35th Annual Symposium on Combinatorial Pattern Matching

CPM 2024, June 25–27, 2024, Fukuoka, Japan

Edited by

Shunsuke Inenaga
Simon J. Puglisi

LIPICS

Editors

Shunsuke Inenaga
Kyushu University, Japan
inenaga.shunsuke.380@m.kyushu-u.ac.jp

Simon J. Puglisi
University of Helsinki, Finland
simon.puglisi@helsinki.fi

ACM Classification 2012
Theory of computation → Algorithm design techniques; Theory of computation → Data compression; Theory of Computation → Design and analysis of algorithms; Theory of computation → Fixed parameter tractability; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Pattern matching; Theory of computation → Problems, reductions and completeness; Theory of computation → Sorting and searching; Theory of computation → Theory and algorithms for application domains; Mathematics of computing → Discrete mathematics; Mathematics of computing → Combinatorics on words; Mathematics of computing → Combinatoric problems; Applied computing → Computational biology

ISBN 978-3-95977-326-3

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/978-3-95977-326-3.

Publication date
June, 2024

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.2024.0

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

Dedicated to our inspirational friend, colleague, and teacher, Masayuki Takeda, 1964–2022.

Contents

Preface

The Annual Symposium on Combinatorial Pattern Matching (CPM) has by now over 30 years of tradition and is considered to be the leading conference for the community working on Stringology. The objective of the annual CPM meetings is to provide an international forum for research in combinatorial pattern matching and related applications such as computational biology, data compression and data mining, coding, information retrieval, natural language processing, and image processing (i.e. 2D strings).

This volume contains the papers presented at the 35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024) held on June 25–27, 2024 in Fukuoka, Japan. The conference program includes 28 contributed papers and three invited talks, by

- Martin Farach-Colton (New York University, USA),
- Zsuzsanna Lipták (University of Verona, Italy), and
- Tetsuo Shibuya (University of Tokyo, Japan).

For the sixth time, CPM includes the "Highlights of CPM" special session, for presenting the highlights of recent developments in combinatorial pattern matching. In this sixth installment we selected as highlight papers "Gapped String Indexing in Subquadratic Space and Sublinear Query Time", by Philip Bille, presented at STACS 2024, and "Optimal-Time Queries on BWT-Runs Compressed Indexes" and "An Optimal-Time RLBWT Construction in BWT-Runs Bounded Space", by Yasuo Tabei / Takaaki Nishimoto, presented at ICALP 2021 and ICALP 2022, respectively.

The contributed papers for CPM 2024 were selected out of 48 submissions, corresponding to an acceptance ratio of 58%. Each submission received at least three reviews. We thank the members of the Program Committee and all the additional external subreviewers, who are listed below, for their hard, invaluable, and collaborative effort that resulted in an excellent scientific program. We also thank the CPM Steering Committee for their support and advice. We thank the CPM 2024 Organising Committee chair Yuto Nakashima and all the other members for their excellent and hard work that made this conference a wonderful one.

The CPM 2024 conference was co-located with a StringMasters workshop held on June 24 and June 28, 2024 in Fukuoka. The StringMasters workshop was organized by Dominik Köppl. Preceding these events in Fukuoka, a two-day summer school was held at the University of Electro-Communications, in Tokyo, which provided lectures by Jesper Jansson (Kyoto University) and Philip Wellnitz (National Institute of Informatics). The summer school was organized by Paweł Gawrychowski, Hideo Bannai, and Takuya Meino.

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, Barcelona, London (Ontario, Canada), Pisa, Lille, New York, Palermo, Helsinki, Bad Herrenalb, Moscow, Ischia, Tel Aviv, Warsaw, Qingdao, Pisa, Copenhagen (on-line), Wrocław, Prague, and Marne-la-Vallée. From 1992 to the 2015 meeting, all proceedings were published in the LNCS (Lecture Notes in Computer Science) series. Since 2016, the CPM proceedings have appeared in the LIPIcs (Leibniz International Proceedings in Informatics) series, as volume 54 (CPM 2016), 78 (CPM 2017), 105 (CPM 2018), 128 (CPM 2019), 161 (CPM 2020), 191 (CPM 2021), 223 (CPM 2022), and 259 (CPM 2023). The entire submission and review process was carried out using the EasyChair conference system.

This proceedings of CPM 2024 is dedicated to the memory of Masayuki Takeda, who passed away in December 2022, in Fukuoka. As a professor at the Department of Informatics, Kyushu University, Masayuki was one of the pioneers of the field of pattern matching and compressed string processing in Japan. He was a passionate researcher, and published over 200 articles, including 34 papers in the CPM series of conferences – more than all but one other author. Masayuki also co-chaired CPM 2002 in Fukuoka with late Alberto Apostolico. Masayuki was a great teacher, and he supervised a number of students at the undergraduate, master, and PhD levels, many of whom have gone on to have successful research careers. Implicitly, and explicitly, he is the reason the CPM conference returns to Fukuoka this year.

Shunsuke Inenaga and Simon J. Puglisi
CPM 2024 Program Committee Chairs

Program Committee

- Golnaz Badkobeh, City, University of London
- Giulia Bernardini, University of Trieste
- Philip Bille, Technical University of Denmark
- Manuel Cáceres, University of Helsinki
- Panagiotis Charalampopoulos, Birkbeck, University of London
- Gabriele Fici, University of Palermo
- Daniel Gibney, University of Texas at Dallas
- Veronica Guerrini, University of Pisa
- Shunsuke Inenaga, Kyushu University (Chair)
- Jesper Jansson, Kyoto University
- Artur Jeż, University of Wrocław
- Dominik Köppl, University of Yamanashi
- Gregory Kucherov, Gustave Eiffel University
- Susana Ladra, University of A Coruna
- Avivit Levy, Shenkar College
- Gonzalo Navarro, University of Chile
- Nicola Prezza, Ca' Foscari University of Venice
- Simon J. Puglisi, University of Helsinki (Chair)
- Giulia Punzi, NII
- Marinella Sciortino, Università di Palermo
- Kana Shimizu, Waseda University

List of External Reviewers

Adrián Gómez Brandón
Alejandro Pacheco
Alessandro Berti
Alexandru I. Tomescu
Arnab Ganguly
B. Riva Shalom
Bartlomiej Dudek
Binhai Zhu
Bojana Kodric
Carlo Tosoni
Chiaki Sakama
Chiara Epifanio
Conrado Martínez
Cristian Urbina
Cyril Gavoille
Daniel Puttini
Davide Rucci
Diego Arroyuelo
Diego Seco
Diptarama Hendrian
Djamal Belazzougui
Dominika Draesslerová
Duncan Adamson
Dustin Cobas
Eitan Kondratovsky
Elena Biagi
Estéban Gabory
Felipe A. Louza
Florin Manea
Francisco Olivares
Gabriel Bathie
Giovanna Rosone
Giuseppa Castiglione
Hideo Bannai
Hongyu Zheng
Itai Boneh
Jakub Radoszewski
Jamshed Khan
Jonas Ellert
Jouni Sirén
Kevin Ryan
Laurent Bulteau
Louisa Seelbach Benkner
Massimo Equi
Md Helal Hossen
Meng He
Mikołaj Marciniak
Nicola Rizzo
Nicolaos Matsakis
Paniz Abedin
Rahul Shah
Rocco Ascone
Romina Doz
Ruben Becker
Sharma V. Thankachan
Shay Golan
Simon Rumle Tarnow
Solon Pissis
Sung-Hwan Kim
Sven Rahmann
Takuya Mieno
Teresa Anna Steiner
Tomasz Walen
Travis Gagie
Vincent Jugé
Vitaly Aksenov
Wiktor Zuba
William Kuszmaul
Wojciech Janczewski
Yinzhan Xu
Yuto Nakashima

Computing the LCP Array of a Labeled Graph

Jarno N. Alanko
University of Helsinki, Finland

Davide Cenzato
Ca' Foscari University of Venice, Italy

Nicola Cotumaccio
University of Helsinki, Finland

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy

Giovanni Manzini
University of Pisa, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

The LCP array is an important tool in stringology, allowing to speed up pattern matching algorithms and enabling compact representations of the suffix tree. Recently, Conte *et al.* [DCC 2023] and Cotumaccio *et al.* [SPIRE 2023] extended the definition of this array to Wheeler DFAs and, ultimately, to arbitrary labeled graphs, proving that it can be used to efficiently solve matching statistics queries on the graph's paths. In this paper, we provide the first efficient algorithm building the LCP array of a directed labeled graph with n nodes and m edges labeled over an alphabet of size σ. The first step is to transform the input graph G into a deterministic Wheeler pseudoforest G_{is} with $O(n)$ edges encoding the lexicographically- smallest and largest strings entering in each node of the original graph. Using state-of-the-art algorithms, this step runs in $O(\min\{m \log n, m + n^2\})$ time on arbitrary labeled graphs, and in $O(m)$ time on Wheeler DFAs. The LCP array of G stores the longest common prefixes between those strings, i.e. it can easily be derived from the LCP array of G_{is}. After arguing that the natural generalization of a compact-space LCP-construction algorithm by Beller *et al.* [J. Discrete Algorithms 2013] runs in time $\Omega(n\sigma)$ on pseudoforests, we present a new algorithm based on dynamic range stabbing building the LCP array of G_{is} in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space. Combined with our reduction, we obtain the first efficient algorithm to build the LCP array of an arbitrary labeled graph. An implementation of our algorithm is publicly available at `https://github.com/regindex/Labeled-Graph-LCP`.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis; Theory of computation → Pattern matching

Keywords and phrases LCP array, Wheeler automata, prefix sorting, pattern matching, sorting

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.1

Related Version *Full Version*: `https://arxiv.org/abs/2404.14235`

Supplementary Material *Software*: `https://github.com/regindex/Labeled-Graph-LCP`

Funding *Jarno Alanko, Nicola Cotumaccio*: Funded by the Helsinki Institute for Information Technology (HIIT).
Davide Cenzato, Sung-Hwan Kim, Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Giovanni Manzini: Funded by the Italian Ministry of Health, POS 2014-2020, project ID T4-AN-07, CUP I53C22001300001, by INdAM-GNCS Project CUP E53C23001670001 and by PNRR ECS00000017 Tuscany Health Ecosystem, Spoke 6 CUP I53C22000780001.

1 Introduction

The LCP array of a string – storing the lengths of the longest common prefixes of lexicographic-adjacent string suffixes – is a data structure introduced by Manber and Myers in [23] that proved very useful in tasks such as speeding up pattern matching queries with suffix arrays [23] and representing more compactly the suffix tree [1]. As a more recent application of this array, Boucher *et al.* augmented the BOSS representation of a de Bruijn graph with a generalization of the LCP array that supports the navigation of the underlying variable-order de Bruijn graph [7]. Recently, Conte *et al.* [9] and Cotumaccio [14] extended this structure to Wheeler DFAs [19] – deterministic edge-labeled graphs admitting a total order of their nodes being compatible with the co-lexicographic order of the strings labeling source-to-node paths – and, ultimately, arbitrary edge-labeled graphs. The main idea when generalizing the LCP array to a labeled graph, is to collect the lexicographic smallest $\inf_u$ and largest $\sup_u$ string entering each node u (following edges backwards, starting from u), sorting them lexicographically, and computing the lengths of the longest common prefixes between lexicographically-adjacent strings in this sorted list [9]. As shown by Conte *et al.* [9] and Cotumaccio *et al.* [14, 16], such a data structure can be used to efficiently find Maximal Exact Matches (MEMs) on the graph's paths and to speed up the navigation of variable-order de Bruijn graphs.

Importantly, [9, 14, 16] did not discuss efficient algorithms for building the LCP array of a labeled graph. The goal of our paper is to design such algorithms.

Overview of our contributions

Let G be a directed labeled graph with n nodes and m edges labeled over an alphabet of cardinality σ. After introducing the main definitions and notation in Section 2, in Section 3 we describe our main three-steps pipeline for computing the LCP array of G: (1) *Pre-processing*. Using the algorithms described in [2, 5, 12] we transform G into a particular deterministic *Wheeler pseudoforest* G_{is} of size $O(n)$, that is, a deterministic Wheeler graph [19] whose nodes have in-degree equal to 1. Graph G_{is} compactly encodes the lexicographically- smallest and largest strings $\inf_u$ and $\sup_u$ (Definition 4) leaving backwards (i.e. following reversed edges) every node u of G, by means of the unique path entering the node. Using state-of-the-art algorithms, this step runs in $O(m)$ time and $O(m)$ words of space if G is a Wheeler semi-DFA [2], and in $O(\min\{m + n^2, m \log n\})$ time and $O(m)$ words of space on arbitrary deterministic graphs [5, 12]. (2) *LCP computation*: we describe a new compact-space algorithm computing the LCP array of G_{is} (see below for more details); (3) *Post-processing*: we turn the LCP array of G_{is} into the LCP array of the original graph G, in $O(m)$ time and $O(m)$ words of space.

As far as step (2) is concerned, we turn our attention to the algorithm of Beller *et al.* [6], working on strings in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space on top of the LCP array (which can be streamed to the output in the form of pairs $(i, LCP[i])$ sorted by their second component). The idea of this algorithm is to keep a queue of suffix array intervals, initially filled with the interval $[1, n]$. After extracting from the queue an interval $[i, j]$ corresponding to all suffixes prefixed by some string α, the algorithm retrieves all *distinct* characters $c_1, \ldots, c_k$ in the Burrows-Wheeler transform [8] interval $BWT[i, j]$ and, by means of backward searching [18], retrieves the intervals of strings $c_1 \cdot \alpha, \ldots, c_k \cdot \alpha$, writing an LCP entry at the beginning of each of those intervals (unless that LCP entry was not already filled, in which case the procedure does not recurse on the corresponding interval). Recently, Alanko *et al.* [3] generalized this algorithm to the BWT of the infimum sub-graph of a de Bruijn graph (also known as the SBWT [4]). In Section 4 we first show a pseudoforest on which the natural generalization of Beller *et al.*'s algorithm [6] performs $\Omega(n\sigma)$ steps of forward search; this implies that, as a function of σ, this algorithm is *exponentially* slower on graphs than it is on strings. Motivated by this fact, we revisit the algorithm. For simplicity, in this paragraph, we sketch the algorithm on strings; see Section 4 for the generalization to graphs. Rather than working with a queue of suffix array intervals, we maintain a queue of LCP indexes $i \in [1, n]$. We furthermore keep a dynamic range-stabbing data structure on all the open intervals $(l, r]$ such that $BWT[l] = BWT[r] = c$ for some character c, and no other occurrence of c appears in $BWT[l, r]$. When processing position i with $LCP[i] = \ell$, we remove the intervals $(l, r]$ stabbed by i, and use them to derive new LCP entries of value $\ell + 1$ (and new positions to be inserted in the queue) by backward-stepping from $BWT[l]$ and $BWT[r]$. On deterministic Wheeler pseudoforests, our algorithm runs in $O(n \log \sigma)$ time and uses $O(n \log \sigma)$ bits of working space (the LCP array can be streamed to output as in the case of Beller *et al.* [6]).

Putting everything together (preprocessing, LCP of G_{is}, and post-processing), we prove (see Section 2 for all definitions):

▶ **Theorem 1.** *Given a labeled graph G with n nodes and m edges labeled over alphabet $[\sigma]$, with $\sigma \leq m^{O(1)}$, we can compute the LCP array of G in $O(m)$ words of space and $O(n \log \sigma + \min\{m \log n, m + n^2\})$ time. If G is a Wheeler semi-DFA, the running time reduces to $O(n \log \sigma + m)$.*

If the input graph G is a Wheeler pseudoforest with all strings $\inf_u$ being distinct, represented compactly as an FM-index of a Wheeler graph [19], we can do even better: our algorithm terminates in $O(n \log \sigma)$ time while using just $O(n \log \sigma)$ bits of working space (Lemma 15).

We implemented our algorithm computing the LCP array of G_{is} (Algorithm 1) and made it publicly available at `https://github.com/regindex/Labeled-Graph-LCP`.

Due to space constraints, some proofs can be found in the full version of the paper.

2 Preliminaries

We work with directed edge-labeled graphs $G = (V, E)$ on a fixed ordered alphabet $\Sigma = [\sigma] = \{1, \ldots, \sigma\}$, where $E \subseteq V \times V \times \Sigma$ and, without loss of generality, $V = [n]$ for some integer $n > 0$. Symbol $n = |V|$ denotes therefore the number of nodes of G. With $m = |E|$ we denote the number of edges of G. Without loss of generality, we assume that there are no nodes with both in-degree and out-degree equal to zero (such nodes can easily be treated separately in the problem we consider in this paper). In particular, this implies that $n \in O(m)$. We require that the alphabet's size is polynomial in the input's size: $\sigma \leq m^{O(1)}$. Notation in_u and out_u, for $u \in V$, indicates the in-degree and out-degree of u, respectively. We say that G is *deterministic* if $(u, v', a), (u, v'', b) \in E \Rightarrow a \neq b$ whenever $v' \neq v''$. We say that any node $u \in V$ with $\mathrm{in}_u = 0$ is a *source*, that G is a *semi-DFA* if G is deterministic, has exactly one source, and all nodes are reachable from the source, and that G is a *pseudoforest* if and only if $\mathrm{in}_u = 1$ for all $u \in V$. If $G = (V, E)$ is a pseudoforest, $\lambda(u) \in \Sigma$, for $u \in V$, denotes the character labeling the unique edge entering v. We say that two labeled graphs $G = (V, E), G' = (V', E')$ on the same alphabet Σ are *isomorphic* if and only if there exists a bijection $\phi : V \to V'$ preserving edges and labels: for every $u, v \in V, u', v' \in V', a \in \Sigma$, $(u, v, a) \in E$ if and only if $(\phi(u), \phi(v), a) \in E'$.

If $\alpha = c_1 c_2 \cdots c_n \in \Sigma^*$ is a finite string, the notation $\overleftarrow{\alpha} = c_n c_{n-1} \cdots c_1$ indicates α reversed. The notation Σ^ω indicates the set of *omega strings*, that is, right-infinite strings of the form $c_1 c_2 c_3 \ldots$, with $c_i \in \Sigma$ for all $i \in \mathbb{N}^{>0}$. As usual, Σ^* and Σ^+ denote the sets of finite (possibly empty) strings and the set of nonempty finite strings from Σ, respectively. For $\alpha = c_1 c_2 \cdots \in \Sigma^\omega \cup \Sigma^*$, $\alpha[i \ldots]$ indicates the suffix $c_i c_{i+1} \ldots$ of α. If $\alpha, \beta \in \Sigma^\omega \cup \Sigma^*$, we write $\alpha \prec \beta$ to indicate that α is lexicographically smaller than β (similarly for $\preceq$: lexicographically smaller than or equal to). Symbol ϵ denotes the empty string, and it holds $\epsilon \prec \alpha$ for all $\alpha, \beta \in \Sigma^\omega \cup \Sigma^+$. Given a set $S \subseteq \Sigma^\omega \cup \Sigma^*$ and a string $\alpha \in \Sigma^\omega \cup \Sigma^*$, notation $S \prec \alpha$ indicates $(\forall \beta \in S)(\beta \prec \alpha)$ (similarly for $\alpha \prec S$, $S \preceq \alpha$, and $\alpha \preceq S$).

If $G = (V, E)$ is deterministic, given $u \in V$ we denote with $\mathsf{OUTL}(u)$ the sorted string of characters labeling outgoing edges from u: $\mathsf{OUTL}(u) = c_1 \ldots c_k$ if and only if $(\forall j \in [k])(\exists v \in V)\big((u, v, c_j) \in E\big)$, with $c_1 \prec c_2 \cdots \prec c_k$. We write $c \in \mathsf{OUTL}(u)$, with $c \in \Sigma$, as a shorthand for $c \in \{\mathsf{OUTL}(u)[1], \ldots, \mathsf{OUTL}(u)[k]\}$. Since we assume that G is deterministic when we use this notation, it holds $\mathrm{out}_u = |\mathsf{OUTL}(u)|$.

Wheeler graphs were introduced by Gagie *et al.* [19]:

▶ **Definition 2.** *Let $G = (V, E)$ be an edge-labeled graph, and let $<$ be a strict total order on V. For every $u, v \in V$, let $u \leq v$ indicate $u < v \vee u = v$. We say that $<$ is a* Wheeler order *for G if and only if:*

1. *(Axiom 1) For every $u, v \in V$, if $\mathrm{in}_u = 0$ and $\mathrm{in}_v > 0$ then $u < v$.*
2. *(Axiom 2) For every $(u', u, a), (v', v, b) \in E$, if $u < v$, then $a \preceq b$.*
3. *(Axiom 3) For every $(u', u, a), (v', v, a) \in E$, if $u < v$, then $u' \leq v'$.*

A graph $G = (V, E)$ is Wheeler if it admits at least one Wheeler order.

Let $u, v \in V$, $\alpha \in \Sigma^+$, and $c \in \Sigma$. We write $u \overset{\alpha}{\rightsquigarrow} v$ to indicate that there exists a path from u to v labeled with string α (that is, we can go from u to v by following edges whose labels, when concatenated, yield α), and write $u \xrightarrow{c} v$ as an abbreviation for $(u, v, c) \in E$.

We use the symbol I_u to denote the set of strings obtained starting from node u and following edges backwards. This process either stops at a node with in-degree 0 (thereby producing a finite string), or continues indefinitely (thereby producing an omega-string). Notice that this notation differs from [2], where edges are followed forwards; we use this slight variation since, as seen below, it is a more natural way to define the LCP array of a graph. More formally:

▶ **Definition 3.** *Let $G = (V, E)$ be a labeled graph. For $u \in V$, $I_u^\omega \subseteq \Sigma^\omega$ denotes the set of* omega-strings *leaving backwards node u:*

$$I_u^\omega = \{c_1 c_2 \ldots \in \Sigma^\omega \ : \ (\exists v_1, v_2, \cdots \in V)(v_1 \xrightarrow{c_1} u \wedge (\forall i \geq 2)(v_i \xrightarrow{c_i} v_{i-1}))\}.$$

The symbol $I_u \subseteq \Sigma^\omega \cup \Sigma^*$ denotes instead the set of all *strings leaving backwards u:*

$$I_u = I_u^\omega \cup \{\overleftarrow{\alpha} \in \Sigma^+ \ : \ (\exists v \in V)(\mathrm{in}_v = 0 \wedge v \overset{\alpha}{\rightsquigarrow} u)\};$$

If $\mathrm{in}_u = 0$, we define $I_u = \{\epsilon\}$.

▶ **Definition 4** (Infimum and supremum strings [22])**.** *Let $G = (V, E)$ and $u \in V$. The infimum string $\inf_u^G = \inf I_u$ and the supremum string $\sup_u^G = \sup I_u$ relative to G are defined as:*

$$\begin{aligned} \inf_u^G &= \gamma \in \Sigma^* \cup \Sigma^\omega \ \textit{s.t.} \ (\forall \beta \in \Sigma^* \cup \Sigma^\omega)(\beta \preceq I_u \rightarrow \beta \preceq \gamma \preceq I_u) \\ \sup_u^G &= \gamma \in \Sigma^* \cup \Sigma^\omega \ \textit{s.t.} \ (\forall \beta \in \Sigma^* \cup \Sigma^\omega)(I_u \preceq \beta \rightarrow I_u \preceq \gamma \preceq \beta) \end{aligned}$$

When G will be clear from the context, we will drop the superscript and simply write $\inf_u$ *and* $\sup_u$*.*

Cotumaccio [14] defines the LCP of a labeled graph as follows:

▶ **Definition 5.** *Let $G = (V, E)$ be a labeled graph. Let $\gamma_1 \preceq \gamma_2 \preceq \cdots \preceq \gamma_{2n}$ be the lexicographically-sorted strings $\inf_u$ and $\sup_u$, for all $u \in V$. The* LCP array $\mathsf{LCP}_G[2, 2n]$ *of G is defined as $\mathsf{LCP}_G[i] = \mathsf{lcp}(\gamma_{i-1}, \gamma_i)$, where $\mathsf{lcp}(\gamma_{i-1}, \gamma_i)$ is the length of the longest common prefix between γ_{i-1} and γ_i.*

In the above definition, observe that $\mathsf{LCP}_G[i] = \infty$ if and only if $\gamma_{i-1} = \gamma_i$ and $\gamma_{i-1}, \gamma_i \in \Sigma^\omega$. We are interested in this definition since, as shown in [14], it allows computing matching statistics on arbitrary labeled graphs.

If $G = (V, E)$ is a pseudoforest, then I_u is a singleton for every $u \in V$ and the above definition can be simplified since $\inf_u = \sup_u$ for every $u \in V$. In this paper we will consider the particular case of pseudoforests for which all the $\inf_u (= \sup_u)$ are distinct. In this particular case, we define the *reduced LCP array* as follows:

▶ **Definition 6.** *Let $G = (V, E)$ be a labeled pseudoforest such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$. Let $\gamma^*_1 \prec \gamma^*_2 \prec \cdots \prec \gamma^*_n$ be the lexicographically-sorted strings $\inf_u$, for all $u \in V$. The* reduced LCP array $\mathsf{LCP}^*_G = \mathsf{LCP}^*_G[2, n]$ *of G is defined as $\mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i)$.*

Since pseudoforests will play an important role in our algorithms, we proceed by proving a useful property of the reduced LCP array of a pseudoforest.

▶ **Lemma 7.** *Let $G = (V, E)$ be a labeled pseudoforest such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$. Let $1 \leq k < \infty$ be such that $\mathsf{LCP}^*_G[i] = k$ for some $2 \leq i \leq n$. Then, there exists $2 \leq i' \leq n$ such that $\mathsf{LCP}^*_G[i'] = k - 1$.*

Proof. Let $\gamma^*_1 \prec \gamma^*_2 \prec \cdots \prec \gamma^*_n$ be the lexicographically-sorted strings $\inf_u$, for all $u \in V$. For all $i \in [n]$ let $1 \leq p(i) \leq n$ be the unique integer such that $\gamma^*_{p(i)} = \gamma^*_i[2 \ldots]$. Such integer always exists and it is unique, since the γ^*_j's are all the strings leaving each node of G (following edges backwards).

Since $1 \leq k < \infty$, then $\gamma^*_{i-1}[1] = \gamma^*_i[1]$ and $\gamma^*_{p(i-1)} = \gamma^*_{i-1}[2 \ldots] \prec \gamma^*_i[2 \ldots] = \gamma^*_{p(i)}$, which implies $p(i-1) < p(i)$. As a consequence, $k = \mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i) = 1 + \mathsf{lcp}(\gamma^*_{p(i-1)}, \gamma^*_{p(i)}) = 1 + \min_{p(i-1)+1 \leq t \leq p(i)} \mathsf{lcp}(\gamma^*_{t-1}, \gamma^*_t) = 1 + \min_{p(i-1)+1 \leq t \leq p(i)} \mathsf{LCP}^*_G[t]$, so there exists $p(i-1) + 1 \leq i' \leq p(i)$ such that $k = 1 + \mathsf{LCP}^*_G[i']$, or equivalently, $\mathsf{LCP}^*_G[i'] = k - 1$. ◀

3 The pipeline: computing LCP_G from G

As mentioned in Section 1, we reduce the computation of LCP_G to three steps: (1) a pre-processing phase building a Wheeler pseudoforest G_{is}, (2) the computation of the reduced LCP array of G_{is}, and (3) a post-processing phase yielding LCP_G. Steps (1) and (3) mainly use existing results from the literature and we illustrate them in this section. Our main contribution is step (2) for which in Section 4 we describe a new algorithm.

3.1 Pre-processing

Let $G = (V, E)$ be the input graph. As the first step of our pre-processing phase, we augment $\Sigma \leftarrow \Sigma \cup \{\#\}$ with a new symbol $\#(= 0)$ lexicographically-smaller than all symbols in the original alphabet $\{1, \ldots, \sigma\}$, and add a self-loop $u \xrightarrow{\#} u$ to all nodes $u \in V$ such that $\mathrm{in}_u = 0$. This will simplify our subsequent steps as now all strings leaving (backwards) any node belong to Σ^ω; from now on, we will therefore work with graphs with no sources.

Consider the set $IS = \{\inf_u, \sup_u : u \in V\}$, and let $N = |IS| \leq 2n$. Note that N could be strictly smaller than $2n$ since some nodes may share the same infimum/supremum string. Moreover, by the definition of infima and suprema strings, for each $\alpha \in IS$ it holds that $\alpha[2\dots] \in IS$: this is true because, if α is the infimum of I_u, then u has a predecessor v such that (i) $v \overset{\alpha[1]}{\rightarrow} u$ and (ii) $\alpha[2\dots]$ is the infimum of I_v (the same holds for suprema strings). Let us give the following definition.

▶ **Definition 8.** *Given $G = (V, E)$ with no sources, let $IS = \{\inf_u, \sup_u : u \in V\}$. We denote with $G'_{is} = (IS, E'_{is})$ the labeled graph with edge set $E'_{is} = \{(\alpha[2\dots], \alpha, \alpha[1]) : \alpha \in IS\}$.*

Observe that (i) each node $\alpha \in IS$ of G'_{is} has exactly one incoming edge, and (ii) G'_{is} is deterministic since $\alpha \overset{a}{\rightarrow} \beta$ and $\alpha \overset{a}{\rightarrow} \beta'$ imply $\beta = a \cdot \alpha = \beta'$. In other words, G'_{is} is a deterministic pseudo-forest. In addition, $\inf_u \neq \inf_v$ for every $u \neq v \in IS$. Let us prove that G'_{is} is a Wheeler graph.

▶ **Lemma 9.** *The lexicographic order $\prec$ on the nodes of G'_{is} is a Wheeler order.*

Proof. We prove that $\prec$ satisfies the three axioms of Definition 2.

(Axiom 1). This axiom holds trivially, since G'_{is} has no nodes with in-degree 0.

(Axiom 2). Let $(\alpha', \alpha, \alpha[1]), (\beta', \beta, \beta[1]) \in E'_{is}$, with $\alpha \prec \beta$. Then, by definition of the lexicographic order $\prec$, it holds $\alpha[1] \preceq \beta[1]$.

(Axiom 3). Let $(\alpha[2\dots], \alpha, a), (\beta[2\dots], \beta, a) \in E'_{is}$, with $a = \alpha[1] = \beta[1]$ and $\alpha \prec \beta$. Then, by the definition of $\prec$ it holds $\alpha[2\dots] \prec \beta[2\dots]$. ◀

Importantly, we remark that our subsequent algorithms will only require the topology and edge labels of G'_{is} to work correctly. In other words, any graph *isomorphic* to G'_{is} will work, and it will not be needed to compute *explicitly* the set IS (an impractical task, since IS contains omega-strings). Figure 1 shows an example of labeled graph G (with sources) pre-processed to remove sources and converted to such a graph isomorphic to G'_{is}. We can compute such a graph using recent results in the literature:

▶ **Theorem 10.** *Let $G = (V, E)$ be a labeled graph with no sources and alphabet of size $\sigma \leq m^{O(1)}$. Let moreover $V_i = \{u_i : u \in V\}$ and $V_s = \{u_s : u \in V\}$ be two duplicates of V. Then, we can compute a graph $G_{is} = ([N], E_{is})$ being isomorphic to $G'_{is} = (IS, E'_{is})$ (Definition 8), together with a function $\mathsf{map} : V_i \cup V_s \rightarrow [N]$ such that:*

1. *for every $u \in V$, $\inf^G_u = \inf^{G_{is}}_{\mathsf{map}(u_i)}$ and $\sup^G_u = \sup^{G_{is}}_{\mathsf{map}(u_s)}$, and*
2. *the total order $<$ on the integers coincides with the Wheeler order of G_{is}. In particular, for all $i, j \in [N]$, $i < j$ if and only if $\inf^{G_{is}}_i = \sup^{G_{is}}_i \prec \inf^{G_{is}}_j = \sup^{G_{is}}_j$.*

Function map is returned as an array of $2n = 2|V|$ words, so that it can be evaluated in constant time. G_{is} and map can be computed from G in $O(m+n^2)$ time [12] or in $O(m \log n)$ time [5]. If G is a Wheeler semi-DFA, the running time reduces to $O(m)$ [2]. All these algorithms use $O(m)$ words of working space.

In the full version of the paper we discuss how Theorem 10 can be obtained using the results in [2, 5, 12] (which were originally delivered for a different purpose: computing the maximum co-lex order [11, 13, 15, 17]). Figure 1 shows an example of G_{is} (right) for a particular labeled graph G (left). Table 1 (right) shows the nodes $[N]$ of such a graph G_{is}, together with the strings entering in each node, sorted lexicographically, and their longest common prefix array $\mathsf{LCP}^*_{G_{is}}$. In Table 1 (left) we sort the duplicated nodes ($V_i \cup V_s$) of G by their infima (V_i) and suprema (V_s) strings, and show the mapping $\mathsf{map} : V_i \cup V_s \rightarrow [N]$ in the second and third columns.

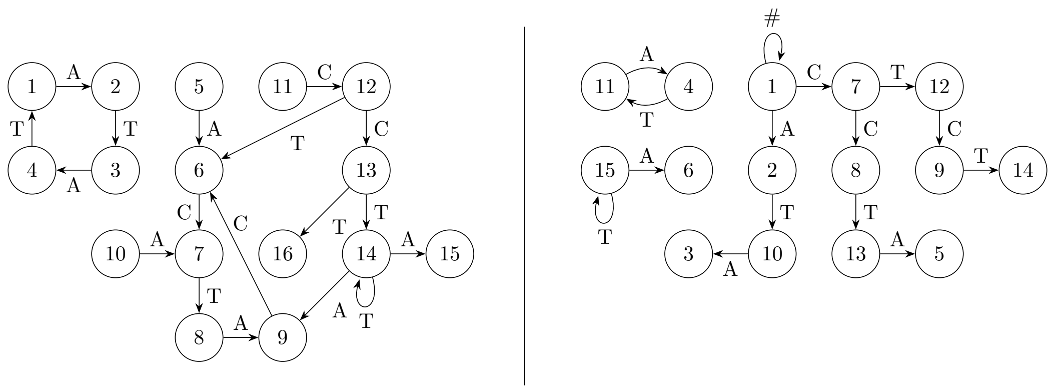


Figure 1 *Left*: a labeled graph G. *Right*: a graph G_{is} isomorphic to G'_{is} (Definition 8) satisfying Theorem 10. Note that in G_{is} the node numbering coincides with the Wheeler order.

