = \varepsilon$. *The integers* d *and* $i - j$ *are called the* order *and* size *of the domain, respectively. The* extended d-domain *of* F_i *is the substring* $\mathsf{extdom}_d(F_i) = \mathsf{dom}_d(F_i) \cdot F_i \cdots F_{i+d-1}$ *of* w.

By the definition and Lemma 2, each domain contains an LZ phrase boundary. For any domain $\mathsf{dom}_d(F_i)$, we say that the leftmost occurrence of $F_i \cdots F_{i+d-1}$ is *associated* with $\mathsf{dom}_d(F_i)$ (Definition 7 of [15]).

▶ **Lemma 5** (Lemma 8 of [15]). *Each substring associated with a domain contains an LZ phrase boundary.*

We show an example of domains in Figure 1.

3.3 Tandem domains

▶ **Definition 6** (Definition 9 of [15]). *Let* $d \geq 1$ *and* $1 \leq i \leq m - d$. *A pair of domains* $\mathsf{dom}_{d+1}(F_i), \mathsf{dom}_d(F_{i+1})$ *is called a* tandem domain *if* $\mathsf{dom}_{d+1}(F_i) \cdot F_i = \mathsf{dom}_d(F_{i+1})$ *or, equivalently, if* $\mathsf{extdom}_{d+1}(F_i) = \mathsf{extdom}_d(F_{i+1})$. *Note that we permit* $\mathsf{dom}_{d+1}(F_i) = \varepsilon$.

Let $\mathsf{dom}_{d+1}(F_i), \mathsf{dom}_d(F_{i+1})$ be a tandem domain. By Lemma 3, F_i can be written as $F_i = F_{i+1} \cdots F_{i+d} \cdot x$ for some $x \in \Sigma^+$. Thus, $F_i \cdots F_{i+d} = F_{i+1} \cdots F_{i+d} \cdot x \cdot F_{i+1} \cdots F_{i+d}$. We say that the occurrence of $x \cdot F_{i+1} \cdots F_{i+d}$ in the leftmost occurrence of $F_i \cdots F_{i+d}$ is *associated* with the tandem domain $\mathsf{dom}_{d+1}(F_i), \mathsf{dom}_d(F_{i+1})$ (Definition 10 of [15]).

In Figure 1, a pair of 3-domain of Lyndon run $w[16..19]$ and 2-domain of Lyndon run $w[20..21]$ is a tandem domain. Moreover, $w[10..13]$ is associated with the tandem domain.

3.4 Groups

▶ **Definition 7.** *Let* $d \geq 1, 2 \leq p \leq m$, *and* $1 \leq i \leq m - d - p + 2$. *A set of* p *domains* $\mathsf{dom}_{d+p-1}(F_i), \ldots, \mathsf{dom}_d(F_{i+p-1})$ *is called a* p-group *if for all* $t = 0, \ldots, p-2$ *the equality* $\mathsf{dom}_{d+p-1-t}(F_{i+t}), \mathsf{dom}_{d+p-2-t}(F_{i+t+1})$ *hols or, equivalently,* $\mathsf{extdom}_{d+p-1}(F_i) = \ldots = \mathsf{extdom}_d(F_{i+p-1})$. *Note that we permit* $\mathsf{dom}_{d+p-1}(F_i) = \varepsilon$.

Let $\mathsf{dom}_{d+p-1}(F_i), \ldots, \mathsf{dom}_d(F_{i+p-1})$ is a p-group. F_i has $F_{i+p-1} \cdots F_{i+p+d-2}$ as a prefix by Lemma 3. Then, $F_i \cdots F_{i+p+d-2} = F_{i+p-1} \cdots F_{i+p+d-2} \cdot x \cdot F_{i+1} \cdots F_{i+p+d-2}$ for some $x \in \Sigma^*$. We say that the occurrence of $x \cdot F_{i+1} \cdots F_{i+p+d-2}$ in the leftmost occurrence of $F_i \cdots F_{i+p+d-2}$ is *associated* with the group.

▶ **Lemma 8** (Lemma 16 of [15]). *The substring associated with a* p*-group is the concatenation, in reverse order, of the* $p - 1$ *substrings associated with the tandem domains belonging to the* p*-group.*

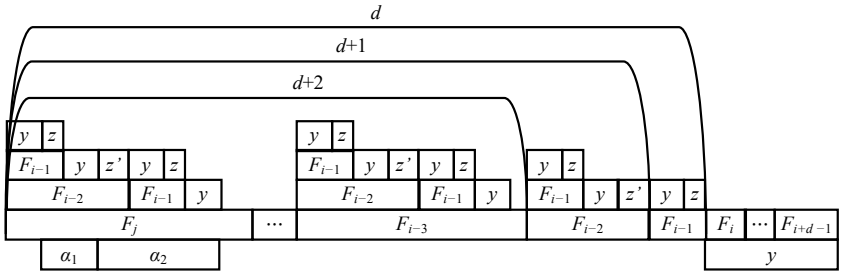


Figure 2 This figure illustrated 3-group $\mathsf{dom}_{d+2}(F_{i-2}), \mathsf{dom}_{d+1}(F_{i-1}), \mathsf{dom}_d(F_i)$. $\alpha_1 (= zy)$ is the substring associated with tandem domain $\mathsf{dom}_{d+1}(F_{i-1}), \mathsf{dom}_d(F_i)$, and $\alpha_2 (= z'yzy)$ is the substring associated with tandem domain $\mathsf{dom}_{d+2}(F_{i-2}), \mathsf{dom}_{d+1}(F_{i-1})$. Moreover, $\alpha_1\alpha_2$ is the substring associated with the 3-group. (This figure imitates Figure 2 of [15].)

Two groups $\mathsf{dom}_{d+p-1}(F_i), \ldots, \mathsf{dom}_d(F_{i+p-1})$ and $\mathsf{dom}_{d'+p'-1}(F_k), \ldots, \mathsf{dom}_{d'}(F_{k+p'-1})$ are said to be *disjoint* if $i+p-1 < k$ or $k+p'-1 < i$. For any disjoint groups, the following property holds.

▶ **Lemma 9** (Lemma 18 of [15]). *Substring associated with disjoint groups do not overlap.*

3.5 Subdomains

▶ **Definition 10** (Definition 19 of [15]). $\mathsf{dom}_e(F_k)$ *is said to be a* subdomain *of* $\mathsf{dom}_d(F_i) = F_j \cdots F_{i-1}$ *if either*

- $k = i$ *and* $e = d$, *or*
- $j \le k < i$ *and* $\mathsf{extdom}_e(F_k)$ *is a substring of* $\mathsf{extdom}_d(F_i)$.

▶ **Lemma 11** (Lemma 20 of [15]). *Let* $\mathsf{dom}_e(F_{k+1}), \mathsf{dom}_{e+1}(F_k)$ *be a tandem domain. If* $\mathsf{dom}_e(F_{k+1})$ *and* $\mathsf{dom}_{e+1}(F_k)$ *are both subdomains of a domain* $\mathsf{dom}_d(F_i)$, *then the substring associated with* $\mathsf{dom}_d(F_i)$ *does not overlap the substring associated with tandem domain* $\mathsf{dom}_e(F_{k+1}), \mathsf{dom}_{e+1}(F_k)$.

From this lemma, if every domain in a group is a subdomain of domain $\mathsf{dom}_d(F_i)$, the substring associated with $\mathsf{dom}_d(F_i)$ does not overlap the substring associated with the group.

3.6 Canonical subdomains

For any domain $\mathsf{dom}_d(F_i) = F_j \cdots F_{i-1}$, we define *canonical subdomain* $C_{i,d}$ as follows. $C_{i,d}$ is the set of subdomains of $\mathsf{dom}_d(F_i)$ which can be obtained by the following conditions. Initially, we set $\delta = d+1, l = i-1$. When $l = j$, then we finish the operations.

- If $\mathsf{dom}_\delta(F_l) = F_j \cdots F_{l-1}$, we add $\mathsf{dom}_\delta(F_l)$ into the set $C_{i,d}$, and set $\delta = \delta + 1, l = l - 1$.
- If $\mathsf{dom}_\delta(F_l) = F_{j'} \cdots F_{l-1}$ $(j < j')$, we add $\mathsf{dom}_\delta(F_l)$ into the set $C_{i,d}$, and set $\delta = 1, l = j' - 1$. All domains that were added to the set in this case are called *loose subdomains*.

We refer to each set of consecutive non-loose subdomains as a *cluster*. Note that the number of clusters is the number of loose subdomains plus one. Since $\mathsf{dom}_{d'}(F_j) = \varepsilon$, the domain w.r.t. F_j is always a cluster.

Let t be the number of loose subdomains in canonical sundomains $C_{i,d}$ of domain $\mathsf{dom}_d(F_i)$. We can discuss the number of LZ phrase boundaries contained in $\mathsf{extdom}_d(F_i)$. Let $\mathsf{dom}_{d_1}(F_{i_1}), \ldots, \mathsf{dom}_{d_t}(F_{i_t})$ $(i_1 < \ldots < i_t)$ be the sequence of loose subdomains, and

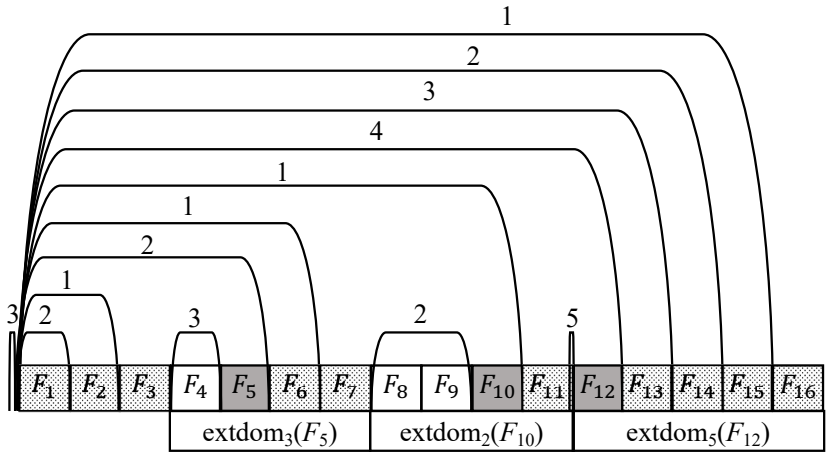


Figure 3 This figure illustrates the canonical subdomains of $\mathsf{dom}_1(F_{16}) = F_1 \cdots F_{15}$. (This figure imitates Figure 3 of [15].)

l (≥ 1) the number of Lyndon runs in the leftmost cluster. By the definition of loose subdomains, we have the following equality.

$$\mathsf{extdom}_d(F_i) = F_j \cdots F_{j+l-1} \cdot \mathsf{extdom}_{d_1}(F_{i_1}) \cdots \mathsf{extdom}_{d_t}(F_{i_t}) \tag{1}$$

Let S be the sum of the number of the LZ phrase boundaries contained in substrings associated with each clusters of $C_{i,d}$. By Lemma 9, these substrings do not overlap each other, and they are in $F_j \cdots F_{j+l-1}$. Moreover, they do not overlap the substring associated with $\mathsf{dom}_d(F_i)$ since they are also subdomains of $\mathsf{dom}_d(F_i)$ (by Lemma 11). Thus, by Lemma 5, there exists an LZ phrase boundary in $F_j \cdots F_{j+l-1}$ which was not counted in S. Let n_h be the number of LZ phrase boundaries which is contained in $\mathsf{extdom}_{d_h}(F_{i_h})$. It is clear that these boundaries are not in $F_j \cdots F_{j+l-1}$. Thus, they do not overlap the substring associated with the group and $\mathsf{dom}_d(F_i)$, respectively. Finally, we can discuss the number $N_{i,d}$ of LZ phrase boundaries in $\mathsf{extdom}_d(F_i)$ by using Equality (2):

$$N_{i,d} \geq 1 + \sum_{h=1}^{t} n_h + S. \tag{2}$$

4 Proof for overlapping LZ factorization

In this section, we prove Theorem 1. Our proof follows a general scheme introduced by Kärkkäinen et al. [15]. However, our analysis leading to the inequality $m < 4z$ is original and seems to be interesting.

4.1 Number of LZ phrase boundaries in groups

In the proof for non-overlapping version, Corollary 17 of [15] is one of the important properties. However, the corollary does not hold for overlapping version of LZ factorization. We want to introduce a new lemma as Lemma 13 for our problem. We start from the following lemma.

▶ **Lemma 12.** *Each substring associated with a 3-group contains an LZ phrase boundary.*

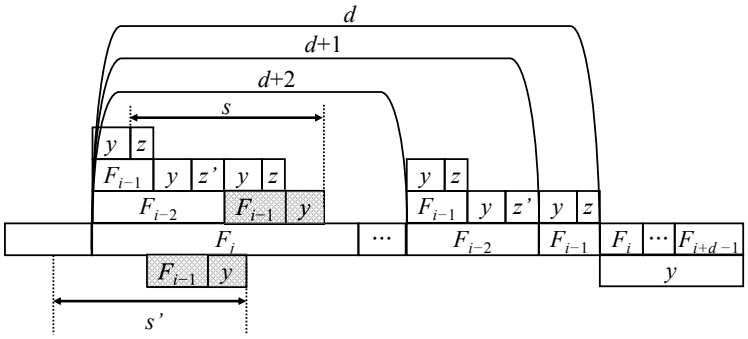


Figure 4 An illustration of the first case of proof for Lemma 12.

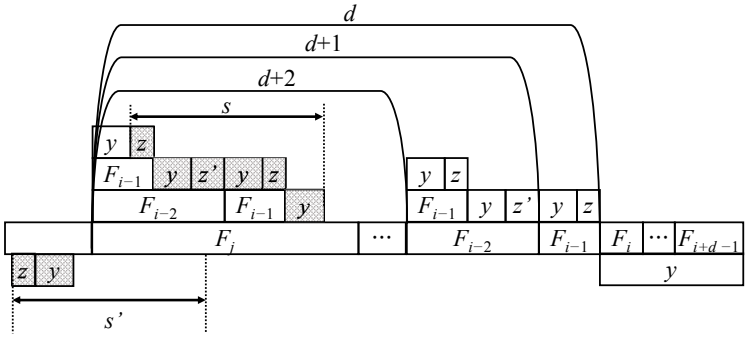


Figure 5 An illustration of the second case of proof for Lemma 12.

Proof. Let $\mathsf{dom}_{d+2}(F_{i-2}), \mathsf{dom}_{d+1}(F_{i-1}), \mathsf{dom}_d(F_i)$ be a 3-group. By the definition of groups, F_{i-1} can be written as $F_i \cdots F_{i+d-1} \cdot z$ for some $z \in \Sigma^+$, and F_{i-2} can be written as $F_{i-2} = F_{i-1} \cdots F_{i+d-1} \cdot z' = F_i \cdots F_{i+d-1} \cdot z \cdot F_i \cdots F_{i+d-1} \cdot z'$ for some $z' \in \Sigma^+$. For convenience, $y = F_i \cdots F_{i+d-1}$. Then, $F_{i-2} \cdots F_{i+d-1} = y \cdot z \cdot y \cdot z' \cdot F_{i-1} \cdot y$.

The substring associated with the 3-group is the suffix $z \cdot y \cdot z' \cdot F_{i-1} \cdot y$ of the leftmost occurrence of $F_{i-2} \cdots F_{i+d-1}$. s denotes the occurrence (see Figure 4). Suppose that $z \cdot y \cdot z' \cdot F_{i-1} \cdot y$ does not have any LZ phrase boundaries at the occurrence. By the definition of LZ factorization, $z \cdot y \cdot z' \cdot F_{i-1} \cdot y$ has an occurrence to the left. Let s' be one of such occurrences of $z \cdot y \cdot z' \cdot F_{i-1} \cdot y$. We consider the suffix $F_{i-1} \cdot y$ of s'. If a prefix of this suffix $F_{i-1} \cdot y$ overlaps a suffix of F_{i-2}(see Figure 4). This fact implies that f_{i-2} has a prefix of $F_{i-1} \cdot y$ as a suffix since $F_{i-2} = f_{i-2}^{e_{i-2}}$. On the other hand, f_{i-2} has $F_{i-1} \cdot y$ as a prefix by Lemma 3. Hence, f_{i-2} has a prefix of $F_{i-1} \cdot y$ as a prefix and also a suffix. This fact contradicts that f_{i-2} is a Lyndon word. Thus, the distance between s and s' has to be at least $|F_{i-1} \cdot y| + 1$. However, this fact also contradicts the leftmost occurrence of y (the leftmost occurrence of y is a prefix of F_j in fact, see also Figure 5). Therefore, every substring associated with a 3-group contains an LZ phrase boundary. ◀

By using this lemma, we can easily obtain the following key lemma.

▶ **Lemma 13.** *Each substring associated with a p-group contains at least $\lfloor \frac{p-1}{2} \rfloor$ LZ phrase boundaries.*

Proof. From Lemma 8, the substring associated with a p-group is the concatenation of $p-1$ substrings associated with tandem domains. The substring associated with 3-group contains an LZ phrase boundary by Lemma 12. Let x and y be the consecutive substrings which are associated with two consecutive tandem domains. Then, either x or y contains an LZ phrase boundary. Therefore, there exists at least $\lfloor \frac{p-1}{2} \rfloor$ LZ phrase boundaries. ◀

4.2 Number of LZ phrase boundaries in extended domains

▶ **Lemma 14.** *Let* $\mathsf{dom}_d(F_i)$ *be a domain of size* $k \geq 0$. $\mathsf{extdom}_d(F_i)$ *contains at least* $\lceil \frac{k-1}{4} \rceil + 1$ *LZ phrase boundaries (namely* $N_{i,d} \geq \lceil \frac{k-1}{4} \rceil + 1$*).*

Proof. Let $\mathsf{dom}_d(F_i) = F_j \cdots F_{i-1}$ be a domain of size $k = i - j$. We prove this lemma by induction on k. If $k = 0$, then the substring associated with $\mathsf{dom}_d(F_i)$ contains an LZ phrase boundary and the statement holds. Now we assume that $k \geq 1$ and the lemma holds for all $k \leq k'$ for some k'.

Firstly, we consider the case when $C_{i,d}$ does not have loose subdomain. In that case, $\mathsf{dom}_{d+k}(F_j), \ldots, \mathsf{dom}_d(F_i)$ is a $(k+1)$-group. By Lemma 5, the substring associated with $\mathsf{dom}_d(F_i)$ contains an LZ phrase boundary. On the other hand, by Lemma 13, the substring associated with the $(k+1)$-group contains $\lfloor \frac{k}{2} \rfloor$ LZ phrase boundaries. Since every domain in the group is a subdomain of $\mathsf{dom}_d(F_i)$, the substring associated with $\mathsf{dom}_d(F_i)$ does not overlap each of them by Lemma 11. Thus, $\mathsf{extdom}_d(F_i)$ contains $\lfloor \frac{k}{2} \rfloor + 1$ LZ phrase boundaries. The statement of the lemma holds for this case since $\lfloor \frac{k}{2} \rfloor + 1 \geq \lceil \frac{k-1}{4} \rceil + 1$.

Suppose that $C_{i,d}$ has t (≥ 1) loose subdomains. Let $\mathsf{dom}_{d_1}(F_{i_1}), \ldots, \mathsf{dom}_{d_t}(F_{i_t})$ be the t loose subdomains of $C_{i,d}$ and k_h the size of loose subdomain $\mathsf{dom}_{d_h}(F_{i_h})$ for any $1 \leq h \leq t$. We can see a lower bound of $N_{i,d}$ by using Equation (2). For the second term of Equation (2), $n_h \geq \lceil \frac{k_h - 1}{4} \rceil + 1$ holds by an induction hypothesis. Now we analyze the sum of k_h for all h. Let l be the number of domains in the leftmost cluster. Then,

$$\sum_{h=1}^{t} k_h = k - l - \sum_{h=1}^{t-1} d_h - (d_t - d) \tag{3}$$

holds. Next, we analyze the third term of Equation (2). Notice that S is the sum of the number of LZ phrase boundaries which are contained in substrings associated with each group that is a cluster in $C_{i,d}$. The leftmost cluster is a l-group, the rightmost cluster is a $(d_t - d)$-group, and each of other clusters is $(d_h - 1)$-group. For convenience, we consider 1-group as a single domain and 0-group as an empty set of domains. It is clear that substrings associated with each of them has no LZ phrase boundary. Thus, S can be written as

$$S = \left\lfloor \frac{l-1}{2} \right\rfloor + \sum_{h=1}^{t-1} \left\lfloor \frac{1}{2}(d_h - 1 - [d_h > 1]) \right\rfloor + \left\lfloor \frac{d_t - d - 1}{2} \right\rfloor \tag{4}$$

by using Knuth's notation [predicate] for the numerical value (0 or 1) of the predicate in brackets. We partition $(t-1)$ clusters (which are not the leftmost and the rightmost) into two sets as;

$$\begin{aligned} T_1 &= \{h \mid d_h \geq 3, h \in [1, t-1]\}, and \\ T_2 &= \{h \mid d_h < 3, h \in [1, t-1]\}. \end{aligned}$$

For any non-negative integer e, $\lfloor \frac{e}{2} \rfloor \geq \frac{e}{2} - \frac{1}{2}$ holds. By using this inequation, the second term in the right-hand side of Equation (4) can be written as

$$\begin{aligned}
& \sum_{h=1}^{t-1} \left\lfloor \frac{1}{2}(d_h - 1 - [d_h > 1]) \right\rfloor \\
= & \sum_{h \in T_1} \left\lfloor \frac{1}{2}(d_h - 1 - [d_h > 1]) \right\rfloor \geq \frac{1}{2} \sum_{h \in T_1} (d_h - 1 - [d_h > 1]) - \frac{|T_1|}{2} \\
= & \frac{1}{2} \sum_{h \in T_1} \left(\frac{d_h}{3} - [d_h > 1] \right) + \frac{1}{3} \sum_{h \in T_1} d_h - |T_1| \geq \frac{1}{3} \sum_{h \in T_1} d_h - |T_1|.
\end{aligned}$$

Thus, S can be also written as

$$S \geq \frac{1}{3} \sum_{h \in T_1} d_h - |T_1| + \alpha \left(\alpha = \left\lfloor \frac{l-1}{2} \right\rfloor + \left\lfloor \frac{d_t - d - 1}{2} \right\rfloor \right).$$

Moreover, Equation (2) can be written as

$$\begin{aligned}
& 1 + \sum_{h=1}^{t} \left(\left\lceil \frac{k_h - 1}{4} \right\rceil + 1 \right) + S \\
\geq & 1 + \frac{3}{4}t + \frac{1}{4} \left(k - l - \sum_{h=1}^{t-1} d_h + d - d_t \right) + S \\
\geq & 1 + \frac{3}{4}t + \frac{1}{4}(k - l + d - d_t) - \frac{1}{4} \sum_{h \in T_1} d_h - \frac{1}{4} \sum_{h \in T_2} d_h + \frac{1}{3} \sum_{h \in T_1} d_h - |T_1| + \alpha \\
\geq & 1 + \frac{3}{4}t + \frac{1}{4}(k - l + d - d_t) - \frac{|T_2|}{2} + \frac{1}{12} \sum_{h \in T_1} d_h - |T_1| + \alpha \\
\geq & 1 + \frac{3}{4}(1 + |T_1| + |T_2|) + \frac{1}{4}(k - l + d - d_t) - \frac{|T_2|}{2} + \frac{|T_1|}{4} - |T_1| + \alpha \\
\geq & \frac{7}{4} + \frac{1}{4}(k - l + d - d_t) + \left\lfloor \frac{l-1}{2} \right\rfloor + \left\lfloor \frac{d_t - d - 1}{2} \right\rfloor.
\end{aligned}$$

Let $\beta = \frac{7}{4} + \frac{1}{4}(k - l + d - d_t) + \lfloor \frac{l-1}{2} \rfloor + \lfloor \frac{d_t - d - 1}{2} \rfloor$. We can prove $\beta \geq \frac{k-1}{4} + 1$ for each of three cases as follows. If $l = 1$, then

$$\begin{aligned}
\beta & \geq \frac{7}{4} + \frac{1}{4}(k - l + d - d_t) + \frac{d_t - d - 1}{2} - \frac{1}{2} \\
& = \frac{3}{4} + \frac{k-1}{4} + \frac{d_t - d}{4} \geq \frac{k-1}{4} + 1.
\end{aligned}$$

If $l > 1$ and $d_t - d - 1 = 1$, then

$$\begin{aligned}
\beta & \geq \frac{7}{4} + \frac{1}{4}(k - l + d - d_t) + \frac{l-1}{2} - \frac{1}{2} \\
& = 1 + \frac{k-1}{4} + \frac{l - (d_t - d)}{4} \geq \frac{k-1}{4} + 1.
\end{aligned}$$

If $l > 1$ and $d_t - d - 1 > 1$, then

$$\begin{aligned}
\beta & \geq \frac{7}{4} + \frac{1}{4}(k - l + d - d_t) + \frac{l-1}{2} - \frac{1}{2} \\
& \quad + \frac{d_t - d - 1}{2} - \frac{1}{2} \\
& = \frac{k-1}{4} + \frac{l}{4} + \frac{d_t - d}{4} \geq \frac{k-1}{4} + 1.
\end{aligned}$$

Therefore, $N_{i,d} \geq \lceil \frac{k-1}{4} \rceil + 1$ holds. ◀

4.3 Proof of Theorem 1

Now, we are ready to prove Theorem 1.