Table 1 *Left:* lexicographically-sorted infima and suprema strings γ_j of graph $G = (V, E)$ of Figure 1, along with the nodes of V they reach (subscripted using the duplicates V_i and V_s of V to show whether the string is an infimum or a supremum), and array LCP_G. The second ($[N]$) and third ($V_i \cup V_s$) columns of the table show the mapping $\mathsf{map} : V_i \cup V_s \to [N]$. *Right:* The sorted infima (equivalently, suprema) γ_j^* of graph G_{is} of Figure 1, and the array $\mathsf{LCP}^*_{G_{is}}$.

j	$[N]$	$V_i \cup V_s$	γ_j	LCP_G
1	1	10_i	ε	-
2	1	10_s	ε	0
3	1	11_i	ε	0
4	1	11_s	ε	0
5	1	5_i	ε	0
6	1	5_s	ε	0
7	2	6_i	`A`	0
8	2	7_i	`A`	1
9	3	9_i	`ATA`	1
10	4	2_i	`ATATA...`	3
11	4	2_s	`ATATA...`	∞
12	4	4_i	`ATATA...`	∞
13	4	4_s	`ATATA...`	∞
14	5	15_i	`ATCC`	2
15	6	15_s	`ATTTT...`	2
16	6	9_s	`ATTTT...`	∞
17	7	12_i	`C`	0
18	7	12_s	`C`	1
19	8	13_i	`CC`	1
20	8	13_s	`CC`	2
21	9	7_s	`CTC`	1
22	10	8_i	`TA`	0
23	11	1_i	`TATAT...`	2
24	11	1_s	`TATAT...`	∞
25	11	3_i	`TATAT...`	∞
26	11	3_s	`TATAT...`	∞
27	12	6_s	`TC`	1
28	13	14_i	`TCC`	2
29	13	16_i	`TCC`	3
30	13	16_s	`TCC`	3
31	14	8_s	`TCTC`	2
32	15	14_s	`TTTTT...`	1

$[N]$	γ_j^*	$\mathsf{LCP}^*_{G_{is}}$
1	`#####...`	-
2	`A####...`	0
3	`ATA##...`	1
4	`ATATA...`	3
5	`ATCC#...`	2
6	`ATTTT...`	2
7	`C####...`	0
8	`CC###...`	1
9	`CTC##...`	1
10	`TA###...`	0
11	`TATAT...`	2
12	`TC###...`	1
13	`TCC##...`	2
14	`TCTC#...`	2
15	`TTTTT...`	1

3.2 Post-processing

In Section 3.1 we have shown how to convert the input $G = (V, E)$ into a Wheeler pseudoforest $G_{is} = ([N], E_{is})$ encoding the infima and suprema strings of G. In Section 4 we will show how to compute the reduced $\mathsf{LCP}^*_{G_{is}}$ of G_{is}. Here, we discuss the last step of our pipeline, converting $\mathsf{LCP}^*_{G_{is}}$ into LCP_G.

Let $\gamma_1^* \prec \gamma_2^* \prec \cdots \prec \gamma_N^*$ be the sorted strings $\inf_j$, for $j \in [N]$, relative to graph G_{is}, and $\gamma_1 \preceq \gamma_2 \preceq \cdots \preceq \gamma_{2n}$ be the sorted strings $\inf_u, \sup_u$, for $u \in V$, relative to graph G. By construction (Section 3.1), the former sequence of strings is obtained from the latter by performing these two operations: (1) duplicates are removed, and (2) finite strings γ_i are turned into omega-strings of the form $\gamma_i \cdot \#^\omega$ (see Table 1, right table). This means that LCP_G is almost the same as $\mathsf{LCP}^*_{G_{is}}$, except in maximal intervals $\mathsf{LCP}_G[i+1, j]$ such that $\gamma_i = \gamma_{i+1} = \cdots = \gamma_j$. In those intervals, we have $\mathsf{LCP}_G[i+1, j] = |\gamma_i|$; notice that this value could be either finite (if $\gamma_i \in \Sigma^*$) or infinite (if $\gamma_i \in \Sigma^\omega$).

As an example, consider the interval $\mathsf{LCP}_G[10,13] = (3,\infty,\infty,\infty)$ in Tabe 1 (left). This interval corresponds to strings `ATATA...`, and corresponds to $\mathsf{LCP}^*_{G_{is}}[4] = 3$ (right). The first value $\mathsf{LCP}_G[10]$ is equal to $\mathsf{LCP}^*_{G_{is}}[4] = 3$, while the others, $\mathsf{LCP}_G[11,13] = (\infty,\infty,\infty)$ are equal to the length (∞) of the omega-string `ATATA...`. A similar example not involving an omega string is $\mathsf{LCP}_G[7,8] = (0,1)$ (string `A`), corresponding to $\mathsf{LCP}^*_{G_{is}}[2] = 0$.

Given $\mathsf{LCP}^*_{G_{is}}$, G_{is}, and map (see Theorem 10) it is immediate to derive LCP_G in $O(m)$ worst-case time and $O(m)$ words of working space, as follows. First of all, we sort $V_i \cup V_s$ (the two duplicates of V, see Theorem 10) according to the order given by the integers $\mathsf{map}(x)$, for $x \in V_i \cup V_s$. Let $u^1, u^2, \dots, u^{2n}$ be the corresponding sequence of sorted nodes, i.e. such that $\mathsf{map}(u^1) \le \mathsf{map}(u^2) \le \dots \le \mathsf{map}(u^{2n})$. The second step is to identify the above-mentioned maximal intervals $\mathsf{LCP}_G[i+1,j]$: these are precisely the maximal intervals such that $\mathsf{map}(u^i) = \mathsf{map}(u^{i+1}) = \dots = \mathsf{map}(u^j)$. For each such interval, we set $\mathsf{LCP}_G[i] = \mathsf{LCP}^*_{G_{is}}[\mathsf{map}(u^i)]$ and $\mathsf{LCP}_G[i+1] = \mathsf{LCP}_G[i+2] = \dots \mathsf{LCP}_G[j] = |\gamma_i|$. In order to compute the length $|\gamma_i|$, observe that $\gamma_i = \gamma^*_{\mathsf{map}(u^i)}$ if $\gamma_i \in \Sigma^\omega$, and $\gamma_i \cdot \#^\omega = \gamma^*_{\mathsf{map}(u^i)}$ if $\gamma_i \in \Sigma^*$. Then, a simple DFS visit of G_{is} starting from nodes u with a self-loop $u \xrightarrow{\#} u$ reveals whether we are in the former or latter case, and allows computing the length (DFS depth) of γ_i in the latter.

4 Computing the LCP array of G_{is}

In view of Sections 3.1 and 3.2, the core task to solve in order to compute LCP_G is the computation of $\mathsf{LCP}^*_{G_{is}}$. To make notation lighter, in this section we denote the input graph with symbol $G = (V,E)$ and assume it is a deterministic Wheeler pseudoforest such that $\inf_u \ne \inf_v$ for all $u \ne v \in V$. In the rest of the section, n and m denote the number of nodes and edges of G. The goal of our algorithms is to compute LCP^*_G.

To solve this problem, we first focused our attention on the algorithm of Beller *et al.* [6], computing the LCP array from the Burrows-Wheeler Transform [8] of the input string in $(1+o(1)) \cdot n\log\sigma + O(n)$ bits of working space (including the indexed BWT and excluding the LCP array, which however can be streamed to output in order of increasing LCP values) and $O(n\log\sigma)$ time. However, as we briefly show in Figure 2, we realized that the natural generalization of this algorithm to pseudoforests runs in $\Omega(n\sigma)$ time in the worst case. Motivated by this fact, in this section we re-design the algorithm by resorting to the dynamic interval stabbing problem (e.g., see [24]), achieving running time $O(n\log\sigma)$ on deterministic Wheeler pseudoforests. This will require designing a novel dynamic range-stabbing data structure that could be of independent interest.

Let $v_1 < v_2 < \dots < v_n$ denote the Wheeler order of G. We define:

▶ **Definition 11** (bridge). *For $1 \le l < r \le n$ and $c \in \Sigma$, a triplet (l,r,c) is said to be a* bridge *of G if and only if (i) both v_l and v_r have an outgoing edge labeled with c and (ii) for every k such that $l < k < r$, v_k does not have any outgoing edge labeled with c.*

Consider two consecutive nodes (in the Wheeler order) v_{i-1} and v_i with the same incoming label $c = \lambda(v_{i-1}) = \lambda(v_i)$ for some $1 < i \le n$. On deterministic pseudoforests there is a one-to-one correspondence between the set of such node pairs and the set of bridges:

▶ **Lemma 12.** *The following bijection exists between the set of bridges and the set of pairs (v_{i-1}, v_i) such that $\lambda(v_{i-1}) = \lambda(v_i)$. Let $\lambda(v_{i-1}) = \lambda(v_i) = c$, and let v_l and v_r be the nodes such that $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$. Then, (l,r,c) is a bridge. Conversely, for every bridge (l,r,c), letting v_h and v_i be the unique nodes such that $v_l \xrightarrow{c} v_h$ and $v_r \xrightarrow{c} v_i$, we have $h = i-1$.*

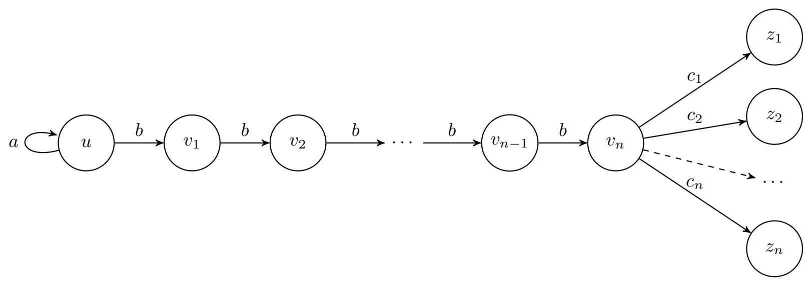


Figure 2 A Wheeler pseudoforest with $V = \{u < v_1 < \cdots < v_n < z_1 < \cdots < z_n\}$ and Wheeler order $<$ where the number of forward search steps in the natural generalization of Beller *et al.*'s algorithm is $\Theta(n\sigma)$. The range of nodes (BWT interval) reached by a path suffixed by b^i, for $1 \le i \le n$, is $[v_i, v_n]$. The algorithm right-extends (via *forward search*) each of these ranges by all the n characters $c_1, \ldots, c_n$; when extending with c_i, we obtain the unary range $[z_i]$ of nodes reached by a path suffixed by $b^i c_i$. This means that Beller *et al.*'s algorithm will perform in total $\Theta(n\sigma) = \Theta(n^2)$ forward search steps. Intuitively, our solution to this problem will be to extend *just one* of those ranges by $c_1, \ldots, c_n$. We achieve this by resorting to a dynamic range stabbing data structure.

Proof. Given such nodes v_{i-1} and v_i, v_l and v_r are unambiguously determined because every node has exactly one incoming edge. To see that $(l, r, \lambda(v_i))$ is a bridge, notice that for every edge $(u, u', \lambda(v_i))$, if $u' < v_{i-1}$, then by Axiom 3 and determinism $u < v_l$, and if $v_i < u'$, then similarly $v_r < u$, so for every k such that $l < k < r$, v_k does not have an outgoing edge labeled $\lambda(v_i)$. For the reverse implication, notice that $h \neq i$ because every node has exactly one incoming edge, and it cannot be $i < h$ otherwise from Axiom 3 we would obtain $v_r < v_l$. Hence $h \le i - 1$. If we had $h < i - 1$, then by Axiom 2 the unique edge entering v_{i-1} should be labeled c, and from Axiom 3 and determinism its start node k should satisfy $l < k < r$, so (l, r, c) would not be a bridge. ◀

The intuition behind our algorithm is as follows. Given a bridge (l, r, c), suppose that $\mathsf{lcp}(\gamma_l^*, \gamma_r^*) = d \ge 0$. Let v_{i-1} and v_i be the nodes such that $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$ (see Lemma 12). These nodes can be obtained from v_l and v_r by one forward search step. Then, we have $\gamma_{i-1}^* = c\gamma_l^*$ and $\gamma_i^* = c\gamma_r^*$, thus $\mathsf{lcp}(\gamma_{i-1}^*, \gamma_i^*) = \mathsf{lcp}(\gamma_l^*, \gamma_r^*) + 1 = d + 1$. By Lemma 9, the Wheeler order $v_1 < \cdots < v_n$ corresponds to the lexicographic order of the node's incoming strings $\gamma_1^* \prec \cdots \prec \gamma_n^*$, hence $\mathsf{lcp}(\gamma_l^*, \gamma_r^*) = \min_{j \in (l,r]} \mathsf{lcp}(\gamma_{j-1}^*, \gamma_j^*) = \min_{j \in (l,r]} \mathsf{LCP}_G^*[j]$. Therefore, the minimum value $d = \min_{j \in (l,r]} \mathsf{LCP}_G^*[j]$ within the left-open interval $(l, r]$ corresponding to a bridge (l, r, c) yields $\mathsf{LCP}_G^*[i] = \mathsf{lcp}(\gamma_{i-1}^*, \gamma_i^*) = d + 1$. This observation stands at the core of our algorithm: if we compute LCP values in nondecreasing order, then the position $j_{min} = \operatorname{argmin}_{j \in (l,r]} \mathsf{LCP}_G^*[j]$ $(1 < j_{min} \le n)$ of the first (smallest) generated LCP value inside $(l, r]$ "stabs" interval $(l, r]$. This yields $\mathsf{LCP}_G^*[i] = \mathsf{LCP}_G^*[j_{min}] + 1$. After this interval stabbing query, bridge (l, r, c) is removed from the set of bridges since the resulting LCP value $\mathsf{LCP}_G^*[i]$ has been correctly computed once for all.

Our procedure for computing LCP_G^* is formalized in Algorithm 1. The algorithm takes as input a deterministic Wheeler pseudoforest G represented as an FM index (Lemma 13 below) and outputs all pairs $(i, \mathsf{LCP}_G^*[i])$, one by one in a streaming fashion, in nondecreasing order of their second component (i.e. LCP value). This is useful in space-efficient applications where one cannot afford storing the whole LCP array in $n \log n$ bits, see for example [26].

▶ **Lemma 13.** *A deterministic Wheeler pseudoforest G can be represented succinctly in $(1 + o(1)) \cdot n \log \sigma + O(n)$ bits of space with a data structure (FM index of a Wheeler graph [19]) supporting the following queries in $O(\log \sigma)$ time:*

1. G.forward_step(i, c)*: given* $1 < i \leq n$ *and a character* $c \in \Sigma$*, let* $k \geq i$ *be the smallest integer such that* v_k *has an outgoing edge labeled with* c*. This query returns the integer* i' *such that* $v_k \xrightarrow{c} v_{i'}$*, or* $\perp$ *if such* k *does not exist.*
2. $\mathsf{OUTL}(v_i)[j]$*: given a node* v_i *and an index* $j \in [\sigma]$*, return the* j*-th outgoing label of* v_i*. Return* $\perp$ *if* $j > \mathrm{out}_{v_i}$*.*
3. $\lambda(v_i)$*: given* $i \in [n]$*, return the incoming label of* v_i*.*

Proof. Following [19], we represent G as a triple $(\mathsf{C}, \mathsf{OUT}, \mathsf{L}) \in \{0, \dots, n\}^\sigma \times \{0,1\}^{2n} \times [\sigma]^n$ defined as:

- $\mathsf{OUT} = 0^{\mathrm{out}_{v_1}}1 \cdot 0^{\mathrm{out}_{v_2}}1 \cdots 0^{\mathrm{out}_{v_n}}1$ is the concatenation of the outdegrees of nodes $v_1, \dots, v_n$, written in unary,
- $\mathsf{L} = \mathsf{OUTL}(v_1) \cdot \mathsf{OUTL}(v_2) \cdots \mathsf{OUTL}(v_n)$ is the concatenation of all the labels of the node's outgoing edges in Wheeler order, and
- $\mathsf{C}[c] = |\{u \in V \ : \ \lambda(u) \prec c\}|$, $c \in \Sigma$, denotes the number of nodes whose incoming edge is labelled with a character c' such that $c' \prec c$. Importantly, C is not actually stored explicitly; as we show below, any $\mathsf{C}[c]$ can be retrieved from L in $O(\log \sigma)$ time.

The only difference with [19] (where arbitrary Wheeler graphs are considered) is that a pseudoforest has n nodes and n edges, and all nodes have in-degree equal to 1. This simplifies the structure, since we do not need to store in-degrees. L is encoded with a wavelet tree [21] and OUT with a succinct bitvector data structure [27]. A root-to leaf traversal of the wavelet tree of L is sufficient to retrieve any $\mathsf{C}[c]$ in $O(\log \sigma)$ time at no additional space cost (see for example [28, Alg 3]). This representation uses $n \log \sigma + O(n)$ bits of space and supports the following operations in $O(\log \sigma)$ time: (1) random access to any of the arrays $\mathsf{C}, \mathsf{OUT}, \mathsf{L}$, (2) $\mathsf{L}.\mathsf{rank}_c(j)$, returning the number of occurrences of c in $\mathsf{L}[1, j]$, and (3) $\mathsf{OUT}.\mathsf{select}_1(j)$, returning the position of the j-th occurrence of bit 1 in bitvector OUT.

Using these queries, we can solve query (1) as follows: $G.\mathsf{forward\_step}(i, c) = \mathsf{C}[c] + \mathsf{L}.\mathsf{rank}_c(\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1)) + 1$. If $\mathsf{L}.\mathsf{rank}_c(\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1)) + 1$ exceeds the number of characters equal to c in L (we discover this in $O(\log \sigma)$ time using the wavelet tree on L), the query returns $\perp$. Query (2) is answered as follows: $\mathsf{OUTL}(v_i)[j] = \mathsf{L}[\mathsf{OUT}.\mathsf{select}_1(i-1) - (i-1) + j]$. If j exceeds the out-degree of v_i (we discover this in constant time using rank and select queries on OUT), the query returns $\perp$. Query (3) $\lambda(v_i)$ can be solved with a root-to-leaf visit of the wavelet tree of L, in $O(\log \sigma)$ time (range quantile queries, see [20]). ◀

We proceed by commenting the pseudocode and proving its correctness and complexity. In Line 3 of Algorithm 1, we compute the set $\mathcal{I}$ of bridges of the input graph using Lemma 14. Each bridge (l, r, c) will "survive" in $\mathcal{I}$ until any $\mathsf{LCP}^*_G[i]$ with $i \in (l, r]$ is computed. Set $\mathcal{I}$ is represented as a dynamic range stabbing data structure (Lemma 14 below) on the set of *character-labeled* intervals $\{(l, r]_c \ : \ (l, r, c) \text{ is a bridge}\}$, where notation $(l, r]_c$ indicates a left-open interval labeled with character c. We require this data structure to support interval stabbing and interval deletion queries. General solutions for this problem solving both queries in amortized $O(\log n / \log\log n)$ time exist [24]. While in the general case this is optimal, in our particular case observe that, by Definition 11, no more than σ intervals get stabbed by a given $i \in [n]$. We exploit this property to develop a more efficient (if $\log \sigma = o(\log n / \log\log n)$) dynamic range stabbing data structure (for the full proof, see the full version of the paper):

▶ **Lemma 14.** *Given a Wheeler pseudoforest* G *represented with the data structure of Lemma 13, in* $O(n \log \sigma)$ *time and* $O(n \log \sigma)$ *bits of working space we can build a* dynamic interval stabbing *data structure* $\mathcal{I}$ *of* $O(n \log \sigma)$ *bits representing the set of bridges of* G

Algorithm 1 Given a Wheeler pseudoforest G, compute LCP^*_G. In Line 8, $\mathcal{I}.\ \mathsf{stab\_and\_remove}(i)$ stabs and removes bridges from $\mathcal{I}$.

```
1:  $\mathsf{LCP}^*_G \leftarrow$ Array $\mathsf{LCP}^*_G[2,n]$, with values initialized to $\infty$
2:  $W \leftarrow \emptyset$                    ▷ $W$: queue of integer pairs of the form $(i, \mathsf{LCP}^*_G[i])$
3:  $\mathcal{I} \leftarrow \{(l,r,c) \in [n] \times [n] \times \Sigma \ :\ (l,r,c)$ is a bridge of $G\}$      ▷ $\mathcal{I}$: bridges of $G$
4:  for all $1 < i \leq n$ such that $\lambda(v_{i-1}) \prec \lambda(v_i)$: $W.\mathsf{push}(i,0)$
5:  while $W \neq \emptyset$ do
6:      $(i,d) \leftarrow W.\mathsf{pop}()$
7:      output $(i,d)$                        ▷ Stream pair $(i, \mathsf{LCP}^*_G[i])$ to output
8:      $R \leftarrow \mathcal{I}.\ \mathsf{stab\_and\_remove}(i)$ ▷ $R \subseteq \Sigma$: set of labels of bridges stabbed and removed
9:      for $c \in R$ do
10:         $i' \leftarrow G.\mathsf{forward\_step}(i,c)$
11:         $W.\mathsf{push}(i', d+1)$
12:     end for
13: end while
```

(Definition 11) and answering the following query: $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$, where $i \in [n]$. Letting $S = \{(l,r,c) \in \mathcal{I} \ :\ l < i \leq r\}$ be the set of stabbed bridges, this query performs the following two operations:

1. *it returns the set of characters $R = \{c \ :\ (l,r,c) \in S\}$ labeling bridges stabbed by i, and*
2. *it removes those bridges: $\mathcal{I} \leftarrow \mathcal{I} \setminus S$.*

Let $\ell = \sum_{(l,r,c)\in S}(r-l+1)$ be the total length of the stabbed bridges. The query is solved in $O(\log\sigma + |R| + \ell/\sigma)$ time.

Proof (Sketch, see the full paper version for all the details). We divide the nodes $v_1,\ldots,v_n$ into non-overlapping blocks $v_{k\sigma},\ldots,v_{(k+1)\sigma}$ of σ nodes each, for $k = 0,\ldots,n/\sigma-1$ (assume σ divides n for simplicity). Let I be the set of labeled intervals $(l,r]_c$ corresponding to all the bridges (l,r,c) of G; the bridges of G can be reconstructed in $O(\log\sigma)$ time each using operation $\mathsf{OUTL}(v_i)[j]$ of Lemma 13. Each interval $(l,r]_c \in I$ overlapping $t > 1$ blocks (i.e. $l+1 \leq k\sigma \leq (k+t-2)\sigma < r$ for some $k \in \{0,\ldots,n/\sigma-1\}$) is broken into t "pieces" $(l_1 = l, r_1]_c,\ldots,(l_t, r_t = r]_c$: a suffix of a block, followed by full blocks, followed by a prefix of a block. Each piece $(l_i,r_i]_c$, overlapping the k-th block for some k, is inserted in interval set I_k. The pieces $(l_1,r_1]_c,\ldots,(l_t,r_t]_c$ are connected using a doubly-linked list. Intervals of I fully contained in a block are not split in any piece and just inserted in I_k, with k being the block they overlap. Since no more than σ intervals can pairwise intersect at any point $i \in [n]$, for every k at most σ "pieces" of at most σ intervals are inserted in I_k: in total, the number of intervals in all the sets I_k is therefore $\sum_{k=0}^{n/\sigma-1}|I_k| \in O(|I| + \sigma\cdot n/\sigma) = O(n)$ (because $|I| \in O(n)$ by Lemma 12). We build an interval tree data structure $\mathcal{T}(I_k)$ ([25, Ch 8.8], [10, Ch 17.3]) on each I_k. Interval stabbing queries are answered locally (on the tree associated with the block containing the stabbing position i). Interval deletion queries require to also follow the linked list associated with the deleted interval, to delete all ℓ/σ "pieces" of the original interval (of length ℓ) of I. Each interval piece is deleted in constant time since we do not need to re-balance the tree. Our claim follows by observing that $|I_k| \in O(\sigma^2)$ for every k (because no more than σ intervals can pairwise intersect at any point), so (i) each $\mathcal{T}(I_k)$ is built in $O(|I_k|\log|I_k|) = O(|I_k|\log\sigma)$ time (overall, all trees are built in $O(n\log\sigma)$ time), and (ii) each pointer (tree edges and linked list pointers) uses just $O(\log\sigma)$ bits: observe that linked list pointers always connect intervals belonging to adjacent trees $\mathcal{T}(I_k)$, $\mathcal{T}(I_{k\pm1})$, so they point inside a memory region of size $O(|I_k|) \subseteq O(\sigma^2)$ and thus require $O(\log|I_k|) \subseteq O(\log\sigma)$ bits each. ◀

After building data structure $\mathcal{I}$, in Line 4 we identify all integers $1 < i \leq n$ such that $\mathsf{LCP}^*_G[i] = 0$: these correspond to consecutive nodes v_{i-1}, v_i with different incoming labels: $\lambda(v_{i-1}) \neq \lambda(v_i)$.

We keep the following invariant before and after the execution of each iteration of the `while` loop at Line 5: for every $1 < i \leq n$, *exactly one* of the following three conditions holds. (i) the pair $(i, \mathsf{LCP}^*_G[i])$ has already been output at line 7, (ii) $(i, d) \in W$ for some $d \geq 0$, in which case it holds that $\mathsf{LCP}^*_G[i] = d$ or (iii) $(l, r, c) \in \mathcal{I}$, where (l, r, c) is the bridge associated with v_i according to Lemma 12.

The invariant clearly holds before entering the `while` loop, because for every $1 < i \leq n$ we either push $(i, 0)$ in W at Line 4 whenever $\lambda(v_{i-1}) \neq \lambda(v_i)$ (thereby satisfying condition (ii) since $\mathsf{LCP}^*_G[i] = \mathsf{lcp}(\gamma^*_{i-1}, \gamma^*_i) = 0$), or insert (l, r, c) in $\mathcal{I}$ at Line 3, where (l, r, c) is the bridge associated with v_i (thereby satisfying condition (iii)). At this point, condition (i) does not hold for any $1 < i \leq n$. Notice that by Definition 11, no bridge is associated with v_i such that $\lambda(v_{i-1}) \neq \lambda(v_i)$ (and vice versa), so the invariant's conditions are indeed mutually exclusive.

We show that the invariant holds after every operation in the body of the main loop. Assume we pop (i, d) from W at Line 6. Then, this means that (before popping) condition (ii) of our invariant holds, and in particular $\mathsf{LCP}^*_G[i] = d$. At line 7 the algorithm correctly outputs $(i, d = \mathsf{LCP}^*_G[i])$, so condition (i) of the invariant now holds for position i (while condition (ii) does not hold anymore, and condition (iii) did not hold even before: remember that the three conditions are mutually exclusive). The invariant is not modified (so it still holds) for the other positions $i' \neq i$.

In Line 8, we retrieve and remove every bridge (l, r, c) such that $l < i \leq r$. Let $v_{i'}$ be the node associated with bridge (l, r, c) (Lemma 12). The fact that we remove (l, r, c) from $\mathcal{I}$ temporarily invalidates the invariant for i' (none of (i-iii) hold), but in the `for` loop at Line 9 we immediately re-establish the invariant by pushing in W pair $(i', d+1)$ and observing that indeed $\mathsf{LCP}^*_G[i'] = d + 1$ (i.e. condition (ii) of the invariant holds for position i'). To see that $\mathsf{LCP}^*_G[i'] = d + 1$ holds true first observe that, since (i) we use a queue W for pairs $(i, \mathsf{LCP}^*_G[i])$, (ii) initially (Line 4), we only push in W pairs of the form $(i, 0)$, and (iii) whenever we pop a pair (i, d) we push pairs of the form $(j, d+1)$ for some $i, j \in [n]$, LCP values are popped in line 6 in nondecreasing order. In particular, for every $d \geq 0$, no pair of the form $(i, d+1)$ is popped from the queue until all pairs of the form (j, d) are popped. From this observation and since i is the first integer stabbing bridge (l, r, c) in Line 8 (since we remove bridges from $\mathcal{I}$ immediately after they are stabbed), it must be $\mathsf{lcp}(\gamma^*_l, \gamma^*_r) = \min_{l<j\leq r} \mathsf{LCP}^*_G[j] = \mathsf{LCP}^*_G[i] = d$. Then, since $\gamma^*_{i'-1} = c\gamma^*_l$ and $\gamma^*_{i'} = c\gamma^*_r$ hold, we have that $\mathsf{LCP}^*_G[i'] = \mathsf{lcp}(\gamma^*_{i'-1}, \gamma^*_{i'}) = d + 1$.

We proved that our invariant always holds true; in particular, it holds when the algorithm terminates. The fact that conditions (i-iii) are mutually exclusive, immediately implies that no LCP value is output more than once, i.e. the first components of the output pairs $(i, \mathsf{LCP}^*_G[i])$ are all distinct.

At the end of the algorithm's execution, $W = \emptyset$ holds. To prove that the algorithm computes every LCP value, suppose for a contradiction, that there exists $i \in [n]$ such that $(i, \mathsf{LCP}^*_G[i])$ is never output in Line 7. Without loss of generality, choose i yielding the smallest such $\mathsf{LCP}^*_G[i]$. Note that $\mathsf{LCP}^*_G[i] > 0$, since all pairs $(j, \mathsf{LCP}^*_G[j] = 0)$ are inserted in W at Line 4, thus they are output at Line 7. Then, condition (i) of our invariant does not hold for i. Also condition (ii) cannot hold, otherwise it would be $(i, \mathsf{LCP}^*_G[i]) \in W \neq \emptyset$. We conclude that condition (iii) must hold for i: $(l, r, c) \in \mathcal{I}$, where (l, r, c) is the bridge associated with v_i by Lemma 12. In turn, this implies that no pair $(j, \mathsf{LCP}^*_G[j])$ for $l < j \leq r$ has been output in Line 7 (otherwise, such a j stabbing (l, r, c) would have caused the removal of (l, r, c) from $\mathcal{I}$). By Lemma 12, $v_l \xrightarrow{c} v_{i-1}$ and $v_r \xrightarrow{c} v_i$ hold. Since we assume that $\inf_u \neq \inf_v$ for all $u \neq v \in V$, we can apply Lemma 7 and obtain that it must hold $\mathsf{LCP}^*_G[j] = \mathsf{LCP}^*_G[i] - 1$ for some $l < j \leq r$, which contradicts to minimality of $\mathsf{LCP}^*_G[i]$. We conclude that the algorithm computes every LCP value.

Next, we analyze the algorithm's running time and working space. By Lemma 14, $\mathcal{I}$ is built in $O(n \log \sigma)$ time using $O(n \log \sigma)$ bits of working space. The `while` loop at Line 5 iterates at most $O(n)$ times because (i) at most n elements are pushed into the queue W at the beginning (Line 4), and (ii) an element $(i, d) \in \mathbb{N}^2$ can be pushed into the queue at Line 11 only after a bridge is stabbed and removed; thus at most $|\mathcal{I}| \in O(n)$ elements can be pushed into the queue. As a result, Line 8 (query $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$) is executed $O(n)$ times. Recall (Lemma 14) that such a query runs in $O(\log \sigma + |R| + \ell/\sigma)$ time, where $|R|$ is the number of characters labeling stabbed intervals (equivalently, the number of stabbed intervals since no two intervals labeled with the same character can intersect) and ℓ is the total cumulative length of the stabbed intervals. Since overall the calls to $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$ will ultimately remove all bridges from $\mathcal{I}$, we conclude that the sum of all cardinalities $|R|$ equals $|\mathcal{I}| \in O(n)$, and the sum of all cumulative lengths ℓ equals the sum of all the bridges' lengths: $\sum_{(l,r,c) \in \mathcal{I}} (r - l + 1) \in O(n\sigma)$ (because no two intervals labeled with the same character can intersect). Appying Lemma 14 we conclude that, overall, all calls to $\mathcal{I}.\mathsf{stab\_and\_remove}(i)$ cost $O(n \log \sigma + n + n\sigma/\sigma) = O(n \log \sigma)$ time. Line 10 takes $O(\log \sigma)$ time by Lemma 13. We represent the queue W in $O(n)$ bits of space, using the same strategy of Beller *et al.* (see [6] for all details): W is represented internally with two queues W_t and W_{t+1}, containing pairs of the form (i, t) and $(i, t+1)$, respectively (in fact, only the first element i of the pair needs to be stored). Pairs are popped from the former queue and pushed into the second. As long as $|W_{t+1}| \leq n/\log n$, W_{t+1} is represented as a simple vector of integers. As soon as $|W_{t+1}| > n/\log n$, we switch to a packed bitvector representation of n bits ($n/\log n$ words) marking with a bit set all i such that $(i, t+1) \in W_{t+1}$. When W_t becomes empty, we delete it, create a new queue W_{t+2}, and start popping from W_{t+1} and pushing into W_{t+2}. If W_{t+1} is still represented as a vector of integers, popping is trivial. Otherwise (packed bitvector), all integers in W_{t+1} can be popped in overall $O(n/\log n + |W_{t+1}|) \subseteq O(|W_{t+1}|)$ time using bitwise operations.

We finally obtain:

▶ **Lemma 15.** *Given a deterministic Wheeler pseudoforest $G = (V, E)$ such that $\inf_u \neq \inf_v$ for all $u \neq v \in V$ represented with the data structure of Lemma 13, the reduced LCP array LCP^*_G of G can be computed in $O(n \log \sigma)$ time and $O(n \log \sigma)$ bits of working space on top of the input. The algorithm does not allocate memory for the output array LCP^*_G: the entries of this array are streamed to output in the form of pairs $(i, \mathsf{LCP}^*_G[i])$ sorted by their second component, from smallest to largest.*

Putting everything together (pre-processing, Lemma 15, and post-processing), we obtain the main result of our paper, Theorem 1.

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 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. *Regular Languages meet Prefix Sorting*, pages 911–930. SIAM, 2020. `doi:10.1137/1.9781611975994.55`.

3 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi. Longest common prefix arrays for succinct k-spectra. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2023. doi:10.1007/978-3-031-43980-3_1.

4 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable κ-spectra via subset rank queries on the spectral burrows-wheeler transform. In Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann, editors, *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.

5 Ruben Becker, Manuel Cáceres, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Francisco Olivares, and Nicola Prezza. Sorting Finite Automata via Partition Refinement. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.15.

6 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.

7 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, 2015. doi:10.1109/DCC.2015.70.

8 M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DEC Systems Research Center, 1994.

9 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on wheeler dfas. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. doi:10.1109/DCC55655.2023.00023.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (4th ed.)*. The MIT Press, 2022.

11 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $o(|e|^2 + |v|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281, 2022. doi:10.1109/DCC52660.2022.00035.

12 Nicola Cotumaccio. Prefix Sorting DFAs: A Recursive Algorithm. In Satoru Iwata and Naonori Kakimura, editors, *34th International Symposium on Algorithms and Computation (ISAAC 2023)*, volume 283 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2023.22.

13 Nicola Cotumaccio. A Myhill-Nerode Theorem for Generalized Automata, with Applications to Pattern Matching and Compression. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2024.26.

14 Nicola Cotumaccio. Enhanced graph pattern matching, 2024. arXiv:2402.16205.

15 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-lexicographically ordering automata and regular languages - part i. *J. ACM*, 70(4), August 2023. doi:10.1145/3607471.

16 Nicola Cotumaccio, Travis Gagie, Dominik Köppl, and Nicola Prezza. Space-time trade-offs for the LCP array of wheeler dfas. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2023. doi:10.1007/978-3-031-43980-3_12.

17 Nicola Cotumaccio and Nicola Prezza. *On Indexing and Compressing Finite Automata*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.

18 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

19 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. doi:10.1016/j.tcs.2017.06.016.

20 Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval*, pages 1–6, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

21 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics.

22 Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza. Faster Prefix-Sorting Algorithms for Deterministic Finite Automata. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2023.16.

23 U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

24 Yakov Nekrich. A Dynamic Stabbing-Max Data Structure with Sub-Logarithmic Query Time. In *Proceedings of the 22nd International Symposium on Algorithms and Computations (ISAAC)*, pages 170–179, 2011. doi:10.1007/978-3-642-25591-5_19.

25 Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

26 Nicola Prezza and Giovanna Rosone. Space-efficient construction of compressed suffix trees. *Theoretical Computer Science*, 852:138–156, 2021.

27 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43–es, November 2007. doi:10.1145/1290672.1290680.

28 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. Special Issue: Combinatorial Pattern Matching (CPM 2010). doi:10.1016/j.ic.2011.03.007.

Reconstructing General Matching Graphs

Amihood Amir
Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
Georgia Tech, College of Computing, Atlanta, GA, USA

Michael Itzhaki
Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract

The classical pattern matching paradigm is that of seeking occurrences of one string in another, where both strings are drawn from an alphabet set Σ. Motivated by many applications, algorithms were developed for pattern matching where the matching relation is not necessarily the "$=$" relation. Examples are pattern matching with "don't cares", approximate matching, less-than matching, Cartesian-tree matching, order preserving matching, parameterized matching, degenerate matching, function matching, and more. Some of the matchings above allow for efficient pattern matching algorithms, while others do not.

Much work has not been done on categorization of the complexity of various string matching queries based on the *type* of matching. For example, when can exact matching be done fast? When can approximate matching be calculated fast? When can tandem or palindrome recognition be efficiently calculated?

This paper defines the *matching graph* of a given string under a matching relation. We show that the type of graph affects various string algorithms. The matching graph can also be a tool for lower bounds. We provide a lower bound for finding palindromes in a general degenerate graph. We also show some results in recognizing the minimum alphabet required for reconstructing a string that presents a given matching graph.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern Matching, Matching Graphs, Reconstruction, $\mathcal{NP}$-hardness

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.2

Funding *Amihood Amir*: Partially supported by ISF grant 168/23 and BSF grant 2018-141.
Michael Itzhaki: Partially supported by ISF grant 168/23.

1 Introduction

In the classical pattern matching model, we seek occurrences of a string, or more generally, a set of strings, in a distinguished string. All strings are comprised of symbols from an alphabet set Σ. The basic problem in this paradigm is that of *standard string matching*, that is, the problem of finding all occurrences of a pattern string of length m in a text string of length n. This problem can be solved in $O(n+m)$ time-independent of the alphabet size $|\Sigma|$ [15, 27, 43].

While the exact matching paradigm, where a *match* means alphabet equality, is a common and important one, historically, many problems were identified where a *match* between symbols has a different meaning. The first such model was the *pattern matching with don't cares*, where a special symbol $\phi \notin \Sigma$ is added, where ϕ matches every symbol in Σ. As we will see, this changes the matching graph, and the matching relation is no longer transitive. Fischer and Paterson [28, 34] showed that convolutions can solve this problem efficiently. Convolutions have been useful in the case of *less-than matching* [8]. Here the alphabet is natural numbers, and a pattern letter p matches the text letter t if $p \leq t$. The new twist in this matching relation is that it is not symmetric. *Approximate* *matching* seeks all pattern occurrences with errors. There are several algorithms for Hamming distance errors, generally convolutions-based [1,13], and dynamic programming algorithms for Levenshtein edit distance [44]. In *Cartesian Tree matching*, two strings match if they have the same Cartesian tree [46], and in *order-preserving matching*, two strings match if the relative order of their elements is the same [30]. *Parameterized Matching* was introduced by Brenda Baker [20]. In this matching, a text and pattern match if there is a bijection that, when applied to the pattern alphabet, will match the text. In *function matching*, the function applied to the pattern alphabet is a general function [14]. In *degenerate string matching*, the alphabet consists of non-empty subsets of alphabet Σ [26,31,35].

The various matchings mentioned above led to many different algorithms. The work of [19] classifies regex-matching problems by their structure. However, except for the latter, a systemic work on categorizing different matching types has not been performed. In this paper, we propose a method of analyzing some matching relations, called the *matching graph*. We give an example where a lower bound can be achieved due to the matching graph and study some of the insights that the matching graph offers.

▶ **Definition 1.** *Let $\mathcal{M} \subseteq \Sigma \times \Sigma$ be a* matching relation *of elements in alphabet Σ. Let $S = S[1], ..., S[n]$ be a string over Σ. The* Matching graph of S *is the graph $G = (V, E)$, whose nodes are $V = \{1, ..., n\}$ and where there is an edge $\overline{ij}$ if $\mathcal{M}(i, j)$.*

▶ **Example 2.** For alphabet $\Sigma = \{a, b, c\}$, the matching graph of string $S = a, a, b, a, b, b, c, c, b$ is the graph consisting of the three cliques $\{1, 2, 4\}, \{3, 5, 6, 9\}$ and $\{7, 8\}$.

▶ **Example 3.** Let $\Sigma = \{a, b\}$, and let ϕ be the don't care symbol. Then the matching graph of $S = a, a, b, \phi, b, a, \phi, a$ can be seen in Fig. 1.

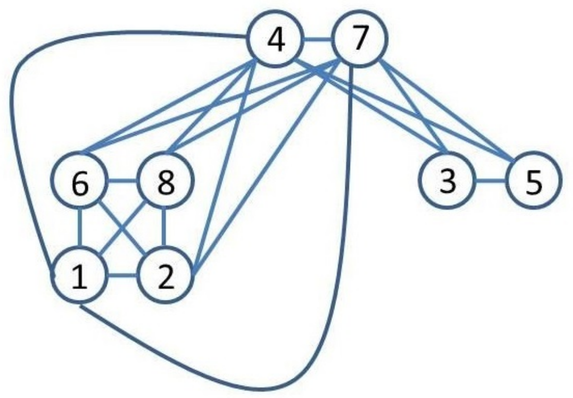


Figure 1 Matching Graph of a string with don't cares.

Note that the graph is always a set of cliques when the matching relation is transitive, as seen in Example 2. When the relation is symmetric, the graph is undirected; otherwise, it is not.

Our contribution

In this paper, we consider the matching relation in a generic string. We prove that any possible matching relation corresponds to a degenerate string, which implies lower bounds on most degenerate string matching algorithms. We also show that the number of characters required to reconstruct a degenerate string is tightly $O(n^2)$, as more characters introduce no new information, and some matching relations cannot be reconstructed with less than $O(n^2)$ alphabet characters. We show that constructing a degenerate string from a matching relation with the smallest possible alphabet is NP-hard.

2 Preliminaries

We begin with basic definitions and notation, generally following [29].

Let $S = S[1]S[2]\dots S[n]$ be a *string* of length $|S| = n$ over an ordered alphabet Σ. By ε we denote the empty string. For two positions i and j on S, we denote by $S[i..j] = S[i]S[i+1]..S[j]$ the *factor* (sometimes called *substring*) of S that begins at position i and ends at position j (it equals ε if $j < i$). A *prefix* of S is a factor that begins at position 1 ($S[1..j]$), and a *suffix* is a factor that ends at position n ($S[i..n]$). We say that S^R is the reversal of S, which is $S[n]S[n-1]...S[1]$.

▶ **Definition 4** (Index Matching Function). *Let S be a string of length $|S| = n$. The $\mathcal{M}_S : [n] \times [n] \to \{0,1\}$ is the matching function of string S and $\mathcal{M}_S(i,j) = 1$ iff $S[i] = S[j]$. The matching function will be denoted as $\mathcal{M}$ if S is clear from the context.*

The matching function of a standard string (as defined above) is transitive, reflexive, and symmetric.

▶ **Definition 5** (Palindrome). *A palindrome is a string S that equals its reversal S^R. Using definition Definition 4, a palindrome is a string that $\forall i, \mathcal{M}(i, n-i) = 1$. A palindrome factor is a string factor $P = S[i..j]$ such that P is a palindrome. A maximal palindromic factor is a palindromic factor $P = S[i..j]$ such that $S[i-1..j+1]$ is either not defined or not a palindrome.*

▶ **Definition 6** (Don't care). *The special character "don't care", denoted as ϕ is a character that matches any other character, including itself. A string S having a don't care at index m satisfies $\forall i < |S|, \mathcal{M}_S(m,i) = \mathcal{M}_S(i,m) = 1$.*

This paper addresses an interesting matching relation - equality in *degenerate strings*.

▶ **Definition 7** (Degenerate string). *Let Σ be an alphabet. S is called a degenerate string, if $S \in \{P(\Sigma)/\phi\}^*$, where $P(\Sigma)$ is the power set of Σ.*

The length *of the string, n, is the number of characters (sets) within that string. The* size *of the string, N, is $\sum\limits_{i=1}^{n} |S[i]|$. We call the sets in S* terminals *and the characters in Σ,* elements*. The empty set can not be a terminal.*

▶ **Example 8.** Let $\Sigma = [5] = \{1,2,3,4,5\}$, and let $S_1 = \{1,4\}\{1,5\}\{4\}\{1,2,3\}$, and $S_2 = \{1\}\{2\}\{5\}\{5\}$. The lengths of S_1, S_2, denoted respectively by n_1, n_2 are both 4. However, the sizes, N_1, N_2 (resp.) are different, where $N_1 = 8$ and $N_2 = 4$.

▶ **Definition 9** (Primitive terminal). *Let c be a terminal of a degenerate string. We say that c is* primitive *if $|c| = 1$.*

▶ **Definition 10** (Terminals equality). *Let c_1, c_2 be two terminals of a degenerate string. We say that c_1* matches *c_2 if $c_1 \cap c_2 \neq \emptyset$. Throughout the paper, we denote terminals equality between $S[i]$ and $S[j]$ as $S[i] = S[j]$.*

▶ **Observation 11.** *Degenerate string matching where the only non-primitive terminal is Σ is equivalent to string matching with don't cares.*

▶ **Definition 12.** *Let $G = (V, E)$ be an undirected graph, meaning that $\forall i, j$ s.t. $(i, j) \in E \rightarrow (j, i) \in E$. An* induced subgraph, or simply a *subgraph $G' = (V', E')$ of G is formed from a subset of the vertices of the original graph, and all of the edges that connect vertices in the subset. Formally, $V' \subseteq V$, $\forall i, j$, $i \in V' \wedge j \in V'$ if and only if $(i, j) \in E'$.*
A clique is a subgraph $G' = (V', E')$ such that $\forall i \neq j \in V'$, $(i, j) \in E'$.

▶ **Definition 13.** *The complete graph K_n is a clique with n vertices. A bipartite graph $G = (V_1 \cup V_2, E)$ is a graph such that all vertices in E connect a vertex in V_1 with a vertex from V_2, formally, $\forall (i, j) \in E$, either $i \in V_1, j \in V_2$ or $i \in V_2, j \in V_1$. The complete bipartite graph $K_{n,m}$ is a bipartite graph $G = (V_1 \cup V_2, E)$ such that $|V_1| = n, |V_2| = m$ and E has all possible edges under the bipartite restriction. Bipartite graphs have no odd-length cycles, and therefore K_3 is not a subgraph of $K_{n,m}$, for any n and m.*

▶ **Definition 14** (Edge Clique Cover)**.** *Let $G = (V, E)$ be a graph. An* edge clique cover of G is a set of subgraphs of G, $\{(V_1, E_1), (V_2, E_2), ..., (V_m, E_m)\}$ such that $E = \bigcup_{i=1}^{m} E_i$. The *edge clique cover number is the size of the smallest possible set that covers G. Deciding if a graph can be covered with less than k cliques is NP-hard, and also hard to estimate.*

3 Matching function in Pattern Matching

The standard definition of a string defines a string as an ordered array of characters, i.e., $S \in \Sigma^*$. However, the alphabet is not crucial for most string algorithms and can be replaced by the numbers $1, 2, ..., |\Sigma|$. This possible replacement is because most algorithms only consider whether two characters are equal. This would not be true for algorithms considering a more complicated relation between the characters, for example, DNA algorithms that can predict a particular illness from a specific DNA subsequence or algorithms concerning the value of the characters, for example, ordered matching or Cartesian tree matching.

We call algorithms that only concern characters equality *alphabet comparison* algorithms. We may perceive the input to such algorithms as *matching oracle* $\mathcal{M}$ rather than a string S.

▶ **Definition 15.** *A* matching oracle *$\mathcal{M}$ is a function $\mathcal{M} : [n] \times [n] \rightarrow \{0, 1\}$, where $\forall i, j$ $M(i, j) = 1$ iff $S[i] = S[j]$.*

3.1 Matching Oracle Properties

While many algorithms claim to be comparison-based, most make additional assumptions about the matching function. The most common assumption is for the the function $\mathcal{M}(i, j)$ to define an equivalence relation, i.e.:

1. $M(i, j) = M(j, i)$ (symmetric)
2. $M(i, i) = 1$ (reflexive)
3. $M(i, j) = 1 \wedge M(j, k) = 1 \rightarrow M(i, k) = 1$ (transitive)

These assumptions work very well for standard equality. However, the last few decades have prompted the evolution of pattern matching from a combinatorial solution of the exact string matching problem to an area concerned with approximate matching of various relationships motivated by computational molecular biology, computer vision, and complex searches in digitized and distributed multimedia libraries [16, 32].

3.1.1 Parameterized strings

An important type of non-exact matching is the *parameterized matching* problem, which was introduced by Baker [21,22]. Her main motivation lay in software maintenance, where program fragments are to be considered "identical" even if variable names are different. Therefore, strings under this model are comprised of symbols from two disjoint sets Σ and Π containing *fixed symbols* and *parameter symbols* respectively. In this paradigm, one seeks *parameterized occurrences*, i.e., exact occurrences up to renaming the pattern string parameter symbols in the respective text location. This renaming is a bijection $b : \Pi \to \Pi$. An optimal algorithm for exact parameterized matching appeared in [9]. It uses the KMP automaton for a linear-time solution over fixed finite alphabet Σ. Approximate parameterized matching was investigated in [17,21,37]. Idury and Schäffer [40] considered multiple matching of parameterized patterns.

Parameterized matching has proven useful in other contexts as well. An interesting problem is searching for images (e.g. [7,18,47]). Assume, for example, that we are seeking a given icon in any possible color map. If the colors were fixed, then this is an exact two-dimensional pattern matching [6]. However, if the color map is different , the exact matching algorithm will not find the pattern. A parameterized two-dimensional search is precisely the algorithm needed. If, in addition, one is also willing to lose resolution, then a two-dimensional function matching search should be used, where the renaming function is not necessarily a bijection [5,14]. Another degenerate parameterized condition appears in DNA matching. Because of the base pair bonding, exchanging A with T and C with G, in both text and pattern, produces a match [38].

As defined, Parameterized matching is not an alphabet comparison matching. However, it has been shown to be equivalent to exact matching on a *prev* array, which is a transitive alphabet comparison matching. A *prev* is an array defined on a string S, where $A[i] = \max_{j<i}\{j \mid S[j] = S[i]\}$, or 0 if no such index exists [21].

3.1.2 Don't cares

Pattern Matching with don't cares is indeed an alphabet comparison matching. It can be defined via a matching oracle, where the don't care symbol ϕ satisfies $\forall i, j$ s.t. $S[i] = \phi$, $M(i,j) = 1$. However, this relation is not an equivalence relation. $\mathcal{M}$ is not transitive.

▶ **Example 16.** Let $S = a\phi b$. $M(1,2) = 1, M(2,3) = 1$ but $M(1,3) = 0$.

However, the matching function is still quite structured even if the transitivity property is omitted. For example, let i, j, k, w be distinct indices such that $M(i,j) = M(j,k) = M(k,w) = 1$. In the don't care settings, we know that $M(i,k) = 1$ or $M(j,w) = 1$. This is true, because if $M(i,k) = 0$, then both $S[i], S[k] \neq \phi$, $S[i] \neq S[k]$, which means that $S[j] = \phi$ and therefore $M(j,w) = 1$. Pattern matching with don't cares has efficient solutions using convolutions [28,34].

3.1.3 Less-than Matching

The matching with don't cares is an example of a non-transitive alphabet comparison matching relation. We do not know of any matching relation that is not reflexive.

Some non-symmetric alphabet comparison matching relations have been researched. Subset matching [12] and less-than matching [8]. The less-than matching problem is:

Input: Text string $T = T[1], ..., T[n]$ and pattern string $P = P[1], ..., P[m]$ where $T[i], P[i] \in N$ (the set of natural numbers).

Output: All locations i in T where $T[i+k] \geq P[k], \quad k = 1, ..., m$.

In words, every matched element of the pattern is not greater than the corresponding text element. If the text and pattern are drawn schematically, we are interested in all positions where the pattern lies below the text.

It turns out that convolutions could be used for efficient matching less-than matching. However, by giving up the symmetry requirement, we may ambiguate some basic strings' constructs - such as periods and palindromes. Therefore, in the remainder of this paper, we will only consider symmetric matching functions.

In the next section, we consider the *degenerate string matching problem*, which generalizes the don't care matching problem.

4 Degenerate string detection

Generalized degenerate strings and elastic degenerate strings are motivated by problems in Computational Biology. Much work has been done on efficient algorithms for matching as well as lower bounds [2–4, 23–26, 31, 35, 41]. In this section we will focus on detecting a *degenerate string* from a matching function. We will show that every symmetric matching graph has a corresponding degenerate string. We will also show that reconstructing a degenerate string from a matching function over a minimal alphabet is $\mathcal{NP}$-hard, and we will eventually show that for certain matching graphs, the degenerate string alphabet Σ is of quadratic order. The following two theorems will be proven in this section.

▶ **Definition 17.** *A degenerate string S is said to* reconstruct a matching graph G if the matching graph of S equals to G [1].

▶ **Theorem 18.** *Every symmetric matching graph has a corresponding degenerate string.*

▶ **Theorem 19.** *Recovering a degenerate string over minimal alphabet from a symmetric matching graph is $\mathcal{NP}$-hard.*

4.1 Matching graph representation

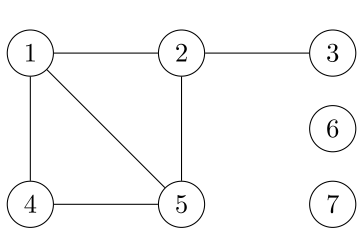
As we have seen, the matching function can be represented as a graph. One of the standard representations of a dense graph is by storing an adjacency matrix. Storing the list of neighbors for each vertices is more efficient if the graph is sparse. As we consider a symmetric function, the graph is undirected.

Because each symmetric matching function is equivalent to an undirected graph, we consider the problem of reconstructing a degenerate string from an arbitrary undirected graph. We begin by showing the existence of such a reconstruction. We later prove that finding a minimal alphabet is a hard problem.

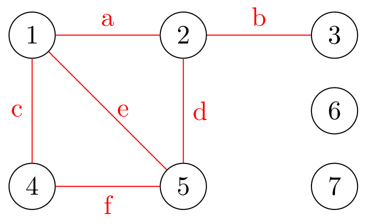
4.2 Reconstructing a degenerate string from an undirected graph

There has been much work recently on reverse engineering data structures [10, 11, 33, 36, 39, 42, 45]. Reverse engineering determines whether a given input is a valid instance of a particular data structure. We also refer to this as *reconstructing* the data structure. As we have seen, reconstructing the string from a matching graph for simple equality matching is simple. Each clique gives the indices of a unique symbol. Any graph that is not a collection of disjointed cliques is an illegal data structure. The matching graph of a degenerate string has the particular property that *every* undirected graph is legal. We now describe how, given an undirected graph G, we reconstruct a degenerate string whose matching graph is G.

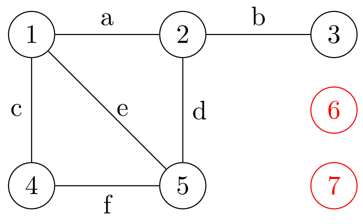
[1] Not isomorphic, as nodes have significance



(a) Matching graph, S is initialized $S = \{\}\{\}\{\}\{\}\{\}\{\}\{\}$.



(b) All edges are labeled with a character $S = \{a, c, e\}\{a, b, d\}\{b\}\{c, f\}\{d, e, f\}\{\}\{\}$.



(c) Empty sets are reconstructed $S = \{a, c, e\}\{a, b, d\}\{b\}\{c, f\}\{d, e, f\}\{g\}\{h\}$.

Figure 2 Reconstructing a degenerate string from the matching graph using Algorithm 1. We reconstruct with characters and not numbers to avoid confusion between edges and nodes.

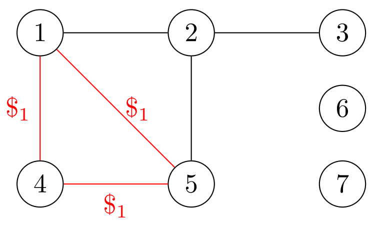
Let $G = (V, E)$ be an undirected graph, where $V = [n] = \{1, 2, ..., n\}$ represent the indices of the degenerate string, and an edge (i, j) exists if and only if the reconstructed string matches between indices i and j, denoted as $S[i] = S[j]$.

Algorithm 1 Reconstruct a degenerate string from an undirected graph.

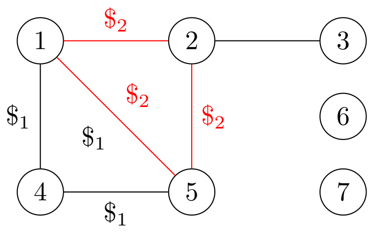
```
   Data: Undirected graph $G = (V, E)$
   Result: Degenerate string $S$
1  $S \leftarrow \{\{\}, \{\}, ...\{\}\}$ // Initialize the output degenerate string with $|V|$
       empty sets
2  $c \leftarrow 1$ for $e = (i, j) \in E$ do
3      $S[i].add(c)$
4      $S[j].add(c)$
5      $c \leftarrow c + 1$
6  for each empty $s$ set in $S$ do
7      $s.add(c)$
8      $c \leftarrow c + 1$
```

The second `For` loop is necessary for the following reason. At the end of the first `For` loop, we may have empty sets for any node that does not match any other node, and empty sets are not allowed in degenerate strings. Thus, we finish the algorithm by adding a new character for every such node, which is the only symbol in that set and does not occur anywhere else. An example can be found at Figure 2.

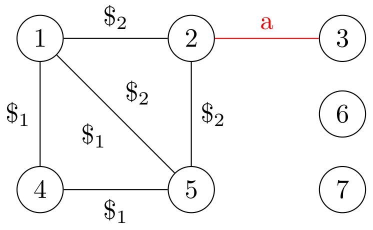
▶ **Lemma 20.** *Algorithm 1 reconstructs a degenerate string S with the same matching graph as the input $G = (V, E)$.*



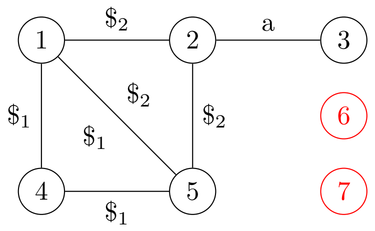
(a) First clique is colored $S = \{\$_1\}\{\}\{\}\{\$_1\}\{\$_1\}\{\}\{\}$.



(b) Second clique is colored $S = \{\$_1, \$_2\}\{\$_2\}\{\}\{\$_1\}\{\$_1, \$_2\}\{\}\{\}$.



(c) Third trivial clique is colored with a regular label, to distinguish it from non trivial cliques. $S = \{\$_1, \$_2\}\{\$_2, a\}\{a\}\{\$_1\}\{\$_1, \$_2\}\{\}\{\}$.



(d) Empty sets are reconstructed $S = \{\$_1, \$_2\}\{\$_2, a\}\{a\}\{\$_1\}\{\$_1, \$_2\}\{b\}\{c\}$.

Figure 3 Reconstructing a degenerate string from the matching graph of Figure 2, using cliques.

Proof. We show that $S[i] = S[j]$ iff $(i, j) \in E$.

We first prove that for every edge (i, j), the terminals $S[i]$ and $S[j]$ match. If (i, j) is an edge, then $S[i]$ and $S[j]$ both have the same character c, and therefore $S[i] \cap S[j] \neq \phi$, hence $S[i]$ matches $S[j]$.

We now prove that for every pair of indices i, j where $S[i]$ matches $S[j]$, the edge (i, j) exists. If $S[i]$ matches $S[j]$, it means that $S[i] \cap S[j] \neq \phi$. Let $c \in S[i] \cap S[j]$. c was either added to the algorithm in the first *For loop or the second. It is clear that if c was added in the first* For loop, then (i, j) is an edge, but in the second loop, all the characters added are unique, so it is impossible that both $S[i]$ and $S[j]$ have the same character that was added in that loop. ◀

The above reconstruction algorithm is simple and linear on the input size but does not produce a degenerate string with a minimal alphabet. Some graphs can be reconstructed to a degenerate string over an alphabet of constant size, while the algorithm will produce a quadratic size.

▶ **Example 21.** Consider the degenerate string $\{a\}^n$, of length n and of size $N = n$. The corresponding matching graph G is a clique of size n. However, after running Algorithm 1 on a clique of size n, the resulting degenerate string will be $\{\{1, 2, ..., n\}, \{1, n + 2, ...\}, \{2, n + 2, 2n + 3, ...\}, ...\}$, a string of length n but of size $N = n^2$.

Is there an efficient algorithm to reconstruct $G = (V, E)$ using a minimal alphabet? The answer is probably no, as we show that this problem is an $\mathcal{NP}$-hard problem.

▶ **Lemma 22.** *Let $G = (V, E)$ be a matching graph, and let $G' = (V', E')$ be an arbitrary sub-clique of G. Applying Algorithm 1 on $G'' = (V, E/E')$ and then adding a new character \$ for all vertices in V' is a valid degenerate string reconstruction.*

Proof. We will use Lemma 20 again. We need to prove that $\forall i, j$, $S[i] = S[j]$ iff $(i, j) \in E$. Lemma 20 shows that for every $(i, j) \in E/E'$, $S[i] = S[j]$. Also, for every $(i, j) \in E'$, we have $S[i] = S[j]$, as we required all of the characters participating in the clique to have a unique new character $\$_i$. We handle the other side similarly. We have a common character for every i, j where $S[i] = S[j]$. If the character is some $\$_i$, there must be an edge between $S[i]$ and $S[j]$, as they participate in the same clique. Otherwise, the terms proof in Lemma 20 holds, which completes the proof. ◀

▶ **Lemma 23.** *Let $G = (V, E)$ be a matching graph, and let $G' = (V', E')$ be a subgraph of G which is not a clique. If the reconstruction algorithm assigns the same character to all indices in V', then the resulting degenerate string does not have G as a matching graph.*

Proof. Let $G = (V, E)$, $G' = (V', E')$ be a graph and a subgraph as defined in the lemma. Let i, j be vertices such that $i, j \in V'$ but $(i, j) \notin E'$. Such a pair must exist, as a subgraph with only one vertex must be a clique, and a subgraph with more than one edge where all distinct vertices are connected is a clique.

If the algorithm assigns all indices i, j in the subgraph G' with the same character c, then $c \in S[i] \cap S[j]$, which means that $S[i] = S[j]$, but $(i, j) \notin E$. ◀

▶ **Observation 24.** *Lemma 22 can be applied iteratively to different cliques of G.*

An example of Algorithm 1 with clique coloring can be found at Figure 3.

▶ **Observation 25.** *Algorithm 1 applies Lemma 22 iteratively to all cliques of size 2, i.e., cliques having exactly two nodes and one edge.*

▶ **Observation 26.** *Let $G = (V, E)$ be a matching graph, and let $e \in E$. Every algorithm reconstructing a degenerate string from G will output a different string to $G = (V, E)$ and $G' = (V, E/\{e\})$.*

▶ **Lemma 27.** *Given a graph $G = (V, E)$ and a degenerate string S that reconstructs it, the string S defines an Edge-Clique-Cover for G.*

Proof. Let Σ be the alphabet of S. For every character $\sigma \in \Sigma$, all string-indices $i_1, i_2, ..., i_k$ whose terminals $S[i_j]$ contain σ are connected in the matching graph G (by the definition of reconstruction) and must form a clique (Lemma 23). Also, every edge $(i, j) \in E$ corresponds to at least one character in Σ (Observation 26), and therefore every character in Σ corresponds to a clique in G, where the vertices are all terminal indices containing σ [2]. ◀

▶ **Observation 28.** *Reconstructing a degenerate string from a matching graph with an alphabet of size k finds a Clique-Edge-Cover of size k to the matching graph, which is $\mathcal{NP}$-hard.*

Degenerate string equivalence

As seen at Observation 28, reconstructing a degenerate string from a matching function over a minimal alphabet is hard. However, reconstructing a degenerate string without limiting the resulting alphabet size is easy. We consider two different degenerate strings that have the same matching relation as *self-equivalent*. As shown in Example 21, every degenerate string can be rewritten as an equivalent string with at most $O(n^2)$ characters and a maximal terminal size of $n - 1$.

[2] Some cliques can be sub-cliques of other cliques.

4.3 Constructing the matching function

We have defined the matching function and matching graph and will use it to prove some lower bounds. Before we proceed, we discuss the complexity of constructing the matching graph. We show that it is at least as hard as boolean matrix multiplication.

▶ **Lemma 29.** *Let S be a degenerate string of length n over an ordered alphabet $\Sigma = \{1, 2, ..., k\}$. Let $d(S[i])$ be the* indicator *of $S[i]$, i.e., a binary vector $w = d(S[i])$ where $w[i] = 1$ iff $i \in S[i]$, and let D be a matrix*

$$D = \begin{bmatrix} \mathbf{d}(S[1]) \\ \mathbf{d}(S[2]) \\ \vdots \\ \mathbf{d}(S[n]) \end{bmatrix}$$

The matching graph of S is $G = (\{1, 2, ..., n\}, E)$, where $E = \{(i, j) \mid (D \times D^T)_{i,j} = 1\}$.

Proof. The vertices of the matching graph are always defined as $[n]$. An edge (i, j) exists if and only if $S[i] = S[j]$. The element $(D \times D^T)_{i,j}$ equals to $\mathbf{d}(S[i]) \cdot \mathbf{d}(S[j])$, and the boolean inner product of binary vectors $\mathbf{v}, \mathbf{w}$ equals one if the vectors are orthogonal, and in our construction it means that $S[i] = S[j]$. ◀

▶ **Lemma 30.** *If finding the matching graph of a degenerate string S of length $2n$ and size $O(n^2)$ can be performed in time $f(n)$, then Boolean Matrix Multiplication can be computed in time $O(f(n))$.*

Proof. Let us denote by $G = (V, E)$ the matching graph constructed from S.

Let A, B be boolean matrices of size $n \times n$. We want to compute $C = A \times B$ in time $f(n)$.

Let S be a degenerate string of length $2n$ over numbers alphabet $[n] = \{1, 2, ..., n\}$. We rewrite:

$$A = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_n \end{bmatrix}, B = \begin{bmatrix} \mathbf{u}_1, \mathbf{u}_n, \dots, \mathbf{u}_n \end{bmatrix}, C = A \times B = \begin{bmatrix} \mathbf{v}_1 \cdot \mathbf{u}_1, \mathbf{v}_1 \cdot \mathbf{u}_2, \dots \mathbf{v}_1 \cdot \mathbf{u}_n \\ \mathbf{v}_2 \cdot \mathbf{u}_1, \mathbf{v}_2 \cdot \mathbf{u}_2, \dots \mathbf{v}_2 \cdot \mathbf{u}_n \\ \vdots \\ \mathbf{v}_n \cdot \mathbf{u}_1, \mathbf{v}_n \cdot \mathbf{u}_2, \dots \mathbf{v}_n \cdot \mathbf{u}_n \end{bmatrix}$$

We choose the terminals of S to be the following:

$$S[i] = \begin{cases} \mathbf{v}_i, & \text{if } i \leq n \\ \mathbf{u}_{i-n}, & \text{otherwise.} \end{cases}$$

Regarding only the elements of v, u and not their vector type. [3]

Let D be the matrix defined in Lemma 29. The edges of the matching graph of S are described by $D \times D^T$. Moreover, $(D \times D^T)_{i,n+j} = \mathbf{v}_i \cdot \mathbf{u}_j$, and therefore $C[i][j] = 1$ iff $(i, j + n) \in E$, hence, completing the proof. ◀

[3] In the definition of $S[i]$ the elements of A are row vectors and the elements of B are column vectors. However, in our definition of degenerate strings, row and column vectors have exactly the same meaning, therefore the direction can be altogether ignored.

5 Palindromes and degenerate strings

In the previous section, we discussed the degenerate string matching problem and showed that in a degenerate string S over an arbitrary alphabet, there are no restrictions on the edges of the matching graph G, whereas, in a regular string, there is a very rigid structure to the graph.

In this subsection, we show how the matching graph can be used to prove unconditional lower bounds for finding maximal palindromes in a degenerate string.

▶ **Lemma 31.** *Let S be an arbitrary degenerate string. A comparison-based algorithm A cannot find all longest palindromes of S using less than $O(n^2)$ time.*

Proof. Let us assume that we have a comparison-based algorithm A that can find all longest palindromes in a degenerate string S using less than $O(n^2)$ time.