Proof of Theorem 1. A string s can be written as the sequence of 1-domains, namely $s = \mathsf{extdom}_1(F_{i_1}) \cdots \mathsf{extdom}_1(F_{i_t})$ where $i_t = m$. Let k_h be the size of $\mathsf{dom}_1(F_{i_h})$. By Lemma 14, $\mathsf{extdom}_1(F_{i_h})$ contains $\left\lceil \frac{k_h - 1}{4} \right\rceil + 1$ LZ phrase boundaries. It is clear that $\sum_{h=1}^{t} k_h = m - t$. Therefore,

$$z \geq \sum_{h=1}^{t} \left(\left\lceil \frac{k_h - 1}{4} \right\rceil + 1 \right) \geq \frac{m - 2t}{4} + t > \frac{m}{4}$$

holds. ◀

5 Conclusion

We discussed the relationship between the size z of overlapping variant of LZ factorization and the size m of Lyndon factorization of the same string. We showed that the inequality $m < 4z$ holds for any string. One of the interesting open questions is whether there exists a better bound. Finally, we conjecture that the inequality $m < 2z$ holds for any string.

References

1. Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, Tomohiro I, Costas S. Iliopoulos, Shunsuke Inenaga, Simon J. Puglisi, and Shiho Sugimoto. Closed factorization. *Discrete Applied Mathematics*, 212:23–29, 2016. doi:10.1016/j.dam.2016.04.009.
2. Hideo Bannai, Travis Gagie, Shunsuke Inenaga, Juha Kärkkäinen, Dominik Kempa, Marcin Piatkowski, and Shiho Sugimoto. Diverse Palindromic Factorization is NP-Complete. *Int. J. Found. Comput. Sci.*, 29(2):143–164, 2018. doi:10.1142/S0129054118400014.
3. Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
4. Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic Length in Linear Time. In *CPM 2017*, pages 23:1–23:12, 2017. doi:10.4230/LIPIcs.CPM.2017.23.
5. Gang Chen, Simon J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, June 2008. doi:10.1007/s11786-007-0024-4.
6. 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. URL: http://www.jstor.org/stable/1970044.
7. Maxime Crochemore, Lucian Ilie, and Liviu Tinta. Towards a Solution to the "Runs" Conjecture. In Paolo Ferragina and Gad M. Landau, editors, *Combinatorial Pattern Matching*, pages 290–302, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
8. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries. In *SPIRE 2016*, pages 22–34, 2016. doi:10.1007/978-3-319-46049-9_3.
9. Marius Dumitran, Florin Manea, and Dirk Nowotka. On Prefix/Suffix-Square Free Words. In *SPIRE 2015*, pages 54–66, 2015. doi:10.1007/978-3-319-23826-5_6.
10. Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014. doi:10.1016/j.jda.2014.08.001.

11 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster Longest Common Extension Queries in Strings over General Alphabets. In *CPM 2016*, pages 5:1–5:13, 2016. doi:10.4230/LIPIcs.CPM.2016.5.

12 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/j.tcs.2016.03.005.

13 Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing Palindromic Factorizations and Palindromic Covers On-line. In *CPM 2014*, pages 150–161, 2014. doi:10.1007/978-3-319-07566-2_16.

14 Hiroe Inoue, Yoshiaki Matsuoka, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing Smallest and Largest Repetition Factorizations in $O(n \log n)$ Time. In *PSC 2016*, pages 135–145, 2016. URL: http://www.stringology.org/event/2016/p12.html.

15 Juha Kärkkäinen, Dominik Kempa, Yuto Nakashima, Simon J. Puglisi, and Arseny M. Shur. On the Size of Lempel-Ziv and Lyndon Factorizations. In *STACS 2017*, pages 45:1–45:13, 2017. doi:10.4230/LIPIcs.STACS.2017.45.

16 Roman Kolpakov and Gregory Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *FOCS 1999*, pages 596–604, Washington, DC, USA, 1999. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=795665.796470.

17 Dmitry Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016. doi:10.1016/j.ipl.2015.11.016.

18 Yoshiaki Matsuoka, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, and Florin Manea. Factorizing a String into Squares in Linear Time. In *CPM 2016*, pages 27:1–27:12, 2016. doi:10.4230/LIPIcs.CPM.2016.27.

19 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

20 Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984. doi:10.1109/MC.1984.1659158.

21 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977. doi:10.1109/TIT.1977.1055714.

22 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

Online Algorithms for Constructing Linear-Size Suffix Trie

Diptarama Hendrian
Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Takuya Takagi
Fujitsu Laboratories Ltd., Kawasaki, Japan
takagi.takuya@fujitsu.com

Shunsuke Inenaga
Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract

The suffix trees are fundamental data structures for various kinds of string processing. The suffix tree of a string T of length n has $O(n)$ nodes and edges, and the string label of each edge is encoded by a pair of positions in T. Thus, even after the tree is built, the input text T needs to be kept stored and random access to T is still needed. The *linear-size suffix tries* (*LSTs*), proposed by Crochemore et al. [Linear-size suffix tries, TCS 638:171-178, 2016], are a "stand-alone" alternative to the suffix trees. Namely, the LST of a string T of length n occupies $O(n)$ total space, and supports pattern matching and other tasks in the same efficiency as the suffix tree without the need to store the input text T. Crochemore et al. proposed an *offline* algorithm which transforms the suffix tree of T into the LST of T in $O(n \log \sigma)$ time and $O(n)$ space, where σ is the alphabet size. In this paper, we present two types of *online* algorithms which "directly" construct the LST, from right to left, and from left to right, without constructing the suffix tree as an intermediate structure. Both algorithms construct the LST incrementally when a new symbol is read, and do not access to the previously read symbols. The right-to-left construction algorithm works in $O(n \log \sigma)$ time and $O(n)$ space and the left-to-right construction algorithm works in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$ space. The main feature of our algorithms is that the input text does not need to be stored.

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

Keywords and phrases Indexing structure, Linear-size suffix trie, Online algorithm, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.30

Funding *Diptarama Hendrian*: Supported by JSPS KAKENHI Grant Number JP19K20208.
Shunsuke Inenaga: Supported by JSPS KAKENHI Grant Number JP17H01697.

Acknowledgements The authors thank Keisuke Goto and Mitsuru Funakoshi for discussions. The authors are also grateful for the anonymous referees for fruitful suggestions.

1 Introduction

Suffix tries are conceptually important string data structures that are the basis of more efficient data structures. While the suffix trie of a string T supports fast queries and operations such as pattern matching, the size of the suffix trie can be $\Theta(n^2)$ in the worst case, where n is the length of T. By suitably modifying suffix tries, we can obtain linear $O(n)$-size string data structures such as suffix trees [24], suffix arrays [20], directed acyclic word graphs (DAWGs) [4], compact DAWGs (CDAWGs) [5], position heaps [10], and so on. In the case of the integer alphabet of size polynomial in n, all these data structures can be constructed in $O(n)$ time and space in an *offline* manner [8, 9, 11, 13, 16, 18, 21]. In the case of a general

ordered alphabet of size σ, there are *left-to-right online* construction algorithms for suffix trees [23], DAWGs [4], CDAWGs [17], and position heaps [19]. Also, there are *right-to-left online* construction algorithms for suffix trees [24] and position heaps [10]. All these online construction algorithms run in $O(n \log \sigma)$ time with $O(n)$ space.

Suffix trees are one of the most extensively studied string data structures, due to their versatility. The main drawback is, however, that each edge label of suffix trees needs to be encoded as a pair of text positions, and thus the input string needs to be kept stored and be accessed even after the tree has been constructed. Crochemore et al. [7] proposed a new suffix-trie based data structure called *linear-size suffix tries* (*LSTs*). The LST of T consists of the nodes of the suffix tree of T, plus a linear-number of auxiliary nodes and suffix links. Each edge label of LSTs is a single character, and hence the input text string can be discarded after the LST has been built. The total size of LSTs is linear in the input text length, yet LSTs support fundamental string processing queries such as pattern matching within the same efficiency as their suffix tree counterpart [7].

Crochemore et al. [7] showed an algorithm which transforms the *given* suffix tree of string T into the LST of T in $O(n \log \sigma)$ time and $O(n)$ space. This algorithm is *offline*, since it requires the suffix tree to be completely built first. No efficient algorithms which construct LSTs *directly* (i.e. without suffix trees) and in an *online* manner were known.

This paper proposes two online algorithms that construct LSTs directly from the given string. The first algorithm is based on Weiner's suffix tree construction [24], and constructs the LST of T by scanning T *from right to left*. On the other hand, the second algorithm is based on Ukkonen's suffix tree construction [23], and constructs the LST of T by scanning T from *left to right*. Both algorithms construct the LST incrementally when a new symbol is read, and do not access the previously read symbols. This also means that our construction algorithms do not need to store the input text, and the currently processed symbol in the text can be immediately discarded as soon as the symbol at the next position is read. The right-to-left construction algorithm works in $O(n \log \sigma)$ time and $O(n)$ space and the left-to-right construction algorithm works in $O(n(\log \sigma + \frac{\log n}{\log \log n}))$ time and $O(n)$ space.

2 Preliminaries

Let Σ denote an *alphabet* of size σ. An element of Σ^* is called a *string*. For a string $T \in \Sigma^*$, the length of T is denoted by $|T|$. The *empty string*, denoted by ε, is the string of length 0. For a string T of length n, $T[i]$ denotes the i-th symbol of T and $T[i:j] = T[i]T[i+1] \dots T[j]$ denotes the *substring* of T that begins at position i and ends at position j for $1 \leq i \leq j \leq n$. Moreover, let $T[i:j] = \varepsilon$ if $i > j$. For convenience, we abbreviate $T[1:i]$ to $T[:i]$ and $T[i:n]$ to $T[i:]$, which are called *prefix* and *suffix* of T, respectively.

2.1 Linear-size suffix trie

The *suffix trie* $\mathsf{STrie}(T)$ of a string T is a trie that represents all suffixes of T. The *suffix link* of each node U in $\mathsf{STrie}(T)$ is an auxiliary link that points to $V = U[2:|U|]$. The *suffix tree* [24] $\mathsf{STree}(T)$ of T is a path-compressed trie that represents all suffixes of T. We consider the version of suffix trees where the suffixes that occur twice or more in T can be represented by non-branching nodes. The *linear-size suffix trie* $\mathsf{LST}(T)$ of a string T, proposed by Crochemore et al. [7], is another kind of tree that represents all suffixes of T, where each edge is labeled by a single symbol. The nodes of $\mathsf{LST}(T)$ are a subset of the nodes of $\mathsf{STrie}(T)$, consisting of the two following types of nodes:

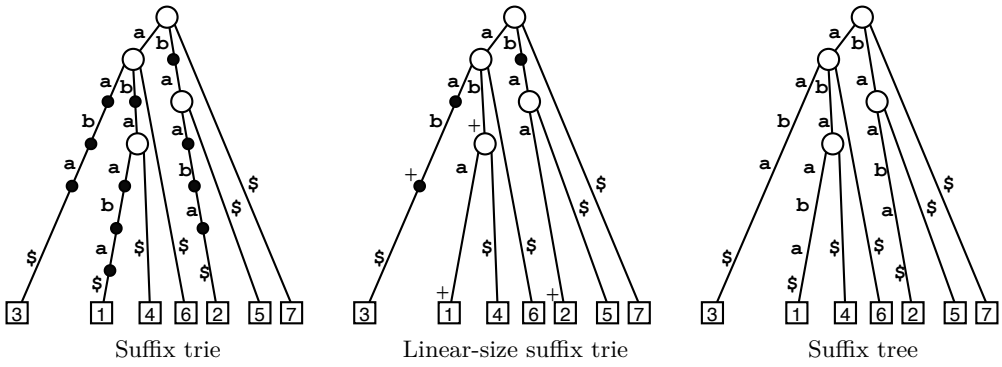


Figure 1 The suffix trie, linear-size suffix trie, and suffix tree of $T =$ abaaba\$.

1. Type-1: The nodes of $\mathsf{STrie}(T)$ whose that also nodes of $\mathsf{STree}(T)$.
2. Type-2: The nodes of $\mathsf{STrie}(T)$ that not type-1 nodes and their suffix links point to type-1 nodes.

A non-suffix type-1 node has two or more children and a type-2 node has only one child. When T ends with a unique terminate symbol \$ that does not occur elsewhere in T, then all type-1 nodes in $\mathsf{LST}(T)$ has two or more children. The nodes of $\mathsf{STrie}(T)$ that are neither type-1 nor type-2 nodes of $\mathsf{LST}(T)$ are called *implicit nodes* in $\mathsf{LST}(T)$.

We identify each node in $\mathsf{LST}(T)$ by the substring of T that is the path label from *root* to the node in $\mathsf{STrie}(T)$. Let U and V be nodes of $\mathsf{LST}(T)$ such that V is a child of U. The edge label of $(U, V) = c$ is the same as the label of the first edge on the path from U to V in $\mathsf{STrie}(T)$. If V is not a child of U in $\mathsf{STrie}(T)$, i.e. the length of the path label from U to V is more than one, we put the $+$ sign on V and we call V a $+$-node. Figure 1 shows an example of a suffix trie, linear-size suffix trie, and suffix tree.

For convenience, we assume that there is an auxiliary node $\perp$ as the parent of the root of $LST(T)$, and that the edge from $\perp$ to the root is labeled by any symbol. This assures that for each symbol appearing in T the root has a non $+$ child. This will be important for the construction of LSTs and pattern matching with LSTs (c.f. Lemma 2).

In the description of our algorithms, we will use the following notations. For any node U, $\mathsf{parent}(U)$ denotes the parent node of U. For any edge (U, V), $\mathsf{label}(U, V)$ denotes the label of the edge connecting U and V, For a node U and symbol c, $\mathsf{child}(U, c)$ denotes the child of U whose incoming edge label is c, if it exists. We denote $+(U) = \mathsf{true}$ if U is a $+$-node, and $+(U) = \mathsf{false}$ otherwise. The suffix link of a node U is defined as $\mathsf{slink}(U) = V$, where $V = U[2 : |U|]$. The reversed suffix link of a node U with a symbol $c \in \Sigma$ is defined as $\mathsf{rlink}(U, c) = V$, if there is a node U such that $cU = V$. It is undefined otherwise. For any type-1 node U, $\mathsf{t1parent}(U)$ denotes the nearest type-1 ancestor of U, and $\mathsf{t1child}(U, c)$ denotes the nearest type-1 descendant of U on c edge. For any type-2 node U, $\mathsf{child}(U)$ is the child of U, and $\mathsf{label}(U)$ is the label of the edge connecting U and its child.

2.2 Pattern matching using linear-size suffix trie

In order to efficiently perform pattern matching on LSTs, Crochemore et al. [7] introduced *fast links* that are a chain of *suffix links of edges*.

▶ **Definition 1.** *For any edge* (U, V)*, let* $\mathsf{fastLink}(U, V) = (\mathsf{slink}^h(U), \mathsf{slink}^h(V))$ *such that* $\mathsf{slink}^h(U) \neq \mathsf{parent}(\mathsf{slink}^h(V))$ *and* $\mathsf{slink}^{h-1}(U) = \mathsf{parent}(\mathsf{slink}^{h-1}(V))$*, where* $\mathsf{slink}^0(U) = U$ *and* $\mathsf{slink}^i(U) = \mathsf{slink}(\mathsf{slink}^{i-1}(U))$.

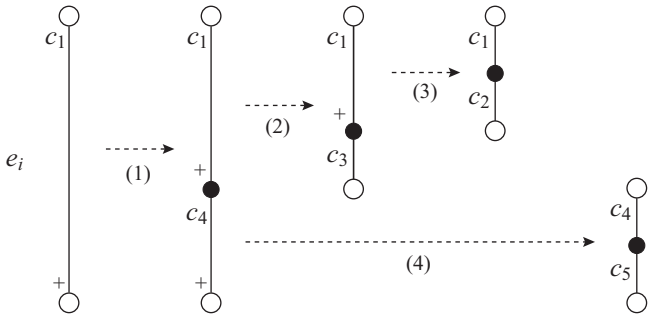


Figure 2 Illustration for our pattern matching algorithm with LST. The dashed arrows represent fast links. The number in parentheses show the orders of applications of fast links when traversing $P_i = c_1c_2c_3c_4c_5$ on the edge e_i.

Here, h is the minimum number of suffix links that we need to traverse so that $\mathsf{slink}^h(U) \neq \mathsf{parent}(\mathsf{slink}^h(V))$. Namely, after taking h suffix links from edge (U, V), there is at least one type-2 node in the path from $\mathsf{slink}^h(U)$ to $\mathsf{slink}^h(V)$. Since type-2 nodes are not branching, we can use the labels of the type-2 nodes in this path to retrieve the label of the edge (U, V) (see Lemma 2 below). Provided that $\mathsf{LST}(T)$ has been constructed, the fast link $\mathsf{fastLink}(U, V)$ for every edge (U, V) can be computed in a total of $O(n)$ time and space [7].

▶ **Lemma 2** ([7]). *The underlying label of a given edge (U, V) of length ℓ can be retrieved in $O(\ell \log \sigma)$ time by using fast links.*

Crochemore et al. [7] claimed that due to Lemma 2 one can perform pattern matching for a given pattern P in $O(|P| \log \sigma)$ time with the LST. However, the proofs provided in [7] for the correctness and time efficiency of their pattern matching algorithm looks unsatisfactory to us, because the algorithm of Crochemore et al. [7] does not seem to guarantee that the label of a given edge is retrieved sequentially from the first symbol to the last one (see also [22]). Still, in the following lemma we present an algorithm which efficiently performs the longest prefix match for a given pattern on the LST with fast links:

▶ **Lemma 3.** *Given $\mathsf{LST}(T)$ and a pattern P, we can find the longest prefix P' of P that occurs in T in $O(|P'| \log \sigma)$ time.*

Proof. Let $P_1P_2 \cdots P_m = P'$ be the factorization of P' such that $P_1 \cdots P_i$ is a node in $\mathsf{LST}(T)$ for $1 \leq i < m$, $P_1 \cdots P_i = \mathsf{parent}(P_1 \cdots P_{i+1})$ for $1 \leq i < m-1$, and $P_1 \cdots P_{m-1}$ is the longest prefix of P' that is a node in $\mathsf{LST}(T)$. If $P_1 \cdots P_{m-1} = P'$, then $P_m = \varepsilon$. In what follows, we consider a general case where $P_m \neq \varepsilon$.

Suppose we have successfully traversed up to $P_1 \cdots P_{i-1}$, and let U be the node representing $P_1 \cdots P_{i-1}$. If U has no out-going edge labeled $c_1 = P_i[1] = P[|P_1 \cdots P_{i-1}| + 1]$, then the traversal terminates on U. Suppose U has an out-going edge labeled c_1 and let V be the child of U with the c_1-edge. We denote this edge by $e_i = (U, V)$. See also Figure 2 for illustration. If V is a not +-node, then we have read c_1 and set $U \leftarrow V$ and continue with the next symbol $c_2 = P_i[2] = P[|P_1 \cdots P_{i-1}| + 2]$. Otherwise (if V is a +-node), then we apply $\mathsf{fastLink}$ from edge (U, V) recursively, until reaching the edge (U', V') such that V' is not a +-node. Then we move onto V'. Note that by the definition of $\mathsf{fastLink}$, V' is always a type-2 node. We then continue the same procedure by setting $U \leftarrow V'$ with the next pattern symbol c_2. This will be continued until we arrive at the first edge (U, V) such that V is a type-1 node. Then, we trace back the chain of $\mathsf{fastLink}$'s from (U, V) until getting back to

the type-2 node V'' whose out-going edge has the next symbol to retrieve. We set $U \leftarrow V''$ and continue with the next symbol. This will be continued until we traverse all symbols c_j in P_i for increasing $j = 1, \ldots, |P_i|$ along the edge e_i, or find the first mismatching symbols.

The correctness of the above algorithm follows from the fact that every symbol in label of the edge e_i is retrieved from a type-2 node that is not branching, except for the first one retrieved from the type-1 node that is the origin of e_i. Since any type-2 node is not branching, we can traverse the edge e_i with P_i iff the underlying label of e_i is equal to P_i for $1 \leq i \leq m-1$. The case of the last edge e_m where the first mismatching symbols are found is analogous.