Let S be a degenerate string of length n such that S has palindromes of length exactly $n/4$ in all centers that fit such a long palindrome. Also, let us assume that no other palindromes exist within S. Such a construction is achievable from Theorem 18. There are $O(n)$ palindromes of size $O(n)$, so comparing all indices within maximal palindromes takes $O(n^2)$ work. However, the algorithm does not perform $O(n^2)$ work, so there is a comparison $S[i], S[j]$ that lays within a maximal palindrome that is not checked, so a similar string $\tilde{S}$ where $\tilde{S}[i] \neq \tilde{S}[j]$, and otherwise is identical to S. Running A on $\tilde{S}$ will result in the same maximal palindromes array, but one of its palindromes is shorter. ◀

The above lemma, in effect, means that all edges in the matching graph must be examined to find the maximum palindrome. The reason is that there are no conditions on the edges of the graph, so one may not infer an edge by knowing other edges.

A conditional lower bound for finding all maximal palindromes from a degenerate string was given by [3]. Recall:

▶ **Theorem 32.** *Given a degenerate string of length 4n over an alphabet of size $\sigma = \omega(logn)$, all maximal GD palindromes cannot be computed in $O(n^{2-\epsilon} \cdot \sigma^O(1))$ time, for any $\epsilon > 0$, unless the Strong Exponential Time Hypothesis fails.*

The difference between M. Alzamel et al. theorem and ours, is that theirs shows a conditional lower bound on SETH, given a degenerate string. At the same time, we give an unconditional lower bound given a general matching graph. Of course, our proof relies on the fact that any general graph is a matching graph of some degenerate string. Our construction requires a quadratic size alphabet. For fixed-sized finite alphabets, the situation may be different. We are aware that, given a fixed finite alphabet, it is not hard to find algorithms that run in time $\tilde{O}(n^2)$ and find all maximal palindromes [4]. However, in that later case, the input size is not quadratic in n, but rather linear. The question is whether our lower bound applies in this case, i.e. can general graphs be matching graphs of degenerate strings over finite alphabets?

▶ **Observation 33.** *Given a matching graph $G = (V, E)$ of any degenerate string, all maximal palindromes can be found in time $O(n^2)$ by checking maximal palindrome around all possible centers.*

We show in Lemma 31 that $O(n^2)$ work is always required in the general case, and in Theorem 32 that $O(n^2)$ work is required under the SETH assumption. Given the matching graph, we also see in Observation 33 that exactly $O(n^2)$ is an upper bound. Therefore, building the matching graph is at least as hard as finding all maximal palindromes on the general case and at least as hard as finding all maximal palindromes in degenerate strings with an alphabet of size $\omega(logn)$ under the SETH assumption.

We show that a quadratic number of characters is necessary to reconstruct a general matching graph.

▶ **Lemma 34.** *There exist matching graphs $G = (V, E)$ that cannot be reconstructed with less than an alphabet Σ of size less than $O(|V|^2)$.*

Proof. Consider the complete bipartite graph $K_{n,n}$. This graph is triangle-free and has a quadratic number of edges. As every character in the reconstructed degenerate string corresponds to a clique, and every clique has exactly one edge, there must be a quadratic number of cliques in the clique cover of the graph, hence a quadratic number of characters in any degenerate string S reconstructing G. ◀

The conclusion from all the above is that we have an unconditional lower bound for finding maximal palindromes in general graphs. The bound is the number of edges in the matching graph, $O(n^2)$. We also know that general matching graphs imply degenerate strings over alphabets of size $O(n^2)$. It may look like we have a tight algorithm, but this is not the case. Our algorithm has two stages:

1. Construct the matching graph $G = (V, E)$ from the degenerate string.
2. Use the matching graph to find all palindromes in time $O(|E|)$.

Indeed, one may construct the matching graph in linear time when the alphabet is finite, but then we are not sure that the matching graph is general, and therefore, the lower bound on finding the palindrome does not apply. Consider the following example:

▶ **Example 35.** Let S be a degenerate string over binary alphabet $\{a, b\}$. Every string element is either $\{a\}$, $\{b\}$ or $\{a, b\}$. Since $\{a, b\}$ matches both $\{a\}$ and $\{b\}$, the problem of finding palindromes in string S is equivalent to the problem of finding palindromes in a regular string over binary alphabets with *don't cares*. As was seen in Example 3, the matching graph in this case is well structured. Hence, there may not be a need to traverse all edges. We also know that pattern matching with don't cares has efficient solutions using convolutions. Accordingly, it may be the case that finding all palindromes in regular strings over binary alphabets with don't care has more efficient solutions than the quadratic.

In the case of alphabets of size $O(n^2)$ (quadratic alphabets), the lower bound applies, and we have an $O(n^2)$ time algorithm for finding palindromes that matches the lower bound, but that algorithm assumes a given matching graph. We have shown a conditional lower bound for constructing the matching graph of a degenerate string over a quadratic alphabet as bounded by the complexity of Boolean matrix multiplication, so our algorithm's time is now dependent on the time to construct the matching graph.

6 Conclusion and Open Problems

We have shown a simple data structure, the *Matching Graph*, that gives information on the matching relation of a pattern matching problem. We can infer from the graph whether a relation is transitive or symmetric. We also show that the graph may be useful for finding lower bounds, as in finding palindromes in degenerate strings.

Some very interesting open problems remain. An important one is finding an optimal algorithm for constructing the matching graph of degenerate strings. Such an algorithm will immediately imply an optimal algorithm for finding all palindromes in a degenerate string. This problem is especially relevant for small alphabets ($O(\log n)$), where no lower bounds are known.

Another intriguing problem is finding optimal algorithms for finding palindromes in degenerate strings over a fixed finite alphabet. A notorious example is finding all palindromes in a string over a binary alphabet, with *don't cares*.

Finally, given a degenerate string over a very large alphabet ($\Omega(n^2)$), we know that there is an equivalent degenerate string over an $O(n^2)$-size alphabet. We have shown that finding an equivalent degenerate string with the minimal alphabet is $\mathcal{NP}$-hard. However, it is easy to construct an equivalent degenerate string over a $O(n^2)$-size alphabet in time $N + n^2|\Sigma|$. Can it be done faster?

References

1 K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

2 M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Degenerate string comparison and applications. In *Proc. 18th International Workshop on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPIcs*, pages 21:1–21:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.WABI.2018.21.

3 M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, and G. Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.

4 M. Alzamel, C. Hampson, C. S. Iliopoulos, Z. Lim, S. P. Pissis, D. Vlachakis, and S. Watts. Maximal degenerate palindromes with gaps and mismatches. *Theor. Comput. Sci.*, 978:114182, 2023. doi:10.1016/J.TCS.2023.114182.

5 A. Amir, A. Aumann, M. Lewenstein, and E. Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.

6 A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

7 A. Amir, K. W. Church, and E. Dar. Separable attributes: a technique for solving the submatrices character count problem. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 400–401, 2002.

8 A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.

9 A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49:111–115, 1994.

10 A. Amir, E. Kondratovsky, G. M. Landau, S. Marcus, and D. Sokol. Reconstructing parameterized strings from parameterized suffix and LCP arrays. *Theor. Comput. Sci.*, 981:114230, 2024. doi:10.1016/J.TCS.2023.114230.

11 A. Amir, E. Kondratovsky, and A. Levy. On suffix tree detection. In *Proc. 30th Int. Symp. on String Processing and Information Retrieval (SPIRE)*, volume 14240 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2023. doi:10.1007/978-3-031-43980-3_2.

12 A. Amir, M. Lewenstein, and E. Porat. Approximate subset matching with "don't care"s. In *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 305–306, 2001.

13 A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

14 A. Amir and I. Nor. Generalized function matching. *J. of Discrete Algorithms*, 5(3):514–523, 2007.

15 O. Amir, A. Amir, D. Sarne, and A. Fraenkel. On the practical power of automata in pattern matching. *SN Computer Science*, 2024. to appear.

16 A. Apostolico and Z. Galil (editors). *Pattern Matching Algorithms.* Oxford University Press, 1997.

17 A. Apostolico, M. Lewenstein, and P. Erdös. Parameterized matching with mismatches. *Journal of Discrete Algorithms*, 5(1):135–140, 2007.

18 G.P. Babu, B.M. Mehtre, and M.S. Kankanhalli. Color indexing for efficient image retrieval. *Multimedia Tools and Applications*, 1(4):327–348, November 1995.

19 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 457–466, December 2016. doi:10.1109/FOCS.2016.56.

20 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation*, pages 71–80, 1993.

21 B. S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

22 B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

23 G. Bernardini, E. Gabory, S. P. Pissis, L. Stougie, M. Sweering, and V. Zuba. Elastic-degenerate string matching with 1 error. In *Proc. 15th Latin American symposium on Theoretical Informatics (LATIN)*, volume 13568 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2022. doi:10.1007/978-3-031-20624-5_2.

24 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *Proc. 46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ICALP.2019.21.

25 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM J. Comput.*, 51(3):549–576, 2022. doi:10.1137/20M1368033.

26 G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone. Approximate pattern matching on elastic-degenerate text. *Theor. Comput. Sci.*, 812:109–122, 2020. doi:10.1016/J.TCS.2019.08.012.

27 R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

28 P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

29 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.

30 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016.

31 M. Crochemore, C. S. Iliopoulos, R. Kundu, M. Mohamed, and F. Vayani. Linear algorithm for conservative degenerate pattern matching. *Eng. Appl. Artif. Intell.*, 51:109–114, 2016. doi:10.1016/J.ENGAPPAI.2016.01.009.

32 M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

33 J.P. Duval, T. Lecroq, and A. Lefebvre. Efficient validation and construction of border arrays and validation of string matching automata. *RAIRO Theor. Informatics Appl.*, 43(2):281–297, 2009.

34 M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

35 E. Gabory, N. M. Mwaniki, N. Pisanti, S. P. Pissis, J. Radoszewski, M. Sweering, and W. Zuba. Comparing elastic-degenerate strings: Algorithms, lower bounds, and applications. In *34th Symp. on Combinatorial Pattern Matching, CPM*, volume 259 of *LIPIcs*, pages 11:1–11:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.11.

36 P. Gawrychowski, A. Jez, and L. Jez. Validating the knuth-morris-pratt failure function, fast and online. *Theory Comput. Syst.*, 54(2):337–372, 2014.

37 C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. In *Proc. 12th Annual European Symposium on Algorithms (ESA 2004)*, pages 414–425, 2004.

38 J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *J. Discrete Algorithms*, 6(1):37–50, 2008.

39 T. I, S. Inenaga, H. Bannai, and M. Takeda. Verifying and enumerating parameterized border arrays. *Theor. Comput. Sci.*, 412(50):6959–6981, 2011.

40 R.M. Idury and A.A Schäffer. Multiple matching of parameterized patterns. In *Proc. 5th Combinatorial Pattern Matching (CPM)*, volume 807 of *LNCS*, pages 226–239. Springer-Verlag, 1994.

41 C. S. Iliopoulos, R. Kundu, and S. P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Inf. Comput.*, 279:104616, 2021. doi:10.1016/J.IC.2020.104616.

42 J. Kärkkäinen, M. Piatkowski, and S. J. Puglisi. String inference from longest-common-prefix array. In *Proc. 44th Intl. Coll. on Automata, Languages, and Programming, ICALP*, volume 80 of *LIPIcs*, pages 62:1–62:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

43 D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

44 G.M. Landau and U. Vishkin. Efficient string matching in the presence of errors. *Proc. 26th IEEE FOCS*, pages 126–126, 1985.

45 Y. Nakashima, T. Okabe, T. I, S. Inenaga, H. Bannai, and M. Takeda. Inferring strings from lyndon factorization. *Theor. Comput. Sci.*, 689:147–156, 2017.

46 S.G. Park, M. Bataa, A. Amir, G.M. Landau, and K. Park. Finding patterns and periods in cartesian tree matching. *Theoretical Computer Sciencr*, 845:181–197, 2020.

47 M. Swain and D. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

Maintaining the Size of LZ77 on Semi-Dynamic Strings

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University (TMDU), Japan

Panagiotis Charalampopoulos
Birkbeck, University of London, UK

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland

Abstract

We consider the problem of maintaining the size of the LZ77 factorization of a string S of length at most n under the following operations: (a) appending a given letter to S and (b) deleting the first letter of S. Our main result is an algorithm for this problem with amortized update time $\tilde{\mathcal{O}}(\sqrt{n})$. As a corollary, we obtain an $\tilde{\mathcal{O}}(n\sqrt{n})$-time algorithm for computing the most LZ77-compressible rotation of a length-n string – a naive approach for this problem would compute the LZ77 factorization of each possible rotation and would thus take quadratic time in the worst case. We also show an $\Omega(\sqrt{n})$ lower bound for the additive sensitivity of LZ77 with respect to the rotation operation. Our algorithm employs dynamic trees to maintain the longest-previous-factor array information and depends on periodicity-based arguments that bound the number of the required updates and enable their efficient computation.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Data compression

Keywords and phrases Lempel-Ziv, compression, LZ77, semi-dynamic algorithm, cyclic rotation

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.3

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP20H04141.
Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2022/46/E/ST6/00463.

1 Introduction

Lempel-Ziv 77 (LZ77) [70] is one of the most well known and most effective compression algorithms that admit efficient implementations. An LZ-like parsing of a string is a partitioning of the string into *phrases*, where each phrase starting at position i is either a single letter that does not occur previously, or is a prefix of the rest of the string of length $\ell \geq 1$ that has a previous occurrence at some position $s < i$. Each phrase can then be encoded by a pair $(1, T[i])$, or (ℓ, s), depending on the type of the phrase. The latter is a reference to a previous occurrence of the phrase, and thus compression can be achieved when there are many long phrases. The LZ-like parsing is also known as a Lempel-Ziv-Storer-Szymanski factorization (LZSS) [63] with self-references or a C-factorization [14]. The number of phrases in an LZ-like parsing can be minimized by adopting a greedy left-to-right approach, which is the so-called LZ77 parsing, and can be computed off-line in $\mathcal{O}(n)$ time and space for linearly sortable alphabets, or in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(n)$ space for general ordered alphabets, where n is the length of the string and σ is the number of distinct letters in the string (also $o(n)$-time algorithms for well-compressible strings over a small alphabet are known [20, 36]). Algorithms for computing the LZ77 parsing of a given (static) string have been studied extensively [1, 18, 16, 54, 4, 31, 32, 38, 33, 27, 28, 69, 24, 53, 25, 57, 44].

In an *on-line* setting, where the string can grow by appending symbols at the end, only the last phrase of an LZ77 parsing can change. The LZ77 parsing of the string can be maintained in amortized $\mathcal{O}(\log \sigma)$ time for each append operation, by a direct adaptation of Ukkonen's suffix tree construction algorithm [67]. There are also results that focus on achieving smaller space [55, 62, 69, 58, 6].

The *fully-dynamic* setting, where edits to the text at any position are allowed, is much more challenging than the on-line setting. The efficient computation of the LZ77 parsing essentially relies on index data structures such as suffix trees or arrays, which allow fast prefix searches. Recent advances have showed that dynamic indices with poly-logarithmic update and query times are possible [53, 37], and together with dynamic longest common extension (LCE) queries [51, 52, 53], this enables the computation of the LZ77 parsing of a dynamic string S in $\tilde{\mathcal{O}}(|\mathsf{LZ}(S)|)$ time, where $|\mathsf{LZ}(S)|$ is the number of phrases in the LZ77 parsing of the current string. Note that this number can be linear in the size of the string.

In this paper, we consider the problem of maintaining the *size* of the LZ77 parsing of a *semi-dynamic* string of length at most n, where the allowed update operations are (a) appending a letter and (b) shrinking the string by deleting the first letter. We present an algorithm that processes each update in strongly sublinear time.

Related Work

Cormode and Muthukrishnan [13] introduced the *substring compression* problem, where the goal is to preprocess a static string so that given any factor of the string, the phrases in its LZ77 parsing can be computed efficiently. Existing solutions for the substring compression problem [13, 35, 42, 43] basically compute the LZ77 parsing one phrase at a time, and thus require $\tilde{\mathcal{O}}(|\mathsf{LZ}(S)|)$ time to answer a query for a factor S. Our aim is to achieve better query time when the size of the LZ77 parsing can be large, by considering the more restricted semi-dynamic setting, where the queried factor moves in a sliding-window fashion over a string (note, however, that the whole string is not necessarily given beforehand).

The notion of *compression sensitivity* [2] measures the degree to which the sizes of compressed representations can change in response to updates. Akagi et al. [2] considered the compression sensitivity of LZ77 under single-letter updates. We extend this result by showing that a cyclic rotation of a length-n string by one letter can change the size of the LZ77 parsing only by $\mathcal{O}(\sqrt{n})$, and there are arbitrarily long strings for which the size of the LZ77 parsing changes by $\Theta(\sqrt{n})$.

The semi-dynamic/sliding window setting has been considered for maintaining the suffix tree [12, 21, 46], the Directed Acyclic Word Graph [29, 60], as well as for other stringology problems [3, 15, 48, 47, 49]. Early studies of the dynamic longest common subsequence and edit distance problems considered models where the allowed updates are a subset of prepend, append, and delete the first or last letter, see [66, 45, 39, 30]. Our results are *not* related to the problem considered in [10], where the "sliding window" considered there is a range in which previous occurrences of the phrases are limited to those starting inside the range.

Our Contributions

We consider the following problem with strings indexed from 0.

Semi-dynamic LZ Compression Size
Maintained object: A string S of length at most n along with $|\mathsf{LZ}(S)|$.
Update: Perform one of the two following operations:
delete: $S \to S[1\,..\,|S|-1]$; append(a) for $a \in \Sigma$: $S \to Sa$.

Our main result can be stated as follows.

▶ **Theorem 1.** *The* Semi-dynamic LZ Compression *problem admits a solution using $\mathcal{O}(n)$ space and $\mathcal{O}(\sqrt{n}\log^2 n)$ amortized update time.*

We also define the problem in the sliding window model.

Sliding Window LZ Compression Size
Input: A string S of length n and an integer $d \in [1\,..\,n]$.
Output: The size of $\mathsf{LZ}(S[i\,..\,i+d))$, for each $i = 0, \ldots, n-d$.

We define the rotation operation on a string S as $\mathsf{rot}(S) = S[1\,..\,|S|)S[0]$. The string obtained from S by r applications of the rotation operation is denoted $\mathsf{rot}^r(S)$, while $\mathsf{rot}^0(S) := S$.

Most LZ-Compressible Rotation
Input: A string S of length n.
Output: An integer $r \in [0\,..\,n)$ such that $\mathsf{LZ}(\mathsf{rot}^r(S))$ has the least number of phrases, that is, $\arg\min_{r\in[0\,..\,n)} |\mathsf{LZ}(\mathsf{rot}^r(S))|$.

This problem is a special case of Sliding Window LZ Compression Size. It suffices to iterate over all length-n factors of string S^2 using a sliding window.

In Fact 25 in Section 6, we show that some rotation of S can have $\Theta(\sqrt{|S|})$ fewer phrases than S, or $\frac{2}{3}z$ phrases, where $z = |\mathsf{LZ}(S)|$. This implies that storing the best rotation value and compressing the rotation can yield better compression.

As a baseline, the Sliding Window LZ Compression Size can be solved using Generalized Substring Compression queries [35] as follows.

▶ **Proposition 2.** *The* Sliding Window LZ Compression Size *problem can be solved in $\mathcal{O}(n\sqrt{\log n} + Z\log\log n)$ time and $\mathcal{O}(n\log\log n)$ space, where $Z = \sum_{i=0}^{n-d} |\mathsf{LZ}(S[i\,..\,i+d))|$ is the total number of phrases in the LZ77 factorizations of all length-d windows of S.*

Proof. Let us recall that in the Generalized Substring Compression problem, we are to preprocess a string S so that, given any factor $S[\ell\,..\,r]$ of S, we can compute the phrases in its LZ77 parsing efficiently. Keller et al. [35] presented an efficient data structure for answering Generalized Substring Compression queries based on range successor queries. If the later range successor data structure of Gao et al. [26] is used, we obtain an $\mathcal{O}(n\log\log n)$-size data structure that can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time and can answer a Generalized Substring Compression query for $S[\ell\,..\,r]$ in $\mathcal{O}(|\mathsf{LZ}(S[\ell\,..\,r])|\log\log n)$ time. ◀

Other trade-offs in the proposition are also possible; see [50, 26, 19]. For example, one can obtain linear space with $\mathcal{O}(n\sqrt{\log n} + Z\log^{\epsilon} n)$ time, for any $\epsilon > 0$ [50, 26]. Still, the time complexity of the algorithm of Proposition 2 is $\tilde{\mathcal{O}}(n+Z)$, which can be as bad as $\tilde{\mathcal{O}}(n^2)$ for poorly compressible texts. As a corollary of Theorem 1, we obtain the following result.

▶ **Corollary 3.** *The* Sliding Window LZ Compression Size *and* Most LZ-Compressible Rotation *problems can be solved in $\mathcal{O}(n\sqrt{n}\log^2 n)$ time using $\mathcal{O}(n)$ space.*

Technical Overview

At the heart of our solution, lies the maintenance of a dynamic tree that encodes the longest-previous-factor array information; the LZ77 factorization corresponds to a single path in this tree and this path's length can be efficiently retrieved by maintaining said tree using link-cut trees [61]. A periodicity-based argument allows us to show that deleting the first letter of S results in $\mathcal{O}(\sqrt{n})$ updates to our dynamic tree. Appending a letter to S might unfortunately lead to $\Omega(n)$ updates. However, all updated edges of the tree have the same target and consecutive sources. Moreover, the structure of those updates allows us to handle them efficiently in batches. Namely, we show that there are $\mathcal{O}(\sqrt{n})$ consecutive intervals of positions $[a \mathinner{.\,.} b]$ such that, for all elements i of each interval, the (rightmost) position of the longest previous factor starting at position i changes from some position j to some position j' and, for some integers x and y, for all $i \in [a \mathinner{.\,.} b]$, $i - j' = x$ and $i - j = y$. All in all, given the endpoints of the intervals and the changes in the (rightmost) positions of the longest previous factors, we update the tree structure in $\tilde{\mathcal{O}}(\sqrt{n})$ time in total by storing all edges with the same target using a joinable balanced binary search tree [64]. In order to exploit the above structural insights, we show that it suffices to maintain a data structure that allows us to efficiently retrieve the value $\mathsf{LPF}_S[i]$ for the elements of an $\tilde{\mathcal{O}}(\sqrt{n})$-size subset of $[1 \mathinner{.\,.} |S|)$.

We obtain this data structure by exploiting ideas that stem from internal string queries, such as the Interval LCP problem [35]. For a static string S, it suffices to combine a suffix tree and a 2D range successor data structure over an $n \times n$ grid, where, for each suffix $S[j \mathinner{.\,.} n)$, we have a point $(\mathsf{RANK}[j], j)$, where $\mathsf{RANK}[j]$ is the lexicographic rank of said suffix among the suffixes of S. For computing the positions of longest previous factors, we enhance this data structure with further range successor data structures. We then maintain such a data structure that efficiently answers the desired queries when the involved factors are contained in $S[0 \mathinner{.\,.} |S| - \mathcal{O}(\sqrt{n})]$. We overcome the technical challenge posed by the need to handle the remaining queries by analysing the periodic structure implied by "hard" such queries and batching them so that we only spend an additive $\mathcal{O}(\sqrt{n})$ factor overhead in the time complexity.

Structure of the paper. The auxiliary data structure for LPF computation in a semi-dynamic setting is described in Section 3. An abstract structure of the LPF-tree is defined in Section 4, where the periodicity-based arguments are also given. Implementations of operations on the tree are provided in Section 5. Finally, Section 6 considers the additive sensitivity of the size of the LZ77 parsing under a single rotation operation.

2 Preliminaries

Let $S = S[0]S[1] \cdots S[n-1]$ be a *string* (or *text*) of length $n = |S|$ over an integer alphabet Σ. The elements of Σ are called *letters*. For two positions i and j of S, we denote by $S[i \mathinner{.\,.} j]$ a string called a *factor* of S that starts at position i and ends at position j (the factor is empty, denoted by ε, if $i > j$). A factor of S can be represented in $\mathcal{O}(1)$ space by specifying the indices i and j of an occurrence of it. We define $S[i \mathinner{.\,.} j+1) = S[i \mathinner{.\,.} j] = S(i-1 \mathinner{.\,.} j]$. A string U is a proper prefix (resp. suffix) of S if there exists a non-empty string V such that $S = UV$ (resp. $S = VU$).

If a string B is both a proper prefix and a proper suffix of a length-n string S, then B is called a *border* of S. A positive integer p is called a *period* of S if $S[i] = S[i+p]$ for all $i \in [0 \mathinner{.\,.} n-p)$. String S has a period p if and only if it has a border of length $n-p$. We refer to the smallest period of S as *the period* of S, and denote it by $\mathsf{per}(S)$. String S is called *periodic* if $\mathsf{per}(S) \le n/2$.

▶ **Lemma 4** (Periodicity Lemma [22], weak version)**.** *If p and q are periods of a string S and satisfy $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

▶ **Lemma 5** ([11, 56, 42])**.** *The set of occurrences of a string X in a string Y can be expressed as a union of $\mathcal{O}(|Y|/|X|)$ arithmetic progressions such that the difference of each progression equals* $\mathsf{per}(X)$*. The intervals spanned by the progressions are disjoint.*

For a string S of length n, we define the following arrays indexed from 0 to $n-1$:

$$\mathsf{LPF}_S[i] = \max(\{\ell \geq 0 \,:\, S[j\mathinner{..}j+\ell) = S[i\mathinner{..}i+\ell),\, j < i\} \cup \{0\})$$
$$\mathsf{LPFpos}_S[i] = \max(\{j < i \,:\, S[j\mathinner{..}j+\mathsf{LPF}_S[i]) = S[i\mathinner{..}i+\mathsf{LPF}_S[i]) \neq \varepsilon\} \cup \{-1\}).$$

See Figure 3 in Page 11 for an example.

▶ **Theorem 6** (Corollary of [8])**.** *For a string S of length n, arrays LPF_S and LPFpos_S can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

Proof. Array LPF_S can be constructed in $\mathcal{O}(n)$ time [17], while array LPFpos_S can be constructed in $\mathcal{O}(n(1+\log\sigma/\sqrt{\log n})) = \mathcal{O}(n\sqrt{\log n})$ time [8]. ◀

Predecessor data structures. For a static set, a combination of x-fast tries [68] and deterministic dictionaries [59] yields the following efficient deterministic data structure.

▶ **Fact 7** ([23, Proposition 2])**.** *A sorted static set $Y \subseteq [1\mathinner{..}U]$ can be preprocessed in $\mathcal{O}(|Y|)$ time and space so that predecessor queries can be performed in $\mathcal{O}(\log\log|U|)$ time.*

A dynamic predecessor data structure over m integer keys can be stored using an exponential search tree [5] in $\mathcal{O}(m)$ space, supporting insertions, deletions, and predecessor queries in $\mathcal{O}(\log^2\log m/\log\log\log m)$ worst-case time.

3 Answering LPF Queries in a Batch in a Semi-dynamic String

In the semi-dynamic model, we will extensively use a solution to the following problem to compute previous occurrences of factors of the maintained string S.

Semi-dynamic Batch LPF
Maintained object: A string S of length at most n.
Update: Perform one of the two following operations:
- delete: $S \to S[1\mathinner{..}|S|-1]$;
- append(a) for $a \in \Sigma$: $S \to Sa$.

Query: Given a set $Y \subseteq [0\mathinner{..}|S|)$, compute $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for each $y \in Y$.

This section is devoted to an $\mathcal{O}(n)$-space solution for this problem with $\mathcal{O}(\sqrt{n\log n})$ update time and $\mathcal{O}(\sqrt{n\log n} + |Y|\log^{\epsilon} n)$ query time, for any $\epsilon > 0$. We start with a discussion of static algorithms for computing LPF and LPFpos.

A solution to the following static problem can be used to answer queries for LPF in a sliding window if the whole string is known in advance.

Interval LCP
Input: A string T of length n.
Query: Given a factor F of T and an interval $[i\mathinner{..}j]$, find the longest prefix F' of F that occurs in T at some position in $[i\mathinner{..}j]$ and a position $k \in [i\mathinner{..}j]$ such that $T[k\mathinner{..}k+|F'|) = F'$.

▶ **Theorem 8** (Keller et al. [35], Belazzougui et al. [8]). *The* Interval LCP *problem admits a solution with $\mathcal{O}(n)$ space, $\mathcal{O}(n\sqrt{\log n})$ construction time, and $\mathcal{O}(\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

To answer queries for LPFpos in a sliding window we would use an auxiliary problem called Interval LCP Position in which, in addition to the output of an Interval LCP query, we compute the rightmost position within the interval $[i\,..\,j]$ where the longest prefix of F occurs. Using range successor queries, one can obtain a solution for the Interval LCP Position problem with the complexities of Theorem 8.

Let us formally define *range successor* queries. We are given a set $\mathcal{P}$ of n points in an $n \times n$ grid. Given a range $[x\,..\,x'] \times [y\,..\,\infty)$ (or $[x\,..\,x'] \times (-\infty\,..\,y]$), we are to report a point of $\mathcal{P}$ that is included in the range and has a minimal y-coordinate (maximal y-coordinate, respectively). Clearly, the x and y coordinates can be interchanged in this definition.

▶ **Lemma 9.** *The* Interval LCP Position *problem admits a solution with $\mathcal{O}(n)$ space, $\mathcal{O}(n\sqrt{\log n})$ construction time, and $\mathcal{O}(\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

Proof. We compute in $\mathcal{O}(n)$ time the suffix array SA, the rank array RANK and the LCP array [34]. Let us recall the definitions of these arrays. The suffix array $\mathsf{SA}[0\,..\,n)$ is a permutation of $[0\,..\,n)$ such that:

$$T[\mathsf{SA}[0]\,..\,n) < T[\mathsf{SA}[1]\,..\,n) < \cdots < T[\mathsf{SA}[n-1]\,..\,n);$$

then $\mathsf{SA}[\mathsf{RANK}[i]] = i$ for all $i \in [0\,..\,n)$. The LCP array is defined as follows:

$$\mathsf{LCP}[i] = \max\{\ell \geq 0 : T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i]+\ell) = T[\mathsf{SA}[i+1]\,..\,\mathsf{SA}[i+1]+\ell)\} \text{ for } i \in [0\,..\,n-1).$$

Next, we perform $\mathcal{O}(n\sqrt{\log n})$-time preprocessing (see [8]) to construct an $\mathcal{O}(n)$-sized data structure for $\mathcal{O}(\log^{\epsilon} n)$-time range successor queries (see [50]) on two sets of points on $n \times n$ grids: set $\mathcal{P}_1 = \{(i, \mathsf{LCP}[i]) : i \in [0\,..\,n-1)\}$ and set $\mathcal{P}_2 = \{(i, \mathsf{RANK}[i]) : i \in [0\,..\,n)\}$; the set $\mathcal{P}_2$ was also used in [35]. Finally, we perform the preprocessing of Theorem 8.

Upon an Interval LCP Position query for a factor F and interval $[i\,..\,j]$, we first use an Interval LCP query to compute the longest prefix F' of F that occurs at some position in $[i\,..\,j]$ and a position $k \in [i\,..\,j]$ such that $T[k\,..\,k+|F'|) = F'$. We would like to compute the maximum index $k' \in [k\,..\,j]$ such that $T[k'\,..\,k'+|F'|) = F'$.

First, we use range successor queries on the set of points $\mathcal{P}_1$ to locate the range $[\ell\,..\,r]$ in the suffix array that contains all the suffixes that have a longest common prefix with suffix $T[k\,..\,n]$ of length at least $|F'|$. Namely, to compute r, we find the smallest x-coordinate of a point from $\mathcal{P}_1$ in the range $[\mathsf{RANK}[k]\,..\,\infty) \times [0\,..\,|F'|)$. If there is no such point, $r = n-1$, and otherwise r is the computed x-coordinate. The computation of ℓ is symmetric.

Now, among the suffixes that correspond to $\mathsf{SA}[\ell\,..\,r]$, we would like to find the suffix occurring at a maximum position that is at most j. We ask a range successor query on the set of points $\mathcal{P}_2$ to find the maximum y-coordinate of a point in $[\ell\,..\,r] \times (-\infty\,..\,j]$; the returned y-coordinate is the sought position k'.

The Interval LCP query and each range successor query take $\mathcal{O}(\log^{\epsilon} n)$ time, for any $\epsilon > 0$ [8, 35]. ◀

The data structure of Lemma 9 essentially consists of the suffix tree of T (which is used in the data structure underlying Theorem 8) and range successor data structures. The corollary below follows from the work of Keller et al. [35] and Lemma 9.

▶ **Corollary 10.** *A string T of length n can be preprocessed in $\mathcal{O}(n\sqrt{\log n})$ time so that, given a string F and an interval $[i\,..\,j]$, computing the longest prefix of F that occurs in T at some position in $[i\,..\,j]$, as well as the rightmost position in $[i\,..\,j]$ at which it occurs, reduces in $\mathcal{O}(\log^{\epsilon} n)$ time, for any $\epsilon > 0$, to computing the locus of L in the suffix tree of T, where L is the longest prefix of F that occurs in T.*

We are now ready to proceed to semi-dynamic computation of LPF and LPFpos.

▶ **Lemma 11.** *The* Semi-dynamic Batch LPF *problem admits an $\mathcal{O}(n)$-space solution with $\mathcal{O}(\sqrt{n\log n})$ update time and $\mathcal{O}(\sqrt{n\log n} + |Y|\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

Proof. We will be rebuilding some data structures over the string S after every $\lfloor\sqrt{n}\rfloor$ updates. We will keep the update-time bound worst-case by using the so-called time slicing technique, that is, splitting the work required for the construction of the data structure into roughly equal chunks and distributing them among the subsequent $\lfloor\sqrt{n}\rfloor$ updates; if the data structures can be constructed in $\tilde{\mathcal{O}}(n)$ time, then we will spend $\tilde{\mathcal{O}}(\sqrt{n})$ time on each single update. Let S_1 be the current string S, S_2 be S_1 after $\lfloor\sqrt{n}\rfloor$ updates, and S_3 be S_2 after $\lfloor\sqrt{n}\rfloor$ updates. The data structure for S_1 will be ready by the time S_2 is processed and will be used until S_3 is reached, at which point the data structure that is constructed for S_2 will be ready. This way, when processing the current string S, we can assume that we have access to data structures for a string $Z = VS[0\,..\,|S|-x)$, where $x \le 2\sqrt{n}$ and V is a string composed of all deleted letters since the construction of this data structure was issued.

The data structures that are stored for Z include the static data structure of Corollary 10 for the Interval LCP Position problem that takes $\mathcal{O}(n)$ space and $\mathcal{O}(n\sqrt{\log n})$ time to construct; the suffix tree of Z augmented in $\mathcal{O}(n)$ time with the weighted-ancestor-queries data structure of [7], which allows one to retrieve in $\mathcal{O}(1)$ time the locus of any given factor of Z in the suffix tree of Z, and other $\mathcal{O}(n)$-time constructible data structures based on the suffix tree of Z to be specified later.

Let us now discuss how to compute $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for all $y \in Y$. We compute at most three candidate values for $\mathsf{LPF}_S[y]$ for each $y \in Y$, together with candidate positions $\mathsf{LPFpos}_S[y]$, and we take the maximum LPF value and the corresponding position in the end.

Case I. $\mathsf{LPFpos}_S[y] \ge d := |S| - 10\lceil\sqrt{n}\rceil$, so $y > d$. We compute the $\mathsf{LPF}_{S'}$ and $\mathsf{LPFpos}_{S'}$ arrays for $S' = S[d\,..\,|S|)$ in $\mathcal{O}(\sqrt{n\log n})$ time (Theorem 6). For each $y \in Y$ with $y > d$, we have a candidate $\mathsf{LPF}_{S'}[y-d]$ for $\mathsf{LPF}_S[y]$ along with candidate $\mathsf{LPFpos}_{S'}[y-d]$ for $\mathsf{LPFpos}_S[y]$.

Case II. $\mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y] < |S| - x$. (Let us note that, however, $y + \mathsf{LPF}_S[y]$ can be as large as $|S|$ in this case.) Our main aim is to compute, for each $y \in Y$, the locus of the longest prefix L_y of $S[y\,..\,|S|)$ that occurs in $S[0\,..\,|S|-x)$. Let the elements of Y in increasing order be $y_0, y_1, \ldots$; we process them in this order. Starting from the locus of $S[y_0\,..\,|S|-x)$ which we compute in $\mathcal{O}(1)$ time using a weighted ancestor query, we go down in the suffix tree letter by letter with the aim of computing L_{y_0}. Note that we follow an edge as long as the node we reach corresponds to a factor of $S[0\,..\,|S|-x)$ (and not just of Z); this can be checked in constant time after a linear-time bottom-up preprocessing of the suffix tree. When we have found the locus of L_{y_0} we do the following. From the suffix tree, we obtain an index i such that $L_{y_0} = Z[i\,..\,i+|L_{y_0}|)$. In constant time, using a weighted ancestor query, we go to the locus of $Z[i+(y_1-y_0)\,..\,i+|L_{y_0}|)$ and start the search for L_{y_1} from there; and so on. We process $|Y|$ suffixes and only have $x = \mathcal{O}(\sqrt{n})$ letters to extend them by. The total time required for the described process is thus $\mathcal{O}(|Y| + \sqrt{n}\log\log n)$ assuming that the children of a node in the suffix tree of Z are stored using the predecessor data structure of Fact 7. We then use Corollary 10 to compute a pair of candidates for $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for each $y \in Y$, spending $\mathcal{O}(\log^{\epsilon} n)$ time for each y.

Case III. $\mathsf{LPFpos}_S[y] < |S| - 10\sqrt{n}$ and $\mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y] \geq |S| - x$. We have

$$y \leq |S| - \mathsf{LPF}_S[y] \leq \mathsf{LPFpos}_S[y] + x \leq \mathsf{LPFpos}_S[y] + 2\sqrt{n}, \text{ and}$$

$$\mathsf{LPF}_S[y] \geq |S| - x - \mathsf{LPFpos}_S[y] > 10\sqrt{n} - x \geq 8\sqrt{n}.$$

Hence, factors $S[y \mathinner{.\,.} y + \mathsf{LPF}_S[y])$ and $S[\mathsf{LPFpos}_S[y] \mathinner{.\,.} \mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y])$ start at most $2\sqrt{n}$ positions apart and overlap by more than $6\sqrt{n}$ positions. This means that $S[\mathsf{LPFpos}_S[y] \mathinner{.\,.} y + \mathsf{LPF}_S[y])$ is periodic with period at most $2\sqrt{n}$. In particular, due to the periodicity lemma (Lemma 4), the period of this factor must be equal to the period of $S[|S| - \lfloor 8\sqrt{n} \rfloor \mathinner{.\,.} |S| - x)$.

We can compute the period p of $S[|S| - \lfloor 8\sqrt{n} \rfloor \mathinner{.\,.} |S| - x)$ in $\mathcal{O}(\sqrt{n})$ time using the Morris-Pratt algorithm [41]. If $p \leq 2\sqrt{n}$, we compute the maximal factor $S[\ell \mathinner{.\,.} r]$ of S that contains $S[|S| - \lfloor 8\sqrt{n} \rfloor \mathinner{.\,.} |S| - x)$ and has the same period; the periodicity can be extended to the left in constant time after a linear-time preprocessing of Z (using longest common extension queries [9]) and to the right in $\mathcal{O}(\sqrt{n})$ time using letter comparisons. Then, for each $y \in [\ell + p \mathinner{.\,.} r]$, we have a candidate $r - y + 1$ for $\mathsf{LPF}_S[y]$ along with candidate $y - p$ for $\mathsf{LPFpos}_S[y]$. ◀

▶ Remark 12. For the purposes of Sliding Window LZ Compression Size problem for a string S, instead of Lemma 11, one could simply build the data structure encapsulated in Lemma 9 for S and use it to compute all required values $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ in the implied instance of the Semi-dynamic Batch LPF problem.

4 Properties of Longest Previous Factors and LPF-Tree

We next show two properties of the LPF and LPFpos arrays. The first of them bounds the number of zeroes in the LPFpos_U array.

To prove the lemma, we show that the values $\mathsf{LPF}_U[i]$ for increasing positions i such that $\mathsf{LPFpos}_U[i] = 0$ are strictly increasing. This, together with the fact that no three factors of the form $U[i \mathinner{.\,.} i + \mathsf{LPF}_U[i])$ for these positions overlap, shows that there are $\mathcal{O}(\sqrt{n})$ such positions. The aforementioned fact follows by periodicity.

▶ **Lemma 13.** *For a string U of length n, the number of positions $i \in [0 \mathinner{.\,.} n)$ such that $\mathsf{LPFpos}_U[i] = 0$ is $\mathcal{O}(\sqrt{n})$.*

Proof. Let $I = \{i \in [1 \mathinner{.\,.} n) : \mathsf{LPFpos}_U[i] = 0\}$. First, let us note that, for any $i, i' \in I$ with $i < i'$, we have $\mathsf{LPF}_U[i] < \mathsf{LPF}_U[i']$. Indeed, if we had $\mathsf{LPF}_U[i] \geq \mathsf{LPF}_U[i']$, then, for $\ell = \mathsf{LPF}_U[i']$, we would have $U[i \mathinner{.\,.} i + \ell) = U[0 \mathinner{.\,.} \ell) = U[i' \mathinner{.\,.} i' + \ell)$, which would imply $\mathsf{LPFpos}_U[i'] \geq i > 0$, yielding a contradiction.

By the above, to prove the statement of the lemma, it suffices to show that if $i \in I$, then there is at most one element $i' \in I \cap (i \mathinner{.\,.} i + \lfloor \frac{1}{2}\mathsf{LPF}_U[i] \rfloor]$.

Assume that such elements i and i' exist. The setting is illustrated in Figure 1. We will show that then i' is determined uniquely for i. For $\ell = \mathsf{LPF}_U[i]$, we have that $U[i \mathinner{.\,.} i + \ell) = U[i' \mathinner{.\,.} i' + \ell)$ is a border of $U[i \mathinner{.\,.} i' + \ell)$. Hence, $U[i \mathinner{.\,.} i' + \ell)$ has a period $p := i' - i \leq \ell/2$ and is thus periodic. Let q be the smallest period of $U[i \mathinner{.\,.} i' + \ell)$. By the periodicity lemma (Lemma 4), q divides p.

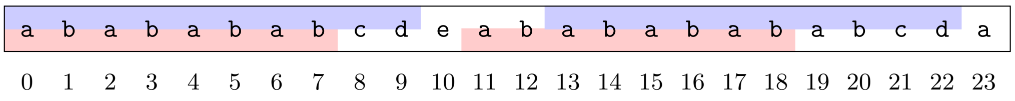


Figure 1 An illustration of the proof of Lemma 13. For $i = 11$ and $i' = 13$, we have $\mathsf{LPFpos}_U[i] = \mathsf{LPFpos}_U[i'] = 0$, $\ell = \mathsf{LPF}_U[i] = 8$ and $\ell' = \mathsf{LPF}_U[i'] = 10$. In this example, we have $q = 2$.

By definition, $U[i\,..\,i+\ell) = U[0\,..\,\ell)$, so $U[0\,..\,\ell)$ has period q. Moreover, $U[i+\ell] = U[i+\ell-q] = U[\ell-q]$, where the first equality follows from the fact that position $i+\ell$ is within the factor $U[i\,..\,i'+\ell)$. Therefore, $U[\ell] \neq U[\ell-q]$, because otherwise we would have $U[i\,..\,i+\ell] = U[0\,..\,\ell]$ and $\mathsf{LPF}_U[i] > \ell$. This means that $U[0\,..\,\ell]$ does *not* have period q, i.e., $U[i'\,..\,i'+\ell]$ does not have period q (as $\mathsf{LPF}_U[i'] > \ell$).

Let r be the smallest position such that $r \geq i+q$ and $U[i\,..\,r]$ does not have period q. We must have $i' = r - \ell$. All in all, i' is uniquely determined by i in U. ◀

▶ Remark 14. The bound from Lemma 13 is tight. Let $a_1, \ldots, a_m$ be distinct letters and consider strings $S_i = a_1 a_2 \cdots a_i$. Then the string $U = S_m S_1 S_2 \cdots S_m$ has length $\Theta(m^2)$ and for each starting position $j > 0$ of some S_i, for $i = 1, \ldots, m$, we have $\mathsf{LPFpos}_U[j] = 0$.

Let us define $\mathsf{LPFpos}'_S[i] = i - \mathsf{LPFpos}_S[i]$.

The next lemma characterizes the positions i such that $i + \mathsf{LPF}_U[i] = |U|$. There can be many such positions, even $\Theta(n)$, say, for a unary string $U = \mathsf{a}^n$. However, there are only $\mathcal{O}(\sqrt{n})$ different values $\mathsf{LPFpos}'_U[i]$ for such positions. Here, we need to consider the LPFpos'_U array and not the LPFpos_U array, as the latter can have $\Theta(n)$ different values for the positions of the considered type; the unary string $U = \mathsf{a}^n$, for which $\mathsf{LPFpos}_U[i] = i-1$ for each $i \in [0\,..\,n)$, is an example.

In the proof it suffices to consider positions $i \leq n - \sqrt{n}$ such that $i + \mathsf{LPF}_U[i] = |U|$. Each such position implies an occurrence of a length-$\lfloor\sqrt{n}\rfloor$ suffix V of U in V. In turn, the occurrence of V determines the value of $\mathsf{LPFpos}'_U[i]$. We consider all possible occurrences of V in U as $\mathcal{O}(\sqrt{n})$ arithmetic progressions (cf. Lemma 5) and use periodicity to show that $\mathcal{O}(1)$ occurrences in each progression can be implied in the aforementioned sense.

▶ **Lemma 15.** *For a string U of length n, among all positions $i \in [0\,..\,n)$ for which $i + \mathsf{LPF}_U[i] = n$, there are $\mathcal{O}(\sqrt{n})$ different values $\mathsf{LPFpos}'_U[i]$.*

Proof. We denote $s = n - \lfloor\sqrt{n}\rfloor$. Obviously, there are at most $\lfloor\sqrt{n}\rfloor - 1$ different values $\mathsf{LPFpos}'_U[i]$ for $i \in (s\,..\,n)$.

Let $V = U[s\,..\,n)$. Note that if $\mathsf{LPFpos}_U[i] = j$ for some $i \in [0\,..\,s]$ with $i + \mathsf{LPF}_U[i] = n$, then there is an occurrence of V in U at position $j + (s-i)$. In this case, we say that the occurrence of V in U at position $j + (s-i)$ is *implied* by position i or that position i *implies* the occurrence.

By Lemma 5, the set of occurrences of V in U consists of $\mathcal{O}(\sqrt{n})$ maximal arithmetic progressions with common difference $\mathsf{per}(V)$. Let J be one of these arithmetic progressions. We will show that there are at most two elements $p \in J$ such that the occurrence of V at position p is implied by any position $i \in [0\,..\,s]$. This will conclude the proof, as for all positions i that imply an occurrence of V at position p, the value $\mathsf{LPFpos}'_U[i] = s - p$ is the same. Let $U[n-d\,..\,n)$ be the longest suffix of U with period $\mathsf{per}(V)$. Figure 2 contains an illustration of possible cases considered below.

Let J_0 be the arithmetic progression containing position s. Assume first that position $i \in [0\,..\,s]$ implies an occurrence of V at a position p in $J = J_0$. We must have $|J_0| > 1$. If $i \geq n - d$, then $U[i\,..\,n)$ has period $\mathsf{per}(V)$. We have $U[j\,..\,j + \mathsf{LPF}_U[i]) = U[i\,..\,n)$, so

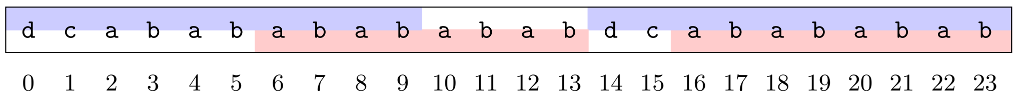


Figure 2 An illustration of the proof of Lemma 15. Let $V = \mathtt{abab}$. We have $n - d = 16$ and $\{i \in [0\,..\,23] : i + \mathsf{LPF}_U[i] = 24\} = [14\,..\,23]$. Let $J = \{2, 4, 6, 8, 10\}$ be an arithmetic progression of occurrences of V in U. Positions 14 and 15 imply the occurrence of V at position 6; the corresponding equality $U[0\,..\,9] = U[14\,..\,23]$ is illustrated using the blue top rectangles. Positions 16 and 17 imply the occurrence of V at position 10; the corresponding equality $U[6\,..\,13] = U[16\,..\,23]$ is illustrated using the red bottom rectangles. For each position $i \in [18\,..\,23]$, we have $\mathsf{LPFpos}'_U[i] = \mathsf{per}(V) = 2$.

$j \geq n - d$, as otherwise $U[j\,..\,j + \mathsf{LPF}_U[i])$ would not have period $\mathsf{per}(V)$. One cannot have $j \in (i - \mathsf{per}(V)\,..\,i)$, as this would imply an additional occurrence of V that is not in the progression. We always select the rightmost position, so $p = s - \mathsf{per}(V)$ is determined uniquely (and $j = i - \mathsf{per}(V)$). Now we need to note that the case that $i < n - d$ is impossible. Indeed, in this case the longest suffix of $U[i\,..\,n)$ that has period $\mathsf{per}(V)$ has length d, the longest suffix of $U[j\,..\,j + \mathsf{LPF}_U[i])$ that has period $\mathsf{per}(V)$ has length $d - (i - j)$, i.e., smaller than d, but $U[j\,..\,j + \mathsf{LPF}_U[i]) = U[i\,..\,n)$.

Henceforth we assume that $J \neq J_0$. For $r = \max J + |V|$, let $U[r - d'\,..\,r)$ be the longest suffix of $U[0\,..\,r)$ with period $\mathsf{per}(V)$. Assume that position $i \in [0\,..\,s]$ implies an occurrence of V at a position in J. If $n - i \leq \min(d, d')$, we have $i \in [n - d\,..\,n)$ and $U[i\,..\,n) = U[\max J + |V| - (n - i)\,..\,\max J + |V|)$. Hence, since $\mathsf{LPFpos}_U[i]$ stores the rightmost value in case of ties, the implied occurrence of V is the one starting at position $\max J$. Otherwise, the factor equality implied by $i + \mathsf{LPF}_U[i] = n$ means that $d \leq d'$. Thus we have $i < n - d$ and $U[i\,..\,n)$ does not have period $\mathsf{per}(V)$. Now, there is exactly one position $p \in J$ such that $U[p + |V| - d\,..\,p + |V|)$ has period $\mathsf{per}(V)$, but $U[p + |V| - d - 1\,..\,p + |V|)$ does not have this period. Namely, $p = \max J - (d' - d)$ and we must have $d' \equiv d \pmod{\mathsf{per}(V)}$. The occurrence of V at position p is the one implied by position i in this case. This concludes the proof that at most two occurrences of V in J can be implied by any position $i \in [0\,..\,s]$, and hence the whole proof. ◀

▶ Remark 16. The bound from Lemma 15 is tight. Let $a_1, \ldots, a_m$ be distinct letters and consider strings $S'_i = a_i a_{i-1} \cdots a_1$. Then the string $U = S'_m S'_{m-1} \cdots S'_1 S'_m$ has length $\Theta(m^2)$, $j + \mathsf{LPF}_U[j] = |U|$ for all $j \in [|U| - m\,..\,|U|)$ and all values $\mathsf{LPFpos}'_U[j]$, for $j \in [|U| - m\,..\,|U|)$, are different.

LPF-tree

Let us define $\mathsf{LPF}'_S[i] = \max(1, \mathsf{LPF}_S[i])$. We define an *LPF-tree* for a string S as a tree with nodes $[0\,..\,|S|]$, among which $|S|$ is the root, and edges $\{(i, i + \mathsf{LPF}'_S[i]) : i = 0, \ldots, |S| - 1\}$. The single edge outgoing from i has *label* equal to $\mathsf{LPFpos}'_S[i]$; see Figure 3.

From Theorem 6 we obtain the following.

▶ **Corollary 17.** *The LPF-tree of a string S of length n can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

The LZ77 parsing of a string S can be computed straightforwardly from the LPF_S and LPFpos_S arrays in a greedy manner; cf. [16, 17]. We make the following simple observation.

▶ **Observation 18.** *Let π be the 0-to-$|S|$ path in the LPF-tree of S. Then, $|\mathsf{LZ}(S)|$ equals the number $|\pi|$ of edges of π. Additionally, for $k \leq |\pi|$, if the k-th edge on π is $(i, i + \mathsf{LPF}'_S[i])$, then the k-th phrase of $\mathsf{LZ}(S)$ is equal to $S[i\,..\,i + \mathsf{LPF}'_S[i])$.*

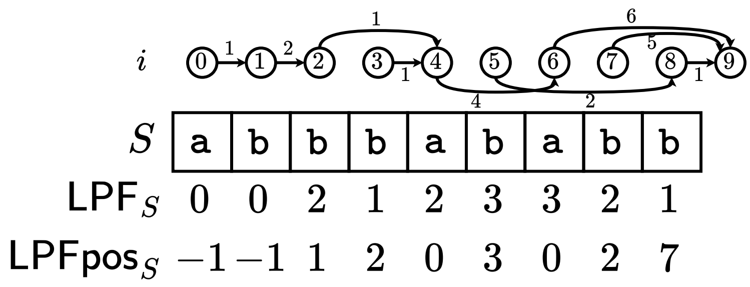

i
0 1 2 3 4 5 6 7 8 9
1 2 1 1 4 6 5 2 1
S a b b b a b a b b
LPF_S 0 0 2 1 2 3 3 2 1
LPFpos_S −1 −1 1 2 0 3 0 2 7

Figure 3 Example of an LPF-tree for the string abbbababb.

We need to extend the definition of an LPF-tree to the semi-dynamic setting. Let U be a semi-dynamic string that was obtained from an initial string by using $a = \mathsf{deletions}(U)$ first-letter deletions and some number of append operations. The nodes and edges of a *semi-dynamic LPF-tree* for U are, respectively, $[a\,..\,a+|U|]$, among which $a+|U|$ is the root, and $\{(i, i+\mathsf{LPF}'_U[i-a]) \,:\, i \in [a\,..\,a+|U|)\}$. The single edge outgoing from i has label equal to $\mathsf{LPFpos}'_U[i-a]$.

▶ **Lemma 19.** *Let us consider any string U and the semi-dynamic LPF-tree T of U. The sources of all edges of T that enter a given node form a set of consecutive integers.*

Proof. Let $a = \mathsf{deletions}(U)$. It suffices to show that for any $i \in [0\,..\,|U|-2]$, we have $i+\mathsf{LPF}'_U[i] \le i+1+\mathsf{LPF}'_U[i+1]$ (equivalently, $i+a+\mathsf{LPF}'_U[i] \le i+1+a+\mathsf{LPF}'_U[i+1]$).

The conclusion is obvious if $\mathsf{LPF}'_U[i] = 1$. Hence, we assume that $\mathsf{LPF}'_U[i] > 1$, so $\mathsf{LPF}'_U[i] = \mathsf{LPF}_U[i]$. Let $\ell = \mathsf{LPF}_U[i]$ and $p = \mathsf{LPFpos}_U[i] < i$. We have $U[p\,..\,p+\ell) = U[i\,..\,i+\ell)$ and hence $U[p+1\,..\,p+\ell) = U[i+1\,..\,i+\ell)$. Consequently, $i+1+\mathsf{LPF}'_U[i+1] \ge i+\ell$, as required. ◀

5 Updating a Semi-dynamic LPF-tree

We start by introducing an abstract data structure that will be used to store labels of edges of an LPF-tree. The data structure stores key-value pairs such that, for each pair, the value is smaller than the key. The set of keys at any time is a set of consecutive integers and is denoted by I. The size of I is denoted by n and the set of values is a subset of $[0\,..\,n)$. The following operations are supported:

(a) insertion of a pair with key $\max I + 1$, where I is the current interval of keys, or deletion of a pair with key $\min I$,

(b) setting the value of all keys in a specified interval $[k_1\,..\,k_2]$ to a given integer v, knowing that their values were all equal to an integer v',

(c) reporting all key-value pairs for which the difference between the key and the value is minimal.

Let us call this data structure $\mathcal{D}$. Let $\pi_n = (\log\log n)^2/\log\log\log n$.

▶ **Lemma 20.** *Data structure $\mathcal{D}$ can be implemented in $\mathcal{O}(n)$ space so that each operation* (a), (b) *is performed in $\mathcal{O}(\pi_n)$ time and each pair in operation* (c) *is reported in $\mathcal{O}(\log\log n)$ time.*

Proof. For each value, we store the set of keys with this value as a collection of maximal intervals in a dynamic predecessor data structure. The predecessor data structures are stored in dynamic predecessor data structure indexed by values 0 through $n-1$.

Moreover, for each value, the minimum key with this value is determined and such key-value pairs are stored in a min-type priority queue ordered by the difference between the key and the value. Whenever an operation is performed on one of the predecessor data structures, the priority queue is updated accordingly.

Insertion in operation (a) requires to insert a singleton interval $\{\max I + 1\}$ to the predecessor data structure for its value, possibly merging the interval with the previous one. Deletion in operation (a) requires to remove the first element of the first interval in the predecessor data structure for this value.

In operation (b) we identify the (at most one) interval I in the predecessor data structure for value v' that contains $[k_1 \mathinner{.\,.} k_2]$ as a sub-interval. Then I is replaced by the at most two intervals $I \setminus [k_1 \mathinner{.\,.} k_2]$, and the interval $[k_1 \mathinner{.\,.} k_2]$ is inserted into the predecessor data structure for v, possibly being merged with any adjacent intervals.

In operation (c), elements are removed from the priority queue one by one and reported until the next element has a different priority. Afterwards they are reinserted to the priority queue.

We use the dynamic predecessor data structure [5] that requires $\mathcal{O}(m)$ space on m elements and supports queries in $\mathcal{O}(\pi_n)$ time. The priority queue requires $\mathcal{O}(\log\log n)$ time per operation [65]. ◀

Let U be a semi-dynamic string with $\mathsf{deletions}(U) = a$. The labels of edges of a semi-dynamic LPF-tree for U will be stored as a set of key-value pairs with keys $i \in [a \mathinner{.\,.} a + |U|)$ and values $\mathsf{LPFpos}'_U[i-a]$ using data structure $\mathcal{D}$. Only edges for which $\mathsf{LPFpos}_U[i-a] \neq -1$ are stored in $\mathcal{D}$.

In the two lemmas below, we use the data structure $\mathcal{D}$ together with the Semi-dynamic Batch LPF data structure (Lemma 11) to efficiently compute the updates that need to be performed on the semi-dynamic LPF-tree upon each of the single-letter updates on the string considered in the semi-dynamic setting. The actual operations on the LPF-tree will be performed in Section 5.1 when we define the data structure representing the LPF-tree.

▶ **Lemma 21.** *Let U be a semi-dynamic string of length n and U' be U after the deletion of its first letter. The semi-dynamic LPF-tree for U' can be obtained from the semi-dynamic LPF-tree for U by updating $\mathcal{O}(\sqrt{n})$ edges.*

The set of edges to be updated, as well as their new labels, can be computed in $\mathcal{O}(\sqrt{n}\log n)$ time. Data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time.

Proof. Let $a = \mathsf{deletions}(U)$. First, the edge from a needs to be removed. No operation on $\mathcal{D}$ is required, as $\mathsf{LPFpos}_U[0] = -1$.

Then, let us note that if $\mathsf{LPFpos}_U[i-a] > 0$ for $i \in (a \mathinner{.\,.} a+n)$, then $\mathsf{LPF}'_U[i-a] = \mathsf{LPF}'_{U'}[i-(a+1)]$ and $\mathsf{LPFpos}_U[i-a] + a = \mathsf{LPFpos}_{U'}[i-(a+1)] + (a+1)$, so $\mathsf{LPFpos}'_U[i-a] = \mathsf{LPFpos}'_{U'}[i-(a+1)]$. Hence, only edges $(i, i + \mathsf{LPF}'_U[i-a])$ with $\mathsf{LPFpos}_U[i-a] = 0$, i.e., $\mathsf{LPFpos}'_U[i-a] = i-a$, remain to be updated. The bound on the number of such edges follows by Lemma 13.

The edges to be updated can be retrieved in $\mathcal{O}(\sqrt{n}\log\log n)$ time using data structure $\mathcal{D}$. Indeed, for such an edge, the difference of the key and the value of the corresponding pair in $\mathcal{D}$ satisfies $i - \mathsf{LPFpos}'_U[i-a] = a$. Moreover, if $\mathsf{LPFpos}_U[i-a] > 0$ for $i \in (a \mathinner{.\,.} a+n)$, then $i - \mathsf{LPFpos}'_U[i-a] = a + \mathsf{LPFpos}_U[i-a] > a$. Therefore, the sought edges can be obtained via operation (c) on data structure $\mathcal{D}$.

For each edge $(i, i + \mathsf{LPF}'_U[i-a])$ with label $\mathsf{LPFpos}'_U[i-a] = i-a$ that we remove, we have to insert edge $(i, i + \mathsf{LPF}'_{U'}[i-(a+1)])$ with label $\mathsf{LPFpos}'_{U'}[i-(a+1)]$. (The edge is inserted to $\mathcal{D}$ as well only if $\mathsf{LPFpos}_{U'}[i-(a+1)] \neq -1$.) We compute the targets and labels of these edges in $\mathcal{O}(\sqrt{n}\log n)$ time using Lemma 11. According to Lemma 20, data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ total time via $\mathcal{O}(\sqrt{n})$ calls to operation (b), each with a singleton interval. ◀

▶ **Lemma 22.** *Let U be a semi-dynamic string of length n and $U' = Uc$, for some letter c. The semi-dynamic LPF-tree for U' can be obtained from the semi-dynamic LPF-tree for U by adding a new root and an edge from the old root to the new root, as well as redirecting some number of edges with consecutive sources that lead to the old root to point to the new root.*

The set of edges to be updated, represented as $\mathcal{O}(\sqrt{n})$ groups of edges with consecutive sources, equal old label and equal new label, can be computed in $\mathcal{O}(\sqrt{n}\log^{1.5} n)$ time. Data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time.

Proof. Let $a = \mathsf{deletions}(U)$. We first create a new node $a+n+1$, designate it to be the root, and insert an edge $(a+n, a+n+1)$ with label decided by an invocation of Lemma 11 in $\mathcal{O}(\sqrt{n}\log n)$ time. (The edge is inserted to $\mathcal{D}$ if only $\mathsf{LPFpos}_{U'}[n] \neq -1$.) Then, since

$$i + \mathsf{LPF}_U[i-a] \leq i + \mathsf{LPF}_{U'}[i-a] \leq i + \mathsf{LPF}_U[i-a] + 1,$$

for all i, all we need to do is compute the nodes i such that $i + \mathsf{LPF}_U[i-a] = a+n$ and $i + \mathsf{LPF}_{U'}[i-a] = a+n+1$. Due to Lemma 19, these nodes form a set R of consecutive integers and, in particular, $R = (r \mathinner{.\,.} a+n)$ for some $r \in [a \mathinner{.\,.} a+n)$.

We can compute r in $\mathcal{O}(\sqrt{n}\log^{1.5} n)$ time using binary search and Lemma 11. It remains to partition R into sub-intervals with the same $\mathsf{LPFpos}'_{U'}$ and LPFpos'_U values. Note that the $\mathsf{LPFpos}'_{U'}$ and LPFpos'_U values in R are non-decreasing since an occurrence of a length-ℓ suffix of U or U' at a position p, implies an occurrence of any suffix of length $\ell-\mu$ for a positive integer μ at position $p+\mu$.

By Lemma 15, there are $\mathcal{O}(\sqrt{n})$ possible values of $\mathsf{LPFpos}'_{U'}[i]$, for $i \in R$. We next show how to compute the partition of R by values $\mathsf{LPFpos}'_{U'}[i]$ in $\tilde{\mathcal{O}}(\sqrt{n})$ time using Lemma 11. We maintain a set of disjoint active intervals whose union is the set of positions for which we have not yet computed the value of $\mathsf{LPFpos}'_{U'}$. Initially, our set of active intervals is $\{R\}$. Then, until there are no active intervals left, for each active interval J, in the order of decreasing size, we do the following. We compute $\mathsf{LPFpos}'_{U'}$ for $\min J$, $\max J$, and the midpoint $j = \lfloor(\min J + \max J)/2\rfloor$ of J. If the interval is of size at most three, we partition it to three singletons which are marked as inactive and labeled with the corresponding $\mathsf{LPFpos}'_{U'}$ values. Else, for the most distant $x, y \in \{\min J, j, \max J\}$ such that $\mathsf{LPFpos}'_{U'}[x-a] = \mathsf{LPFpos}'_{U'}[y-a]$, if they exist, we designate $[x \mathinner{.\,.} y]$ as inactive and label it with $\mathsf{LPFpos}'_{U'}[x-a]$. The remaining positions yield at most two active intervals, by splitting J at its midpoint j. We can think of this algorithm proceeding in levels, where at level λ we process those active intervals that have been obtained via λ splits. At each level, we sweep the intervals in a left-to-right manner, repeatedly merging consecutive intervals with the same label, so that the resulting interval inherits that label. Since in each level other than the first one each active interval contains an endpoint of the sought partition of R, we have at most $\mathcal{O}(\sqrt{n})$ active intervals in each level. As the sizes of active intervals decrease by a constant factor in each level, we have $\mathcal{O}(\log n)$ levels. We batch the $\mathcal{O}(\sqrt{n})$ queries for each level and answer them using Lemma 11. The total time required for partitioning R is thus $\mathcal{O}(\sqrt{n}\log^{1.5} n)$.

Next, we partition R by values $\mathsf{LPFpos}'_U[i]$ using the same algorithm in $\tilde{\mathcal{O}}(\sqrt{n})$ time. Finally, we compute an intersection of the two partitions, as desired, in $\mathcal{O}(\sqrt{n})$ time.

To update the data structure $\mathcal{D}$ using Lemma 20, we insert the edge from the old root to the new root using operation (a) and then perform the operation (b) on each of the $\mathcal{O}(\sqrt{n})$ groups of edges that are being redirected. In total, the data structure is updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time, as desired. ◀

5.1 Implementation of a Semi-Dynamic LPF-tree using Link-cut Trees and Joinable Balanced BSTs

A link-cut tree is a classic data structure [61] that represents a forest of rooted trees containing n nodes in total. Each node stores an integer weight. The data structure has size $\mathcal{O}(n)$ and supports the following operations in amortized $\mathcal{O}(\log n)$ time:

- add a tree consisting of a single node to the forest;
- remove a tree consisting of a single node from the forest;
- attach a root node to another node as its child (*link* operation);
- given a node in one of the trees, disconnect it (and its subtree) from the tree of which it is part to form a separate tree (*cut* operation);
- add a given value α to the weights of all the descendants of a node;
- return the weight of a given node.

For implementations of operations including weights, see for example [40, Appendix: Splay trees and link-cut trees].

It is well-known (cf. [64, pp. 45-56]) that a collection of red-black trees (RB trees) containing n integer keys in total can support the following operations, each in $\mathcal{O}(\log n)$ time:

- insert an element to an RB tree;
- delete a given element from an RB tree;
- join two RB trees into one RB tree, provided that all keys in one of the trees are smaller than all keys in the other (the arguments of the join operation are not kept);
- split an RB tree into two RB trees, one containing all keys smaller than a specified parameter k and the other containing the remaining keys (again, the initial RB tree is not kept).

We note that every operation on an RB tree (adding a new leaf, removing a leaf, rotation) can be simulated using $\mathcal{O}(1)$ link and cut operations. We obtain the following observation.

▶ **Observation 23.** *A collection of RB trees on n nodes can be simulated using link-cut trees. The amortized cost of every operation on an RB tree is then $\mathcal{O}(\log^2 n)$.*

The semi-dynamic LPF-tree is represented using link-cut trees. A straightforward implementation would be sufficient to cover first-letter deletions (Lemma 21). However, when a new letter is appended to the string, the total number of single-edge updates could be $\Theta(n)$. To guarantee efficiency, edges need to be redirected in batches (Lemma 22). To this end, we use joinable balanced BSTs, such as RB trees, to represent all edges leading to a single node.