To analyze the time complexity, we consider the number of applications of fastLink. For each $1 \leq i \leq m-1$, the number of applications of fastLink is bounded by the length of the underlying label of edge e_i, which is $|P_i|$. This is because each time we follow a fastLink, at least one new symbol is retrieved. Hence we can traverse $P_1 \cdots P_{m-1}$ in $O(|P_1 \cdots P_{m-1}| \log \sigma)$ time. For the last fragment P_m, we consider the number of applications of fastLink until we find the type-2 node X whose out-going edge has the first mismatching symbol. Since the first application of fastLink for P_m begins with an edge whose destination has string depth $|P_1 \cdots P_{m-1}|$ and since each symbol appearing in T is represented by a node as a child of the root, the number of applications of fastLink until finding X is bounded by $|P_1 \cdots P_{m-1}|$. Note that this is independent of the length of the edge e_m which can be much longer than P_m. After finding X, we can traverse P_m as in the same way to previous P_i's. Thus, we can traverse P_m in $O(|P_1 \cdots P_m| \log \sigma)$. Overall, it takes $O(|P_1 \cdots P_m| \log \sigma)$ time to traverse $P' = P_1 \cdots P_m$. This completes the proof. ◀

Algorithm 6 in Appendix shows a pseudo-code of our pattern matching algorithm with the LST in Lemma 3.

3 Right-to-left online algorithm

In this section, we present an online algorithm that constructs $\mathsf{LST}(T)$ by reading T from right to left. Let $\mathcal{T}_i = \mathsf{LST}(T[i:])$ for $1 \leq i \leq n$. Our algorithm constructs $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ incrementally when $c = T[i]$ is read. For simplicity, we assume that T ends with a unique terminal symbol \$ such that $T[i] \neq \$$ for $1 \leq i < n$.

We remark that the algorithm does not construct fast links of the LSTs. The fast links can easily be constructed in $O(n)$ time after $\mathsf{LST}(T)$ has been constructed.

Let us first recall Weiner's suffix tree contraction algorithm on which our right-to-left LST construction algorithm is based. Weiner's algorithm uses the reversed suffix links of the suffix tree called *hard Weiner links*. We in particular consider the version of Weiner's algorithm that also explicitly maintains *soft-Weiner links* [6] of the suffix tree. In the suffix tree of a text T, there is a soft-Weiner link for a node V with a symbol c iff cV is a substring of T but cV is not a node in the suffix tree. It is known that the hard-Weiner links and the soft-Weiner links are respectively equivalent to the primary edges and the secondary edges of the *directed acyclic word graph* (*DAWG*) for the reversal of the input string [4].

Given the suffix tree for $T[i+1:]$, Weiner's algorithm walks up from the leaf representing $T[i+1:]$ and first finds the nearest branching ancestor V such that aV is a substring of $T[i+1:]$, and then finds the nearest branching ancestor V' such that $cV' = U'$ is also a branching node, where $c = T[i]$. Then, Weiner's algorithm finds the insertion point for a new leaf for $T[i:]$ by following the reversed suffix link (i.e. the hard-Weiner link) from V' to U', and then walking down the corresponding out-edge of U' with the difference of the string depths of V and V'. A new branching node U is made at the insertion point if necessary.

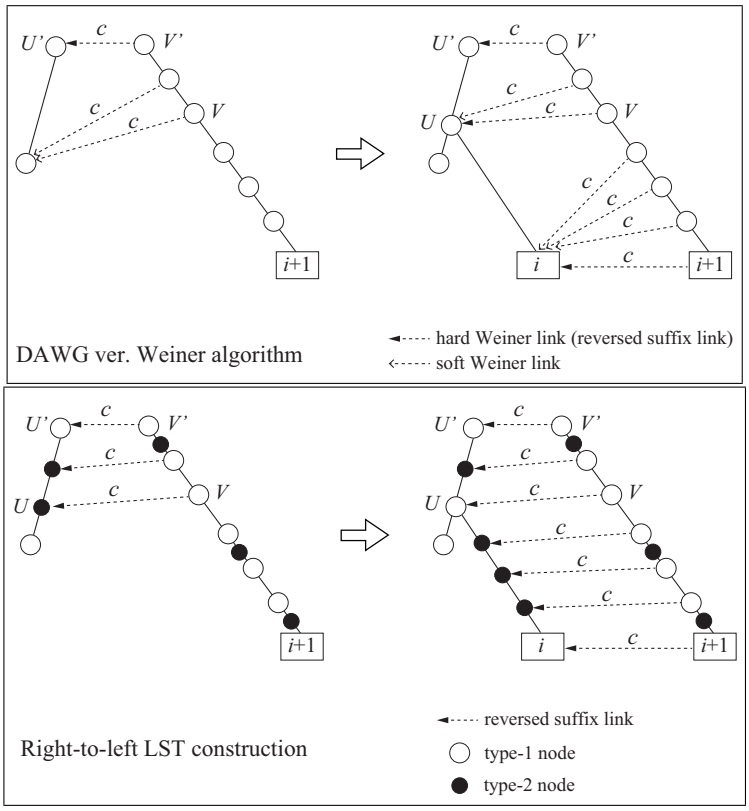


Figure 3 Upper: The DAWG version of Weiner's algorithm when updating the suffix tree for $T[i+1:]$ to the suffix tree for $T[i:]$. Lower: Our right-to-left LST construction when updating $\mathcal{T}_{i+1} = \mathsf{LST}(T[i+1:])$ to $\mathcal{T}_i = \mathsf{LST}(T[i:])$.

New soft-Weiner links are created from the nodes between the leaf for $T[i+1:]$ and V to the new leaf for $T[i:]$.

Now we consider our right-to-left LST construction. See the lower diagram of Figure 3 for illustration. The major difference between the DAWG version of Weiner's algorithm and our LST construction is that in our LST we explicitly create type-2 nodes which are the destinations of the soft-Weiner links. Hence, in our linear-size suffix trie construction, for every type-1 node between V and the leaf for $T[i+1:]$, we explicitly create a unique new type-2 node on the path from the insertion point to the new leaf for $T[i:]$, and connect them by the reversed suffix link labeled with c. Also, we can directly access the insertion point U by following the reversed suffix link of V, since U is already a type-2 node before the update.

The above observation also gives rise to the number of type-2 nodes in the LST. Blumer et al. [4] proved that the number of secondary edges in the DAWG of any string of length n is at most $n-1$. Hence we have:

▶ **Lemma 4.** *The number of type-2 nodes in the LST of any string of length n is at most $n-1$.*

The original version of Weiner's suffix tree construction algorithm only maintains a Boolean value indicating whether there is a soft-Weiner link from each node with each symbol. We note also that the number of pairs of nodes and symbols for which the indicators are true is the same as the number of soft-Weiner links (and hence the DAWG secondary edges).

We have seen that LSTs can be seen as a representation of Weiner's suffix trees or the DAWGs for the reversed strings. Another crucial point is that Weiner's algorithm only needs to read the first symbols of edge labels. This enables us to easily extend Weiner's suffix tree algorithm to our right-to-left LST construction. Below, we will give more detailed properties of LSTs and our right-to-left construction algorithm.

Let us first observe relations between $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$.

▶ **Lemma 5.** *Any non-leaf type-1 node U in $\mathcal{T}_i$ exists in $\mathcal{T}_{i+1}$ as a type-1 or type-2 node.*

Proof. If there exist two distinct symbols $a, b \in \Sigma$ such that Ua, Ub are substrings of $T[i+1:]$, then clearly U is a type-1 node in $\mathcal{T}_{i+1}$. Otherwise, then let b be a unique symbol such that Ub is a substring of $T[i+1:]$. This symbol b exists since U is not a leaf in $\mathcal{T}_i$. Also, since U is a type-1 node in $\mathcal{T}_i$, there is a symbol $a \neq b$ such that Ua is a substring of $T[i:]$. Note that in this case Ua is a prefix of $T[i:]$ and this is the unique occurrence of Ua in $T[i:]$. Now, let $U' = U[2:]$. Then, $U'a$ is a prefix of $T[i+1:]$. Since $U'b$ is a substring of $T[i+1:]$, U' is a type-1 node in $\mathcal{T}_{i+1}$ and hence U is a type-2 node in $\mathcal{T}_{i+1}$. ◀

As was described above, only a single leaf is added to the tree when updating $\mathcal{T}_{i+1}$ to $\mathcal{T}_i$. The type-2 node of $\mathcal{T}_i$ that becomes type-1 in $\mathcal{T}_i$ is the *insertion point* of this new leaf.

▶ **Lemma 6.** *Let U be the longest prefix of $T[i:]$ such that U is a prefix of $T[j:]$ for some $j > i$. U is a node in $\mathcal{T}_{i+1}$.*

Proof. If $U = \varepsilon$ then U is the root. Otherwise, since U occurs twice or more in $T[i:]$ and $T[i : i+|U|] \neq T[j : j+|U|]$, U is a type-1 node in $\mathcal{T}_i$. By Lemma 5, U is a node in $\mathcal{T}_{i+1}$. ◀

By Lemma 6, we can construct $\mathcal{T}_i$ by adding a branch on node U, where U is the longest prefix of $T[i:]$ such that U is a prefix of $T[j:]$ for some $j > i$. This node U is the insertion point for $\mathcal{T}_i$. The insertion point U can be found by following the reversed suffix link labeled by c from the node $U[2:]$ i.e. $U = \mathsf{rlink}(U[2:], c)$. Since U is the longest prefix of $T[i:]$ where $U[2:]$ occurs at least twice in $T[i+1:]$, $U[2:]$ is the deepest ancestor of the leaf $T[i+1:]$ that has the reversed suffix link labeled by c. Therefore, we can find U by checking the reversed suffix links of the ancestors of $T[i+1:]$ walking up from the leaf. We call this leaf representing $T[i+1:]$ as the *last leaf* of $\mathcal{T}_{i+1}$.

After we find the insertion point, we add some new nodes. First, we consider the addition of new type-1 nodes.

▶ **Lemma 7.** *There is at most one type-1 node U in $\mathcal{T}_i$ such that U is a type-2 node in $\mathcal{T}_{i+1}$. If such a node U exists, then U is the insertion point of $\mathcal{T}_i$.*

Proof. Assume there is a type-1 node U in $\mathcal{T}_i$ such that U is a type-2 node in $\mathcal{T}_{i+1}$. There are suffixes UV and UW such that $|V| > |W|$ and $V[1] \neq W[1]$. Since U is a type-2 node in $\mathcal{T}_{i+1}$, $UV = T[i:]$ and $UW = T[j:]$ for some $j > i$. Clearly, such a node is the only one which is the branching node. ◀

From Lemma 7, we know that new type-1 node is added at the insertion point if it is a type-2 node. The only other new type-1 node is the new leaf representing $T[i:]$.

Next, we consider the addition of the new branch from the insertion point. By Lemma 7, there are no type-1 nodes between the insertion point and the leaf for $T[i:]$ in $\mathcal{T}_i$. Thus, any node V in the new branch is a type-2 node and this node is added if $V[2:]$ is a type-1 node. This can be checked by ascending from leaf $T[i+1:]$ to $U[2:]$, where U is the insertion point. Regarding the labels of the new branch, for any new node V and its parent W, the label of

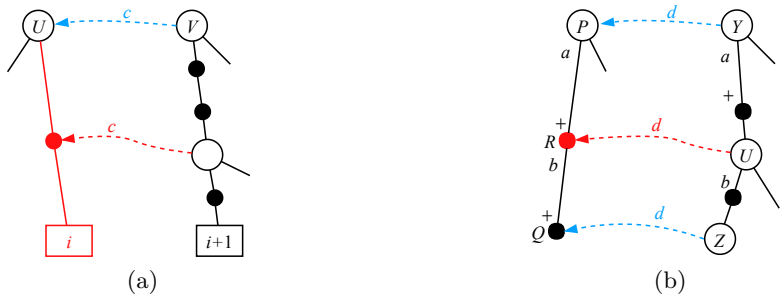


Figure 4 Illustration of (a) new branch addition and (b) type-2 nodes addition. The new nodes, edges, and reverse suffix link are colored red.

(W, V) edge is the same as the label of the first edge between $W[2:]$ and $V[2:]$. The node V is a $+$-node if $V[2:]$ is a $+$-node or there is a node between $W[2:]$ and $V[2:]$. Figure 4 (a) shows an illustration of the branch addition: V can be found by traversing the ancestors of $i+1$ leaf. After we find the insertion point $U = \mathsf{rlink}(V, c)$, we add a new leaf i and type-2 nodes for each type-1 node between $i+1$ leaf and V.

Last, consider the addition of type-2 nodes when updating the insertion point U to a type-1 node. In this case, we add a type-2 node dU for any $d \in \Sigma$ such that dU occurs in $T[i:]$.

▶ **Lemma 8.** *Let U be the insertion point of $\mathcal{T}_i$. Consider the case where U is a type-2 node in $\mathcal{T}_{i+1}$. Let Z be the nearest type-1 descendant of U and Y be the nearest type-1 ancestor of U in $\mathcal{T}_{i+1}$. For any node Q such that $Q = \mathsf{rlink}(Z, d)$ for some $d \in \Sigma$, $P = \mathsf{rlink}(Y, d)$ is the parent of Q in $\mathcal{T}_{i+1}$ and there is a type-2 node R between P and Q in $\mathcal{T}_i$.*

Proof. First, we prove that P is the parent of Q in $\mathcal{T}_{i+1}$. Assume on the contrary that P is not the parent of Q. Then, there is a node $Q[:j] = dZ[:j-1]$ for some $|P| < j < |Q|$. Thus, $Z[:j-1]$ is a type-1 ancestor of Z and a type-1 descendant of Y, however this contradicts the definition of Z or Y.

Second, we prove that there is a type-2 node between P and Q in $\mathcal{T}_i$. Since U is a type-2 node in $\mathcal{T}_{i+1}$ and $Q = dZ$ is a node in $\mathcal{T}_{i+1}$, dU occurs in $T[i+1:]$ but is not a node in $\mathcal{T}_{i+1}$. Since U is a type-1 node in $\mathcal{T}_i$, dU is a type-2 node $\mathcal{T}_i$. ◀

See Figure 4 (b) for an illustration of type-2 nodes addition. It follows from Lemma 8 that we can find the position of new type-2 nodes by first following the reversed suffix link of the nearest type-1 descendant Z of U in $\mathcal{T}_{i+1}$. Then, we obtain the parent P of Z, and obtain Y by following the suffix link of P. The string depth of a new type-2 node R equal to the string depth of U plus one. We can determine whether R is a $+$-node using the difference of the string depths of Y and U. By Lemma 5, the total number of type-2 nodes added this way for all positions $1 \le i \le n$ is bounded by the number of type-1 and type-2 nodes in $\mathcal{T}_n$ for the whole text T.

Algorithm 1 in Appendix shows a pseudo-code of our right-to-left linear-size suffix trie construction algorithm. For each symbol $c = T[i]$ read, the algorithm finds the deepest node U in the path from the root to the last leaf for $T[i+1:]$ for which $\mathsf{rlink}(U, c)$ is defined, by walking up from the last leaf (line 5). If the insertion point $\mathit{insertPoint} = \mathsf{rlink}(U, c)$ is a type-1 node, the algorithm creates a new branch. Otherwise (if $\mathit{insertPoint}$ is a type-2 node), then the algorithm updates $\mathit{insertPoint}$ to type-1 and adds a new branch. The branch addition is done in lines 10–21.

Also, the algorithm adds nodes R such that $R = \mathsf{rlink}(\mathit{insertPoint}, d)$ for some $d \in \Sigma$ in $\mathcal{T}_i$. The algorithm finds the locations of these nodes by checking the reversed suffix links of the nearest type-1 ancestor and descendant of $\mathit{insertPoint}$ by using $\mathsf{createType2}(\mathit{insertPoint})$.

Let Y be the nearest type-1 ancestor of *insertPoint* and Z be the nearest type-1 descendant of *insertPoint*. For a symbol d such that $\mathsf{rlink}(Z, d)$ is defined, let $P = \mathsf{rlink}(Y, d)$ and $Q = \mathsf{rlink}(Z, d)$: the algorithm creates type-2 node R and connects it to P and Q.

A snapshot of right-to-left LST construction is shown in Figure 8 of Appendix.

We discuss the time complexity of our right-to-left online LST construction algorithm. Basically, the analysis follows the amortization argument for Weiner's suffix tree construction algorithm. First, consider the cost for finding the insertion point for each i.

▶ **Lemma 9.** *Our algorithm finds the insertion point of $\mathcal{T}_i$ in $O(\log \sigma)$ amortized time.*

Proof. For each iteration, the number of type-1 and type-2 nodes we visit from the last leaf to find the insertion point is at most $\mathsf{depth}(L_{i+1}) - \mathsf{depth}(U_i) + 1$, where L_{i+1} is the leaf representing $T[i+1:]$ and U_i is the insertion point for the new leaf representing $T[i:]$ in $\mathcal{T}_i$, respectively, and $\mathsf{depth}(X)$ denotes the depth of any node X in $\mathcal{T}_i$. See also the lower diagram of Figure 3 for illustration. Therefore, the total number of nodes visited is $\sum_{1 \leq i < n} \mathsf{depth}(L_{i+1}) - \mathsf{depth}(U_i) + 1 \leq 2n$. Since finding each reversed suffix link takes $O(\log \sigma)$ time, the total cost for finding the insertion points for all $1 \leq i \leq n$ is $O(n \log \sigma)$, which is amortized to $O(\log \sigma)$ per iteration. ◀

Last, the computation time of a new branch addition in each iteration is as follows.

▶ **Lemma 10.** *Our algorithm adds a new leaf and new type-2 nodes between the insertion point and the new leaf in $\mathcal{T}_i$ in $O(\log \sigma)$ amortized time.*

Proof. Given the insertion point for $\mathcal{T}_i$, it is clear that we can insert a new leaf in $O(\log \sigma)$ time. For each new type-2 node in the path from the insertion point and the new leaf for $T[i:]$, there is a corresponding type-1 node in the path above the last leaf $T[i+1:]$ (see also the lower diagram of Figure 3). Thus the cost for inserting all type-2 nodes can be charged to the cost for finding the insertion point for $\mathcal{T}_i$, which is amortized $O(\log \sigma)$ per a new type-2 node by Lemma 9. ◀

By Lemmas 9 and 10, we get the following theorem:

▶ **Theorem 11.** *Given a string T of length n, our algorithm constructs $\mathsf{LST}(T)$ in $O(n \log \sigma)$ time and $O(n)$ space online, by reading T from the right to the left.*

4 Left-to-right online algorithm

In this section, we present an algorithm that constructs the linear-size suffix trie of a text T by reading the symbols of T from the left to the right. Our algorithm constructs a slightly-modified data structure called the pre-LST defined as follows: The pre-LST $\mathsf{preLST}(T)$ of a string T is a subgraph of $\mathsf{STrie}(T)$ consisting of two types of nodes,

1. Type-1: The root, branching nodes, and leaves of $\mathsf{STrie}(T)$.
2. Type-2: The nodes of $\mathsf{STrie}(T)$ that are not type-1 nodes and their suffix links point to type-1 nodes.

The main difference between $\mathsf{preLST}(T)$ and $\mathsf{LST}(T)$ is the definition of type-1 nodes. While $\mathsf{LST}(T)$ may contain non-branching type-1 nodes that correspond to non-branching internal nodes of $\mathsf{STree}(T)$ which represent repeating suffixes, $\mathsf{preLST}(T)$ does not contain such type-1 nodes. When T ends with a unique terminal symbol \$, the pre-LST and LST of T coincide.

Our algorithm is based on Ukkonen's suffix tree construction algorithm [23]. For each prefix $T[:i]$ of T, there is a unique position k_i in $T[:i]$ such that $T[k_i:i]$ occurs twice or more in $T[:i-1]$ but $T[k_i - 1:i]$ occurs exactly once in $T[:i]$. In other words, $T[k_i - 1:i]$ is

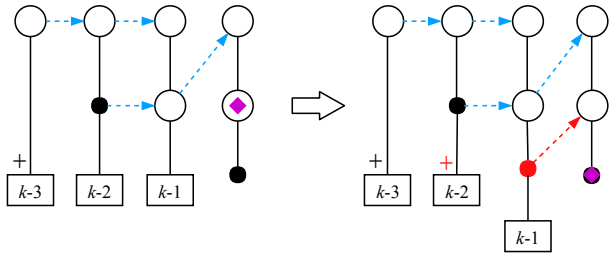


Figure 5 Illustration for updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j < k_i$. The purple diamond shows the active point. The new + sign, node, and its suffix link are colored red.

the shortest suffix of $T[: i]$ that is represented as a leaf in the current pre-LST $\mathsf{preLST}(T[: i])$, and $T[k_i : i]$ is the longest suffix of $T[: i]$ that is represented in the "inside" of $\mathsf{preLST}(T[: i])$. The location of $\mathsf{preLST}(T[: i])$ representing the longest repeating suffix $T[k_i : i]$ of $T[: i]$ is called the *active point*, as in the Ukkonen's suffix tree construction algorithm. We also call k_i the *active position* for $T[: i]$. Our algorithm keeps track of the location for the active point (and the active position) each time a new symbol $T[i]$ is read for increasing $i = 1, \ldots, n$. We will show later that the active point can be maintained in $O(\log \sigma)$ amortized time per iteration, using a similar technique to our pattern matching algorithm on LSTs in Lemma 3. In order to "neglect" extending the leaves that already exist in the current tree, Ukkonen's suffix tree construction algorithm uses the idea of *open leaves* that do not explicitly maintain the lengths of incoming edge labels of the leaves. However, we cannot adapt this open leaves technique to construct pre-LST directly, since we need to add type-2 node on the incoming edges of some leaves. Fortunately, there is a nice property on the pre-LST so we can update it efficiently. We will discuss the detail of this property later. Below, we will give more detailed properties of pre-LSTs and our left-to-right construction algorithm.

Let $\mathcal{P}_i = \mathsf{preLST}(T[: i])$ be the pre-LST of $T[: i]$. Our algorithm constructs $\mathcal{P}_i$ from $\mathcal{P}_{i-1}$ incrementally when a new symbol $c = T[i]$ is read.

There are two kinds of leaves in $\mathsf{preLST}(T[: i])$, the one that are +-nodes and the other ones that are not +-nodes. There is a boundary in the suffix link chain of the leaves that divides the leaves into the two groups, as follows:

▶ **Lemma 12.** *Let $T[j : i]$ be a leaf of $\mathcal{P}_i$, for $1 \le j < k$. There is a position l such that $T[j : i]$ is a +-node for $1 \le j < l$ and not a +-node for $l \le j < k_i$.*

Proof. Assume on the contrary there is a position j such that $T[j : i]$ is not a +-node and $T[j+1 : i]$ is a + node. Since $T[j : i]$ is not a +-node, $T[j : i-1]$ is a node. By definition, $T[j+1 : i-1]$ is also a node. Thus $T[j+1 : i]$ is not a +-node, which is a contradiction. ◀

Intuitively, the leaves that are +-nodes in $\mathcal{P}_i$ are the ones that were created in the last step of the algorithm with the last read symbol $T[i]$.

When updating $\mathcal{P}_{i-1}$ into $\mathcal{P}_i$, the active position k_{i-1} for $T[: i-1]$ divides the suffixes $T[j : i-1]$ into two parts, the $j < k_{i-1}$ part and the $j \ge k_{i-1}$ part. First, we consider updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j < k_{i-1}$.

▶ **Lemma 13.** *For any leaf $T[j : i-1]$ of $\mathcal{P}_{i-1}$ with $j < k_{i-1} - 1$, $T[j : i-1]$ is implicit in $\mathcal{P}_i$.*

Proof. Consider updating $\mathcal{P}_{i-1}$ to $\mathcal{P}_i$. $T[k_{i-1} - 1 : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$. Therefore, $T[k_{i-1}-2 : i-1]$ is implicit in $\mathcal{P}_i$. $T[j : i-1]$ for $j < k_{i-1}-1$ are also implicit. ◀

▶ **Lemma 14.** *If $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, then $T[j : i]$ is a $+$-leaf in $\mathcal{P}_i$, where $1 \leq j < k_{i-1} - 1$.*

Proof. Assume on the contrary that $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$ but $T[j : i]$ is not a $+$-leaf in $\mathcal{P}_i$. Then $T[j : i-1]$ is a node in $\mathcal{P}_i$. Since $T[j : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, $T[j : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$. Moreover, $T[j+1 : i-1]$ is a leaf in $\mathcal{P}_{i-1}$, thus $T[j+1 : i-1]$ cannot be a type-1 node in $\mathcal{P}_i$ and $T[j : i-1]$ cannot be a type-2 node in $\mathcal{P}_i$. Therefore, $T[j : i-1]$ is neither type-1 nor type-2 node in $\mathcal{P}_i$, which contradicts the assumption. ◀

Lemma 13 shows that we do not need to add nodes on the leaves of $\mathcal{P}_{i-1}$ besides $T[k-1 : i]$ leaf and Lemma 14 shows that we can update all leaves $T[j : i]$ for $l \leq j < k-1$ to a $+$-leaf. Therefore, besides the leaf for $T[k-1 : i]$, once we update a leaf to $+$ node, we do not need to update it again. Figure 5 shows an illustration of how to update this part.

Next, we consider updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j \geq k_{i-1}$. If $T[k_{i-1} : i]$ exists in the current LST (namely $T[k_{i-1} : i]$ occurs in $T[: i-1]$), then the $j \geq k_{i-1}$ part of the current LST does not need to be updated. Then we have $k_i = k_{i-1}$ and $T[k_i : i]$ is the active point of $\mathcal{P}_i$. Otherwise, we need to create new nodes recursively from the active point that will be the parents of new leaves. There are three cases for the active point $T[k_{i-1} : i-1]$ in $\mathcal{P}_{i-1}$:

Case 1. $T[k_{i-1} : i-1]$ is a type-1 node in $\mathcal{P}_{i-1}$. Let $T[p : i]$ be the longest suffix of $T[k_{i-1} : i]$ that exists in $\mathcal{P}_{i-1}$. Since $T[k_{i-1} : i-1]$ is a type-1 node, $T[j : i-1]$ is also a type-1 node for $k_{i-1} \leq j < p$. Therefore, we can obtain $\mathcal{P}_i$ by adding a leaf from the node representing $T[j : i-1]$ for every $k \leq j < p$, with edge label c by following the suffix link chain from $T[k_{i-1} : i-1]$. In this case, we only need to add one new type-2 node, which is $T[k_{i-1} - 1 : i-1]$ that is connected to the type-1 node $T[k_{i-1} : i-1]$ by the suffix link. Moreover, p will be the active position for $T[: i]$, namely $k_i = p$.

Case 2. $T[k_{i-1} : i-1]$ is a type-2 node in $\mathcal{P}_{i-1}$. Similarly to Case 1, we add a leaf from the node representing $T[j : i-1]$ for every $k_{i-1} \leq j < p$ with edge label c by following the suffix link chain from $T[k_{i-1} : i-1]$, where p is defined as in Case 1.. Then, $T[k_{i-1} : i-1]$ becomes a type-1 node, and a new type-2 node $T[k_{i-1} - 1 : i-1]$ is added and is connected to this type-1 node $T[k_{i-1} : i-1]$ by the suffix link. Moreover, for any symbol d such that $dT[k_{i-1} : i-1]$ is a substring of $T[: i]$, a new type-2 node for $dT[k_{i-1} : i-1]$ is added to the tree, and is connected by the suffix link to this new type-1 node $T[k_{i-1} : i-1]$. These new type-2 nodes can be found in the same way as in Lemma 8 for our right-to-left LST construction. Finally, p will become the active position for $T[: i]$, namely $k_i = p$.

Case 3. $T[k_{i-1} : i-1]$ is implicit in $\mathcal{P}_{i-1}$. In this case, there is a position $p > k_{i-1}$ such that $T[p : i-1]$ is a type-2 node. We create new type-1 nodes $T[j : i-1]$ and leaves $T[j : i]$ for $k \leq j < p$, then do the same procedure as Case 2 for $T[j : i-1]$ for $p \leq j$.

Figure 6 shows an illustration of how to add new leaves. Algorithm 3 shows a pseudo-code of our left-to-right online algorithm for constructing LSTs. In Case 1 or Case 2, the algorithm checks whether there is an out-going edge labeled with $c = T[i]$, and performs the above procedures (lines 19–29). In Case 3, we perform readEdge to check if the active point can proceed with c on the edge. The function readEdge returns the location of the new active point and sets *mismatch* = false if there is no mismatch, or it returns the mismatching position and sets *mismatch* = true if there is a mismatch. If there is no mismatch, then we just update the $T[j : i-1]$ part of the current LST for $j < k_{i-1}$. Otherwise, then we create new nodes as explained in Case 3, by split in the pseudo-code.

A snapshot of right-to-left LST construction is shown in Figure 9 of the Appendix.

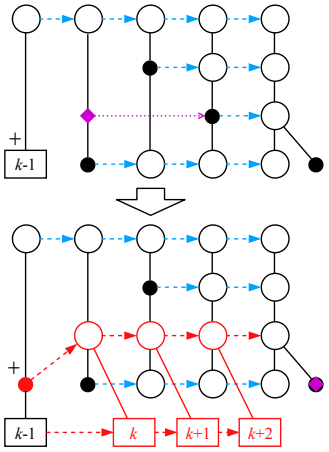


Figure 6 Illustration for updating the parts of $\mathcal{P}_{i-1}$ that correspond to $T[j : i-1]$ for $j \geq k_{i-1}$. The purple diamond and arrow show the active point and its virtual position when reading the edge. The new branches, nodes, and their suffix links are colored red.

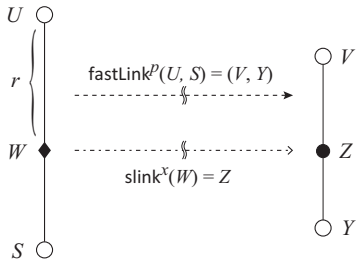


Figure 7 Illustration for our analysis of the cost to maintain the active point. The diamond shows the current location of the active point. New leaves will be created from W to Z by following the (virtual) suffix link chain of length x. When we have reached the edge (V, Y), we have already retrieved the corresponding prefix of the label between U and W. The rest of the label can be retrieved by at most r applications of fastLink from edge (V, Z).

We discuss the time complexity of our left-to-right online construction for LSTs. To maintain the active point for each $T[: i]$, we use a similar technique to Lemma 3.

▶ **Lemma 15.** *The active point can be maintained in $O(f(n) + \log \sigma)$ amortized time per each iteration, where $f(n)$ denotes the time for accessing* fastLink *in our growing LST.*

Proof. We consider the most involved case where the active point lies on an implicit node W on some edge (U, S) in the current LST. The other cases are easier to show. Let $r = |W| - |U|$, i.e., the active point is hanging off U with string depth r. Let Z be the type-2 node from which a new leaf will be created. By the monotonicity on the suffix link chain there always exists such a type-2 node. See Figure 7 for illustration. Let p be the number of applications of fastLink from edge (U, S) until reaching the edge (V, Y) on which Z lies. Since such a type-2 node Z always exists, we can sequentially retrieve the first r symbols with at most r applications of fastLink by the same argument to Lemma 3. Thus the number of applications of fastLink until finding the next location of the active point is bounded by $p + r$. If x is the number of (virtual) suffix links from W to Z, then $p \leq x$ holds. Recall that we create at least $x + 1$ new leaves by following the (virtual) suffix link chain from W to Z. Now r is charged to the number of text symbols read on the edge from U, and p is charged to the number of newly created leaves, and both of them are amortized constant as in Ukkonen's suffix tree algorithm. Thus the number of applications of fastLink is amortized constant, which implies that it takes $O(f(n) + \log \sigma)$ amortized time to maintain the active point. ◀

To maintain fastLink in our growing (suffix link) tree, we use the nearest marked ancestor (NMA) data structure [1] that allows marking, unmarking, and NMA query in an online manner in $O(\log n / \log \log n)$ time each, using $O(n)$ space on a dynamic tree of size n. By maintaining the tree of suffix links of edges enhanced with the NMA data structure, we have $f(n) = O(\log n / \log \log n)$ for Lemma 15. This leads to the final result of this section.

▶ **Theorem 16.** *Given a string T of length n, our algorithm constructs* LST(T) *in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$ space online, by reading T from the left to the right.*

5 Conclusions and Future Work

In this paper we proposed a right-to-left online algorithm which constructs linear-size suffix trees (LSTs) in $O(n \log \sigma)$ time and $O(n)$ space, and a left-to-right online algorithm which constructs LSTs in $O(n(\log \sigma + \log n / \log \log n))$ time and $O(n)$ space, for an input string of length n over an ordered alphabet of size σ. Unlike the previous construction algorithm by Crochemore et al. [7], our algorithms do not construct suffix trees as an intermediate structure, and do not require to store the input string. Fischer and Gawrychowski [12] showed how to build suffix trees in a right-to-left online manner in $O(n(\log \log n + \log^2 \log \sigma / \log \log \log \sigma))$ time for an integer alphabet of size $\sigma = n^{O(1)}$. It might be possible to extend their result to our right-to-left online LST construction algorithm. An improvement of the running time of left-to-right online LST construction is also left for future work.

Takagi et al. [22] proposed *linear-size CDAWGs* (*LCDAWG*), which are edge-labeled DAGs obtained by merging isomorphic subtrees of LSTs. They showed that the LCDAWG of a string T takes only $O(e + e')$ space, where e and e' are respectively the numbers of right and left extensions of the maximal repeats in T, which are always smaller than the text length n. Belazzougui and Cunial [2] proposed a very similar CDAWG-based data structure that uses only $O(e)$ space. It is not known whether these data structures can be efficiently constructed in an online manner, and thus it is interesting to see if our algorithms can be extended to these data structures. The key idea to both of the above CDAWG-based structures is to implement edge labels by *grammar-compression* or *straight-line programs*, which are enhanced with efficient grammar-compressed data structures [14, 3]. In our online setting, the underlying grammar needs to be dynamically updated, but these data structures are static. It is worth considering if these data structures can be efficiently dynamized by using recent techniques such as e.g. [15].

References

1 Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked Ancestor Problems. In *Proc. FOCS 1998*, pages 534–544, 1998. doi:10.1109/SFCS.1998.743504.

2 Djamal Belazzougui and Fabio Cunial. Fast Label Extraction in the CDAWG. In *Proc. SPIRE 2017*, pages 161–175, 2017. doi:10.1007/978-3-319-67428-5_14.

3 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random Access to Grammar-Compressed Strings and Trees. *SIAM J. Comput.*, 44(3):513–539, 2015. doi:10.1137/130936889.

4 Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M.T. Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985. doi:10.1016/0304-3975(85)90157-4.

5 Anselm Blumer, J. Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

6 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. doi:10.1016/j.jda.2012.07.003.

7 Maxime Crochemore, Chiara Epifanio, Roberto Grossi, and Filippo Mignosi. Linear-size suffix tries. *Theoretical Computer Science*, 638:171–178, 2016. doi:10.1016/j.tcs.2016.04.002.

8 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching*, pages 116–129, 1997. doi:10.1007/3-540-63220-4_55.

9 Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science: A Selection of Essays in Honor of A. Ehrenfeucht*, pages 192–211. Springer Berlin Heidelberg, 1997. doi:10.1007/3-540-63246-8_12.

10 Andrzej Ehrenfeucht, Ross M. McConnell, Nissa Osheim, and Sung-Whan Woo. Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 9(1):100–121, 2011. doi:10.1016/j.jda.2010.12.001.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

12 Johannes Fischer and Pawel Gawrychowski. Alphabet-Dependent String Searching with Wexponential Search Trees. In *Proc. CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

13 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets. In *MFCS 2016*, pages 38:1–38:14, 2016. doi:10.4230/LIPIcs.MFCS.2016.38.

14 Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-Time Traversal in Grammar-Based Compressed Files. In *Proc. DCC 2005*, page 458, 2005. doi:10.1109/DCC.2005.78.

15 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal Dynamic Strings. In *Proc. SODA 2018*, pages 1509–1528, 2018. doi:10.1137/1.9781611975031.99.

16 Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/j.tcs.2016.03.005.

17 Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005. doi:10.1016/j.dam.2004.04.012.

18 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

19 Gregory Kucherov. On-line construction of position heaps. *Journal of Discrete Algorithms*, 20:3–11, 2013. doi:10.1016/j.jda.2012.08.002.

20 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

21 Kazuyuki Narisawa, Hideharu Hiratsuka, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Computation of Substring Equivalence Classes with Suffix Arrays. *Algorithmica*, 79(2):291–318, 2017. doi:10.1007/s00453-016-0178-z.

22 Takuya Takagi, Keisuke Goto, Yuta Fujishige, Shunsuke Inenaga, and Hiroki Arimura. Linear-Size CDAWG: New Repetition-Aware Indexing and Grammar Compression. In *SPIRE 2017*, volume 10508, pages 304–316, 2017. doi:10.1007/978-3-319-67428-5_26.

23 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

24 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

A Supplementary Figures

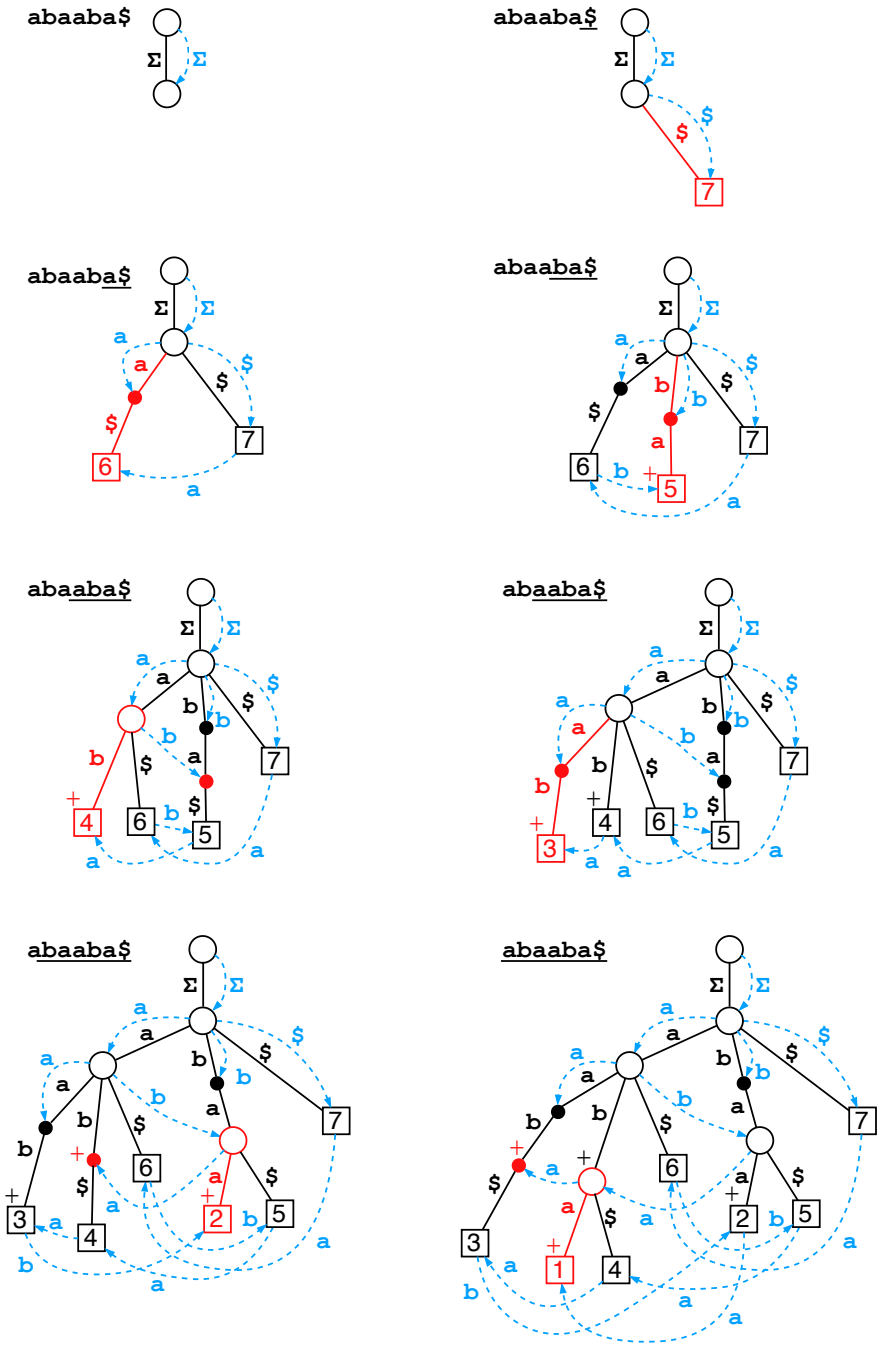


Figure 8 A snapshot of right-to-left online construction of LST(T) with T = abaaba\$ by Algorithm 1. The white circles show Type-1 nodes, the black circles show Type-2 nodes, and the rectangles show leaves. The reverse suffix links and its label are colored blue. The new branches and nodes are colored red.

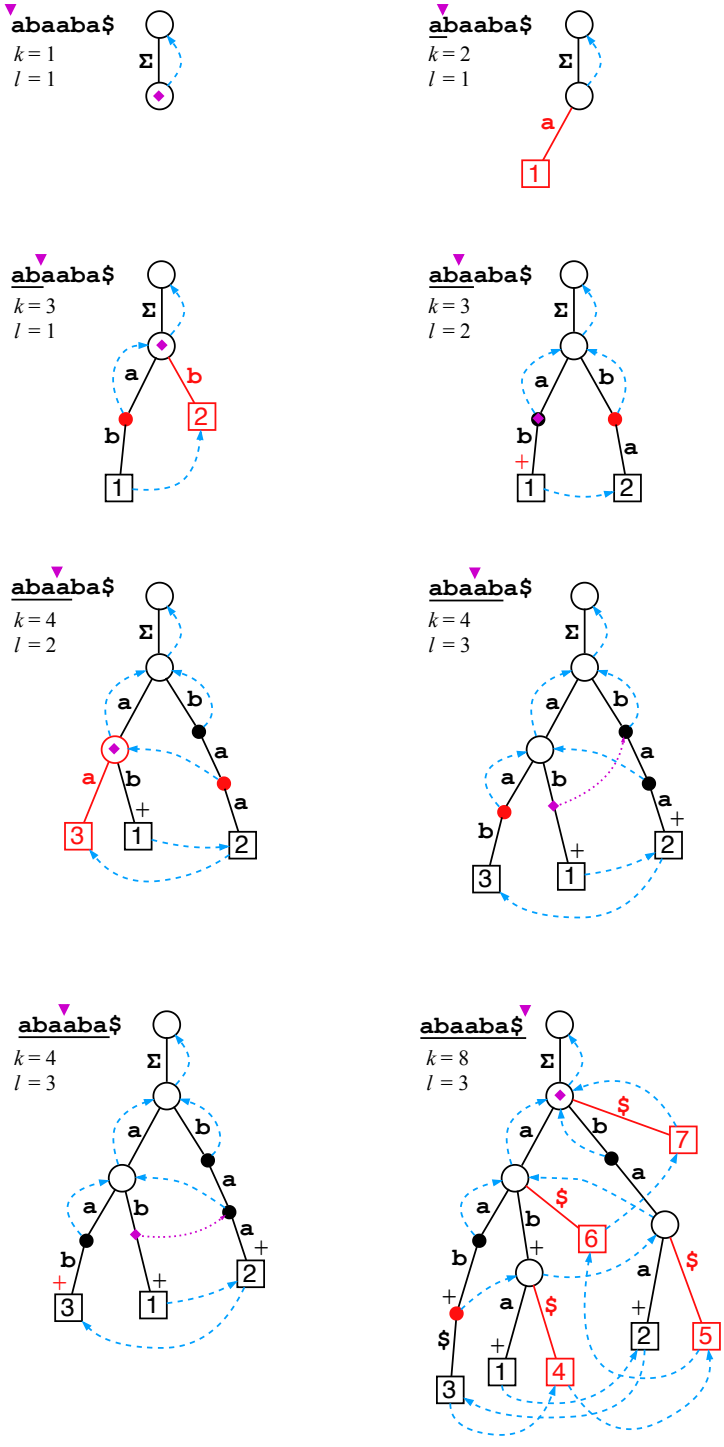


Figure 9 A snapshot of left-to-right online construction of $\mathsf{LST}(T)$ with T = abaaba\$ by Algorithm 3. The purple diamond and arrow represent the active point and its virtual position when reading the edge label. The suffix links are colored blue. The new branches and nodes are colored red. k is the active position and l is the boundary position for +-leaves and non-+ leaves defined in Lemma 12.