More formally, the whole LPF-tree is stored in one link-cut tree. There is a 1-to-1 correspondence between nodes of the LPF-tree and nodes of the link-cut tree. Assume node v of the LPF-tree is not a leaf and that the sources of all edges with target v are $u_1, u_2, \dots, u_p$, with $u_1 < u_2 < \dots < u_p$. Then the link-cut tree arranges the nodes $u_1, u_2, \dots, u_p$ into an RB-tree and the root of this tree is joined with an edge with v.

In Lemma 21, we need to make $\mathcal{O}(\sqrt{n})$ single edge updates. Each of them requires the move of a node from one RB tree to another RB tree in our link-cut tree, which gives $\mathcal{O}(\sqrt{n}\log^2 n)$ time by Observation 23. In Lemma 22, in addition to operations on $\mathcal{O}(1)$ nodes and edges on the link-cut tree, we need to move a batch of consecutive nodes from an RB tree to a new RB tree leading to the new root. By Observation 23, the operations on the link-cut tree in this lemma require only $\mathcal{O}(\log^2 n)$ time. The resulting update time complexity $\mathcal{O}(\sqrt{n}\log^2 n)$ dominates the remaining operations from Lemmas 21 and 22.

We assume that edges inside RB trees have weight 0 and all the remaining edges have weight 1. Then, in the data structure the weights of nodes will be updated so that at each moment, the weight of a node will be equal to the sum of weights of edges on the path to the root. Then, for a node $i \in [a\,..\,a+|U|]$, where $a = \mathsf{deletions}(U)$, the weight will correspond to the length of the path in the LPF-tree from a to $a+|U|$. The weights can be maintained with $\mathcal{O}(1)$ "add" and "query" operations (which gives $\mathcal{O}(\log n)$ amortized time) per link or cut operation to satisfy this definition of weights. Indeed, no changes to weights are required in link or cut operations implementing a rotation on an RB tree; when moving a batch of edges to a different RB tree, we first query for the weights of the roots of the old RB tree and the new RB tree and then add the difference of these weights to the whole moved subtrees; adding a new root requires incrementing the weights of all the existing nodes.

By Lemmas 21 and 22, we obtain the following result, which together with Observation 18 implies our main result, Theorem 1.

▶ **Lemma 24.** *A semi-dynamic LPF-tree of a string of length at most n can be maintained in $\mathcal{O}(\sqrt{n}\log^2 n)$ amortized time per update operation, such that the length of the path from the root to any node can be retrieved in $\mathcal{O}(\log n)$ amortized time.*

6 Additive Sensitivity of LZ77 for Rotations

Note that as each rotation can be emulated with two edit operations, the work of Akagi et al. [2, Section 8] implies that, for any string S, $\frac{1}{6}|\mathsf{LZ}(\mathsf{rot}(S))| \le |\mathsf{LZ}(S)| \le 6|\mathsf{LZ}(\mathsf{rot}(S))|$.

▶ **Fact 25.** *There are infinitely many strings S for which $|\mathsf{LZ}(\mathsf{rot}(S))| \ge |\mathsf{LZ}(S)| + \Theta(\sqrt{|S|})$ and $|\mathsf{LZ}(\mathsf{rot}(S))| \ge \frac{3}{2}|\mathsf{LZ}(S)| - 2$.*

Proof. Let $a_1, \dots, a_m$ be distinct letters and let $S_i = a_1 a_2 \cdots a_i$. Now, consider the string $S = S_m S_1 S_2 \cdots S_m$, which is of length $\Theta(m^2)$.

We have that $|\mathsf{LZ}(S)| = 2m$ since the phrases of $\mathsf{LZ}(S)$ are:

$$(a_1, a_2, \dots, a_m, S_1, S_2, \dots, S_m).$$

Let $S' = \mathsf{rot}(S) = a_2 a_3 \cdots a_m S_1 S_2 \cdots S_m a_1$. Then, we have $|\mathsf{LZ}(S')| = 3m-2$ since the phrases of $\mathsf{LZ}(S')$ are:

$$(a_2, a_3, \dots, a_m, a_1,\ S_1, a_2,\ S_2, a_3,\ S_3, a_4,\ \dots, S_{m-2}, a_{m-1},\ S_{m-1}, a_m a_1). \qquad ◀$$

The next fact provides a corresponding upper bound on $|\mathsf{LZ}(\mathsf{rot}(S))|$. Further, let us note that the rot operation can decrease the number of LZ phrases by one; for example, $|\mathsf{LZ}(abaa)| = 4$ and $|\mathsf{LZ}(baaa)| = 3$. Fact 26 also shows that a larger decrease is not possible.

▶ **Fact 26.** *For every string S, we have $|\mathsf{LZ}(S)| - 1 \le |\mathsf{LZ}(\mathsf{rot}(S))| \le |\mathsf{LZ}(S)| + \Theta(\sqrt{|S|})$ and $|\mathsf{LZ}(\mathsf{rot}(S))| \le 2|\mathsf{LZ}(S)|$.*

Proof. The second inequality follows by Lemma 13. Indeed, let us consider the LZ parsing of S into phrases $F_1, F_2, \dots, F_k$. We have $|F_1| = 1$. We can transform it into a parsing of $\mathsf{rot}(S)$ as follows: move F_1 to the end and for each phrase F_i whose previous occurrence was only at the leftmost position of S, partition F_i into a one-letter phrase and the remaining phrase. By Lemma 13, $\mathcal{O}(\sqrt{|S|})$ phrases will be partitioned. The resulting parsing cannot have fewer phrases than the LZ parsing of $\mathsf{rot}(S)$ by the fact that greedy is optimal in this case. The parsing has size $|\mathsf{LZ}(S)| + \Theta(\sqrt{|S|})$ and, simultaneously, size at most $2|\mathsf{LZ}(S)|$, as required.

Let us prove the first inequality. Let S' be S without its first letter. If suffices to show that $|\mathsf{LZ}(S')| \ge |\mathsf{LZ}(S)| - 1$, as appending letters can only increase the number of phrases.

For a position a in S, by $\mathit{next}_S(a)$ we denote the smallest ending position b of a phrase in $\mathsf{LZ}(S)$ such that $b \geq a$. We show by induction that if i is the ending position of a phrase in $\mathsf{LZ}(S')$, then $|\mathsf{LZ}(S'[0\,..\,i])| \geq |\mathsf{LZ}(S[0\,..\,j])| - 1$ for $j = \mathit{next}_S(i+1)$.

The base case for $i = 0$ holds with equality for $j = \mathit{next}_S(1)$. Assume that $|\mathsf{LZ}(S'[0\,..\,i])| \geq |\mathsf{LZ}(S[0\,..\,j])| - 1$ holds for i being an ending position of a phrase in S' and $j = \mathit{next}_S(i+1)$. Let $i' > i$ be the next ending position of a phrase in S' and $j' = \mathit{next}_S(i'+1)$. If $j' = j$, then the desired inequality holds as $|\mathsf{LZ}(S'[0\,..\,i'])| = |\mathsf{LZ}(S'[0\,..\,i])| + 1$ and $|\mathsf{LZ}(S[0\,..\,j'])| = \mathsf{LZ}(S[0\,..\,j])|$. Otherwise, we have that $j' = j''$, where $j'' = \mathit{next}_S(j+1)$: Indeed, if $j' > j''$, then $S'[j\,..\,j'-1]$ would have an earlier occurrence in S', so $S[j+1\,..\,j'] = S'[j\,..\,j'-1]$ would have an earlier occurrence in S, which contradicts the greediness of the algorithm computing the parsing. By the inductive assumption, this concludes that

$$|\mathsf{LZ}(S'[0\,..\,i'])| = |\mathsf{LZ}(S'[0\,..\,i])|+1 \geq |\mathsf{LZ}(S[0\,..\,j])| = |\mathsf{LZ}(S[0\,..\,j''])|-1 = |\mathsf{LZ}(S[0\,..\,j'])|-1,$$

as required. ◀

7 Conclusions

We have shown that the size of the Lempel-Ziv-Storer-Szymanski factorization (LZSS) with self-references of a length-n semi-dynamic string S can be updated in $\tilde{\mathcal{O}}(\sqrt{n})$ time. The same approach with minor adaptations can store the size of the classic LZ77 parsing (with self-references), in which the phrase of S starting at position i is $S[i\,..\,\min(i+\mathsf{LPF}_S[i], |S|-1)]$ (i.e., it includes the position that immediately follows the longest previous factor), also with $\tilde{\mathcal{O}}(\sqrt{n})$ update time. Future work includes storing the size of other types of compression in the semi-dynamic setting. The main open problems are, however, if the update time can be decreased and if strictly sublinear update time is possible in the fully dynamic setting.

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. doi:10.1016/S1570-8667(03)00065-0.

2 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Information and Computation*, 291:104999, 2023. doi:10.1016/J.IC.2022.104999.

3 Tooru Akagi, Yuki Kuhara, Takuya Mieno, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Combinatorics of minimal absent words for a sliding window. *Theoretical Computer Science*, 927:109–119, 2022. doi:10.1016/J.TCS.2022.06.002.

4 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 Computing Surveys*, 45(1):5:1–5:17, 2012. doi:10.1145/2379776.2379781.

5 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.

6 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020. doi:10.1016/J.TCS.2019.08.005.

7 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, pages 8:1–8:15, 2021. doi:10.4230/LIPICS.CPM.2021.8.

8 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2053–2071, 2016. doi:10.1137/1.9781611974331.CH143.

9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

10 Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. Lempel-Ziv compression in a sliding window. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 15:1–15:11, 2017. doi:10.4230/LIPICS.CPM.2017.15.

11 Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995. doi:10.1007/BF01294132.

12 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. doi:10.3390/A11080118.

13 Graham Cormode and S. Muthukrishnan. Substring compression problems. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 321–330. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070478.

14 Maxime Crochemore. Linear searching for a square in a word. *Bulletin-European Association for Theoretical Computer Science*, 24(1):66–72, 1984.

15 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Information and Computation*, 270, 2020. doi:10.1016/J.IC.2019.104461.

16 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008. doi:10.1016/J.IPL.2007.10.006.

17 Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. Computing the longest previous factor. *European Journal of Combinatorics*, 34(1):15–26, 2013. doi:10.1016/J.EJC.2012.07.011.

18 Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *2008 Data Compression Conference (DCC 2008)*, pages 482–488, 2008. doi:10.1109/DCC.2008.36.

19 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, M. Sohel Rahman, German Tischler, and Tomasz Waleń. Improved algorithms for the range next value problem and applications. *Theoretical Computer Science*, 434:23–34, 2012. doi:10.1016/J.TCS.2012.02.015.

20 Jonas Ellert. Sublinear time Lempel-Ziv (LZ77) factorization. In *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023*, pages 171–187, 2023. doi:10.1007/978-3-031-43980-3_14.

21 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.

22 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

23 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching, CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

24 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015*, pages 172–184, 2015. doi:10.1007/978-3-319-19929-0_15.

25 Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018. doi:10.1007/S00453-017-0333-1.

26 Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In *28th Annual European Symposium on Algorithms, ESA 2020*, pages 54:1–54:18, 2020. doi:10.4230/LIPICS.ESA.2020.54.

27 Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *2013 Data Compression Conference, DCC 2013*, pages 133–142, 2013. doi:10.1109/DCC.2013.21.

28 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Data Compression Conference, DCC 2014*, pages 163–172, 2014. doi:10.1109/DCC.2014.62.

29 Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *Journal of Discrete Algorithms*, 2(1):33–51, 2004. doi:10.1016/S1570-8667(03)00064-9.

30 Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In *15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, pages 563–574, 2005. doi:10.1007/11537311_49.

31 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms, 12th International Symposium, SEA 2013*, volume 7933, pages 139–150, 2013. doi:10.1007/978-3-642-38527-8_14.

32 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013*, pages 189–200, 2013. doi:10.1007/978-3-642-38905-4_19.

33 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Data Compression Conference, DCC 2014*, pages 153–162, 2014. doi:10.1109/DCC.2014.78.

34 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

35 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

36 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1344–1357, 2019. doi:10.1137/1.9781611975482.82.

37 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *STOC 2022: 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1657–1670, 2022. doi:10.1145/3519935.3520061.

38 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013*, pages 103–112, 2013. doi:10.1137/1.9781611972931.9.

39 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.

40 Philip Klein and Shay Mozes. Optimization algorithms for planar graphs, 2023. URL: https://planarity.org/.

41 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

42 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551, 2015. doi:10.1137/1.9781611973730.36.

43 Dominik Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2):44, 2021. doi:10.3390/A14020044.

44 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *2016 Data Compression Conference, DCC 2016*, pages 3–12, 2016. doi:10.1109/DCC.2016.38.

45 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

46 N. Jesper Larsson. Extended application of suffix trees to data compression. In *Proceedings of the 6th Data Compression Conference (DCC 1996)*, pages 190–199, 1996. doi:10.1109/DCC.1996.488324.

47 Takuya Mieno, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing minimal unique substrings for a sliding window. *Algorithmica*, 84(3):670–693, 2022. doi:10.1007/S00453-021-00864-1.

48 Takuya Mieno and Mitsuru Funakoshi. Shortest unique palindromic substring queries in semi-dynamic settings. In *Combinatorial Algorithms - 33rd International Workshop, IWOCA 2022*, pages 425–438, 2022. doi:10.1007/978-3-031-06678-8_31.

49 Takuya Mieno, Kiichi Watanabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Palindromic trees for a sliding window and its applications. *Information Processing Letters*, 173:106174, 2022. doi:10.1016/J.IPL.2021.106174.

50 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops*, pages 271–282, 2012. doi:10.1007/978-3-642-31155-0_24.

51 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, pages 72:1–72:15, 2016. doi:10.4230/LIPICS.MFCS.2016.72.

52 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. *CoRR*, abs/1605.01488, 2016. arXiv:1605.01488.

53 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete and Applied Mathematics*, 274:116–129, 2020. doi:10.1016/J.DAM.2019.01.014.

54 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011*, pages 15–26, 2011. doi:10.1007/978-3-642-21458-5_4.

55 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Algorithms - ESA 2008, 16th Annual European Symposium*, pages 696–707, 2008. doi:10.1007/978-3-540-87744-8_58.

56 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In *Automata, Languages and Programming, 25th International Colloquium, ICALP 1998*, pages 731–742, 1998. doi:10.1007/BFB0055097.

57 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.

58 Nicola Prezza and Giovanna Rosone. Faster online computation of the succinct longest previous factor array. In *Beyond the Horizon of Computability - 16th Conference on Computability in Europe, CiE 2020*, pages 339–352, 2020. doi:10.1007/978-3-030-51466-2_31.

59 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *International Colloquium on Automata, Languages and Programming, ICALP 2008*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

60 Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008*, pages 109–120, 2008. doi:10.1007/978-3-540-89097-3_12.

61 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

62 Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012*, pages 789–799, 2012. doi:10.1007/978-3-642-32589-2_68.

63 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

64 Robert Endre Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, 1983. doi:10.1137/1.9781611970265.

65 Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, 2007. `doi:10.1145/1314690.1314692`.

66 Alexandre Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2007. `arXiv:0707.3619`.

67 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. `doi:10.1007/BF01206331`.

68 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. `doi:10.1016/0020-0190(83)90075-3`.

69 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In *31st International Symposium on Theoretical Aspects of Computer Science, STACS 2014*, pages 675–686, 2014. `doi:10.4230/LIPICS.STACS.2014.675`.

70 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. `doi:10.1109/TIT.1977.1055714`.

Internal Pattern Matching in Small Space and Applications

Gabriel Bathie
DIENS, École normale supérieure de Paris, PSL Research University, France
LaBRI, Université de Bordeaux, France

Panagiotis Charalampopoulos
Birkbeck, University of London, UK

Tatiana Starikovskaya
DIENS, École normale supérieure de Paris, PSL Research University, France

Abstract

In this work, we consider pattern matching variants in small space, that is, in the read-only setting, where we want to bound the space usage on top of storing the strings. Our main contribution is a space-time trade-off for the Internal Pattern Matching (IPM) problem, where the goal is to construct a data structure over a string S of length n that allows one to answer the following type of queries: Compute the occurrences of a fragment P of S inside another fragment T of S, provided that $|T| < 2|P|$. For any $\tau \in [1\mathinner{.\,.} n/\log^2 n]$, we present a nearly-optimal $\tilde{O}(n/\tau)$-size[1] data structure that can be built in $\tilde{O}(n)$ time using $\tilde{O}(n/\tau)$ extra space, and answers IPM queries in $O(\tau + \log n \log^3 \log n)$ time. IPM queries have been identified as a crucial primitive operation for the analysis of algorithms on strings. In particular, the complexities of several recent algorithms for approximate pattern matching are expressed with regards to the number of calls to a small set of primitive operations that include IPM queries; our data structure allows us to port these results to the small-space setting. We further showcase the applicability of our IPM data structure by using it to obtain space-time trade-offs for the longest common substring and circular pattern matching problems in the *asymmetric streaming* setting.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases internal pattern matching, longest common substring, small-space algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.4

Funding *Gabriel Bathie*: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR) and a Royal Society International Exchanges Award.
Panagiotis Charalampopoulos: Research visits during which some of the presented ideas were conceived were funded by a Royal Society International Exchanges Award.
Tatiana Starikovskaya: Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR) and a Royal Society International Exchanges Award.

Acknowledgements We would like to dedicate this work to our dear friend and colleague Paweł Gawrychowski on the occasion of his 40th birthday. We thank Solon Pissis for helpful suggestions.

1 Introduction

In the fundamental text indexing problem, the task is to preprocess a text T into a data structure (index) that can answer the following queries efficiently: Given a pattern P, find the occurrences of P in T. The Internal Pattern Matching problem (IPM) is a variant of the text indexing problem, where both the pattern P and the text T are fragments of a longer string S, given in advance.

[1] Throughout this work, the $\tilde{O}(\cdot)$ notation suppresses factors polylogarithmic in the input-size.

Introduced in 2009 [47], IPM queries are a cornerstone of the family of internal queries on strings. The list of internal queries, primarily executed through IPM queries, comprises of period queries, prefix-suffix queries, periodic extension queries, and cyclic equivalence queries; see [52, 53, 50]. Other problems that have been studied in the internal setting include shortest unique substring [1], longest common substring [5], suffix rank and selection [9, 50], BWT substring compression [9], shortest absent string [10], dictionary matching [32, 21, 20], string covers [31], masked prefix sums [34], circular pattern matching [44], and longest palindrome [61].

The primary distinction between the classical and internal string queries lies in how the pattern is handled during queries. In classical queries, the input is explicitly provided at query time, whereas in internal queries, the input is specified in constant space via the endpoints of fragments of string S. This distinction enables notably faster query times in the latter setting, as there is no need to read the input when processing the query. This characteristic of IPM and similar internal string queries renders them particularly valuable for bulk processing of textual data. This is especially advantageous when S serves as input for another algorithm, as illustrated by multiple direct and indirect (via other internal queries) applications of IPM: pattern matching with variables [56, 36], detection of gapped repeats and subrepetitions [55, 41], approximate period recovery [2, 4], computing the longest unbordered substring [51], dynamic repetition detection [3], computing string covers [31], identifying two-dimensional maximal repetitions, enumeration of distinct substrings [25], dynamic longest common substring [5], approximate pattern matching [26, 27], approximate circular pattern matching [23, 24], (approximate) pattern matching with wildcards [11], RNA folding [33], and the language edit distance problem for palindromes and squares [12].

Below we assume $|T| < 2|P|$, which guarantees that the set of occurrences of P in T forms an arithmetic progression and can be thus represented in $O(1)$ space.

With no preprocessing ($O(1)$ extra space), IPM queries on a string S of length n can be answered in $O(n)$ time by a constant-space pattern matching algorithm (see [17] and references therein). On the other side of the spectrum, Kociumaka, Radoszewski, Rytter, and Waleń [52] showed that for every string $S \in [0\,..\,\sigma]^n$, there exists a data structure of size $O(n/\log_\sigma n)$ which answers IPM queries in optimal $O(1)$ time and can be constructed in $O(n/\log_\sigma n)$ time given the packed representation of S (meaning that S divided into blocks of $\log_\sigma n$ consecutive letters, and every block is stored in one machine word). The problem has been equally studied in the compressed and dynamic settings [26, 49, 48].

1.1 Our Main Contribution: Small-space IPM

As our main contribution, we provide a trade-off between the constant-space and $O(n)$ query time and the $O(n/\log_\sigma n)$-space and constant query time data structures. We consider the IPM problem in the read-only setting, where one assumes random read-only access to the input string(s) and only accounts for the extra space, that is, the space used by the algorithm/data structure on top of the space needed to store the input.

▶ **Corollary 1.1.** *Suppose that we have read-only random access to a n-length string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time using $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer the following internal pattern matching queries in time $O(\tau + \log n \log^3 \log n)$: given $p, p', t, t' \in [1\,..\,n]$ such that $t' - t \le 2(p' - p)$, return all occurrences of $P = S[p\,..\,p']$ in $T = S[t\,..\,t']$.*

Our data structure is nearly optimal: First, when n/τ is polynomial, the construction time is linear; and secondly, the product of the query time and space of our data structure is optimal up to polylogarithmic factors (**Lemma 3.8**).

Technical overview for IPM queries. Our solution relies heavily on utilizing the concept of τ-partitioning sets, as introduced by Kosolobov and Sivukhin [57]. For a string of length n, a τ-partitioning is a subset of $O(n/\tau)$ positions that satisfies some density and consistency criteria. We use the positions of such a set as anchor points for identifying pattern occurrences, provided that the pattern avoids a specific periodic structure. To detect these anchored occurrences, we employ sparse suffix trees alongside a three-dimensional range searching structure. In cases where the pattern does not avoid said periodic structure, we employ a different strategy, leveraging the periodic structure to construct the necessary anchor points.

We next provide a brief comparison of the outlined approach with previous work. String anchoring techniques have been proven very useful in and been developed for text indexing problems, such as the longest common extension (LCE) problem, in small space [57, 16]. One of the most technically similar works to ours is that of Ben-Nun et al. [14] who considered the problem of computing a long common substring of two input strings in small space. They use an earlier variant of τ-partitioning sets, due to Birenzwige et al. [16], that has slightly worse guarantees than that of Kosolobov and Shivukhin [57]. The construction of anchors for substrings with periodic structure is quite similar to that of Ben-Nun et al. [14]. After computing a set of anchors, they aim to identify a synchronised pair of anchors that yields a long common substring; they achieve this via mergeable AVL trees. As IPM queries need to be answered in an online manner, we instead construct an appropriate orthogonal range searching data structure over a set of points that correspond to anchors. Using orthogonal range searching is a by-now classical approach for text indexing, see [58] for a survey.

1.2 Applications

Several internal queries reduce to IPM queries, and hence we obtain efficient implementations of them in the small-space setting. Additionally, we port several efficient approximate pattern matching algorithms to the small-space setting since IPM was the only primitive operation that they rely on that did not have an efficient small-space implementation to this day. See Section 4 for details on these applications.

Longest Common Substring (LCS). The LCS problem is formally defined as follows.

Longest Common Substring (LCS)
Input: Strings S and T of length at most n.
Output: The length of a longest string that appears as a (contiguous) fragment in both S and T.

The length of the longest common substring is one of the most popular string-similarity measures. The by-now classical approach to the LCS problem is to construct the suffix tree of S and T in $O(n)$ time and space. The longest common substring of the two strings appears as a common prefix of a pair of suffixes of S and T and hence its length is the maximal string-depth of a node of the suffix tree with leaf-descendants corresponding to suffixes of both strings; this node can be found in $O(n)$ time in a bottom-up manner.

Starikovskaya and Vildhøj [64] were the first to consider the problem in the read-only setting. They showed that for any $n^{2/3} < \tau \leq n$, the problem can be solved in $O(\tau)$ extra space and $O(n^2/\tau)$ time. Kociumaka et al. [54] extended their bound to all $1 \leq \tau \leq n$, which in particular resulted in a constant-space read-only algorithm running in time $\tilde{O}(n^2)$.

In an attempt to develop even more space-efficient algorithms for the LCS problem, it might be tempting to consider the streaming setting, which is particularly restrictive: in this setting, one assumes that the input arrives letter-by-letter, as a stream, and must account for all the space used. Unfortunately, this setting does not allow for better space complexity: any streaming algorithm for LCS, even randomised, requires $\Omega(n)$ bits of space (**Theorem 5.2**). In the asymmetric streaming setting, which is slightly less restrictive and was introduced by Andoni et al. [7] and Saks and Seshadhri [63], the algorithm has random access to one string and sequential access to the other. Mai et al. [60] showed that in this setting, LCS can be solved in $\tilde{O}(n^2)$ time and $O(1)$ space. By utilising (a slightly more general variant of) IPM queries, we extend their result and show that for every $\tau \in [\sqrt{n}\log n(\log\log n)^3 \mathinner{.\,.} n]$, there is an asymmetric streaming algorithm that solves the LCS problem in $O(\tau)$ space and $\tilde{O}(n^2/\tau)$ time (**Theorem 6.1**). Note that these bounds almost match the bounds of Kociumaka et al. [54], while the setting is stronger.

Circular Pattern Matching (CPM). The CPM problem is formally defined as follows.

CIRCULAR PATTERN MATCHING (CPM)
Input: A pattern P of length m, a text T of length n.
Output: All occurrences of rotations of P in T.

The interest in occurrences of rotations of a given pattern is motivated by applications in Bioinformatics and Image Processing: in Bioinformatics, the starting position of a biological sequence can vary significantly due to the arbitrary nature of sequencing in circular molecular structures or inconsistencies arising from different standards of linearization applied to sequence databases; and in Image Processing, the contour of a shape can be represented using a directional chain code, which can be viewed as a circular sequence, particularly when the orientation of the image is irrelevant [8].

For strings over an alphabet of size σ, the classical read-only solution for CPM via the suffix automaton of $P \cdot P$ runs in $O(n \log \sigma)$ time and uses $O(m)$ extra space [59]. Recently, Charalampopoulos et al. showed a simple $O(n)$ time and $O(m)$ extra space solution. The problem has been also studied from the practical point of view [65, 40, 29] and in the text indexing setting [45, 43, 42].

It is not hard to see that the CPM and the LCS problems are closely related: occurrences of rotations of P in T are exactly the common substrings of $P \cdot P$ and T of length m. Implicitly using this observation, we show that in the streaming setting, the CPM problem requires $\Omega(m)$ bits of space (**Theorem 5.3**) and that in the asymmetric streaming setting, for every $\tau \in [\sqrt{m}\log m(\log\log m)^3 \mathinner{.\,.} m]$, there exists an algorithm that solves the CPM problem in time $\tilde{O}(mn/\tau)$ using $O(\tau)$ extra space (**Corollary 6.5**). Finally, in the read-only setting, we give an *online* $O(n)$-time, $O(1)$-space algorithm (**Theorem 7.1**).

2 Preliminaries

For integers $i, j \in \mathbb{Z}$, denote $[i \mathinner{.\,.} j] = \{k \in \mathbb{Z} : i \le k \le j\}$, $[i \mathinner{.\,.} j) = \{k \in \mathbb{Z} : i \le k < j\}$. We consider an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ of size polynomially bounded in the length of the input string(s). The elements of the alphabet are called letters, and a string is a finite sequence of letters. For a string T and an index $i \in [1 \mathinner{.\,.} n]$, the i-th letter of T is denoted by $T[i]$. We use $|T| = n$ to denote the length of T. For two strings S, T, we use ST or $S \circ T$ indifferently to denote their concatenation $S[1] \cdots S[|S|]T[1] \cdots T[|T|]$. For integers i, j, $T[i \mathinner{.\,.} j]$ denotes the *fragment* $T[i]T[i+1] \cdots T[j]$ of T if $1 \le i \le j \le n$ and the empty string ε

otherwise. We extend this notation in a natural way to $T[i\,..\,j+1) = T[i\,..\,j] = T(i-1\,..\,j]$. When $i = 1$ or $j = n$, we omit these indices, i.e., $T[\,..\,j] = T[1\,..\,j]$ and $T[i\,..\,] = T[i\,..\,n]$. A string P is a *prefix* of T if there exists $j \in [1\,..\,n]$ such that $P = T[\,..\,j]$, and a *suffix* of T if there exists $i \in [1\,..\,n]$ such that $P = T[i\,..\,]$. We denote the reverse of a string T by $\mathrm{rev}(T) = T[n]T[n-1]\cdots T[2]T[1]$. For an integer $\Delta \in [1\,..\,n]$, we say that a string $T[\Delta+1\,..\,n] \circ T[1\,..\,\Delta]$ is a *rotation* of T. A fragment $T[i\,..\,j]$ of a string T is called an *occurrence* of a string P if $T[i\,..\,j] = P$; in this case, we say that P *occurs* at position i of T. A positive integer ρ is a *period* of a string T if $T[i] = T[i+\rho]$ for all $i \in [1\,..\,|T|-\rho]$. The smallest period of T is referred to as *the period* of T and is denoted by $\mathsf{per}(T)$. If $\mathsf{per}(T) \le |T|/2$, T is called *periodic.*

▶ **Fact 2.1** (Corollary of the Fine–Wilf periodicity lemma [37]). *The starting positions of the occurrences of a pattern P in a text T form $O(|T|/|P|)$ arithmetic progressions with difference $\mathsf{per}(P)$.*

We assume a reader to be familiar with basic data structures for string processing, see, e.g., [59]. Recall that a suffix tree for a string S is essentially a compact trie representing the set of all suffixes of S, whereas a sparse suffix tree contains only a subset of these suffixes.

▶ **Fact 2.2** ([57, Theorem 3]). *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $b = \Omega(\log^2 n)$, one can construct in $O(n \log_b n)$ time and $O(b)$ space the sparse suffix tree for arbitrarily chosen b suffixes.*

▶ **Fact 2.3** ([17]). *There is a read-only online algorithm that finds all occurrences of a pattern P of length m in a text T of length $n \ge m$ in $O(n)$ time and $O(1)$ space.*

▶ **Fact 2.4** ([38, Lemma 6]). *Given read-only random access to a string S of length n, one can decide in $O(n)$ time and $O(1)$ space if S is periodic and, if so, compute $\mathsf{per}(S)$.*

▶ **Fact 2.5** ([35]). *Given read-only random access to a string S of length n, the lexicographically smallest rotation of a string S can be computed in $O(n)$ time and $O(1)$ space.*

Static predecessor. For a static set, a combination of x-fast tries [66] and deterministic dictionaries [62] yields the following efficient deterministic data structure; cf. [39].

▶ **Fact 2.6** ([39, Proposition 2]). *A sorted static set $Y \subseteq [1\,..\,U]$ can be preprocessed in $O(|Y|)$ time and space so that predecessor queries can be performed in $O(\log\log |U|)$ time.*

Weighted ancestor queries. Let $\mathcal{T}$ be a rooted tree with integer weights on nodes. A *weighted ancestor* query for a node u and weight d must return the highest ancestor of u with weight at least d.

▶ **Fact 2.7** ([6]). *Let $\mathcal{T}$ be a rooted tree of size n with integer weights on nodes. Assume that each weight is at most n, with the weight of the root being zero, and the weight of every non-root node being strictly larger than its parent's weight. $\mathcal{T}$ can be preprocessed in $O(n)$ time and space so that weighted ancestor queries on it can be performed in $O(\log\log n)$ time.*

If $\mathcal{T}$ is the suffix tree of a string and the weights are the string-depths of the nodes, this result can be improved further:

▶ **Fact 2.8** ([13]). *The suffix tree $\mathcal{T}$ of a string of length n can be preprocessed in $O(n)$ time and $O(n)$ space so that weighted ancestor queries on it can be performed in $O(1)$ time.*

3D range emptiness. A three-dimensional orthogonal range emptiness query asks whether a range $[a_1 \times a_2] \times [b_1 \times b_2] \times [c_1 \times c_2]$ is empty.

▶ **Fact 2.9** ([46, Theorem 2]). *There exists a data structure that answers three-dimensional orthogonal range emptiness queries on a set of n points from a $[U] \times [U] \times [U]$ grid in $O(\log\log U + (\log\log n)^3)$ time, uses $O(n \log n (\log\log n)^3)$ space, and can be constructed in $O(n \log^4 n \log\log n)$ time. If the query range is not empty, the data structure also outputs a point from it.*

▶ Remark 2.10. Better space vs. query-time tradeoffs than the above are known for the 3D range emptiness problem; cf [19] and references therein. We opted for the data structure encapsulated of Fact 2.9 due to its efficient construction algorithm. Note that a data structure capable of reporting all points in an orthogonal range over a $[U] \times [U] \times [U]$ grid with n points in time $O(Q_1(U, n) + Q_2(U, n) \cdot |\mathsf{output}|)$ can answer range emptiness queries, also returning a witness in the case the range is not empty, in time $O(Q_1(U, n) + Q_2(U, n))$.

3 Internal Pattern Matching

We consider a slightly more powerful variant of IPM queries, as required by our applications. A reader that is only interested in IPM queries can focus on the case when $a = \varepsilon$.

EXTENDED IPM (DECISION)
Input: A string S of length n over an integer alphabet to which we have read-only random access.
Query: Given $p, p', t, t' \in [1\,..\,n]$ and $a \in \Sigma \cup \{\varepsilon\}$, return whether $P := S[p\,..\,p']a$ occurs in $T := S[t\,..\,t']$ and, if so, return a witness occurrence.

Our solution for EXTENDED IPM (DECISION) heavily relies on a solution for the following auxiliary problem.

ANCHORED IPM
Input: A string S of length n over an integer alphabet Σ to which we have read-only random access and a set $\mathcal{A} \subseteq [1\,..\,n]$.
Query: Given $p, x, p', t, t' \in [1\,..\,n]$ with $p \le x \le p'$, $x \in \mathcal{A}$, and $a \in \Sigma \cup \{\varepsilon\}$, for $P := S[p\,..\,p']a$, decide whether there exists an occurrence of P at some position $j \in [t\,..\,t' - |P| + 1]$ such that $j + (x - p) \in \mathcal{A}$ and, if so, return a witness.

▶ **Lemma 3.1.** *There exists a data structure for the ANCHORED IPM problem that can be built using $O(n \log_{|\mathcal{A}|} n) + O(|\mathcal{A}| \log^4 |\mathcal{A}| \log\log |\mathcal{A}|)$ time and $O(|\mathcal{A}| \log |\mathcal{A}| (\log\log |\mathcal{A}|)^3)$ extra space, and answers queries in $O(\log^3 \log n)$ time.*

Proof. For an integer $y \in [1\,..\,n]$, denote $P_y := \mathrm{rev}(S[\,..\,y))$ and $S_y := S[y\,..\,]$. Consider a family $\mathcal{X} := \{(P_y\$, S_y\$) : y \in \mathcal{A}\}$ of pairs of strings, where $\$ \notin \Sigma$ is a letter lexicographically smaller than all others. Using Fact 2.2, we build a sparse suffix tree RSST for the first components of the elements of $\mathcal{X}$ and a sparse suffix tree SST for the second components of the elements of $\mathcal{X}$.