B Pseudo-codes

Algorithm 1: Right-to-left linear-size suffix trie construction algorithm.

```
1  child(⊥, c) := root for any c ∈ Σ; rlink(⊥, c) := root for all c ∈ Σ;
2  prevInsPoint := ⊥; prevLeaf := root; prevLabel := NULL;
3  for i = n to 1 do
4      c := T[i]; U := prevInsPoint;
5      while rlink(U, c) = NULL do U := parent(U);
6      insertPoint := rlink(U, c);
7      if type(insertPoint) = 2 then
8          createType2(insertPoint);
9          type(insertPoint) := 1;
10     create a leaf newLeaf;
11     V := prevLeaf; U := prevInsPoint; Y := newLeaf;
12     while rlink(U, c) = NULL do
13         create a type-2 node X;
14         if U = prevInsPoint then a = prevLabel else a = label(U, V);
15         if +(V) = true or child(U, a) ≠ V then +(Y) := true;
16         child(X, a) := Y; rlink(U, c) := X; Y := X;
17         V := U;
18         repeat U := parent(U) until type(U) = 1;
19     if U = ⊥ then a = c else a = label(U, V);
20     if +(V) = true or child(U, a) ≠ V then +(Y) := true;
21     child(insertPoint, a) := Y;
22     prevInsPoint := insertPoint; prevLeaf := newLeaf; prevLabel := a;
```

Algorithm 2: createType2(U).

```
1  Function createType2(U)
2      V := U; b = label(U); Z := t1child(U, b);
3      for d such that rlink(Z, d) ≠ NULL do
4          Q := rlink(Z, d);
5          P := parent(Q);
6          if slink(P) ≠ NULL then
7              a := label(P, Q);
8              Y := slink(P);
9              create a type-2 node R;
10             child(P, a) := R; child(R, b) := Q;
11             if child(Y, a) ≠ U or +(child(Y, a)) = true then +(R) := true;
12             if child(U, b) ≠ Z or +(child(U, b)) = true then +(Q) := true;
```

Algorithm 3: Left-to-right linear-size suffix trie construction algorithm.

```
1  create root and ⊥; child(⊥, c) := root for any c ∈ Σ;
2  activePoint = root; i := 1; l := 1; k := 1;
3  while i ≤ n do
4      c := T[i];
5      if child(activePoint, c) ≠ NULL then
6          V := child(activePoint, c);
7          (U, i', mismatch) := readEdge((activePoint, V), i);
8          if type(activePoint) = 1 then
9              create a type-2 node W;
10             V := parent(leaf[k − 1]);
11             child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
12             +(W, c) := +(leaf[k − 1]); slink(W) := activePoint;
13         else +(leaf[k − 1]) := true;
14         while j ≠ k − 1 do +(leaf[l]) := true; l := l + 1;
15         if mismatch = false then
16             if +(U) = true then +(leaf[k − 1]) := true;
17         else split(U, activePoint, c, i, i');
18         activePoint := U; i := i';
19     else
20         if type(activePoint) = 2 then
21             createType2(activePoint); type(activePoint) := 1;
22         while l ≠ k − 1 do +(leaf[l]) := true; l := l + 1;
23         create a type-2 node W; V := parent(leaf[k − 1]);
24         child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
25         +(W, c) := +(leaf[k − 1]); slink(W) := activePoint;
26         while child(activePoint, c) = NULL do
27             create a leaf U;
28             child(activePoint, c) := U; slink(leaf[k − 1]) := U;
29             k := k + 1; leaf[k − 1] := U; activePoint = slink(activePoint);
```

Algorithm 4: readEdge$((U, V), i)$.

```
1  Function readEdge(U, V, i)
2      while U ≠ V do
3          c := T[i];
4          if child(U, c) = NULL then return (U, i, true);
5          else
6              if +(child(U, c)) = true then
7                  (W, i, mismatch) := readEdge(fastLink(U, child(U, c)), i);
8                  if mismatch = true then return (W, i, true);
9                  U := W;
10             else U := child(U, c); i := i + 1;
11     return (U, i, false);
```

Algorithm 5: split(U, X, a, i, i').

```
1  Function split(U, X, a, i, i')
2      b = label(U, child(U)); c' := T[i'];
3      create a type-1 node W;
4      V := parent(leaf[k − 1]);
5      child(W, c) := leaf[k − 1]; child(V, label(V, leaf[k − 1])) := W;
6      +(W) := +(leaf[k − 1]); newNode := W;
7      k := k + 1; Y' := leaf[k − 1];
8      while X ≠ U do
9          if type(x) = 1 then Y := child(X, a);
10         d = STrieDepth(Y) − STrieDepth(X);
11         while d < i' − i do
12             X := Y; i := i + d;
13             Y := child(X); d := STrieDepth(Y) − STrieDepth(X);
14         if X ≠ U then
15             create a type-2 node Z; create a leaf Y'; a := label(X, Y);
16             child(X, a) := Z; child(Z, b) := Y; createType2(Z);
17             type(Z) := 1; child(Z, c') := Y';
18             if i' − 1 > 1 then +(Z) := true;
19             if d − (i' − 1) > 1 then +(Y) := true;
20             slink(newNode) := Z; slink(leaf[k − 1]) := Y';
21             k := k + 1; leaf[k − 1] := Y';
22             newNode := Z; X := slink(X);
23     slink(newNode) := U;
```

Algorithm 6: Fast pattern matching algorithm with the LST.

```
1  let P be a pattern and i be a global index.
2  Function fastMatching(P)
3      U := root; i := 1;
4      while i ≤ |P| do
5          if child(U, P[i]) ≠ NULL then
6              U := fastDecompact(U, child(U, P[i]));
7              if U = NULL then return false;
8          else return false;
9      return true;
10 Function fastDecompact(U, V)
11     while U ≠ V do
12         if child(U, P[i]) ≠ NULL then
13             if +(child(U, P[i])) = false then
14                 U := child(U, P[i]);
15                 i := i + 1;
16             else U = fastDecompact(fastLink(U), fastLink(child(U, P[i])));
17             if i > |P| then return V;
18         else return NULL;
19     return V;
```

Searching Long Repeats in Streams

Oleg Merkurev
Ural Federal University, Ekaterinburg, Russia
o.merkuryev@gmail.com

Arseny M. Shur
Ural Federal University, Ekaterinburg, Russia
arseny.shur@urfu.ru

Abstract

We consider two well-known related problems: Longest Repeated Substring (LRS) and Longest Repeated Reversed Substring (LRRS). Their streaming versions cannot be solved exactly; we show that only approximate solutions by Monte Carlo algorithms are possible, and prove a lower bound on consumed memory. For both problems, we present purely linear-time Monte Carlo algorithms working in $O(E + \frac{n}{E})$ space, where E is the additive approximation error. Within the same space bounds, we then present nearly real-time solutions, which require $O(\log n)$ time per symbol and $O(n + \frac{n}{E} \log n)$ time overall. The working space exactly matches the lower bound whenever $E = O(n^{0.5})$ and the size of the alphabet is $\Omega(n^{0.01})$.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Longest repeated substring, longest repeated reversed substring, streaming algorithm, Karp–Rabin fingerprint, suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.31

Funding *Oleg Merkurev*: Supported by the Russian Ministry of Education and Science, project 1.3253.2017.
Arseny M. Shur: Supported by the Russian Ministry of Education and Science, project 1.3253.2017.

Acknowledgements The authors are grateful to D. Kosolobov for very useful comments.

1 Introduction

The streaming model of computation became popular in string processing during the last decade. In this model, there is no random access to the input: the input string of length n arrives symbol by symbol and the space available to the algorithm is sublinear in n. In the design of algorithms for this model, the two main goals are minimization of the total space required and of the worst-case time in which a single symbol is processed (update time). If a problem can be solved only approximately, there is usually a trade-off between the approximation error and the required space/time. A wide use of Monte Carlo randomized algorithms is another distinctive feature of the streaming model.

Surprisingly, the exact pattern matching, which is the fundamental problem in stringology, can be solved in the streaming model very efficiently. The first efficient algorithm was presented by Porat and Porat [18]; soon after that Breslauer and Galil showed [3] that $O(\log m)$ space is enough to find all occurrences of a length-m pattern in a length-n text in real time (=constant update time), just with a small probability of false positive. More complicated pattern matching problems, like approximate or multiple matching, were also analysed [5, 6, 7, 12, 13, 19], as well as the related problem of estimating the Hamming distance between the pattern and length-m substrings of the text [8].

Another class of string problems concerns the search of repetitions (periods, repeats, palindromes, etc). As shown in [11], a longest palindrome in a stream can be found approximately by a real-time Monte Carlo algorithm spending $O(M)$ words of memory,

where $M = \frac{n}{E}$ for the additive error E and $M = \frac{\log n}{\log(1+\varepsilon)}$ for the multiplicative error $1+\varepsilon$; matching lower bounds for the required space were also proved there. A longest palindrome with at most d errors can be found within the space and update time which are both polylog in n and inverse polynomial in the error [14].

In this paper, we consider the problem of finding the longest string occurring at least twice in the input (longest repeating substring, LRS) and the problem of finding the longest string occurring in the input together with its reversal (longest repeating reversed substring, LRRS). Solving LRS in the standard RAM model in linear time is a well-known application of the suffix tree, first mentioned in the pioneering paper by Weiner [21]. LRRS can be solved in the same way (e.g., by building the suffix tree for the reversal of the input string). Note that LRS and LRRS also admit efficient parallel algorithms [2]. However, in the streaming model these problems can be solved only approximately and only with high probability, as shown in Section 6. Our main contributions are Monte Carlo algorithms for both problems, solving them with an additive approximation error E in $O(\frac{n}{E} + E)$ space. We give two algorithms for each problem; "simple" algorithms work in $O(n)$ total time, but have the update time $O(\frac{n}{E})$, while more elaborate versions use $O(n + \frac{n}{E}\log n)$ total time and allow updates in $\mathcal{O}(\log n)$ time. After preliminaries, we describe the main idea of reaching sublinear memory in Section 3, algorithms for LRRS in Section 4, algorithms for LRS in Section 5, and memory lower bounds in Section 6.

2 Model and Definitions

Let S denote a string of length n over an alphabet $\Sigma = \{1, \ldots, N\}$, where N is polynomial in n. We write $S[i]$ for the ith symbol of S and $S[i..j]$ for its *substring* (or *factor*) $S[i]S[i+1]\cdots S[j]$; thus, $S[1..n] = S$. A *prefix* (resp. *suffix*) of S is a substring of the form $S[1..j]$ (resp., $S[j..n]$). A *period* of S is a positive integer p such that $S[1..n-p] = S[p+1..n]$; here the string $S[1..n-p]$ is a *border* of S. The *reversal* of S is the string $\bar{S} = S[n]S[n-1]\cdots S[1]$. If $S = \bar{S}$, then S is a *palindrome*. A repeat in S is a pair of equal substrings $S[i..i+l-1] = S[j..j+l-1]$, where $i < j$; we denote repeats by the triples (i, j, l) and write $L(S)$ for the maximum length of a repeat in S. Similar, the condition $S[i..i+l-1] = \bar{S}[j..j+l-1]$, $i \le j$, defines the *reversed repeat* (i, j, l) in S; we write $\bar{L}(S)$ for the maximum length of a reversed repeat in S. Note that a palindrome $S[i..i+l-1]$ is the reversed repeat (i, i, l); moreover, if $i + l \ge j - 1$ for a reversed repeat (i, j, l), then $S[i..j+l-1]$ is a palindrome of length $l + j - i$. Thus the longest reversed repeat in S is either non-overlapping (the "left occurrence" $S[i..i+l-1]$ ends before the "right occurrence" $S[j..j+l-1]$ begins), or a palindrome.

We write log for the binary logarithm and $a \overline{\bmod} b$ instead of $(a-1) \bmod b + 1$.

We work in the *streaming model* of computation: the input string $S[1..n]$ (the *stream*) is read left to right, one symbol at a time, and cannot be stored, because the available space is sublinear in n. The space is counted as the number of $\mathcal{O}(\log n)$-bit machine words. The *update* time is the worst-case time spent between two reads.

An approximation algorithm for a maximization problem has *additive error* E if it finds a solution with the cost at least $OPT - E$, where OPT is the cost of optimal solution; here E can be a function of the size of the input. For an instance LRS(S) of the LRS problem, $OPT = L(S)$; similar for LRRS(S) and $\bar{L}(S)$.

A *Las Vegas algorithm* always returns a correct answer, but its working time and memory usage on length-n inputs are random variables. A *Monte Carlo algorithm* gives a correct answer with high probability (at least $1 - \frac{1}{n}$) and has deterministic working time and space.

▶ **Observation 1.** *A longest palindrome in a stream can be approximated in $\mathcal{O}(\frac{n}{E})$ space and $\mathcal{O}(1)$ update time by Monte Carlo Algorithm A of [11], which is better than we can achieve for* LRRS*. So for an algorithm designed to solve* LRRS *it suffices to process only* some *reversed repeats (including* all non-overlapping ones*) in a stream; we assume that the algorithm for the longest palindrome is run in parallel, and the longer of two results is returned.*

Karp–Rabin fingerprints [15] is a hash function widely used for streaming string algorithms. Let p be a fixed prime from the range $[n^{3+\alpha}, n^{4+\alpha}]$ for some $\alpha > 0$, and r be a fixed integer randomly chosen from $\{1, \dots, p-1\}$. For a string S, its forward hash and reversed hash are defined, respectively, as $\phi^F(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^i\right) \bmod p$ and $\phi^R(S) = \left(\sum_{i=1}^{n} S[i] \cdot r^{n-i+1}\right) \bmod p$. Clearly, the forward hash of a string coincides with the reversed hash of its reversal; this fact is used to detect reversed repeats. The probability of hash collision for two strings of length m is at most m/p; thus, a linear-time algorithm faces a collision with probability $O(n^{-1-\alpha})$ by the choice of p. All further considerations assume that no collisions happen. For an input stream S, we denote $F^F(i,j) = \phi^F(S[i..j])$ and $F^R(i,j) = \phi^R(S[i..j])$. Hashes of substrings can be extracted in constant time from the hashes of prefixes, as the next observation shows.

▶ **Proposition 2** ([3])**.** *The following equalities hold:*

$$F^F(i,j) = r^{-(i-1)} \left(F^F(1,j) - F^F(1,i-1)\right) \bmod p\,,$$
$$F^R(i,j) = F^R(1,j) - r^{j-i+1} F^R(1,i-1) \bmod p\,.$$

For an input stream S, the tuple $I(i) = (i, F^F(1,i-1), F^R(1,i-1), r^{1-i} \bmod p, r^i \bmod p)$ is its *i-th frame*. The proposition below is immediate from definitions and Proposition 2.

▶ **Proposition 3.** *Given $I(i)$ and $S[i]$, the tuple $I(i+1)$ can be computed in $\mathcal{O}(1)$ time.*

Let $\mathcal{T}(S)$ denote the suffix tree of a length-n string S. Recall that $\mathcal{T}(S)$ is the compressed trie of all suffixes of S, has $\mathcal{O}(n)$ nodes and edges, and occupies $\mathcal{O}(n)$ words of memory; S is a part of the data structure. Edges are labelled by substrings of S, each node is identified with the label of the path from the root to it. Not all substrings of S correspond to nodes; in general, substrings are addressed by *positions*. A position in the suffix tree is a pair $pos = (v, raise)$, where v is a node and $raise \geq 0$. This position corresponds to the prefix of v of length $|v| - raise$ and points to the "locus" in $\mathcal{T}(S)$ on the incoming edge of v, $raise$ symbols above v. The walks on the tree are navigated by (direct) *suffix links*; for a node v, $link(v)$ is the longest proper suffix of v. A similar link from an arbitrary position is called *implicit suffix link* and is not stored; computing such links is a crucial primitive for the work with suffix trees.

3 Reduction to Packed Repeats

The main difficulty of LRS and LRRS is that they are not local: the two occurrences of a repeat can be separated by as much as $\Omega(n)$ symbols. To avoid storing the whole input, we hash blocks of some fixed size b, consider hashes as new symbols, called hashletters, and search for repeated strings of hashletters. We define *direct trace* and *reversed trace* of a stream S, with the block size b and shift r, as the strings

$$P_{r,b} = F^F(r, r+b-1)F^F(r+b, r+2b-1) \cdots F^F(r+x_{r,b}b, r+(x_{r,b}+1)b-1) \text{ and}$$
$$Q_{r,b} = F^R(r+x_{r,b}b, r+(x_{r,b}+1)b-1) \cdots F^R(r+b, r+2b-1)F^R(r, r+b-1),$$

respectively, where $x_{r,b} = \left\lfloor \frac{|S|+1-r}{b} \right\rfloor - 1$ and the alphabet is $0, \dots, p-1$.

Let us fix b and consider only the traces with block size b and shifts $r = 1, \dots, b$, writing P_r, Q_r, x_r instead of $P_{r,b}, Q_{r,b}, x_{r,b}$; the traces P_b, Q_b are called *main*. We say that a substring $S[l..t]$ is contained in the traces P_r and Q_r if $l \equiv_b r$ and $t - l + 1 \equiv_b 0$. This condition means that $S[l..t]$ can be partitioned into blocks such that the corresponding hashletters after the forward (resp., backward) hashing constitute a substring of the trace P_r (resp., Q_r).

We store only the main trace (P_b for LRS and Q_b for LRRS) and search for repeats with the left occurrence contained in the main trace. We call these repeats *packable*. By the length argument, the right occurrence of a packable repeat is also contained in some trace. Long packable repeats approximate the solutions for LRS and LRRS:

▶ **Observation 4.** *For each repeat (reversed repeat) (i, j, l) such that $l > 2b - 2$, a stream S contains a packable repeat (resp., packable reversed repeat) (i', j', l') such that $l' \geq l - 2b + 2$. Namely, take $i' = \lceil \frac{i}{b} \rceil b$, $j' = j + i' - i$, $l' = \lfloor \frac{i+l-i'}{b} \rfloor b$.*

A *packed repeat* in a string S, denoted by a quadruple (r, i_l, i_r, k), is a pair of equal substrings $P_b[i_l \,..\, i_l{+}k{-}1] = P_r[i_r \,..\, i_r{+}k{-}1]$, where $i_l < i_r$ and $1 \leq r \leq b$ (see Fig. 1). Similar, a *packed reversed repeat* is a pair $Q_b[x_r{+}2{-}i_l \,..\, x_r{+}3{-}i_l{-}k] = P_r[i_r \,..\, i_r{+}k{-}1]$.

▶ **Observation 5.** *If (i, j, l) is a packable repeat (packable reversed repeat), then the tuple $(j \bmod b, i/b, \lfloor j/b \rfloor + 1, l/b)$ is a packed repeat (packed reversed repeat). Conversely, if (r, i_l, i_r, k) is a packed repeat (packed reversed repeat), then $(i_l b, (i_r - 1)b + r, kb)$ is a packable repeat (resp., packable reversed repeat), up to a hash collision.*

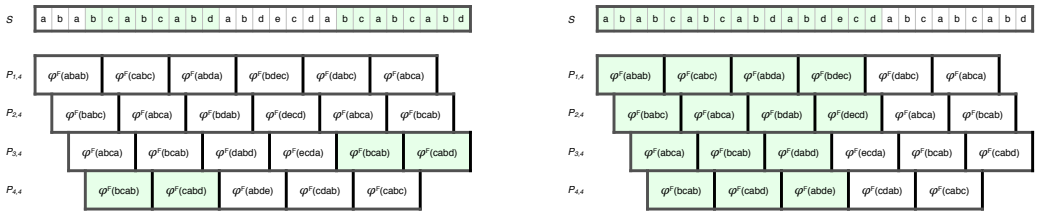


Figure 1 Traces and packed repeats. Left: packable repeat $(4, 19, 8)$ and its packed repeat $(3, 1, 5, 2)$ (colored). Right: hashletters available in all direct traces after reading $S[17]$ (colored).

Due to Observations 4, 5, we reduced the initial problems to the problems of finding a longest packed repeat and a longest packed reversed repeat in a stream; their solutions will solve, w.h.p., LRS and LRRS with an additive error less than $2b$. By Observation 1, the algorithm for packed reversed repeats can skip some of them if the corresponding packable reversed repeats are overlapping.

A hashletter has the form $F^F(j, j + b - 1)$ or $F^R(j, j + b - 1)$. This hashletter becomes *available* when we read the symbols $S[j], S[j{+}1], \dots, S[j{+}b{-}1]$, i.e., after reading $S[j{+}b{-}1]$. Hence, after each read, starting from $S[b]$, a new hashletter becomes available for one trace and one reversed trace. To compute this hashletter $F^F(j, j + b - 1)$ or $F^R(j, j + b - 1)$ it is sufficient to know the frames $I(j + b)$ and $I(j)$. Before reading $S[i]$ we store $I(i - b + 1), I(i - b + 2), \dots, I(i)$. The numbers of these frames are all distinct modulo b, so the frames can be stored in a cyclic queue of length b. The prefix of the trace P_r (resp., the suffix of Q_r) available after reading $S[i]$ is denoted by P_r^i (resp., Q_r^i). Note that $P_r^i = P_r[1..\lfloor \frac{i-r+1}{b} \rfloor]$, similar for Q_r^i.

4 Search of a Longest Reversed Repeat

We first approach LRRS. By Observation 1, it is sufficient to analyse any set of reversed repeats which includes all non-overlapping ones. If a reversed repeat is non-overlapping, then the packable reversed repeat obtained from it (Observation 4) is also non-overlapping.

Let us define the packed counterpart of non-overlapping repeats. A *handy repeat* is a packed reversed repeat (r, i_l, i_r, k) such that its left occurrence is a substring of $Q_b^{i_r b-1}$.

▶ **Lemma 6.** *For a non-overlapping packable reversed repeat, the corresponding packed reversed repeat is handy.*

Proof. Let (i, j, l) be a non-overlapping packable reversed repeat; the corresponding packed reversed repeat is $(j \bmod b, i/b, \lfloor j/b \rfloor{+}1, l/b)$ by Observation 5. Its left occurrence is contained in Q_b^{i+l-1}, and $i + l - 1 < j - 1 < (\lfloor j/b \rfloor + 1)b - 1$, so it is handy by definition. ◀

By Lemma 6, to get the desired approximation to LRRS it is enough to find the longest handy repeat in S. We use Weiner's algorithm [21] to maintain a dynamic suffix tree for Q_b which equals $\mathcal{T}(Q_b^j)$ after processing $S[j]$, for each $j \le n$. Processing a symbol $S[i]$, we add a new hashletter to the trace P_r, where $r = (i{+}1) \bmod b$, and find the longest handy repeat, the right occurrence of which is a suffix of P_r^i; we denote this suffix by $suff_r^i$. Since the repeat is handy, the last hashletter of the left occurrence is added to the tree before the first hashletter of the right occurrence becomes available. This condition implies that if the repeat is longer than one hashletter, it extends a handy repeat found on the iteration $i - b$.

▶ **Observation 7.** *By definition of a handy repeat, the condition $P_r^i = P_r^{i-1}$ implies $suff_r^i = suff_r^{i-1}$. Hence $suff_r^{i-1} = suff_r^{i-b}$ whenever $i + 1 \equiv_b r$. That is, the update of the main trace Q_b does not affect the longest handy repeat ending with a suffix of P_r^i.*

For each $r = 1, \dots, b$, we maintain the position of the string $suff_r^i$ in the tree $\mathcal{T}(Q_b^i)$. This position is denoted by pos_r^i.

▶ **Observation 8.** *The equality $suff_r^{i-1} = suff_r^{i-b}$ stated in Observation 7 does not necessarily imply $pos_r^{i-1} = pos_r^{i-b}$. Namely, these pairs are equal iff it is still valid to refer to the position of $w = suff_r^{i-b}$ in the tree $\mathcal{T}(Q_b^{i-1})$ with the pair $pos_r^{i-b} = (v, raise)$. However new nodes can appear during the last update of the tree, splitting the incoming edge of v such that the position of w will be above some ancestor of v. Still, if we go up by $raise$ symbols from the node v, we will reach the position of w.*

After reading $S[i]$, we need to compute pos_r^i, where $r = (i{+}1) \bmod b$. We may suppose that we have computed pos_r^{i-b} after reading $S[i - b]$. By Observation 8, one can get pos_r^{i-1} from $pos_r^{i-b} = (v, raise)$ walking up the tree from v to the lower end of the edge containing the position which is $raise$ symbols above v.

▶ **Proposition 9.** *If $i + 1 \equiv_b r$, the longest proper prefix of $suff_r^i$ is a suffix of $suff_r^{i-1}$.*

Proof. Let $(r, i_l, \frac{i-r+1}{b} - k + 1, k)$, $k \ge 1$, be the handy repeat corresponding to $suff_r^i$. The longest proper prefix of $suff_r^i$ corresponds to the handy repeat $(r, i_l + 1, \frac{i-r+1}{b} - k + 1, k - 1)$; its right occurrence ends at the position $\frac{i-r+1}{b} - 1$ and thus is a suffix of P_r^{i-b}. Then this occurrence is a suffix of $suff_r^{i-b}$ by definition. By Observation 7, $suff_r^{i-b} = suff_r^{i-1}$. ◀

By Proposition 9, we can find the position pos_r^i in the tree $\mathcal{T}(Q_b^i)$ as follows: take the longest suffix w of $suff_j^{i-1}$ such that its position in the tree has a transition by the hashletter $a = F^F[i{-}b{+}1..i]$, and follow this transition to get pos_r^i; if the root of $\mathcal{T}(Q_b^i)$ has no transition by a, pos_r^i coincides with the root. To find w, we scan suffixes of $suff_j^{i-1}$ in the order of

decreasing length, following suffix links. If the current position is on the edge (u, v), t symbols from its upper end u, then its implicit link can be found by descending t symbols from the node $link(u)$. Finally, if $i \overline{\bmod} b = b - 1$, a new hashletter is added to Q_b, and an iteration of Weiner's algorithm is run to update the suffix tree. Now we can formulate

▶ **Theorem 10.** *There is an algorithm solving, w.h.p.,* LRRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n)$ total time, and $\mathcal{O}(\frac{n}{E})$ update time.*

Proof. Let us fix $b = \lfloor \frac{E}{2} \rfloor$ and write $\mathcal{T}_j$ for the suffix tree $\mathcal{T}(Q_b^j)$. Here we present Algorithm 1 which finds, w.h.p., the longest handy repeat with the block size b in a length-n stream S within the space and time bounds stated in the theorem. Comparing the corresponding packable repeat with the longest palindrome in S found by [11, Algorithm A] and taking the longer of two, we obtain, w.h.p., the solution for LRRS(S) with the error, space, and time bounds as in the theorem; see Observations 1, 4, 5, and Lemma 6.

During the algorithm we maintain a suffix tree $\mathcal{T}$, an array $pos[1..b]$ of positions in $\mathcal{T}$, an array $SI[1..b]$ of recent frames and the last frame I. By *ith iteration* we mean all operations starting with the read of $S[i]$ and preceding the read of $S[i+1]$. After $(i-1)$th iteration, $\mathcal{T} = \mathcal{T}_{i-1}$, $pos[r]$ contains the latest computed value of pos_r^j (i.e., $j \geq i - b$ for each r), and SI contains the frames $I(j)$ for $j = i - b + 1, \ldots, i$ such that $I(j)$ is stored in $SI[(j-1) \overline{\bmod} b]$. The ith iteration looks as follows:

Algorithm 1 : Algorithm AdditiveReversedRepeat, i-th iteration ($i \geq b$).

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FF = F^F(i - b + 1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ by traversing $\mathcal{T}$ from pos_r^{i-1} by the hashletter FF
6: update *answer* by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: compute $FR = F^R(i - b + 1, i)$ from I and $SI[r]$
9: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR by Weiner's algorithm
10: $SI[r] = I$

The update of $pos[r]$ in lines 4-5 is correct by Observation 8 and Proposition 9. In order to update the answer in line 6, we store in each node of the suffix tree its string depth sd (the length of the corresponding string). Then the new value $(v, raise)$ of $pos[r]$ gives us the length of the corresponding handy repeat as $sd(v) - raise$. From the length and the number of the iteration we get the right occurrence of the repeat. To find the left occurrence, recall that the edge of a suffix tree with the lower end v is labeled by the last position of an occurrence of v in the string; subtracting $sd(v) - 1$ from this position, we get the first position of the left occurrence of the repeat. Thus Algorithm 1 correctly computes the longest handy repeat. Consider its time/space costs. The suffix tree is of size $\mathcal{O}(\frac{n}{E})$, both the array of positions and the array of frames are of size $\mathcal{O}(E)$, giving the required space bound.

We store transitions in the suffix tree in a single hash table with the pair (node, symbol) as the key, using the dynamic perfect hashing [10]. This method provides a dictionary with constant-time lookup; with probability at least $1 - \frac{1}{n}$ all updates work in constant time as well. If an update takes more than a prescribed constant time, we output a "repeat" $(1, 1, n)$ and stop. This results in an improbable error on the same side as the error caused by a hash collision. Thus the descents in the suffix tree require constant time per edge.

At an iteration, potentially heavy computations are in the lines 5 and 9. Each of them, in the worst case, requires the time proportional to the size of the suffix tree, i.e., $O(\frac{n}{E})$. Note that line 4 requires a constant number of operations because Weiner's algorithm adds to the tree at most one internal node per iteration.

To estimate the total time, count the total number of elementary operations in each of the lines 5 and 9 during all iterations. In line 9, a suffix tree is built for a string of length $\mathcal{O}(\frac{n}{E})$ over a polynomial alphabet (p is polynomial in n and thus in $\frac{n}{E}$). This can be done in $\mathcal{O}(\frac{n}{E})$ time using the hash table. Now consider line 5, grouping all iterations with the same trace P_r. Consider the evolution of the string depth of $pos[r]$. Initially, $sd = 0$. Each suffix link transition decreases it by 1; it can increase by 1 once per iteration, if the transition by the current symbol is found. Hence the total number of suffix link transitions is $\mathcal{O}(\frac{n}{E})$, constant time per each. Finally compute the number of descents which follow suffix link transitions. Consider the tree depth td of $pos[r]$. Initially, $td = 0$, and td is bounded by the depth of the tree, which is $\mathcal{O}(\frac{n}{E})$. A suffix link transition decreases td by at most 1; each ascent in line 4 decreases it by 1; updates of the tree do not decrease it. Since the total number of ascents in lines 4 and suffix link transitions in line 5 is $\mathcal{O}(\frac{n}{E})$, the total number of descents is $\mathcal{O}(\frac{n}{E})$ as well. Summing over all traces, we get the total time $\mathcal{O}(n)$, as required.

Thus, LRRS can be solved within the space/time costs stated in the theorem. ◀

Next we modify Algorithm 1 to avoid slow updates.

▶ **Theorem 11.** *There is an algorithm solving, w.h.p.,* LRRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n + \frac{n}{E}\log n)$ total time, and $\mathcal{O}(\log n)$ update time.*

Proof. Algorithm 1 has two heavy parts: updating the suffix tree and searching in it. For the first part, we replace Weiner's construction with its modification by Amir et al. [1]. This modification can work with the polynomial integer alphabet and has $O(\log n)$ update time and $O(n \log n)$ total time, thus adding $\frac{n}{E}\log n$ to the total time cost of our algorithm.

We should note that the algorithm by Amir et al. makes no use of suffix links; instead, the positions where to add new suffixes are determined through queries to the "balanced indexed structure" (BIS). BIS is capable of finding such a position in $\mathcal{O}(\log|\mathcal{T}|)$ time; in addition, it can find the longest common prefix of an arbitrary string with $\mathcal{T}$, also in $\mathcal{O}(\log|\mathcal{T}|)$ time. This allows us to build direct suffix links of new nodes within the same time bounds.

The search will use delayed operations. For each shift $r = 1, \ldots, b$ we maintain a "conveyor" C_r, consisting of the position $pos = pos[r]$ in the tree $\mathcal{T}$, a queue *queue* of hashletters awaiting processing (it can be viewed as a suffix of the string P_r), and a flag $flag$ set to 0 if the algorithm should skip the update of the answer with the current value of *pos*. One iteration of our algorithm is shown below.

A new hashletter is added to the current conveyor in line 3; the function $lazyProcess$, called in line 4, performs two delayed transitions in $\mathcal{T}$ (by suffix links or by queue symbols). If only one transition is available (by the only queue symbol), one transition is performed.

Let $C = C_r$ be the current conveyor, *maxlen* be the maximum string depth among all positions *pos* seen in C (= the length of the longest handy repeat with the right occurrence in P_r, found so far). After each call to $lazyProcess$, the following semi-invariant is kept: $balance = maxlen - sd(pos) - 2|queue| \geq 0$. Adding a symbol to *queue* decreases *balance* by 2; a suffix link transition increases it by 1. A transition by a symbol updates *pos* and so is followed by a deletion from *queue*; thus it increases *balance* by 1 or even by 2 (if the new *pos* changes *maxlen*). Thus if $lazyProcess$ performed two transitions, then *balance* has not decreased, and if there was one transition, then $|queue| = 0$ and $balance \geq 0$ by definition.

Algorithm 2 : FastAdditiveReversedRepeat, ith iteration ($i \geq b$).

$r = i \bmod b + 1$
read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
compute $FF = F^F(i-b+1, i)$ from I and $SI[r]$; put (C_r, FF)
$C_r = lazyProcess(C_r)$; $flag = [queue \text{ is empty}]$
if $flag = 1$ **then**
    update *answer* by value determined by C_r
if $r = b$ **then**
    compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
    update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
$SI[r] = I$

Because of the semi-invariant, $|queue| > 0$ implies $sd(pos) + |queue| < maxlen$. This means that after processing the whole queue both *maxlen* and the longest handy repeat cannot be updated. Hence at the end of an iteration $flag$ is set to 0 if $|queue| > 0$ and to 1 otherwise.

▶ **Observation 12.** *Between the iteration when a hashletter was added to a queue, and an iteration when it was processed to get pos, the suffix tree can be updated, and a wrong (too deep) pos can be found. However, while the queue is nonempty, the answer is not updated because $flag = 0$. When it becomes empty, the position pos will be correct, since the newest symbol in the queue appeared after the last update of $\mathcal{T}$.*

Observation 12 implies that Algorithm 2 performs exactly the same updates of the answer as Algorithm 1, and hence is correct. Now consider its time and space costs.

We cannot store *queue* explicitly due to memory restrictions. Instead, we factorize the string *queue* into a prefix (or head) *head* and a suffix (or tail) *tail*, storing them separately. We use four words of memory in total and support three operations in $\mathcal{O}(1)$ time: append a symbol to *tail*; delete the first symbol of *head*; set *head* to *tail* if *head* is empty. This provides full functionality of a queue; let us consider details.

The tail is stored as the position $posT = (v, raise)$ of the string *tail* in the tree $\mathcal{T}$. The head is represented by two numbers L and R such that the interval $[L..R]$ of the main trace Q_b equals *head*; recall that Q_b is a part of the suffix tree data structure. To delete the first hashletter from the queue, we increment L by 1. If now $L > R$ (= the head is empty), we compute the interval of Q_b equal to *tail*: if $posT = (v, raise)$, the incoming edge of v is marked by the last position x of an occurrence of v is Q_b, so $tail = Q_b[x-sd(v)+1..x-raise]$. Then we set $[L..R]$ to this interval and $posT$ to $(root, 0)$, thus moving *tail* to *head*.

To add a hashletter c to the queue, we try to make a transition by c from $posT$. If it succeeds, we just update $posT$ with the position reached. If it fails, the string $tail \cdot c$ does not occur in Q_b and we update the current conveyor as follows: $pos = posT$; $posT = (root, 0)$; $L = R$ points to an occurrence of c. If there is no such occurrence, we put $pos = posT = (root, 0)$; $L = R = -1$. The justification for such an update is Observation 12: since *maxlen* cannot be updated before the queue is emptied, we abandon the currently processed update of *pos* and skip some subsequent updates which cannot lead to the update of the answer (because $flag = 0$); then we use the tail (without the new symbol) to define the new value of $suff_r$.

Next we optimize the computations in the suffix tree using the dynamic weighted ancestor tree (DWAT) structure [17] for $\mathcal{T}$, which is maintained in parallel with $\mathcal{T}$; this tree has the size $\mathcal{O}(|\mathcal{T}|)$, and its update requires $\mathcal{O}(\frac{\log^2 \log |\mathcal{T}|}{\log \log \log |\mathcal{T}|})$ time per one update of the suffix tree.

DWAT was used to find implicit suffix links in the suffix tree for the order-preserving model [9] as follows: descend to the nearest node, follow its suffix link, and ascend using one $\mathcal{O}(\log\log|\mathcal{T}|)$-time query to DWAT. If we adopt this line, the total time spent by the algorithm will increase to $\Theta(n\log\log n)$. To avoid this, we exploit the following trick. First we try to find the implicit link by the usual procedure: ascend to a nearest node, follow suffix link, and descend the same distance. If after $\log\log|\mathcal{T}|$ operations we have not reached the destination edge, we abandon this try and find the link querying DWAT. This approach allows us to unite both bounds: $\mathcal{O}(\log\log n)$ update time from DWAT and $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}(\frac{n}{E})$ total time for each conveyor, which gives $O(n)$ in total for all conveyors.

With all these optimizations, Algorithm 2 works in the same space as Algorithm 1; the update time is dominated by $\mathcal{O}(\log n)$ for the suffix tree update (line 8), including the update of DWAT; *lazyProcess* (line 4) requires only $\mathcal{O}(\log\log n)$ time per iteration, as shown above.

The total time of the tree update (including DWAT) is $\mathcal{O}(\frac{n}{E}\log n)$; summing this with the bound $\mathcal{O}(n)$ for all conveyors, we get the result of the theorem. ◀

▶ Remark 13. The heaviest operation in Algorithm 2 is the update of the suffix tree. However, in the literature we found no algorithm which gives better worst-case update time and is suitable for a polynomial integer alphabet. E.g., the algorithm by Breslauer and Italiano [4] has $\mathcal{O}(\log\log n)$ update time, but only for constant alphabets, while the algorithm of Kopelowitz [16] is based on the y-fast trie [22], which has $\mathcal{O}(\log n)$ worst-case update time.

5 Search of a Longest Repeat

Now we approach LRS. A simple solution is very close to Algorithm 1; in fact it is easier, because we do not need to take a special care about overlapping repeats: a symbol in the right occurrence of a repeat becomes available later than its counterpart from the left occurrence no matter whether the repeat is overlapping or not. It is enough to make the following changes in Algorithm 1: (i) use P_b instead of Q_b, (ii) build suffix tree left to right by Ukkonen's algorithm [20], enhanced with the dynamic perfect hashing, instead of Weiner's algorithm (line 9; line 8 is no longer needed), and (iii) redefine suff_r^i to be the longest suffix of P_r^i occurring in P_b^{i-r}. This gives us

▶ **Theorem 14.** *There is an algorithm solving, w.h.p.,* LRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ working space, $\mathcal{O}(n)$ total time, and $\mathcal{O}(\frac{n}{E})$ update time.*

However, there is a problem with enhancing this algorithm to get better update time. Namely, each iteration of Weiner's algorithm produces exactly one leaf and at most one internal node, while in Ukkonen's algorithm the number of new nodes at one iteration can be linear in the size of the tree. Hence for some updates of the tree $\mathcal{T}$ the time $\Theta(|\mathcal{T}|)$ can be inevitable (postponed updates may affect the search results). Instead, we work with Q_b and Q_r, comparing reversed strings of "reversed" hashletters. This allows us to stick to Weiner's construction and prove an analog of Theorem 11.

▶ **Theorem 15.** *There is an algorithm solving, w.h.p.,* LRS *in a stream with a given additive error $E = \mathcal{O}(n^{0.99})$ in $\mathcal{O}(\frac{n}{E} + E)$ space, $\mathcal{O}(n + \frac{n}{E}\log n)$ total time, and $\mathcal{O}(\log n)$ update time.*

We define pref_r^i be the longest prefix of Q_r^i occurring in Q_b^{i-r}, and redefine pos_r^i to be the position of pref_r^i in the current tree $\mathcal{T}(Q_b^i)$. Similar to Observation 7, we get

▶ **Observation 16.** *One has $\mathit{pref}_r^{i-1} = \mathit{pref}_r^{i-b}$ whenever $i + 1 \equiv_b r$.*

As with reversed repeats, after reading $S[i]$ we take $r = (i+1) \overline{\bmod}\, b$ and compute pos_r^{i-1} from $pos_r^{i-b} = (v, raise)$ walking up the tree until the edge with the position $raise$ symbols above v will be found. After this we compute pos_r^i, using a direct analog of Proposition 9.

▶ **Proposition 17.** *The longest proper suffix of $pref_r^i$ is a prefix of $pref_r^{i-1}$.*

Proof of Theorem 15. We define the block size $b = \lfloor \frac{E}{2} \rfloor$ and write $\mathcal{T}_j$ for the suffix tree $\mathcal{T}(Q_b^j)$. Algorithm 3 is presented below.

Algorithm 3 : FastAdditiveRepeat, ith iteration ($i \geq b$).

1: $r = i \bmod b + 1$
2: read $S[i]$; compute $I(i+1)$ from $I(i)$; $I = I(i+1)$
3: compute $FR = F^R(i-b+1, i)$ from I and $SI[r]$
4: update $pos[r] = pos_r^{i-b}$ to pos_r^{i-1} by walking up
5: compute $pos[r] = pos_r^i$ as longest prefix of $FR{\cdot}pref_r^{i-1}$ that exists in $\mathcal{T}$
6: update $answer$ by the value determined by $pos[r]$
7: **if** $r = b$ **then**
8: update $\mathcal{T}$ to $\mathcal{T}_i$, appending FR and building suffix links using BIS
9: $SI[r] = I$

Line 4 requires $\mathcal{O}(1)$ time, since only one internal node can appear in $\mathcal{T}$ between the corresponding iterations; line 8 requires $\mathcal{O}(\log n)$ time per iteration and thus $\mathcal{O}(\frac{n}{E} \log n)$ in total (see Algorithm 2). It remains to explain the computation in line 5. As in Theorem 11, we use two different ways to benefit from both good total time and good update time.

One way is to use *hard inverse links*, which are inverses of suffix links (i.e., link a node u to all nodes of the form au where a is a symbol). Clearly, they can be added to the tree simultaneously with suffix links and stored in a hash table similar to the one used for transitions by letters. To compute pos_r^i, one walks up the tree from the position pos_r^{i-1} until a node with a defined hard inverse link by the hashletter FR is reached, follows this link, and possibly walks some letters down inside one edge (however, the number of operations needed to compute the length of descent equals, in the worst case, the number of nodes passed on ascent). Note that one step up decreases the tree depth of a node by 1, the transition by hard link cannot increase it more than by 1, and the descent inside one edge does not affect it. Thus the total number of steps up in processing of a fixed trace is $O(|\mathcal{T}|)$. This gives us $\mathcal{O}(n)$ total time for all computations in line 5.

The second way is to use a one $\mathcal{O}(\log n)$-time query to BIS. Finally, the solution is to perform $\log n$ steps of the search in the tree, and if the requires position is not found, query BIS. This gives us both $\mathcal{O}(n)$ total time and $\mathcal{O}(\log n)$ update time. ◀

6 Lower Bounds

In this section we use Yao's minimax principle [23] to show the lower space bounds to the LRS problem, where the input length n and the input alphabet Σ are given; we use the notation $LRS_{|\Sigma|}[n]$. For LRRS, the same results can be obtained by straightforward adaptations of theorems from [11, Section 2], so we omit them. For LRS, the proofs of Lemma 20 and Theorem 22 also follow the scheme from [11], while Theorem 19 and Lemma 21 are proved by different arguments.

▶ **Theorem 18** (Yao's principle). *Let $\mathcal{X}$ be the set of inputs for a problem, $\mathcal{A}$ be the set of all deterministic algorithms solving it, and $c(a,x) \geq 0$ be the cost of running $a \in \mathcal{A}$ on $x \in \mathcal{X}$.*

Let p be a probability distribution over $\mathcal{A}$, and let A be an algorithm chosen at random according to p. Let q be a probability distribution over $\mathcal{X}$, and let X be an input chosen at random according to q. Then $\max_{x\in\mathcal{X}} \mathbf{E}[c(A,x)] \geq \min_{a\in\mathcal{A}} \mathbf{E}[c(a,X)]$.

First we prove that any Las Vegas online algorithm solving LRS with a given additive error needs at least linear space and thus cannot be used for streaming. For Las Vegas algorithms, the class $\mathcal{A}$ consists of all correct algorithms, and $c(a,x)$ is the memory usage.

▶ **Theorem 19.** *Let* A *be a Las Vegas streaming algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with an additive error $E \leq 0.49n$ using $s(n)$ bits of memory. Then* $\mathbf{E}[s(n)] = \Omega(n \log |\Sigma|)$.

Proof. We give the proof for $\Sigma = \{0,1\}$ to simplify computations. It can be extended for arbitrary alphabet by encoding letters by binary strings of uniform length. Let $\mathcal{P}$ be the uniform distribution over all strings of even length n. Consider an arbitrary deterministic algorithm D, solving LRS with an additive error E. An input z is "good" if D spends at most $0.004n$ bits of memory on it, and "bad" otherwise. First assume that at least half of inputs are good. Then good inputs cover at least half, or $2^{0.5n-1}$, possible prefixes of length $n/2$ of inputs. After reading such a prefix of a good input, D is in one of at most $2^{0.004n}$ states. Then we can choose a class C, containing at least $2^{0.496n-1}$ prefixes sharing the same state of D. Note that there exist at most $n^2 \cdot 2^{0.495n}$ prefixes which contain a repeat of length $0.005n$. Further, any input string xx has a common substring of length $0.005n$ with at most $n^2 \cdot 2^{0.495n}$ strings of length $n/2$. Hence for n big enough, C contains two strings x and y such that neither of them contains a repeat of length $0.005n$ and xx has no common substring of length $0.005n$ with y. Then the length of the longest repeated substring in xx is $0.5n$, while for yx it is less than $2 \cdot 0.005n = 0.01n$. Since D is in the same state after reading x and y, it gives the same answer for xx and yx; one of the answers must be erroneous.

Thus our assumption about good inputs was wrong, and at least half of the inputs are bad. Then the expected memory usage of D is at least $0.004n/2 = \Omega(n \log |\Sigma|)$ bits. The reference to Theorem 18 finishes the proof. ◀

For Monte Carlo algorithms, $\mathcal{A}$ consists of all (non necessarily correct) algorithms, and $c(a,x)$ is the correctness indicator (0 if correct, 1 otherwise). First we show that the exact answer to LRS cannot be found using sublinear memory.