Consider a three-dimensional grid $[1\,..\,n] \times [1\,..\,n] \times [1\,..\,n]$. In this grid, create a set Π of points, which contains, for each element $(P_y\$, S_y\$)$ of $\mathcal{X}$, a point $(\mathsf{rank}_{\mathrm{rev}}(y), \mathsf{rank}(y), y)$, where $\mathsf{rank}_{\mathrm{rev}}(y)$ is the lexicographic rank of $P_y\$$ among the first components of the elements of $\mathcal{X}$ and $\mathsf{rank}(y)$ is the lexicographic rank of $S_y\$$ among the second components of the elements of $\mathcal{X}$.

Upon a query, we first retrieve the leaves corresponding to $P_x\$$ and $S_x\$$ in RSST and SST, respectively. This can be done in $O(\log\log n)$ time with the aid of Fact 2.6 built over the elements of $\mathcal{A}$, with $x \in \mathcal{A}$ storing pointers to the corresponding leaves as satellite information. Next, we retrieve the (possibly implicit) nodes u and v corresponding to $\mathrm{rev}(S[p\mathinner{.\,.} x))$ in RSST and $S[x\mathinner{.\,.} p']a$ in SST, respectively. This can be done in $O(\log\log n)$ time after an $O(|\mathcal{A}|)$-time preprocessing of (a) the two trees according to Fact 2.7 and (b) the edge-labels of the outgoing edges of each node using Fact 2.6. Now, it suffices to check if there is some integer j such that the leaf corresponding to $P_j\$$ is a descendant of u, the leaf corresponding to $S_j\$$ is a descendant of v, and $j \in [t+(x-p)\mathinner{.\,.} t'-(p'+|a|-x)]$. After a linear-time bottom-up preprocessing of RSST and SST, we can retrieve in $O(1)$ time the following ranges:

- $R_1 = \{\mathsf{rank}_{\mathrm{rev}}(y) :$ the node of RSST corresponding to $P_y\$$ is a descendant of $u\}$;
- $R_2 = \{\mathsf{rank}(y) :$ the node of SST corresponding to $S_y\$$ is a descendant of $v\}$.

The query then reduces to deciding whether the orthogonal range $R_1 \times R_2 \times [t+(x-p)\mathinner{.\,.} t'-(p'+|a|-x)]$ contains any point in Π, and returning a witness if it does. We can do this efficiently by building the data structure encapsulated in Fact 2.9 for Π: the query time is $O(\log^3\log n)$, while the construction time is $O(n\log_{|\mathcal{A}|} n) + O(|\mathcal{A}|\log^4|\mathcal{A}|\log\log|\mathcal{A}|)$ and the space usage is $O(|\mathcal{A}|\log|\mathcal{A}|(\log\log|\mathcal{A}|)^3)$. ◀

For an integer parameter τ, we next present a data structure for Extended IPM (Decision) that uses $\tilde{O}(n/\tau)$ space on top of the space required to store S and answers queries in nearly-constant time provided that P is of length greater than 5τ. We achieve this result using the so-called τ-partitioning sets of Kosolobov and Sivukhin [57] as *anchors* for the occurrences if P avoids a certain periodic structure, and by exploiting said periodic structure to construct anchors in the remaining case.

▶ **Definition 3.2** (τ-partitioning set). *Given an integer $\tau \in [4\mathinner{.\,.} n/2]$, a set of positions $\mathcal{P} \subseteq [1\mathinner{.\,.} n]$ is called a* τ-partitioning set *if it satisfies the following properties:*

(a) *if $S[i-\tau\mathinner{.\,.} i+\tau] = S[j-\tau\mathinner{.\,.} j+\tau]$ for $i,j \in [\tau+1\mathinner{.\,.} n-\tau]$, then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$;*

(b) *if $S[i\mathinner{.\,.} i+\ell] = S[j\mathinner{.\,.} j+\ell]$, for $i,j \in \mathcal{P}$ and some $\ell \geq 0$, then, for each $d \in [0\mathinner{.\,.}\ell-\tau)$, $i+d \in \mathcal{P}$ if and only if $j+d \in \mathcal{P}$;*

(c) *if $i,j \in [1\mathinner{.\,.} n]$ with $j-i > \tau$ and $(i\mathinner{.\,.} j) \cap \mathcal{P} = \emptyset$, then $S[i\mathinner{.\,.} j]$ has period at most $\tau/4$.*

▶ **Theorem 3.3** ([57]). *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4\mathinner{.\,.} O(n/\log^2 n)]$ and $b = n/\tau$, one can construct in $O(n\log_b n)$ time and $O(b)$ extra space a τ-partitioning set $\mathcal{P}$ of size $O(b)$. The set $\mathcal{P}$ additionally satisfies the property that if a fragment $S[i\mathinner{.\,.} j]$ has period at most $\tau/4$, then $\mathcal{P} \cap [i+\tau\mathinner{.\,.} j-\tau] = \emptyset$.*

▶ **Definition 3.4** (τ-runs). *A fragment F of a string S is a τ-run if and only if $|F| > 3\tau$, $\mathsf{per}(F) \leq \tau/4$, and F cannot be extended in either direction without its period changing. The Lyndon root of a τ-run R is the lexicographically smallest rotation of $R[1\mathinner{.\,.}\mathsf{per}(R)]$.*

The following fact follows from the proof of Lemma 10 in the full version of [22], where the definition of τ-runs is slightly different, but captures all of our τ-runs.

▶ **Fact 3.5** (cf. [22, proof of Lemma 10]). *Two τ-runs can overlap by at most $\tau/2$ positions. The number of τ-runs in a string of length n is $O(n/\tau)$.*

▶ **Lemma 3.6.** *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4\mathinner{.\,.} O(n/\log^2 n)]$, all τ-runs in S can be computed and grouped by Lyndon root in $O(n\log_b n)$ time using $O(b)$ extra space, where $b = n/\tau$. Within the same complexities, we can compute, for each τ-run, the first occurrence of its Lyndon root in it.*

Proof. We first compute a τ-partitioning set $\mathcal{P}$ for S using Theorem 3.3. Due to Property c, its converse that is stated in Theorem 3.3, and Fact 3.5 there is a natural injection from the τ-runs to the maximal fragments of length at least τ that do not contain any position in $\mathcal{P}$ – the τ-run corresponding to such a maximal fragment may extend for τ more positions in each direction. We can find the period of each maximal fragment in time proportional to its length using $O(1)$ extra space due to Fact 2.4. We then try to extend the maximal fragment to a τ-run using $O(\tau)$ letter comparisons. Additionally, we compute the Lyndon root of each computed τ-run R in $O(\tau) = O(|R|)$ time by applying Fact 2.5 to $R[1\mathinner{.\,.}\mathsf{per}(R)]$. The first occurrence of the Lyndon root in the τ-run can be computed in constant time since we know which rotation of $R[1\mathinner{.\,.}\mathsf{per}(R)]$ equals the Lyndon root. Over all τ-runs, the total time is $O(n)$ due to Fact 3.5. ◀

We next prove the main result of this section.

▶ **Theorem 3.7.** *For any $\ell \in [20\mathinner{.\,.}O(n/\log^2 n)]$, there is a data structure for* Extended IPM *(*Decision*) that can be built using $O(n \log_{n/\ell} n) + O((n/\ell)\cdot \log^4 n \log\log n)$ time and $O((n/\ell)\cdot \log n (\log\log n)^3)$ extra space given random access to S and answers queries in $O(\log^3 \log n)$ time, provided that $|P| > \ell$.*

Proof. Let $\tau = \lfloor \ell/5 \rfloor$. We use Theorem 3.3 and Lemma 3.6 with parameter τ to compute a partitioning set $\mathcal{P}$ of size $O(n/\tau)$ and all τ-runs in S, grouped by Lyndon root, each one together with the first occurrence of its Lyndon root. We create a static predecessor structure $\mathcal{R}$ using Fact 2.6, where we insert the starting position of each run R with the following satellite information: R's ending position, the first occurrence of R's Lyndon root in R, and an identifier of its group. We additionally create a data structure $\mathcal{Q}$, where, for each group of τ-runs with a common root L, indexed by their identifiers, we construct, using Fact 2.6, a predecessor data structure for a set $Q_L := \{(y, s, e) : S[s\mathinner{.\,.}e] \text{ is the longest } \tau\text{-run with a suffix } L \circ L[1\mathinner{.\,.}y]\}$, with the first components being the keys and the remaining components being stored as satellite information. The sets Q_L can be straightforwardly constructed in $O(n \log n/\tau)$ time.

Now, let $\mathcal{L}$ be a set that contains the ending position of each τ-run as well as the starting (resp. ending) positions of the first (resp. last) two occurrences of the Lyndon root in this τ-run; $\mathcal{L}$ can be straightforwardly constructed in $O(n/\tau)$ time given the information returned by the application of Lemma 3.6. We then construct a set $\mathcal{A} := \mathcal{P} \cup \mathcal{L}$ and preprocess the string S and the set $\mathcal{A}$ according to Lemma 3.1.

Our query comprises of two steps.

Step 1. First, we deal with the case when both P and T have period at most $\tau/4$. Since P and T are of length at least 5τ, due to Fact 3.5, each of them can be only contained in the τ-run whose starting position is closest to it in the left. We can thus check whether they both have period at most $\tau/4$ in $O(\log\log n)$ time by performing two predecessor queries on $\mathcal{R}$. If this turns out to be the case, we then check whether the two corresponding τ-runs belong to the same group. If they do not, then P does not occur in T due to Fact 3.5. Otherwise, let the common Lyndon root of the two runs be L. We can compute in constant time non-negative integers $x_P, x_T, y_P, y_T < |L|$ and e_P, e_T such that $P = L(|L| - x_P\mathinner{.\,.}] \circ L^{e_P} \circ L[\mathinner{.\,.}y_P]$ and $T = L(|L| - x_T\mathinner{.\,.}] \circ L^{e_T} \circ L[\mathinner{.\,.}y_T]$. Note that P occurs in T if and only if at least one of the following conditions is met: (1) $e_P = e_T$, $x_P \le x_T$, and $y_P \le y_T$; or (2) $e_P = e_T - 1$ and $x_P \le x_T$; or (3) $e_P = e_T - 1$ and $y_P \le y_T$; or (4) $e_P \le e_T - 2$. In each of the four cases, we can compute an occurrence of P in T in constant time.

Step 2. In the second step of the query, we consider the case when $\mathsf{per}(T) > \tau/4$ and distinguish between two cases depending on whether $\mathsf{per}(S[p\mathinner{.\,.}p + 3\tau]) \le \tau/4$. In each case, it suffices to perform at most two anchored internal pattern matching queries.

Case I: $\mathsf{per}(S[p\mathinner{.\,.}p+3\tau]) > \tau/4$. Due to Property c, $[p\mathinner{.\,.}p+3\tau] \cap \mathcal{P} \neq \emptyset$. Let $x = \min([p\mathinner{.\,.}p+3\tau] \cap \mathcal{P})$. Additionally, due to Property b, for any occurrence of P in S at position j, we have $[p\mathinner{.\,.}p+3\tau] \cap \mathcal{P} = (p-j) + ([j\mathinner{.\,.}j+3\tau] \cap \mathcal{P})$, and hence $j + (x-p) \in \mathcal{P}$. Thus, an anchored IPM query returns the desired answer in $O(\log^3 \log n)$ time.

Case II: $\mathsf{per}(S[p\mathinner{.\,.}p+3\tau]) \leq \tau/4$. We distinguish between two subcases depending on whether $\mathsf{per}(P) > \tau/4$; we can check this in $O(\log\log n)$ time with the aid of data structure $\mathcal{R}$ by comparing p' with the ending position of the τ-run that contains $S[p\mathinner{.\,.}p+3\tau]$ and checking if $a = P[|P| - \mathsf{per}(S[p\mathinner{.\,.}p+3\tau])]$ if $a \neq \varepsilon$.

Subcase (a): $\mathsf{per}(P) > \tau/4$. In this case, for any occurrence of P in T, the ending position of the τ-run that is a prefix of P must be aligned with the ending position of a τ-run in T, which belongs to $\mathcal{L} \subseteq \mathcal{A}$.
Recall that $P = S[p\mathinner{.\,.}p']a$. If the period of $S[p\mathinner{.\,.}p']$ is greater than $\tau/4$, we retrieve the ending position of the τ-run containing $S[p\mathinner{.\,.}p+3\tau]$, which is in $\mathcal{L} \subseteq \mathcal{A}$ as well and issue an anchored internal pattern matching query. Assume now that the period of $S[p\mathinner{.\,.}p'+1]$ is at most $\tau/4$ and $\varepsilon \neq a \neq P[|P| - \mathsf{per}(S[p\mathinner{.\,.}p'])]$, in which case p' might not be in $\mathcal{A}$. In this case, we retrieve a fragment $S[q\mathinner{.\,.}q']$ equal to $S[p\mathinner{.\,.}p']$, such that q' is an ending position of a τ-run in $O(\log\log n)$ time using the data structure $\mathcal{Q}$, if such a fragment exists, and use $q \in \mathcal{L} \subseteq \mathcal{A}$ as the anchor to our internal anchor query, effectively searching for $S[q\mathinner{.\,.}q']a = P$. Observe that if such a fragment $S[q\mathinner{.\,.}q']$ does not exist, P cannot have any occurrence in T.

Subcase (b): $\mathsf{per}(P) \leq \tau/4$. We consider an occurrence of P in the τ-run that contains P that starts in its first $\mathsf{per}(P)$ positions and one that ends in its last $\mathsf{per}(P)$ positions. Let these two occurrences be at positions p_1 and p_2, respectively. Each of these occurrences contains at least one element of $\mathcal{L}$; let those elements be denoted q_1 for the occurrence at p_1 and q_2 for the occurrence at p_2.
Note that these elements can be straightforwardly computed given the endpoints of the τ-run, the endpoints of P, and the first occurrence of the Lyndon root in the τ-run, which we already have in hand. We then issue anchored internal pattern matching queries for $(p_1, q_1, p_1 + |P| - 1, t, t')$ and $(p_2, q_2, p_2 + |P| - 1, t, t')$ as both q_1 and q_2 are in $\mathcal{L}$. These queries are answered in $O(\log^3 \log n)$ time. As we show next, if P has an occurrence in T, this occurrence will be returned by those queries.
Consider an occurrence of P in $S[t\mathinner{.\,.}t']$ and denote the τ-run that contains this occurrence by R. Since $\mathsf{per}(T) > \tau/4$, R does not contain $S[t\mathinner{.\,.}t']$. Without loss of generality, let us assume that R does not extend to the left of $S[t\mathinner{.\,.}t']$, the remaining case is symmetric. Let the first occurrence of the Lyndon root L of the τ-run in P be at position $i = q_1 - p_1 + 1$ of P, noting that $i \leq \mathsf{per}(P)$. Then, in the leftmost occurrence of P in R, position i must be aligned with either the first or the second position where L occurs in R. By the construction of the set $\mathcal{L}$, it follows that both of these positions are in $\mathcal{L}$, and hence the anchored internal pattern matching query will return an occurrence. ◀

▶ **Corollary 1.1.** *Suppose that we have read-only random access to a n-length string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time using $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer the following internal pattern matching queries in time $O(\tau + \log n \log^3 \log n)$: given $p, p', t, t' \in [1\mathinner{.\,.}n]$ such that $t' - t \leq 2(p' - p)$, return all occurrences of $P = S[p\mathinner{.\,.}p']$ in $T = S[t\mathinner{.\,.}t']$.*

Proof. If the length of P is at most $\max\{\tau, 20\}$, we compute its occurrences in T, whose length is $O(\tau)$, in $O(\tau)$ time using Fact 2.3. In what follows, we assume that $|P| > \max\{\tau, 20\}$.

We build the Extended IPM (Decision) data structure of Theorem 3.7 for S with $\ell = \max\{\tau, 20\}$. This allows us to efficiently answer the decision version of the desired IPM queries, also returning a witness, in $O(\log^3 \log n)$ time. If the query does not return an occurrence of P in T, we are done. Otherwise, we have to compute all occurrences of P in T represented as an arithmetic progression (cf Fact 2.1). Let the witness returned by the data structure be $S[x \mathinner{.\,.} x']$. Consider the rightmost occurrence of P in $S[t \mathinner{.\,.} x')$, or, if it does not exist, the leftmost occurrence in $S(x \mathinner{.\,.} t']$. Such an occurrence can be found by binary search. If no such occurrence exists, we are again done, as P has a single occurrence in T. Otherwise, the occurrences of P in T form an arithmetic progression with difference equal to the difference d of x and the starting position of the found occurrence due to Fact 2.1. We compute the extreme values of this arithmetic progression using binary search as well: we compute the minimum and the maximum $j \in \mathbb{Z}$ such that $S[x + j \cdot d \mathinner{.\,.} x' + j \cdot d] = P$ and $t \le x + j \cdot d \le x' + j \cdot d \le t'$ using $O(\log n)$ IPM queries in total; the complexity follows. ◀

Lower Bound for an IPM data structure

We now show that the product of the query time and the space achieved in Corollary 1.1 is optimal up to polylogarithmic factors, via a reduction from the following problem.

Longest Common Extension (LCE)
Input: A string S of length n.
Query: Given $i, j \in [1 \mathinner{.\,.} n]$, return the largest ℓ such that $S[i \mathinner{.\,.} i + \ell] = S[j \mathinner{.\,.} j + \ell]$.

Bille et al. [15, Lemma 4] showed that any data structure for LCE for n-length strings that uses s bits of extra space on top of the input has query time $\Omega(n/s)$.

▶ **Lemma 3.8.** *In the non-uniform cell-probe model, any IPM data structure that uses s bits of space on top of the input for a string of length n, has query time $\Omega(n/(s \log n))$.*

Proof. We prove Lemma 3.8 by reducing LCE queries in a string S of length n to IPM queries in S. Consider an IPM data structure with space s and query time q and observe that IPM queries can be used to check substring equality since $S[i \mathinner{.\,.} i'] = S[j \mathinner{.\,.} j']$ if and only if $S[i \mathinner{.\,.} i']$ occurs inside the interval $[j \mathinner{.\,.} j']$ and $j' - j = i' - i$. Using binary search, we can thus answer any LCE query via $O(\log n)$ IPM queries. Hence, we have $q \log n = \Omega(n/s)$, which concludes the proof the lemma. ◀

Lemma 3.8 implies a similar lower bound for the word RAM model, which is weaker than the non-uniform cell-probe model.

4 Other Internal Queries and Approximate Pattern Matching

In the `PILLAR` model of computation [26] the runtimes of algorithms are analysed with respect to the number of calls made to standard word-RAM operations and a few primitive string operations. It has been used to design algorithms for internal queries [52, 53, 50], approximate pattern matching under Hamming distance [26] and edit distance [27], circular approximate pattern matching under Hamming distance [24] and edit distance [28], and (approximate) wildcard pattern matching under Hamming distance [11]. Space-efficient implementations of the `PILLAR` model immediately result in space-efficient implementations of the above algorithms.

In the PILLAR model, one is given a family of strings $\mathcal{X}$ for preprocessing. The elementary objects are fragments $X[i\,..\,j]$ of strings $X \in \mathcal{X}$. Each fragment S is represented via a handle, which is how S is passed as input to PILLAR operations. Initially, the model provides a handle to each $X \in \mathcal{X}$. Handles to other fragments can be obtained through an Extract operation:

- $\mathsf{Extract}(S, \ell, r)$: Given a fragment S and positions $1 \leq \ell \leq r \leq |S|$, extract $S[\ell\,..\,r]$.

Furthermore, given elementary objects S, S_1, S_2 the following primitive operations are supported in the PILLAR model:

- $\mathsf{Access}(S, i)$: Assuming $i \in [1\,..\,|S|]$, retrieve $S[i]$.
- $\mathsf{Length}(S)$: Retrieve the length $|S|$ of S.
- Longest common prefix $\mathsf{LCE}(S_1, S_2)$: Compute the length of the longest common prefix of S_1 and S_2.
- $\mathsf{LCE}^R(S_1, S_2)$: Compute the length of the longest common suffix of S_1 and S_2.
- Internal pattern matching $\mathsf{IPM}(S_1, S_2)$: Assuming that $|S_2| < 2|S_1|$, compute the set of the starting positions of occurrences of S_1 in S_2 represented as one arithmetic progression.

All PILLAR operations other than LCE, LCE^R, and IPM admit trivial constant-time and constant-space implementations in the read-only setting. For any $\tau = O(n/\log^2 n)$, Kosolobov and Sivukhin [57] showed that after $O(n \log_{n/\tau} n)$-time, $O(n/\tau)$-space preprocessing, LCE and LCE^R queries can be supported in $O(\tau)$ time. For IPM queries, we use Corollary 1.1.

In [52, 53, 50] it is (implicitly) shown that the following internal queries can be efficiently implemented in the PILLAR model.

- A *cyclic equivalence query* takes as input two equal-length fragments $U = S[i\,..\,i+\ell]$ and $V = S[j\,..\,j+\ell]$, and returns all rotations of U that are equal to V. Any cyclic equivalence query reduces to $O(1)$ LCE queries and $O(1)$ $\mathsf{IPM}(P,T)$ queries with $|T|/|P| = O(1)$.
- A *period query* takes as input a fragment $U = S[i\,..\,j]$, and returns all periods of U. Such a period query reduces to $O(\log|U|)$ LCE queries and $O(\log|U|)$ $\mathsf{IPM}(P,T)$ queries with $|T|/|P| = O(1)$.
- A *2-period* query takes as input a fragment $U = S[i\,..\,j]$, checks if U is periodic and, if so, it also returns U's period. Such a query reduces to $O(1)$ LCE queries and $O(1)$ $\mathsf{IPM}(P,T)$ queries with $|T|/|P| = O(1)$.

▶ **Corollary 4.1.** *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time and $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space and can answer cyclic equivalence and 2-period queries on S in $O(\tau + \log n \log^3 \log n)$ time, and period queries on S in $O(\tau \log n + \log^2 n \log^3 \log n)$ time.*

By plugging this implementation of the PILLAR model into [26, 27, 24, 11, 28], we obtain the following:

▶ **Corollary 4.2.** *Suppose that we have read-only random access to a text T of length n, a pattern P of length m over an integer alphabet. Given an integer threshold k, for any integer $\tau = O(m/\log^2 m)$, we can compute:*

- *the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n + k^2\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^2)$ extra space;*
- *the approximate occurrences of P in T under the edit distance in $\tilde{O}(n + k^{3.5}\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^{3.5})$ extra space;*
- *the approximate occurrences of all rotations of P in T under the Hamming distance in $\tilde{O}(n + k^3\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^3)$ extra space;*

- *the approximate occurrences of all rotations of P in T under the edit distance in $\tilde{O}(n + k^5\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^5)$ extra space;*
- *in the case where P has D wildcard letters arranged in G maximal intervals, the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n+(D+k)(G+k)\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + (D+k)(G+k))$ extra space.*

To the best of our knowledge, the only work that has considered approximate pattern matching in the read-only model is due to Bathie et al. [12]. They presented online algorithms both for the Hamming distance and the edit distance; for the Hamming distance their algorithm uses $O(k \log m)$ extra space and $O(k \log m)$ time per letter of the text, and for the edit distance $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortised time per letter.

5 LCS and CPM in the Streaming Setting

In the streaming setting, we receive a stream composed of the concatenation of the input strings, e.g., the pattern and the text in the case of CPM. We account for all the space used, including the space needed to store any information about the input strings. We exploit the well-known connection between streaming algorithms and communication complexity to prove linear-space lower bounds for streaming algorithms for LCS and CPM.

Lower Bounds for Streaming Algorithms

Our streaming lower bounds are based on a reduction from the following problem:

AUGMENTED INDEX
Alice holds a binary string S of length n.
Bob holds an index $i \in [1\,..\,n]$ and the string $S[..\,i-1]$.
Output: Bob is to return the value of $S[i]$.

In the one-way communication complexity model, Alice performs an arbitrary computation on her input to create a message $\mathcal{M}$ and sends it to Bob who must compute the output using this message and his input. The communication complexity of a protocol is the size of $\mathcal{M}$ in bits. The protocol is randomised when either Alice or Bob use randomised computation.

▶ **Theorem 5.1** ([18, Theorem 2.3]). *The randomised one-way communication complexity of* AUGMENTED INDEX *is $\Omega(n)$ bits.*

▶ **Theorem 5.2.** *In the streaming setting, any algorithm for LCS for strings of length at most n uses $\Omega(n)$ bits of space.*

Proof. We show the bound by a reduction from the AUGMENTED INDEX problem. Consider an input $S, (i, S[..\,i-1])$ to the AUGMENTED INDEX problem, where $|S| = n$. We observe that for $A = 0^n\$S$ and $B = 0^n\$S[..\,i-1]1$, where $\$ \notin \{0,1\}$, we have $\mathsf{LCS}(A,B) = n+i+1$ if and only if $S[i] = 1$. Now, if we have a streaming algorithm for LCS that uses b bits of space, we can develop a one-way protocol for the AUGMENTED INDEX problem with message size b bits as follows. Alice runs the algorithm on A. When she reaches the end of A, she sends the memory state of the algorithm and n (in binary) to Bob. Bob continues running the algorithm on B, which he can construct knowing n and $S[..\,i-1]$, and returns 1 if and only if $\mathsf{LCS}(A,B) = n+i+1$. Theorem 5.1 implies that $b + \log n = \Omega(n)$, and hence $b = \Omega(n)$. ◀

▶ **Theorem 5.3.** *In the streaming setting, any algorithm for CPM uses $\Omega(m)$ bits of space, where m is the size of the pattern.*

Proof. We show the bound by a reduction from the AUGMENTED INDEX problem. Consider an input $S, (i, S[\,..\, i-1])$ to the AUGMENTED INDEX problem, where $|S| = m$. Let $A = S\$$ and $B = S\$S[\,..\, i-1]1$, where $\$ \notin \{0,1\}$. B ends with an occurrence of a rotation of A if and only if $S[i] = 1$. Now, if we have a streaming algorithm for CPM that uses b bits of space, we can develop a one-way protocol for the AUGMENTED INDEX problem with message size b bits as follows. Alice runs the algorithm on the pattern $A = S\$$ and the first $|S| + 1$ letters of the string B. She then sends the memory state of the algorithm to Bob. Bob continues running the algorithm on the remainder of B, i.e., on $S[1\,..\, i-1]1$, and returns 1 if and only if the algorithm reports an occurrence of a rotation of A ending at position $n + i + 1$. By Theorem 5.1, we have $b = \Omega(m)$. ◀

6 LCS and CPM in the Asymmetric Streaming Setting

In this section, we use Theorem 3.7 to show that for any $\tau \in [\tilde{\Omega}(\sqrt{m})\,..\, O(m/\log^2 m)]$, there are asymmetric streaming algorithms for LCS and CPM that use $O(\tau)$ space and $\tilde{O}(m/\tau)$ time per letter. We start by giving an algorithm for a generalization of the LCS problem that can be used to solve both LCS and CPM. For two strings S, T, a fragment $T[t\,..\, t']$ is a *T-maximal common substring* of S and T if it is a occurs in S and neither $T[t-1\,..\, t']$ (assuming $t > 1$) nor $T[t\,..\, t'+1]$ (assuming $t' < n$) occurs in S.

▶ **Theorem 6.1.** *Assume to be given read-only random access to a string S of length m and streaming access to a string T of length n over an integer alphabet, where $n \geq m$. For all $\tau \in [\sqrt{m}\log m(\log\log m)^3\,..\, O(m/\log^2 m)]$, there is an algorithm that reports all T-maximal common substrings of S and T using $O(\tau)$ space and $O(nm/\tau \cdot \log\log\sigma)$ time.*

Proof. We cover T with windows of length 2τ (except maybe for the last) that overlap by τ letters: there are $O(n/\tau)$ such windows. After reading such a window W, we apply the procedure encapsulated in the following claim with $A = W$ and $B = S$:

▷ Claim 6.2. Let A, B be strings of respective lengths a and b, where $a < b < a^{O(1)}$, over an integer alphabet of size σ. Given read-only random access to A and B, we can compute all B-maximal common substrings of A and B, and the length $\mathsf{LCSuf}(A, B)$ of the longest suffix of A that occurs in B in $O(b\log\log\sigma)$ time using $O(a)$ extra space.

Proof. We start by building the suffix tree for A and preprocessing it for constant-time weighted ancestor queries: this takes $O(a)$ time (see Fact 2.2 and Fact 2.8). Additionally, we preprocess the labels of edges outgoing from each node according to Fact 2.6. Then, the algorithm traverses the tree maintaining the following invariant: at every moment, it is at a node (maybe implicit) corresponding to a substring $B[i\,..\, j]$ of B. It starts at the root of the tree with $i = 1$ and $j = 0$. In each iteration, the algorithm tries to go down the tree from the current node using $B[j+1]$; this takes $O(\log\log\sigma)$ time. If it succeeds, it increments j and continues. Otherwise, it considers two cases. If it is at the root, it increments both i and j. Otherwise, it jumps to the node corresponding to $B[i+1\,..\, j]$ via a weighted ancestor query in $O(1)$ time and increments i. The nodes reached by an edge traversal and abandoned with the use of a weighted ancestor query in the next iteration are in one-to-one correspondence with the B-maximal common substrings of A and B. The LCSuf of A and B is the depth of the deepest visited node that corresponds to a suffix of A. As at least one of the indices i, j gets incremented at every step of the traversal, the total runtime is $O(b\log\log\sigma)$. ◁

The above sliding-window procedure takes $O(m\log\log\sigma)$ time per window and uses $O(\tau)$ space, which adds up to $O(nm/\tau \cdot \log\log\sigma)$ time in total, and finds all T-maximal common substrings of S and T that have length at most τ.

We run another procedure in parallel in order to compute T-maximal common substrings of length at least τ. During preprocessing, we build the Extended IPM (Decision) data structure (Theorem 3.7) for the string S with $\ell = \tau - 2$ in $O(m \log_{m/\tau} m) = O(nm/\tau)$ time using $O((m/\tau) \cdot \log m (\log\log m)^3) = O(\tau)$ space.

Assume that while reading a window $W = T[\ell\,..\,r]$, the sliding-window procedure found an LCSuf $T[i\,..\,r]$ of length at least τ. We start a search for a common substring starting in W. Let $j \geq r$ be the current letter of T, and $T[i\,..\,j]$, $\ell \leq i \leq r$, be the longest suffix of $T[\ell\,..\,j]$ that occurs in S. We assume that we know a position where $T[i\,..\,j]$ occurs in S, which is the case for $j = r$. When $T[j+1]$ arrives, we update i using the following observation:

▶ **Observation 6.3.** *If $T[i\,..\,j]$ is the longest suffix of $T[1\,..\,j]$ that occurs in S, and $T[i'\,..\,j+1]$ is the longest suffix of $T[1\,..\,j+1]$ that occurs in S, then $i \leq i'$.*

By using binary search and IPM queries, we can find the smallest $i \leq i'$ such that $T[i'\,..\,j+1]$ occurs in S and a witness occurrence, if the corresponding string has length at least τ: namely, if $S[x\,..\,x']$ is a witness occurrence of $T[i\,..\,j]$ in S, we search for occurrences of $P = S[x + (i' - i)\,..\,x']T[j+1]$ in S. If $j - i' < \tau$, we stop the search, and otherwise we set $i' = i$ and continue. It is evident that all T-maximal common substrings of S and T that are of length greater than τ can be extracted during the execution of the above procedure: a maintained suffix of length greater than τ is such a fragment if the last update to it was an increment of its right endpoint, while the next update is an increment of its left endpoint. For every letter, we run at most one binary search which uses $O(\log m)$ IPM queries and hence takes $O(\log m (\log\log m)^3)$ time. As $\tau = O(m/\log^2 m)$, the m/τ term dominates the per-letter running time. The correctness of the described procedure follows from the fact that any substring of T of length greater than τ is either fully contained in the first window or crosses the boundary of some window. ◀

▶ **Corollary 6.4.** *Assume to be given random access to an m-length string S and streaming access to a n-length string T, where $n \geq m$. For all $\tau \in [\sqrt{m} \log m (\log\log m)^3\,..\,O(m/\log^2 m)]$, there is an algorithm that computes LCS(S,T) using $O(nm/\tau \cdot \log\log\sigma)$ time and $O(\tau)$ space.*

Proof. Note that the longest common substring of S and T is a T-maximal substring of S and T. We use the algorithm of Theorem 6.1 with the same value of τ to iterate over all T-maximal common substrings $T[t\,..\,t']$ of S and T, and store the pair of indices t, t' that maximizes $t' - t$. ◀

▶ **Corollary 6.5.** *Assume to be given random access to an m-length pattern P and streaming access to an n-length text T, where $n \geq m$. For all $\tau \in [\sqrt{m} \log m (\log\log m)^3\,..\,O(m/\log^2 m)]$, there is an algorithm that solves the CPM problem for P, T using $O(m/\tau \cdot \log\log\sigma)$ time per letter of T and $O(\tau)$ space.*

Proof. We use the algorithm of Theorem 6.1 with threshold τ on the string $P \cdot P$, to which we have random access, and a streaming string T. The occurrence of any rotation of P in T implies a common substring of $P \cdot P$ and T of length $m \geq 2\tau$. The algorithm of Theorem 6.1 allows us to find such occurrences in $O(m/\tau \cdot \log\log\sigma)$ amortized time per letter of T using $O(\tau)$ space. By noticing that none of the m-length substrings are fully contained in $T(|T| - \tau\,..\,|T|]$, we can deamortise the algorithm using the standard time-slicing technique, cf [30]. ◀

7 CPM in the Read-only Setting

In this section, we present a deterministic read-only online algorithm for the CPM problem.

▶ **Theorem 7.1.** *There is a deterministic read-only online algorithm that solves the CPM problem on a pattern P of length m and a text T of length n using $O(1)$ space and $O(1)$ time per letter of the text.*

Proof. In this proof, we assume that $n \leq 2m-1$. If this is not the case, we can cover T with $2m$-length windows overlapping by $m-1$ letters, and process the text window by window; the last window might be shorter. Every occurrence of a rotation of P belongs to exactly one of the windows and hence will be reported exactly once.