▶ **Lemma 20.** *There is a constant γ such that any Monte Carlo online algorithm solving* $\mathsf{LRS}_{\Sigma}[n]$ *exactly with probability $1 - \frac{1}{n}$ uses at least $\gamma n \log \min\{|\Sigma|, n\}$ bits of memory.*

The proof is similar to the proof of [11, Lemma 3] and can be found in Appendix.

For Monte Carlo algorithms with additive error, we first prove an auxiliary rough bound and then a sharp bound, using a reduction from the exact Monte Carlo algorithm.

▶ **Lemma 21.** *Any randomized Monte Carlo algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with the additive error $E \leq 0.49n$ and the error probability $\frac{1}{n}$ uses $\Omega(\log n)$ bits of memory.*

Proof. Let $\Sigma = \{0,1\}$ and consider the uniform distribution $\mathcal{P}$ over any set P of $2^{n/2+1}$ strings of even length n with the following property: each string $x \in \Sigma^{n/2}$ occurs twice as the left half of a string from P such that one of these strings is xx and the other is xy where y is a fully randomly chosen string of length $n/2$. Thus, half of strings in P have repeats of length $n/2$, and the other strings are fully random and thus their longest repeats are of length $\Theta(\log n)$ with probability $1 - \frac{1}{n}$ (cf. [11, Lemma 22]). Hence if an algorithm stores

any sketch of $s(n) = o(\log n)$ bits of information about x, it distinguishes between x and y in the second part with a too small probability: y shares this sketch with x with probability $2^{-s(n)} > \frac{1}{n}$. ◀

▶ **Theorem 22** (Monte Carlo additive approximation). *Any randomized streaming Monte Carlo algorithm solving the problem* $\mathsf{LRS}_{|\Sigma|}[n]$ *with the additive error* $E \leq 0.49n$ *with probability* $1 - \frac{1}{n}$ *uses* $\Omega(\frac{n}{E} \log \min\{|\Sigma|, \frac{n}{E}\})$ *bits of memory.*

Proof. Let $\sigma = \min\{|\Sigma|, \frac{n}{E}\}$. Since the bound $\Omega(\frac{n}{E} \log \sigma)$ becomes $\Omega(\log n)$ if $E = \Omega(\frac{n \log \sigma}{\log n})$, we further suppose that $E = o(\frac{n \log \sigma}{\log n})$.

Assume that there is a Monte Carlo streaming algorithm A solving $\mathsf{LRS}_{|\Sigma|}[n]$ with additive error E with probability $1 - \frac{1}{n}$, using $o(\frac{n}{E} \log \sigma)$ bits of memory. Let $n' = \lfloor \frac{n-E}{E+1} \rfloor$. We define new Monte Carlo streaming Algorithm A', which processes a string $x[1..n']$ as follows: run Algorithm A on $x' = 0^E x[1] 0^E x[2] \cdots 0^E x[n'] 0^E$, using $\log E \leq \log n$ additional bits of memory to count to E, get an answer R, and return the number $\lfloor \frac{R}{E+1} \rfloor$. If the longest repeat in x has length r, the longest repeat in x' has length $(r+1)E + r$ (each occurrence consists of r letters of x and $r+1$ blocks of 0's). Since Algorithm A has additive error E, one gets $r(E+1) \leq R \leq (r+1)E + r$, so A' must return r. Hence Algorithm A' solves $\mathsf{LRS}_{|\Sigma|}[n']$ exactly with probability $1 - \frac{1}{n} \geq 1 - \frac{1}{n'}$ using $o(n' \log \sigma) + \log n$ bits of memory. By the above assumption on E, this number is $o(n' \log \sigma)$, contradicting Lemma 20, which requires the memory usage for exact LRS to be $\Omega(n' \log \sigma)$. ◀

7 Conclusion

In this paper, two classical string problems (LRS and LRRS) are considered in the streaming model. We proved that they can be solved only by Monte Carlo approximation algorithms; for the additive approximation error, we presented efficient algorithms which are space optimal whenever $E = \mathcal{O}(\sqrt{n})$ and $|\Sigma| = \Omega(n^{0.01})$. The algorithms are based on suffix trees, which is rather exotic for the streaming model.

Two intriguing open problems about LRS and LRRS streaming solutions are the existence of algorithms with the additive error $E > \sqrt{n}$ within $o(\sqrt{n})$ memory and the existence of efficient algorithms with multiplicative error. We also note that an efficient solution of the LRRS problem can be important for solving the problem of the longest gapped palindrome.

References

1 Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *International Symposium on String Processing and Information Retrieval*, pages 67–78. Springer, 2005.

2 Alberto Apostolico, Costas Iliopoulos, Gad M. Landau, Baruch Schieber, and Uzi Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(1-4):347–365, 1988.

3 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In *Combinatorial Pattern Matching*, volume 6661 of *LNCS*, pages 162–172, Berlin, 2011. Springer.

4 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *Journal of Discrete Algorithms*, 18:32–48, 2013.

5 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In *Algorithms-ESA 2015*, pages 361–372. Springer, 2015.

6 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 2039–2052. Society for Industrial and Applied Mathematics, 2016.

7 Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1106–1125. SIAM, 2019.

8 Raphaël Clifford and Tatiana Starikovskaya. Approximate Hamming distance in a stream. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *LIPIcs*, pages 20:1–20:14, 2016.

9 Maxime Crochemore, Costas S Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016.

10 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Dynamic hashing in real time. In *Informatik*, pages 95–119. Springer, 1992.

11 Paweł Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemysław Uznański. Tight Tradeoffs for Real-Time Approximation of Longest Palindromes in Streams. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, pages 18:1–18:13, 2016.

12 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

13 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In *25th Annual European Symposium on Algorithms, ESA 2017*, volume 87 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

14 Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming for aibohphobes: Longest palindrome with mismatches. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017*, volume 93 of *LIPIcs*, pages 31:1–31:13, 2017.

15 Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

16 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 283–292. IEEE, 2012.

17 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574. Society for Industrial and Applied Mathematics, 2007.

18 Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 315–323. IEEE, 2009.

19 Jakub Radoszewski and Tatiana A. Starikovskaya. Streaming K-Mismatch with Error Correcting and Applications. In *2017 Data Compression Conference, DCC 2017*, pages 290–299. IEEE, 2017.

20 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

21 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

22 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–84, 1983.

23 Andrew Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

A Appendix

Proof of Lemma 20. We use the auxiliary problem $\mathsf{LCP}_\Sigma[n]$ which is to find the longest common prefix of left and right halves of the input. First we prove that if A is a Monte Carlo online algorithm solving $\mathsf{LCP}_\Sigma[n]$ exactly using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, then its error probability is at least $\frac{1}{n|\Sigma|}$.

By Theorem 18, it is enough to construct probability distribution $\mathcal{P}$ over Σ^n such that for any deterministic algorithm D using less than $\lfloor \frac{n}{2} \log |\Sigma| \rfloor$ bits of memory, the expected probability of error on a string chosen according to $\mathcal{P}$ is $\geq \frac{1}{n|\Sigma|}$. To do this, let $n' = \frac{n}{2}$. For any $x \in \Sigma^{n'}$, $k = 1, \ldots, n'$, $c \in \Sigma$, let $w(x,k,c) = x[1..n']x[1..k{-}1]cx[k{+}1..n']$. Now $\mathcal{P}$ is the uniform distribution over all such $w(x,k,c)$. Choose an arbitrary maximal matching of strings from $\Sigma^{n'}$ into pairs (x,x') such that D is in the same state after reading either x or x'. At most one string per state of D is left unpaired, that is at most $2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ strings in total. Since there are $|\Sigma|^{n'} = 2^{n' \log |\Sigma|} \geq 2 \cdot 2^{\lfloor \frac{n}{2} \log |\Sigma| \rfloor - 1}$ possible strings of length n', at least half of the strings are paired. Let s be the longest common prefix of x and x', so $x = scv$, $x' = sc'v'$, where $c \neq c'$ are distinct letters. Then D returns the same answer on $w(x,|s|,c)$ and $w(x',|s|,c)$, although $\mathsf{LCP} = |s|$ in one case and $\mathsf{LCP} \geq |s|+1$ in the other. Similarly, D errs on either $w(x,|s|,c')$ or $w(x',|s|,c')$. Thus the error probability is at least $\frac{1}{2n'|\Sigma|} = \frac{1}{n|\Sigma|}$.

Now we prove the lemma for $LCP_\Sigma[n]$ with an amplification trick. Assume we have a Monte Carlo streaming algorithm, which solves $LCP_\Sigma[n]$ exactly with error probability ε using $s(n)$ bits of memory. Then we can run its k instances simultaneously and return the most frequent answer. The new algorithm uses $\mathcal{O}(k \cdot s(n))$ bits of memory and its error probability ε_k satisfies the inequality $\varepsilon_k \leq \sum_{2i<k} \binom{k}{i} (1-\varepsilon)^i \varepsilon^{k-i} \leq 2^k \cdot \varepsilon^{k/2} = (4\varepsilon)^{k/2}$. Let $\kappa = \frac{1}{6} \frac{\log(4/n)}{\log(1/(n|\Sigma|))}$, so $\kappa = \frac{1}{6} \frac{1-o(1)}{1+\log|\Sigma|/\log n} = \Theta\left(\frac{\log n}{\log n + \log|\Sigma|}\right) = \gamma \cdot \frac{1}{\log|\Sigma|} \log\min\{|\Sigma|, n\}$ for some constant γ. Assume that A uses less than $\kappa \cdot n \log|\Sigma| = \gamma \cdot n \log\min\{|\Sigma|, n\}$ bits of memory. Then running $\lfloor \frac{1}{2\kappa} \rfloor \geq \frac{3}{4} \frac{1}{2\kappa}$ (which holds since $\kappa < \frac{1}{6}$) instances of A in parallel requires less than $\lfloor \frac{n}{2} \log|\Sigma| \rfloor$ bits of memory. But then the error probability of the new algorithm is bounded from above by $\left(\frac{4}{n}\right)^{3/16\kappa} = \left(\frac{1}{n|\Sigma|}\right)^{18/16} \leq \frac{1}{n|\Sigma|}$, which we have already shown to be impossible.

The lower bound for LCP can be converted into a lower bound for solving LRS exactly by padding the input so that the longest repeat is the common prefix of the whole string and its right half. Let $x = x[1..n]$ be the input for $\mathsf{LCP}_{|\Sigma|}[n]$, with the answer k. We define $w(x) = 0^n 1 x[1..\frac{n}{2}] 0^n 1 x[\frac{n}{2}+1..n]$, where $0, 1 \notin \Sigma$; clearly, $w(x)$ contains a palindrome of length at least $n+k+2$. On the other hand, any repeated substring of $w(x)$ of length $\geq n{+}1$ must contain the substring $0^{n/2}1$, which has just two occurrences in w. Thus we have reduced solving $\mathsf{LCP}_{|\Sigma|}[n]$ to solving $\mathsf{LRS}_{|\Sigma|}[3n+2]$. We already know that solving $\mathsf{LCP}[n]$ with probability $1 - \frac{1}{n}$ requires $\gamma \cdot n \log\min\{|\Sigma|, n\}$ bits of memory, so solving $LRS_{|\Sigma|}[3n+2]$ with probability $1 - \frac{1}{3n+2} \geq 1 - \frac{1}{n}$ requires $\gamma \cdot n \log\{|\Sigma|, n\} \geq \gamma' \cdot (3n+2) \log\min\{|\Sigma|, 3n+2\}$ bits of memory. The reduction needs $\mathcal{O}(\log n)$ additional bits of memory to count up to n, but for large n this is much smaller than the lower bound if we choose $\gamma' < \frac{\gamma}{4}$. ◀

Computing the Antiperiod(s) of a String

Hayam Alamro
Department of Informatics, King's College London, UK
Department of Information Systems, Princess Nourah bint Abulrahman University, Riyadh, KSA
hayam.alamro@kcl.ac.uk

Golnaz Badkobeh
Department of Computing, Goldsmiths, University of London, UK
g.badkobeh@gold.ac.uk

Djamal Belazzougui
Centre de Recherche sur I'nformation Scientifique et Technique, Algeria
dbelazzougui@cerist.dz

Costas S. Iliopoulos
Department of Informatics, King's College London, UK
costas.iliopoulos@kcl.ac.uk

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Abstract

A string $S[1, n]$ is a *power* (or repetition or tandem repeat) of order k and period n/k, if it can be decomposed into k consecutive identical blocks of length n/k. Powers and periods are fundamental structures in the study of strings and algorithms to compute them efficiently have been widely studied. Recently, Fici et al. (Proc. ICALP 2016) introduced an *antipower* of order k to be a string composed of k distinct blocks of the same length, n/k, called the antiperiod. An arbitrary string will have antiperiod t if it is prefix of an antipower with antiperiod t. In this paper, we describe efficient algorithm for computing the smallest antiperiod of a string S of length n in $O(n)$ time. We also describe an algorithm to compute all the antiperiods of S that runs in $O(n \log n)$ time.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases antiperiod, antipower, power, period, repetition, run, string

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.32

Acknowledgements This work was supported by the Academy of Finland under grant 319454. We thank the anonymous reviewers for their detailed comments that greatly improved the quality of this article, in particular for the improvement in Section 5.2 and for pointers on previous works on weighted-level ancestors data structures.

1 Introduction

Algorithms and data structures for finding repeating patterns or *regularities* in strings (see, e.g., [6, 16, 24]) are central to several fields of computer science including computational biology, pattern matching, data compression, and randomness testing. The nature and extent of regularity in strings is also of immense combinatorial interest in its own right [22].

One of the most fundamental notions of regularity is that of powers (also known as repetitions or tandem repeats). A power of order k is defined by a concatenation of k identical blocks of symbols, where k is at least 2. The study of powers began in the early 1900s with the work of Thue [25], who studied a class of strings that do not contain any substrings that are powers. Powers in various forms later came to be important structures in computational biology, where they are associated with various regulatory mechanisms

and play an important role in genomic fingerprinting (for further reading see, e.g., [19] and references therein). More recently it has been shown that the number of so-called *maximal* powers (or runs) in a string is less than the length of the string itself [2] – positively settling a long-standing conjecture [20, 5].

Antipowers are an orthogonal notion to that of powers, that were introduced recently by Fici et al. in [9, 10]. In contrast to powers, antipowers insist instead on the diversity of consecutive blocks: an antipower of order k is a concatenation of k pairwise distinct strings of equal length.

In the short period of time following Fici et al.'s work, antipowers have received a great deal of attention, especially from a combinatorial perspective. In a follow-up article, Fici et al. [10], further showed that every infinite string contains powers of any order or antipowers of any order. Moreover, Defant in [7] (see also Narayanan [23]) studied the sequence of lengths of the shortest prefixes of the Thue-Morse string that are k-antipowers, and proved that this sequence grows linearly in k. Gaetz [12] has since extended Defant's results to substrings. Burcroff [4] studied the avoidability of k antipowers in infintite strings, generalizing Fici et al.'s results.

The algorithmic study of antipowers was initiated recently by Badkobeh et al. [1], who describe an algorithm that, given a string S of length n and a parameter k, locates all substrings of S that are antipowers of order k. The algorithm takes $\Theta(n^2/k)$ time, which the authors show to be optimal in the sense that there exist strings containing $\Theta(n^2/k)$ distinct antipowers of order k. Very recently Kociumaka et al. [18] have shown an output sensitive algorithm for the same problem with running time $O(nk \log k + C)$, where C is the number of reported antipowers. They also show that they can be counted within time $O(nk \log k)$.

New Results. In this paper, we describe algorithms for computing the smallest antiperiod and all the antiperiods of a string S of length n. Our definition of antiperiod is slightly more general than that considered by Badkobeh et al., and we define it formally in the next section. The starting point for both our algorithms is an efficient solution to the monotone weighted level ancestor problem, which we describe in Section 3. This leads to an $O(n \log n)$ time algorithm for computing all the antiperiods of a given string, described in Section 4. We then show how to exploit combinatorial properties of antipowers to obtain an algorithm for computing the smallest antiperiod, of S in time $O(n)$. We notice that a very similar result to our monotone weighted level ancestor solution was already shown by Kociumaka et al. [17] (see also Barton et al. [3]). Such a result was unknown to us prior to the initial submission of this paper.

2 Preliminaries

Let $S = S[1, n]$ be a string of length $|S| = n$ over an alphabet Σ of size $|\Sigma| = \sigma$. The empty string ε is the string of length 0. For $1 \leq i \leq j \leq n$, $S[i]$ denotes the ith symbol of S, and $S[i, j]$ the contiguous sequence of symbols (called *factor* or *substring*) $S[i]S[i + 1] \dots S[j]$. A substring $S[i, j]$ is a suffix of S if $j = n$ and it is a prefix of S if $i = 1$. A string p is a *repeat* of S iff p has at least two occurrences in S. In addition p is said to be *right-maximal* in S iff there exist two positions $i < j$ such that $S[i, i + |p| - 1] = S[j, j + |p| - 1] = p$ and either $j + |p| = n + 1$ or $S[i, i + |p|] \neq S[j, j + |p|]$.

The *suffix tree* T for a string S of length n over the alphabet Σ is a rooted directed compacted trie built on the set of suffixes of S. The suffix tree has n leaves and its internal nodes have at least two children whiles its edges are labelled with substrings of S. The labels

of all outgoing edges from a given node start with a different character. All leaves of the suffix tree are labelled with an integer i, where $i \in \{1 \cdot \cdot n\}$ and the concatenation of the labels on the edges from the root to the leaf gives us the suffix of S which starts at position i. The nodes of the (non-compacted) trie which have branching nodes and leaves of the tree are called *explicit nodes*, while the others are called *implicit nodes*. The occurrence of a substring P in S is represented on T by either an explicit node or implicit node and called the *locus* of P. The *suffix tree* T can be constructed in $O(n)$ time and space. In order to have one-to-one correspondence between the suffixes of S and the leaves of T, a character $\$ \notin \Sigma$ is added to the end of the label edge for each suffix i to ensure that no suffix is a prefix of another suffix. To each node α in T is also associated an interval of leaves $[i..j]$, where $[i..j]$ is the set of labels of the leaves that have α as an ancestor (or the interval $[i..i]$ if α is a leaf labelled by i). The intervals associated with the children of α (if α is an internal node) form a partition of the interval associated with α (the intervals are disjoints sub-intervals of $[i..j]$ and their union equals $[i..j]$). For any internal node α in the *suffix tree* T, the concatenation of all edge labels in the path from the root to the node α is denoted by $\bar{\alpha}$ and the string depth of a node α is denoted by $|\bar{\alpha}|$. For brevity, in what follows, we will always talk about depth instead of string depth [1].

A *power of order* k (or *k-power*) is a string that is the concatenation of k identical strings. The *period* of a k-power of length n is n/k. For example, *abaaba* is a 2-power (or a *square*) of period 3.

Our main subject in this paper is a complementary structure to a power, called an antipower, which we now formally define.

▶ **Definition 1** (Fici et al. [9]). *An* antipower *of order k (or k-*antipower*) is a string obtained by the concatenation of k pairwise-distinct strings of identical length. The* antiperiod *of a k-antipower of length n is n/k.*

For example, *ababbabab* is a 3-antipower of antiperiod 3.

We now extend the notion of antiperiod to strings that are not necessarily antipowers.

▶ **Definition 2.** *A string s has an antiperiod t if it is a prefix of some k-antipower $w = p_1 p_2 \cdots p_k$ whose antiperiod is t.*

Note that by this definition every string has antiperiods: $\lfloor n/2 \rfloor + 1, \ldots, n-1$.

▶ **Example 3.** Suppose we have the following string:

$s = aabbbbaaaabb |s| = 12$

$t = 6$: s is a 2-antipower with antiperiod 6. $s = (aabbbb)(aaaabb)$;
$t = 5$: s is a prefix of a word, w, that is a 3-antipower with antiperiod 5:
$w = (aabbb)(baaaa)(bbaaa)$;
$t = 4$: s is not a 3-antipower because $s = (aabb)(bbaa)(aabb)$ and it therefore cannot be a prefix of any antipower with antiperiod 4;
$t = 3$: s is a 4-antipower with antiperiod 3. $s = (aab)(bbb)(aaa)(abb)$;
$t = 2$: s is not a 6-antipower because $s = (aa)(bb)(bb)(aa)(aa)(bb)$ and so cannot be a prefix of any antipower with antiperiod 2.
Therefore, the smallest antiperiod of the string s is 3.

[1] Notice that string depth of a suffix tree node is different from the usual definition of node depth in arbitrary trees which refers to the number of nodes in the path from the root to the reffered node.

▶ **Example 4.** Suppose we have the following string:

$s = ababbbaaaaabaa|s| = 14$

$t = 7$: s is a 2-antipower with antiperiod 7. $s = (ababbba)(aaaabaa)$;
$t = 6$: s is a prefix of a 3-antipower, w with antiperiod 6. $w = (ababbb)(aaaaab)(aaaaaa)$;
$t = 5$: s is a prefix of a 3-antipower, w with antiperiod 5. $w = (ababb)(baaaa)(abaaa)$;
$t = 4$: s is a prefix of a 4-antipower, w with antiperiod 4. $w = (abab)(bbaa)(aaab)(aaaa)$;
$t = 3$: s is a prefix of a 5-antipower, w with antiperiod 3. $w = (aba)(bbb)(aaa)(aab)(aac)$;
$t = 2$: s is not a 7-antipower. $s = (ab)(ab)(bb)(aa)(aa)(ab)(aa)$.
Therefore, the smallest antiperiod of the string s is 3.

We study two problems in this paper related to the computation of antiperiods.

▶ **Problem 1.** *Given a string S, find the smallest antiperiod of S.*

▶ **Problem 2.** *Given a string S, find all the antiperiods of S.*

In what follows we denote by $\log x$ the function $\log_2 x$ and by $\ln(x)$ the natural logarithm of x. We will also make use of the iterated logarithm function. We denote by $\log^{(1)}(x) = \log(x)$ and for integer $k > 1$ define $\log^{(k)}(x) = \log(\log^{(k-1)}(x))$. We let $k = \log^*(x)$ be the number k such that $\log^{(k)}(x) \leq 1$.

3 Monotone Weighted Level Ancestor Queries

In this section, we are interested in weighted level ancestor queries over leaves in the suffix tree. We will use such queries in subsequent sections to obtain efficient algorithms for computing antiperiods.

An internal node α *spans* depth d iff the depth of α is at least d and *depth* of the parent of α is less than d. A leaf ℓ spans depth d iff the parent of ℓ has depth less than d.

Abusing notation we will identify each leaf of T by its label. A weighted level ancestor query consists of a pair (ℓ, d), where ℓ is a leaf in the suffix tree and d is a depth. The goal is to return the highest ancestor of ℓ that has depth at least d. Although there exist some solutions to this problem, none of them is satisfactory for our purpose. The solution described in [15] supports constant time queries, but does not have an efficient construction algorithm[2]. All the other solutions to this problem have linear preprocessing time, but query time $O(\log \log n)$ [8, 21]. Here we will describe a solution that can be used to efficiently answer *sequences* of weighted-level ancestor queries as long as the depth argument of the queries in the sequence is non-decreasing.

One of the most frequent applications of weighted level ancestor queries is the determination of the locus of substrings of S. More precisely, the locus of a substring $S[i, j]$ can be determined as follows: first determine the leaf ℓ that corresponds to suffix $S[i, n]$. Then, the locus of $S[i, j]$ is obtained by issuing a weighted level ancestor query with pair $(\ell, j - i + 1)$.

▶ **Definition 5.** *A sequence of weighted-level ancestor queries $(\ell_1, d_1), (\ell_2, d_2) \ldots (\ell_t, d_t)$ is called monotone, iff we have that $d_i \leq d_{i+1}$ for all $i \in [1..t - 1]$.*

▶ **Definition 6.** *A split-find data structure is a data structure over the interval $[1..n]$ that starts with a set of disjoint non-empty sub-intervals of $[1..n]$ whose union equals $[1..n]$ (the*

[2] Although not specifically analysed in [15], preprocessing time is $O(n \log^4(n))$ [14].

subintervals form a partition of $[1..n]$*) and in which the query* `find`(k) *returns the only subinterval* $[i..j]$ *such that* $i \leq k \leq j$*, and the update query* `split`$([i..j], k)$ *with* $k \in [i..j-1]$ *removes the interval* $[i..j]$ *and insert intervals* $[i..k]$ *and* $[k+1..j]$.

Before describing our proposed solution, we will start by stating some useful lemmas.

▶ **Lemma 7.** *A node alpha associated with an interval* $[i_\alpha, j_\alpha]$ *is an ancestor of a node* β *associated with interval* $[i_\beta, j_\beta]$ *iff* $[i_\beta, j_\beta] \subseteq [i_\alpha, j_\alpha]$

Proof. The lemma is immediate from the definition of the suffix tree and the definition of the intervals associated with the suffix tree nodes. ◀

▶ **Lemma 8.** *The answer to a weighted-level ancestor query* (ℓ, d) *is a node* α *that spans depth* d *and has an associated interval* $[i..j]$ *such that* $\ell \in [i..j]$.

Proof. The fact that α needs to span d is immediate from the definition of the query. The fact that $[i..j]$ needs to include ℓ follows from the definition of the query and from Lemma 7. ◀

▶ **Lemma 9.** *The collection of intervals associated with the set of nodes that span depth* d *forms a partition of the interval* $[1, n]$

Proof. We will first prove that the intervals are disjoint and then prove that their union equals $[1, n]$. We will prove that by showing that every leaf is included in exactly one interval that spans depth d. Suppose that a leaf ℓ is included in two intervals $[i_\alpha, j_\alpha]$ and $[i_\beta, j_\beta]$, then one of the nodes α or β is an ancestor of the other (by Lemma 7), which means that it has depth less than d thus does not span d. We will now prove that ℓ is included in at least one interval. Suppose that the parent of ℓ has stringdepth less than d. Then, ℓ spans d and thus the interval $[\ell, \ell]$ is in the collection. Otherwise, clearly one of the ancestors of ℓ spans d and its corresponding interval includes ℓ. ◀

We will now prove the following lemma.

▶ **Lemma 10.** *The collection of intervals corresponding to nodes that span depth* $d+1$ *can be obtained from the collection of intervals that corresponds to nodes that span depth* d *by replacing the intervals that correspond to internal node at depth* d *with intervals of their children.*

Proof. First of all, observe that each node that spans depth $d+1$ either spans depth d or has a parent that spans depth d. Thus generating all nodes that span depth $d+1$ amounts to consider all (intervals of) nodes that span depth d and for each such node either add it (its interval) to the output set or add (intervals of) its children. We will now look at all nodes that span depth d. Observe that a node α that spans depth d will have depth at least d. Consider now two cases:

1. If α has depth more than d, then it will clearly also span depth $d+1$, since it obviously has depth at least $d+1$ and its parent has depth less than d and thus less than $d+1$. It thus suffices to keep the interval that corresponds to α.
2. If α has depth exactly d, then the depth of all its children is at least $d+1$. Those children will obviously span d, since their depth is at least $d+1$ and the depth of their parent α is $d < d+1$. It thus suffices to replace the interval of α with the intervals of its children. ◀

We are now ready to prove the main theorem of this section.

▶ **Theorem 11.** *Suppose that we have a split-find algorithm over n elements in the interval $[1..n]$ that has initialization time $O(n)$ and that supports the* `find` *operation in constant time and any sequence of k* `split` *operations in amortized $O(1 + \frac{n}{k})$ time per operation. Then we can use such an algorithm to support any sequence of k monotone weighted-level ancestor queries over a suffix tree with n leaves in amortized $O(1 + \frac{n}{k})$ time per query.*

Proof. We assume depth $d_1 \geq 1$ (level ancestor queries for depth 0 are trivial – the returned node is the root). Each node α in the suffix tree has a depth d and an associated interval $[i..j]$, where $[i..j]$ is the set of leaves that have α as an ancestor. We initialize the weighted level ancestor query data structure in time $O(n)$ with the intervals associated with the children of the root. We will use an auxiliary table $P[1..n]$ which will store the data associated with each interval.

More precisely, the data associated with interval $[i..j]$ will be stored in the cell $P[i]$, and in our case will consist of a pair (P_1, P_2), where P_1 is a pointer to the suffix tree node to which the interval $[i..j]$ is associated and P_2 is a pointer to the internal representation of the interval $[i..j]$ in the split-find data structure. We also preprocess the suffix tree, and store a table $L[1..n-2]$, where $L[d]$ points to the list of all nodes at depth d (nodes α such that $|\bar{\alpha}| = d$). This preprocessing can clearly be done in $O(n)$ time [3].

We now describe the algorithm. We proceed in $n-2$ *update steps* numbered from 2 to $n-1$. These steps are intermingled with the queries. More precisely, if we have $d_i > d_{i-1}$ for some pair of consecutive queries $(\ell_{i-1}, d_{i-1}), (\ell_i, d_i)$, we proceed to all steps $d_{i-1}+1, \ldots, d_i$, thus preparing the split-find data structure for the next queries. At update step number $u \in [d_{i-1}+1, d_i]$, we will induce the collection of intervals of nodes that span depth u from the collection of intervals that span depth $u-1$. Following Lemma 10, we traverse the list of nodes $L[u-1]$ and for every node α with associated interval $[i..j]$ and split the interval $[i..j]$ in the split-find structure by using the pointer $P[i].P_2$ that points to the internal representation of the interval in the split-find representation. The interval of α will be replaced with the intervals of its children. If α has t children, then $[i..j]$ will be split into k subintervals $[i..i_1], \ldots, [i_{t-1}+1..j]$. This splitting can be done by successively splitting $[i..j]$ into $[i..i_1]$ and $[i_1+1..j]$, then split $[i_1+1, j]$ into $[i_1+1, i_2]$ and $[i_2+1, j]$ and so on, until we have completed the splitting of the interval $[i_{t-2}+1, j]$ into subintervals $[i_{t-2}+1, i_{t-1}]$ and $[i_{t-1}+1, j]$ (each time using and updating cells $P[i], P[i_1+1] \ldots$). At the end, the cells $P[i], P[i_1+1] \ldots P[i_{t-1}+1]$ will point to the suffix tree nodes with associated intervals $[i..i_1], \ldots, [i_{t-1}+1..j]$.

Notice that after step d_i, the subintervals stored in the split-find data structure will precisely form the collection of intervals associated to nodes that span depth d_i. It is clear that this set of nodes is precisely the set of nodes that can be answers to weighted level ancestor queries for depth d_i. Moreover answering a query (ℓ_i, d_i) amounts to finding the subinterval $[i..j]$ such that $i \leq \ell_i \leq j$ and from there retrieve the pointer to the suffix tree node pointed by $P[i]$.

It remains to analyze the amortized time of queries to the weighted-level ancestor query data structure. The total time can be decomposed into three components: the preprocessing time, the update time and the query time. The preprocessing time is dominated by the construction of lists L which as argued above is $O(n)$. The update time is dominated by the updates to the split-find data structure. We argue that the total time for the updates (between queries) is $O(n)$ assuming that the union-find data structure supports a sequence

[3] We traverse all nodes in the tree in depth-first order, computing the depth of every node α from the depth of its parent β and the length of the label of the edge that connects β to α.

of k updates in amortized time $O(1 + n/k)$ which in total gives $O(n + k)$. It is clear that the total number of split operations is upper bounded by the number of children of internal nodes in the suffix tree, since we perform $t - 1$ update operations for an internal node with t children. The total number of children of all internal nodes is clearly upper bounded by the total number of nodes in the suffix tree which is $2n - 1$. We thus conclude that the total time for all updates is $O(n + 2n - 1) = O(n)$.

Finally the amortized query time is constant for each query, since it is dominated by the query time of the split-find data structure which is constant time. Thus the sum of all query times is $O(k)$. We thus conclude that the total time for preprocessing, updates and queries is $O(n)$ and thus the amortized time per operation is $O((n + k)/k) = O(1 + n/k)$. ◀

Since there exists a split-find algorithm over n elements that has initialization time $O(n)$, uses space $O(n)$ working over the interval $[1..n]$, that supports the `find` operation in constant time and any sequence of t `split` in amortized constant $O(1 + \frac{n}{t})$ time per operation [11], we can state the following corollary.

▶ **Corollary 12.** *Given a suffix tree T with n leaves, we can compute the result of any sequence of t monotone weighted-level ancestor queries over T in $O(1 + \frac{n}{t})$ amortized time.*

4 Computing All the Antiperiods of a String

In the smallest antiperiod problem, we are given a string S of length n and have to find the smallest t such that the string S is a prefix of some string $p_0p_1 \dots p_k$, where $|p_i| = t$ for all $i \in [0..k]$ and $p_i \neq p_j$ for all $i \neq j$. This problem can be solved using the suffix tree of S by making use of the following observation:

▶ **Lemma 13.** *Two substrings $S[i, i + d - 1]$ and $S[j, j + d - 1]$ are equal iff they have the same locus in the suffix tree of S.*

The algorithm proceeds in (at most) $\lfloor n/2 \rfloor$ phases, where in phase i, it is tested whether value $t = i$ is an antiperiod of S. Such a test can be carried out via $\lfloor n/i \rfloor$ monotone weighted level-ancestor queries. That is, in phase i, we compute weighted level ancestor queries for (weighted) level i for the leaves labeled with positions 1, $i + 1$, $2i + 1$, $3i + 1$, and so on. If at any point we visit the same node (i.e. ancestor) a second time, then S cannot have antiperiod i because at least two of the substrings of length i starting at the tested positions are equal. On the other hand if all ancestors are unique so are the corresponding substrings of length i (at tested positions), implying that i is an antiperiod.

Let t be the first phase in which all ancestor nodes returned by the monotone level ancestor queries are unique. It is clear that t is the smallest antiperiod of S. We now analyse the running time. The ith phase involves n/i level ancestor queries. Therefore if we stop the algorithm at phase t, the total number of queries will be $N = O(\sum_{i=1}^{t} n/i) = O(n \log t)$. We now bound the total time taken by all queries. First of all, notice that queries have the monotonicity property, which means that we can apply Theorem 11. Now, by Theorem 11, each of the N queries is supported in amortized $O(1 + \frac{n}{N}) = O(1)$ time per query which means that the total time for all N queries is $O(N) = O(n \log t)$. We have thus obtained the following theorem.

▶ **Theorem 14.** *We can find the smallest antiperiod of a string of length n in $O(n \log t)$ time, where t is the length of the antiperiod.*

Clearly, if instead of stopping at the point at which we have computed the smallest antiperiod we continue to run the algorithm up to phase $\lfloor n/2 \rfloor$, we will compute all the antiperiods of the string, obtaining the following theorem.

▶ **Theorem 15.** *We can find the all the antiperiods of a string of length n in $O(n \log n)$ time.*

5 Faster Computation of the Smallest Antiperiod

In this section we show how to obtain a faster algorithm for the smallest antiperiod problem via the following two easily proved properties of antiperiods.

▶ **Lemma 16.** *If t is a an antiperiod, then any multiple of t is also an antiperiod.*

▶ **Lemma 17.** *If t is the smallest antiperiod, then $1, 2 \ldots, t-1$ are not antiperiods.*

Using the two observations above, we can improve the solution described in the previous section. More precisely, we can reduce the time down to $O(n \log \log t)$ as follows. We first test whether the value $i = 1$ is an antiperiod. If not we proceed as follows. Let us first define $f(x) = x\lceil \log^2 x \rceil$. For increasing values of $i = 2, 3, \ldots$, we test antiperiod $f(i)$, and stop whenever we find a value i such that $f(i)$ is an antiperiod. Clearly this first step takes time:

$$O(n) + \sum_{i=2}^{\infty} O\left(\frac{n}{i\lceil \log^2 i \rceil}\right) = O\left(n + n \sum_{i=2}^{\infty} \frac{1}{i\lceil \log^2 i \rceil}\right) = O(n).$$

This is because the series $\sum_{i=2}^{\infty} 1/(i\lceil \log^2 i \rceil)$ converges to a constant (for completeness this is shown in the appendix).

Now, for any $j \in [2..i-1]$, we have the following two facts:

1. $f(j)$ is multiple of j.
2. $f(j)$ is not an antiperiod.

By Lemma 16, these two facts imply that j is not an antiperiod. We thus have proved that the smallest antiperiod is in the range $[i..f(i)]$. Then we test all antiperiods between i and $f(i)$ (in increasing order), taking time:

$$\begin{aligned} O(n) + \sum_{j=i}^{f(i)} O\left(\frac{n}{j}\right) &= O\left(n + \sum_{j=1}^{f(i)} \frac{n}{j} - \sum_{j=1}^{i} \frac{n}{j} + \frac{n}{i}\right) \\ &= O\left(n + n(\log f(i) - \log(i))\right) \\ &= O\left(n + n(\log(i\lceil \log^2 i \rceil) - \log(i))\right) \\ &= O(n \log\lceil \log^2 i \rceil) = O(n \log \log i) \end{aligned}$$

Clearly we have that $t \in [i..f(i)]$ and so $O(n \log \log i) \in O(\log \log t)$. We thus have proved the following theorem

▶ **Theorem 18.** *We can find the smallest antiperiod t of a string of length n in time $O(n \log \log t)$.*

5.1 Recursive solution

By recursing on the solution described above multiple times we get total time $O(n \log^* t)$ to find the smallest antiperiod. That is, at the second step, instead of testing all antiperiods

between i and $f(i)$, we instead test every $\ell = \lceil \log \log i \rceil$ value, spending total time $O(n)$. That is, we test the antiperiods $i \cdot \ell, (i+1) \cdot \ell, \ldots, \lceil \frac{f(i)}{\ell} \rceil \cdot \ell$. The complexity is this time:

$$\begin{aligned} O(n) + \sum_{j=i}^{\lceil \frac{f(i)}{\ell} \rceil} O\left(\frac{n}{j\ell}\right) &= O\left(n + \frac{n}{\ell} \cdot \sum_{j=i}^{\lceil \frac{f(i)}{\ell} \rceil} \frac{1}{j}\right) \\ &\in O\left(n + \frac{n}{\ell} \cdot \sum_{j=i}^{f(i)} \frac{1}{j}\right) \\ &= O(n + \frac{n}{\ell} \cdot (\log(f(i)) - \log(i) + \frac{1}{i}) \\ &= O(n + \frac{n}{\ell} \cdot (\log \log i)) = O(n). \end{aligned}$$

In the formulae above, the $O(n)$ terms in the beginning is due to the monotone weighted-level ancestor data structure. At the end of this second step we will have determined an interval $[i_2, i_2 \cdot \ell]$. In third step, we continue doing the tests but testing every $\ell_2 = \lceil \log \log \log i \rceil$ value as a candidate antiperiod, obtaining interval $[i_3, i_3 \cdot \ell_2]$. We continue that way, at each step k using value $\ell_{k-1} = \lceil \log^{(k)}(i) \rceil$ until we get into an interval $[i_k..i_k \cdot \lceil \log^{(k)}(i) \rceil]$, such that $\log^{(k)}(i) \leq 3$. Then, we can test all values inside that interval. We also have $\log^* i = O(\log^* t)$. Now, since we have $O(\log^* t)$ steps and each step takes $O(n)$ time, the total execution time of our algorithm is $O(n \log^* t)$.

▶ **Theorem 19.** *We can find the smallest antiperiod t of a string of length n in $O(n \log^* t)$ time.*

5.2 Linear time solution

We now show how to further reduce the time for finding the smallest antiperiod from $O(n \log^* t)$ to $O(n)$. We will make use of the substring hashing data structure described by Gawrychowski [13]. Given a string $S[1..n]$, the data structure allows to associate to each distinct substring of S a unique integer in the range $[1..O(n^3)]$. The data structure can be precomputed in $O(n)$ time and returns the unique identifier associated to a given substring $S[s..e]$ in constant time. For testing an antiperiod p, we will make $\lfloor n/p \rfloor$ queries to the substring hashing data structure and collect the obtainted integer identifiers. We can check whether two substrings are equal by using radix-sort to sort all the identifiers in time $O(\sqrt{n} + (n/p) \cdot \frac{\log n}{\log n/2}) = O(\sqrt{n} + n/p)$ using buckets of size $2^{\lceil \log(n)/2 \rceil}$. At the end of the sorting we scan the identifiers to check whether any identifier occurs twice, and if not, conclude that p is an antiperiod. We will modify the recursive algorithm as follows.

At each step $k+1$ we will use $\ell_k = \lceil (\log^{(k)}(i))^2 \rceil$ instead of $\ell_k = \lceil \log^{(k)}(i) \rceil$. The running time of step $k+1$ will thus be

$$O\left(n \frac{\log \ell_{k-1}}{\ell_k}\right) = O\left(n \frac{2 \log^{(k)}(i)}{(\log^{(k)}(i))^2}\right) = O\left(\frac{n}{\log^{(k)}(i)}\right)$$

instead of $O(n)$ (notice that we avoind the $O(n)$ term due to the weighted-level ancestor data structure). Summing up over all the steps, the running time will be:

$$O\left(n + \frac{n}{\log \log i} + \frac{n}{\log \log \log i} + \ldots + n\right) = O(n)$$

We thus have proved the following theorem.

▶ **Theorem 20.** *We can find the smallest antiperiod of a string of length n in $O(n)$ time.*

References

1 Golnaz Badkobeh, Gabriele Fici, and Simon J. Puglisi. Algorithms for Anti-Powers in Strings. *Information Processing Letters*, 137:57–60, 2018. arXiv:1805.10042.

2 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "Runs" Theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

3 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P Pissis, and Jakub Radoszewski. Indexing weighted sequences: neat and efficient. *arXiv preprint*, 2017. arXiv:1704.07625.

4 Amanda Burcroff. (k, λ)-Anti-Powers and Other Patterns in Words. *Electronic Journal of Combinatorics*, 25(4):#P4.41, 2018.

5 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theor. Comput. Sci.*, 412(27):2931–2941, 2011.

6 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology.* World Scientific, 2002.

7 Colin Defant. Anti-Power Prefixes of the Thue-Morse Word. *Electronic Journal of Combinatorics*, 24(1):#P1.32, 2017.

8 Martin Farach and S Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Annual Symposium on Combinatorial Pattern Matching*, pages 130–140. Springer, 1996.

9 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-Powers in Infinite Words. In *43rd International Colloquium on Automata, Languages, and Programming, (ICALP)*, volume 55 of *LIPIcs*, pages 124:1–124:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

10 Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-powers in infinite words. *J. Comb. Theory, Ser. A*, 157:109–119, 2018. doi:10.1016/j.jcta.2018.02.009.

11 Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

12 Marisa Gaetz. Anti-power j-fixes of the Thue-Morse word. arXiv:1808.01528.

13 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *European Symposium on Algorithms*, pages 421–432. Springer, 2011.

14 Paweł Gawrychowski. Personal Communication, 2018.

15 Paweł Gawrychowski, Moshe Lewenstein, and Patrick K Nicholson. Weighted ancestors in suffix trees. In *European Symposium on Algorithms*, pages 455–466. Springer, 2014.

16 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

17 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A Linear Time Algorithm for Seeds Computation. *arXiv preprint*, 2011. arXiv:1107.2422.

18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Efficient representation and counting of antipower factors in words. In *International Conference on Language and Automata Theory and Applications*, pages 421–433. Springer, 2019.

19 Roman M. Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*, 31(13):3672–3678, 2003.

20 Roman M. Kolpakov and Gregory Kucherov. Finding Maximal Repetitions in a Word in Linear Time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604, 1999.

21 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574. Society for Industrial and Applied Mathematics, 2007.

22 M. Lothaire. *Algebraic Combinatorics on Words.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002. doi:10.1017/CBO9781107326019.

23 Shyam Narayanan. Functions on antipower prefix lengths of the Thue-Morse word. arXiv:1705.06310.

24 W. F. Smyth. *Computing Patterns in Strings.* Pearson Addison Wesley, United Kingdom, 2003.

25 Axel Thue. Uber unendliche Zeichenreihen. *Norske vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana*, 7:1–22, 1906.

A Complement of Section 4

We show that the series $\sum_{i=2}^{\infty} 1/(i\lceil \log^2 i \rceil)$ converges to a constant. This can be proved as follows. Since the function $f(x) = 1/(x(\ln x)^2)$ is monotonically decreasing over $[2, \infty)$, we can upper bound the sum

$$\sum_{i=2}^{\infty} \frac{1}{i(\ln i)^2}$$

by the integral

$$\int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x$$

By replacing $x = e^u$, we get

$$\int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x = \int_{\ln 2}^{\infty} \frac{1}{e^u u^2} (e^u \cdot \mathrm{d}u) = \int_{\ln 2}^{\infty} \frac{1}{u^2} \cdot \mathrm{d}u.$$

The antiderivative of $1/u^2$ being equal to $-1/u$, we get that

$$\int_{\ln 2}^{\infty} \frac{1}{u^2} \cdot \mathrm{d}u = \frac{1}{\ln 2} - \frac{1}{\infty} = \frac{1}{\ln 2}.$$

We thus have

$$\begin{aligned}
\sum_{i=2}^{\infty} \frac{1}{i\lceil \log^2 i \rceil} &\leq \sum_{i=2}^{\infty} \frac{1}{i \log^2 i} \\
&= 1 + \sum_{i=2}^{\infty} \frac{(\ln 2)^2}{i(\ln i)^2} \\
&\leq 1 + (\ln 2)^2 \cdot \int_2^{\infty} \frac{1}{x(\ln x)^2} \cdot \mathrm{d}x \\
&= 1 + (\ln 2)^2 \cdot \frac{1}{\ln 2} = 1 + \ln 2.
\end{aligned}$$