We partition P into four fragments P_1, P_2, P_3, P_4, each of length either $\lfloor m/4 \rfloor$ or $\lceil m/4 \rceil$.[2] By applying Fact 2.4, we compute the periods of each of P and P_i for $i \in [1\mathinner{.\,.} 4]$, if it is are periodic. We also compute, for each $i \in [1\mathinner{.\,.} 4]$, the occurrences of P_i in P^2 using Fact 2.3, and store them in $O(1)$ space due to Fact 2.1. Overall, the preprocessing step takes $O(m)$ time and uses constant space.

We compute all occurrences of all P_i in T in an online manner using Fact 2.3. Due to Fact 2.1, we can represent all computed occurrences of each P_i using a constant number of arithmetic progressions with difference $\mathsf{per}(P_i)$ in $O(1)$ space.

▶ **Observation 7.2.** *Assume that $T(j-m\mathinner{.\,.} j] = P[\Delta+1\mathinner{.\,.} m] \circ P[\mathinner{.\,.}\Delta]$. There is an occurrence of P_i at a position ℓ of T such that $j-m < \ell \leq j-|P_i|+1$ if and only if there is an occurrence of P_i at position $p = \Delta+\ell-j+m$ of P^2.*

Now, note that for every rotation P' of P, some P_i occurs at one of the first $\phi := 2\lceil m/4 \rceil$ positions of P'. We will use such occurrences as anchors to compute the occurrences of rotations of P in T. Fix i such that there is an occurrence of P_i in the first ϕ positions of $T(j-m\mathinner{.\,.} j]$. We consider two cases depending on whether the period of P_i is large or small.

Case I: $\mathsf{per}(P_i) > |P_i|/4$. By Fact 2.1, there are $O(1)$ occurrences of P_i in each of T and P^2. Suppose that P_i occurs at position ℓ of T. If $T(j-m\mathinner{.\,.} j] = P[\Delta+1\mathinner{.\,.} m] \circ P[\mathinner{.\,.}\Delta]$ for some Δ, then, by Observation 7.2, P_i occurs at position $p = \Delta+\ell-j+m$ of P^2 and we must have that the length of the longest common suffix of $T[1\mathinner{.\,.}\ell)$ and $P^2[1\mathinner{.\,.} p)$ is at least $\ell-(j-m)$ and the length of the longest common prefix of $T[\ell+|P_i|\mathinner{.\,.}]$ and $P^2[p+|P_i|\mathinner{.\,.}]$ is at least $j-\ell-|P_i|$. As we only need to consider occurrences of P_i in the first ϕ positions of rotations of P, we can work under the assumption that $\ell-(j-m) \leq \phi$. Hence, it suffices to compute, for every occurrence of P_i at a position p in P^2 and every occurrence of P_i at a position ℓ in T, values

- $x := \max\{\phi, \mathsf{LCE}^R(T[1\mathinner{.\,.}\ell), P^2[1\mathinner{.\,.} p))\}$, the maximum of ϕ and the length of the longest common suffix of $T[1\mathinner{.\,.}\ell)$ and $P^2[1\mathinner{.\,.} p)$;
- $y := \mathsf{LCE}^R(T[1\mathinner{.\,.}\ell), P^2[1\mathinner{.\,.} p))$, the length of the longest common prefix of $T[\ell+|P_i|\mathinner{.\,.}]$ and $P^2[p+|P_i|\mathinner{.\,.}]$.

The length y is computed naively as new letters arrive, while, in order to compute x, we perform a constant number of letter comparisons for each letter that arrives. Since $\ell-(j-m) = O(j-\ell-|P_i|)$, we will have completed the extension to the left when the j-th letter of the text arrives. As there is a constant number of pairs (p,ℓ) to be considered, we perform a total number of $O(1)$ letter comparisons per letter of the text.

[2] The sole reason for partitioning P into four fragments instead of two is to guarantee that there is an occurrence of some P_i close the the starting position of each rotation of P. This allows us to obtain a worst-case rather than an amortised time bound for processing each letter of the text.

Case II: $\mathsf{per}(P_i) \le |P_i|/4$. For brevity, denote $\rho = \mathsf{per}(P_i)$. Below, when we talk about arithmetic progressions of occurrences of P_i, we mean maximal arithmetic progressions of starting positions of occurrences of P_i with difference ρ. Consider the first element ℓ and the last element r of the rightmost computed arithmetic progression of occurrences of P_i in $T(j-m\mathinner{..}j]$. We next distinguish between two cases depending on whether $\mathsf{per}(T(j-m\mathinner{..}j]) = \rho$. This information can be easily maintained in $O(1)$ time per letter using $O(1)$ space as follows. In particular, for each arithmetic progression of occurrences of P_i in T, we perform at most $\rho-1$ letter comparisons to extend the periodicity to the left; we can do this lazily upon computing the first element of each progression, by performing at most one letter comparison for each of the next $\rho-1$ letter arrivals. Further, as at most one arithmetic progression corresponds to occurrences of P_i in T that contain a position in $(j-\rho\mathinner{..}j]$, the extensions to the right take $O(1)$ time per letter as well.

Subcase (a): $\mathsf{per}(T(j-m\mathinner{..}j]) \ne \rho$. Suppose that $T(j-m\mathinner{..}j] = P[\Delta+1\mathinner{..}m] \circ P[\mathinner{..}\Delta]$ for some Δ. Then, due to Observation 7.2, one of the two following holds:

1. ℓ and $p_\ell = \Delta+\ell-j+m$ are the first elements in arithmetic progressions of occurrences of P_i in $T(j-m\mathinner{..}j]$ and P^2, respectively;
2. r and $p_r = \Delta+r-j+m$ are the last elements in arithmetic progressions of occurrences of P_i in $T(j-m\mathinner{..}j]$ and P^2, respectively.

We handle this case by considering a subset of pairs of occurrences of P_i and treating them similarly to Case I. Namely, we consider (a) pairs that are first in their respective arithmetic progressions in P^2 and T and (b) pairs that are last in their respective arithmetic progressions in P^2 and $T(j-m\mathinner{..}j]$. By Fact 2.1, there are only a constant number of such elements in P^2 and a constant number of such elements in the text at any time (a previously last element in the text may stop being last when a new occurrence of P_i is detected). For pairs of first elements there are no changes required to the algorithm for Case I. We next argue that, for each pair (r, p_r) of last elements, it suffices to perform only $O(\rho)$ letter comparisons to check how far the periodicity extends to the left, and that this is all we need to check. Due to this, we do not restrict our attention to the case when $r \in (j-m\mathinner{..}j-m+\phi]$, but rather consider all last elements of arithmetic progressions. Let ℓ' be the first element of the arithmetic progression in $T(j-m\mathinner{..}m]$ that contains r. If $\ell' > \rho+j-m$, we avoid extending to the left since either $\ell' \in (j-m\mathinner{..}j-m+\phi]$ and the sought occurrence of a rotation of P, if it exists, will be computed by the algorithm when it processes pair $(\ell', \Delta+\ell'-j+m)$ or the sought occurrence will be computed when processing a different arithmetic progression of occurrences of P_i or a different P_j. Further note that the extension to the left has been already computed; either ℓ' is not the first element in the arithmetic progression of occurrences of P_i in T (we have assumed that it is in $T(j-m\mathinner{..}j]$), in which case we are trivially done, or ℓ' is the first element of an arithmetic progression in T and hence we extended the periodicity via a lazy computation when the occurrence of P_i at position ℓ' was detected. As the occurrences of P_i in T are spaced at least ρ positions away, the above procedure takes $O(1)$ time per letter of the text.

Subcase (b): $\mathsf{per}(T(j-m\mathinner{..}j]) = \rho$. Using $O(m)$ time and $O(1)$ extra space, we can precompute all $1 \le j \le \rho$ such that $Q_i^\infty[j\mathinner{..}j+m)$ occurs in P^2, where $Q_i = P_i[1\mathinner{..}\rho]$; it suffices to extend the periodicity for each of the $O(1)$ arithmetic progressions of occurrences of P_i in P^2 and to perform standard arithmetic. In particular, the output consists of a constant number of intervals. Then, if $\mathsf{per}(T(j-m\mathinner{..}j]) = \rho$, $T(j-m\mathinner{..}j]$ equals a rotation of P if and only if $\ell-(j-m) \pmod{\rho}$ is in one of the computed intervals and this can be checked in constant time. ◀

References

1 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11), 2020. doi:10.3390/a13110276.

2 Amihood Amir, Mika Amit, Gad M. Landau, and Dina Sokol. Period recovery of strings over the Hamming and edit distances. *Theoretical Computer Science*, 710:2–18, 2018. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). doi:10.1016/j.tcs.2017.10.026.

3 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *Proc. of ESA*, pages 5:1–5:18, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

4 Amihood Amir, Ayelet Butman, Eitan Kondratovsky, Avivit Levy, and Dina Sokol. Multidimensional period recovery. *Algorithmica*, 84(6):1490–1510, 2022. doi:10.1007/S00453-022-00926-Y.

5 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

6 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algor.*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

7 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proc. of FOCS*, pages 377–386, 2010. doi:10.1109/FOCS.2010.43.

8 Lorraine A.K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. doi:10.1016/j.patrec.2017.01.018.

9 Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proc. of SODA*, pages 572–591, 2015. doi:10.1137/1.9781611973730.39.

10 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theoretical Computer Science*, 922:271–282, 2022. doi:10.1016/j.tcs.2022.04.029.

11 Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern matching with mismatches and wildcards. *CoRR*, abs/2402.07732, 2024. doi:10.48550/ARXIV.2402.07732.

12 Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya. Small-space algorithms for the online language distance problem for palindromes and squares. In *Proc. of ISAAC*, pages 10:1–10:17, 2023. doi:10.4230/LIPICS.ISAAC.2023.10.

13 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *Proc. of CPM*, pages 8:1–8:15, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

14 Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In *Proc. of CPM*, pages 5:1–5:14, 2020. doi:10.4230/LIPICS.CPM.2020.5.

15 Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/J.JDA.2013.06.003.

16 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proc. of SODA*, pages 607–626, 2020. doi:10.1137/1.9781611975994.37.

17 Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theoretical Computer Science*, 483:2–9, 2013. doi:10.1016/J.TCS.2012.11.040.

18 Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM J. Comput.*, 42(1):61–83, 2013. doi:10.1137/100816481.

19 Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the RAM, revisited. In *Proc. of SoCG*, pages 1–10, 2011. doi:10.1145/1998196.1998198.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *Proc. of CPM*, pages 8:1–8:15, 2020. doi:10.4230/LIPICS.CPM.2020.8.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

22 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *Proc. of ESA*, pages 30:1–30:17, 2021. Full version: https://arxiv.org/abs/2105.03106. doi:10.4230/LIPICS.ESA.2021.30.

23 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular pattern matching with k mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.

24 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In *Proc. of ESA*, pages 35:1–35:19, 2022. doi:10.4230/LIPICS.ESA.2022.35.

25 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Efficient enumeration of distinct factors using package representations. In *Proc. of SPIRE*, volume 12303, pages 247–261. Springer, 2020. doi:10.1007/978-3-030-59212-7_18.

26 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. of FOCS*, pages 978–989, 2020. doi:10.1109/FOCS46700.2020.00095.

27 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *Proc. of FOCS*, pages 698–707, 2022. doi:10.1109/FOCS54457.2022.00072.

28 Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching under edit distance. In *Proc. of STACS*, pages 24:1–24:22, 2024. doi:10.4230/LIPIcs.STACS.2024.24.

29 Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *The Computer Journal*, 57(5):731–743, March 2013. doi:10.1093/comjnl/bxt023.

30 Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. *Inf. Comput.*, 209(4):731–736, 2011. doi:10.1016/J.IC.2010.12.007.

31 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Internal quasiperiod queries. In *Proc. of SPIRE*, pages 60–75, 2020. doi:10.1007/978-3-030-59212-7_5.

32 Jiangqi Dai, Qingyu Shi, and Tingqiang Xu. Faster algorithms for internal dictionary queries. *CoRR*, abs/2312.11873, 2023. doi:10.48550/ARXIV.2312.11873.

33 Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for Dyck edit distance and RNA folding. In *Proc. of ICALP*, pages 49:1–49:20, 2022. doi:10.4230/LIPIcs.ICALP.2022.49.

34 Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, and Kaiyu Wu. Internal masked prefix sums and its connection to fully internal measurement queries. In *Proc. of SPIRE*, pages 217–232, 2022. doi:10.1007/978-3-031-20643-6_16.

35 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

36 Henning Fernau, Florin Manea, Robert Mercaş, and Markus L. Schmid. Pattern matching with variables: Efficient algorithms and complexity results. *ACM Trans. Comput. Theory*, 12(1), February 2020. doi:10.1145/3369935.

37 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.

38 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. of ESA*, volume 9294, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.

39 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. of CPM*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

40 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.

41 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. doi:10.1007/S00224-017-9794-5.

42 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, and Sharma V. Thankachan. Space-efficient construction algorithm for the circular suffix tree. In *Proc. of CPM*, pages 142–152, 2013. doi:10.1007/978-3-642-38905-4_15.

43 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Proc. of ISAAC*, pages 673–682, 2011. doi:10.1007/978-3-642-25591-5_69.

44 Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Linear-time computation of cyclic roots and cyclic covers of a string. In *Proc. of CPM*, pages 15:1–15:15, 2023. doi:10.4230/LIPICS.CPM.2023.15.

45 Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and indexing circular patterns. In *Algorithms for Next-Generation Sequencing Data: Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.

46 Marek Karpinski and Yakov Nekrich. Space efficient multi-dimensional range reporting. In *Proc. of COCOON*, volume 5609, pages 215–224. Springer, 2009. doi:10.1007/978-3-642-02882-3_22.

47 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

48 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *Proc. of FOCS*, pages 1002–1013, 2020. doi:10.1109/FOCS46700.2020.00097.

49 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. of STOC*, pages 1657–1670, 2022. Full version at http://arxiv.org/abs/1910.10631. doi:10.1145/3519935.3520061.

50 Tomasz Kociumaka. *Efficient data structures for internal queries in texts*. PhD thesis, University of Warsaw, Warsaw, Poland, October 2018. Available at https://depotuw.ceon.pl/handle/item/3614.

51 Tomasz Kociumaka, Ritu Kundu, Manal Mohamed, and Solon P. Pissis. Longest unbordered factor in quasilinear time. In *Proc. of ISAAC*, pages 70:1–70:13, 2018. doi:10.4230/LIPIcs.ISAAC.2018.70.

52 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Optimal data structure for internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235, 2013. arXiv:1311.6235.

53 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proc. of SODA*, pages 532–551, 2015. doi:10.1137/1.9781611973730.36.

54 Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Proc. of ESA*, pages 605–617, 2014. doi:10.1007/978-3-662-44777-2_50.

55 Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. *Journal of Discrete Algorithms*, 46-47:1–15, 2017. doi:10.1016/j.jda.2017.10.004.

56 Dmitry Kosolobov, Florin Manea, and Dirk Nowotka. Detecting one-variable patterns. In *Proc. of SPIRE*, pages 254–270, 2017. doi:10.1007/978-3-319-67428-5_22.

57 Dmitry Kosolobov and Nikita Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. In *Proc. of CPM*, 2024.

58 Moshe Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302, 2013. doi:10.1007/978-3-642-40273-9_18.

59 M. Lothaire. *Applied Combinatorics on Words.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005.

60 Tung Mai, Anup Rao, Ryan A Rossi, and Saeed Seddighin. Optimal space and time for streaming pattern matching. *arXiv preprint arXiv:2107.04660*, 2021.

61 Kazuki Mitani, Takuya Mieno, Kazuhisa Seto, and Takashi Horiyama. Internal longest palindrome queries in optimal time. In *Proc. of WALCOM*, pages 127–138, 2023.

62 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. of ICALP*, volume 5125, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

63 Michael Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *Proc. of SODA*, pages 1698–1709, 2013. doi:10.1137/1.9781611973105.122.

64 Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proc. of CPM*, pages 223–234, 2013. doi:10.1007/978-3-642-38905-4_22.

65 Robert Susik, Szymon Grabowski, and Sebastian Deorowicz. Fast and simple circular pattern matching. In *Man-Machine Interactions 3*, pages 537–544, 2014.

66 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Random Wheeler Automata

Ruben Becker
Ca' Foscari University of Venice, Italy

Davide Cenzato
Ca' Foscari University of Venice, Italy

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy

Bojana Kodric
Ca' Foscari University of Venice, Italy

Riccardo Maso
Ca' Foscari University of Venice, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

Wheeler automata were introduced in 2017 as a tool to generalize existing indexing and compression techniques based on the Burrows-Wheeler transform. Intuitively, an automaton is said to be Wheeler if there exists a total order on its states reflecting the natural co-lexicographic order of the strings labeling the automaton's paths; this property makes it possible to represent the automaton's topology in a constant number of bits per transition, as well as efficiently solving pattern matching queries on its accepted regular language. After their introduction, Wheeler automata have been the subject of a prolific line of research, both from the algorithmic and language-theoretic points of view. A recurring issue faced in these studies is the lack of large datasets of Wheeler automata on which the developed algorithms and theories could be tested. One possible way to overcome this issue is to generate random Wheeler automata. Motivated by this observation of practical nature, in this paper we initiate the theoretical study of random Wheeler automata, focusing our attention on the deterministic case (Wheeler DFAs – WDFAs). We start by naturally extending the Erdős-Rényi random graph model to WDFAs, and proceed by providing an algorithm generating uniform WDFAs according to this model. Our algorithm generates a uniform WDFA with n states, m transitions, and alphabet's cardinality σ in $O(m)$ expected time ($O(m \log m)$ time w.h.p.) and constant working space for all alphabets of size $\sigma \leq m/\ln m$. The output WDFA is streamed directly to the output. As a by-product, we also give formulas for the number of distinct WDFAs and obtain that $n\sigma + (n - \sigma)\log\sigma$ bits are necessary and sufficient to encode a WDFA with n states and alphabet of size σ, up to an additive $\Theta(n)$ term. We present an implementation of our algorithm and show that it is extremely fast in practice, with a throughput of over 8 million transitions per second.

2012 ACM Subject Classification Theory of computation → Generating random combinatorial structures; Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis

Keywords and phrases Wheeler automata, Burrows-Wheeler transform, random graphs

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.5

Related Version *Full Version*: `https://arxiv.org/abs/2307.07267`

Supplementary Material
Software (Source Code): `https://github.com/regindex/Wheeler-DFA-generation`
archived at `swh:1:dir:8c1d3690d81c3e6600423dfcbe8ce8bffb769f64`

Funding *Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Wheeler automata were introduced by Gagie et al. in [11] in an attempt to unify existing indexing and compression techniques based on the Burrows-Wheeler transform [5]. An automaton is said to be Wheeler if there exists a total order of its states such that (i) states reached by transitions bearing different labels are sorted according to the underlying total alphabet's order, and (ii) states reached by transitions bearing the same label are sorted according to their predecessors (i.e. the order propagates forward, following pairs of equally-labeled transitions). Equivalently, these axioms imply that states are sorted according to the co-lexicographic order of the strings labeling the automaton's paths. Since their introduction, Wheeler automata have been the subject of a prolific line of research, both from the algorithmic [7, 3, 10, 12, 9, 6, 13] and language-theoretic [2, 1, 8] points of view. The reason for the success of Wheeler automata lies in the fact that their total state order enables *simultaneously* to index the automaton for pattern matching queries and to represent the automaton's topology using just $O(1)$ bits per transition (as opposed to the general case, requiring a logarithmic number of bits per transition).

A recurring issue faced in research works on Wheeler automata is the lack of datasets of (large) Wheeler automata on which the developed algorithms and theories could be tested. As customary in these cases, a viable solution to this issue is to randomly generate the desired combinatorial structure, following a suitable distribution. The most natural distribution, the uniform one, represents a good choice in several contexts and can be used as a starting point to shed light on the combinatorial objects under consideration; the case of random graphs generated using the Erdős-Rényi random graph model [15] is an illuminating example. In the case of Wheeler automata, we are aware of only one work addressing their random generation: the WGT suite [6]. This random generator, however, does not guarantee a uniform distribution over the set of all Wheeler automata.

Our contributions

Motivated by the lack of formal results in this area, in this paper we initiate the theoretical study of random Wheeler automata, focusing our attention on the algorithmic generation of uniform deterministic Wheeler DFAs (WDFAs). We start by extending the Erdős-Rényi random graph model to WDFAs: our uniform distribution is defined over the set $\mathcal{D}_{n,m,\sigma}$ of all Wheeler DFAs over the *effective* alphabet (i.e. all labels appear on some edge) $[\sigma] = \{1, \ldots, \sigma\}$, with n states $[n]$, m transitions, and Wheeler order $1 < 2 < \ldots < n$. We observe that, since any WDFA can be encoded using $O(n\sigma)$ bits [11], the cardinality of $\mathcal{D}_{n,m,\sigma}$ is at most $2^{O(n\sigma)}$. On the other hand, the number of DFAs with n states over alphabet of size σ is $2^{\Theta(n\sigma \log n)}$ [15]. As a result, a simple rejection sampling strategy that uniformly generates DFAs until finding a WDFA (checking the Wheeler property takes linear time on DFAs [1]) would take expected exponential time to terminate. To improve over this naive solution, we start by defining a new combinatorial characterization of WDFAs: in Section 3, we establish a bijection that associates every element of $\mathcal{D}_{n,m,\sigma}$ to a pair formed by a binary matrix and a binary vector. This allows us to design an algorithm to uniformly sample WDFAs, based on the above-mentioned representation. Remarkably, our sampler uses *constant* working space and streams the sampled WDFA directly to output:

▶ **Theorem 1.** *There is an algorithm to generate a uniform WDFA from $\mathcal{D}_{n,m,\sigma}$ in $O(m)$ expected time ($O(m \log m)$ time with high probability) using $O(1)$ words of working space, for all alphabets of size $\sigma \leq m/\ln m$. The output WDFA is directly streamed to the output as a set of labeled edges.*

As a by-product of our combinatorial characterization of WDFAs, in Theorem 19 we give an exact formula for the number $|\mathcal{D}_{n,m,\sigma}|$ of distinct WDFAs with n nodes and m edges labeled from alphabet $[\sigma]$ and in Theorem 20 we give a tight asymptotic formula for the number $|\mathcal{D}_{n,\sigma}|$ of distinct WDFAs with n nodes and any number of edges labeled from $[\sigma]$, obtaining that $n\sigma + (n - \sigma)\log \sigma$ bits are necessary and sufficient to encode WDFAs from such a family up to an additive $\Theta(n)$ term.

We conclude by presenting an implementation of our algorithm, publicly available at `https://github.com/regindex/Wheeler-DFA-generation`, and showing that it is very fast in practice while using a negligible (constant) amount of working space.

2 Preliminaries and Problem Statement

With $\ln x$ and $\log x$, we indicate the natural logarithm and the logarithm in base 2 of x, respectively. For an integer $k \in \mathbb{N}^+$, we let $[k]$ denote the set of all integers from 1 to k. For a bit-vector $v \in \{0,1\}^k$, we denote with $\|v\| = \sum_{i \in [k]} v_i$ the L_1-norm of v, i.e., the number of set bits in v. For an integer $\ell \leq k$, we denote with $v[1 : \ell]$ the bit-vector $(v_1, \dots, v_\ell)$ consisting only of the first ℓ bits of v. For a bit-matrix $A \in \{0,1\}^{\ell \times k}$ and a column index $j \in [k]$, we denote the j'th column of A by A_j and the element at row i and column j as $A_{i,j}$. We let $\|A\| = \sum_{i \in [k], j \in [\ell]} A_{i,j}$ be the $L_{1,1}$-norm of A, which again counts the number set bits in A. For a bit-vector $v \in \{0,1\}^k$, we use the notation $\mathrm{rank}(v, i)$ to denote the number of occurrences of 1 in $v[1 : i]$. For completeness, we let $\mathrm{rank}(v, 0) = 0$. We generalize this function also to matrices as follows. For a bit-matrix $A \in \{0,1\}^{\ell \times k}$, we let $\mathrm{rank}(A, (i, j)) = \sum_{r \in [j-1]} \mathrm{rank}(A_r, \ell) + \mathrm{rank}(A_j, i)$. We sometimes write bit-vectors from $\{0,1\}^k$ in string form, i.e., as a sequence of k bits.

In this paper we are concerned with deterministic finite automata.

▶ **Definition 2** (Determinisitic Finite Automaton (DFA)). *A Determinisitic Finite (Semi-) Automaton (DFA) D is a triple (Q, Σ, δ) where $Q = [n]$ is a finite set of n states with $1 \in Q$ being the source state, $\Sigma = [\sigma]$ is the finite alphabet of size σ, and $\delta : Q \times \Sigma \to Q$ is a transition function containing m transitions.*

We omit to specify the final states of DFAs, since they do not play a role in the context of our problem. We use the shorthand $\delta_j(v)$ for $\delta(v, j)$. Furthermore, we write $\delta^{out}(v) := \{\delta_j(v) : j \in \Sigma\}$ for the set of all out-neighbors of a state $v \in Q$ and $\delta^{in}(v) := \{u \in Q : \exists j \in \Sigma \text{ with } v \in \delta_j(u)\}$ for the set of all in-neighbors of v. We assume DFAs to have non-zero in-degree for exactly the non-source states, i.e., $\delta^{in}(v) \neq \emptyset$ if and only if $v > 1$; This choice simplifies our exposition and it is not restrictive from the point of view of the languages accepted by such DFAs. We do not require the transition function δ to be complete; This choice is motivated by the fact that requiring completeness restricts the class of Wheeler DFAs [2]. Furthermore, we do not require DFAs to be connected; Also this choice is customary as it allows, for instance, to use our WDFA sampler to empirically study properties such as connectivity phase transition thresholds.

We say that the alphabet Σ is *effective* if and only if $(\forall j \in \Sigma)(\exists u, v \in Q)(\delta_j(u) = v)$, i.e. if every character of Σ labels at least one transition. We assume that the alphabet $\Sigma = [\sigma]$ is totally ordered according to the standard order among integers. Wheeler DFAs constitute a special class of DFAs that can be stored compactly and indexed efficiently due to an underlying order on the states: the *Wheeler order* (see Definition 3). As said in Definition 2, in this paper the states Q of an automaton D are represented by the integer set $[n]$ for some positive integer n; note that in the following definition we use the order on integers $<$ to denote the Wheeler order on the states.

▶ **Definition 3** (Wheeler DFA [11]). *A* Wheeler DFA *(WDFA) is a DFA D such that $<$ is a* Wheeler order, *i.e. for $a, a' \in \Sigma$, $u, v, u', v' \in Q$:*
(i) *If $u' = \delta_a(u)$, $v' = \delta_{a'}(v)$, and $a \prec a'$, then $u' < v'$.*
(ii) *If $u' = \delta_a(u) \neq \delta_a(v) = v'$ and $u < v$, then $u' < v'$.*

We note that the source axiom present in [11], which requires that the source state is first in the order, vanishes in our case as the ordering $<$ on the integers directly implies that the source state is ordered first. Notice that property (i) in Definition 3 implies that a WDFA is *input-consistent*, i.e., all in-going transitions to a given state have the same label.

▶ **Definition 4.** *With $\mathcal{D}_{n,m,\sigma}$ we denote the set of all Wheeler DFAs with effective alphabet $\Sigma = [\sigma]$, n states $Q = [n]$, m transitions, and Wheeler order $1 < 2 < \ldots < n$.*

Clearly, $\mathcal{D}_{n,m,\sigma}$ is a subset of the set $\mathcal{A}_{n,m,\sigma}$ of all finite (possibly non-deterministic) automata over the ordered alphabet $[\sigma]$ with n states $[n]$ and m transitions.

In this paper we investigate the following algorithmic problem:

▶ **Problem 5.** *For given n, m, and σ, generate an element from $\mathcal{D}_{n,m,\sigma}$ uniformly at random.*

Note that, since in Definition 4 we require $1 < 2 < \ldots < n$ to be the Wheeler order, Problem 5 is equivalent to that of uniformly generating pairs formed by a Wheeler DFA D and a valid Wheeler order for the states $Q = [n]$ of D, not necessarily equal to the integer order $1 < 2 < \cdots < n$. Throughout the whole paper, we assume that $n - 1 \leq m \leq n\sigma$ and $\sigma \leq n - 1$ (due to input consistency), as otherwise $\mathcal{D}_{n,m,\sigma} = \emptyset$ and the problem is trivial.

3 An Algorithm for Uniformly Generating WDFAs

Our strategy towards solving Problem 5 efficiently is to associate every element D from $\mathcal{D}_{n,m,\sigma}$ to exactly one pair (O, I) of elements from $\mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ (see Definition 6 below) via a function $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ ("r" stands for *representation*). Formally, the two sets appearing in the co-domain of r are given in the following definition.

▶ **Definition 6.** *Let*

$$\mathcal{O}_{n,\sigma,m} := \left\{O \in \{0,1\}^{n\times\sigma} : \|O\| = m \text{ and } \|O_j\| \geq 1 \text{ for all } j \in [\sigma]\right\} \quad \text{and}$$
$$\mathcal{I}_{m,n} := \left\{I \in \{0,1\}^{m} : \|I\| = n - 1\right\}.$$

The intuition behind the two sets $\mathcal{O}_{n,\sigma,m}$ and $\mathcal{I}_{m,n}$ is straightforward: their elements encode the outgoing labels and the in-degrees of the automaton's states, respectively. In order to describe more precisely this intuition, let us fix an automaton $D = (Q, \delta, \Sigma) \in \mathcal{D}_{n,m,\sigma}$ and consider its image $r(D) = (O, I) \in \mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ (see Figures 1 and 2 for an illustration):

- The matrix O is an encoding of the labels of the out-transitions of D. A 1-bit in position $O_{u,j}$ means that there is an out-going transition from state u labeled j. Formally,

$$O_{u,j} := \begin{cases} 1 & \text{if } \exists v : v = \delta_j(u) \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

- The vector I is a concise encoding of the in-degrees of all states. It is defined as

$$I := (\underbrace{1, 0, \ldots, 0}_{|\delta^{in}(2)|}, \underbrace{1, 0, \ldots, 0}_{|\delta^{in}(3)|}, \ldots, \underbrace{1, 0, \ldots, 0}_{|\delta^{in}(n)|}), \tag{2}$$

i.e, for all states i other than the source (that has no in-transitions), the vector contains exactly one 1-bit followed by $|\delta^{in}(i)| - 1$ 0-bits.

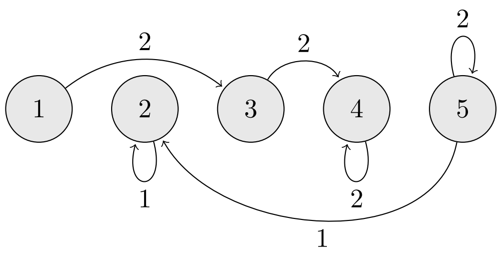


Figure 1 Running example: a WDFA D with $n = 5$ states, $m = 6$ edges, alphabet cardinality $\sigma = 2$, and Wheeler order $1 < 2 < 3 < 4 < 5$. Note that the WDFA has two connected components.

	1	2
1	0	1
2	1	0
3	0	1
4	0	1
5	1	1

2		3	4		5
1	0	**1**	1	0	1

Figure 2 Matrix O (left) and bit-vector I (right) forming the encoding $r(D) = (O, I)$ of the WDFA D of Figure 1. In matrix O, column names are characters from $\Sigma = [\sigma]$ and row names are states from $Q = [n]$. In bit-vector I, each state (except state 1) is associated with a bit set, in Wheeler order. Cells containing a set bit are named with the name of the corresponding state. Bits in bold highlight the states on which the character that labels the state's incoming transitions changes (i.e. state 2 is the first whose incoming transitions are labeled 1, and state 3 is the first whose incoming transitions are labeled 2).

Let us proceed with two remarks.

▶ Remark 7. As $\|O\| = m$ there are m transitions in total. As $\|O_j\| \geq 1$ for all $j \in [\sigma]$, the alphabet is effective, i.e., every character labels at least one transition.

▶ Remark 8. The vector I does not encode the letter on which a transition is in-going to a given state. Notice however that as D is a WDFA all these transitions have to be labeled with the same letter and we can reconstruct this letter for a given I once we know the total number of transitions labeled with each letter. This is because property (i) of Definition 3 guarantees that the node order is such that the source state (that has no in-going transitions) is ordered first followed by nodes whose in-transitions are labeled with character 1, followed by nodes with in-transitions labeled with character 2, etc. The information on how many transitions are labeled with each character is carried by the matrix O for which $r(D) = (O, I)$.

Let (O, I) be a pair from the image of r, i.e, $r(D) = (O, I)$ for some D. Then it will always be the case that I is contained in a subset $\mathcal{I}_O$ of $\mathcal{I}_{m,n}$ that can be defined as follows.

▶ **Definition 9.** *For a matrix $O \in \mathcal{O}_{n,\sigma,m}$, let*

$$\mathcal{I}_O := \left\{ I \in \mathcal{I}_{m,n} : I_{1+\sum_{k=1}^{j-1} \|O_k\|} = 1 \textit{ for all } j \in [\sigma] \right\}.$$

Using our running example of Figure 2, the bits $I_{1+\sum_{k=1}^{j-1}\|O_k\|}$ that we force to be equal to 1 are those highlighted in bold, i.e. I_1 and I_3: noting that bits in I correspond to edges, those bits correspond to the leftmost edge labeled with a given character j (for any $j \in \Sigma$).

This leads us to define the following subset of $\mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$:

▶ **Definition 10.** $\mathcal{R}_{n,m,\sigma} := \{(O, I) : O \in \mathcal{O}_{n,\sigma,m}$ *and* $I \in \mathcal{I}_O\}$.

Based on the above definition, we can prove:

▶ **Lemma 11.** *For any* $D \in \mathcal{D}_{n,m,\sigma}$, $r(D) \in \mathcal{R}_{n,m,\sigma}$.

Proof. Note that the integers $\sum_{k=1}^{j-1}\|O_k\|$ for $j \in [\sigma]$ correspond to the number of edges labeled with letters $1, \dots, j-1$, hence the positions $1 + \sum_{k=1}^{j-1}\|O_k\|$ correspond to a change of letter in the sorted (by destination node) list of edges. Recalling that WDFAs are input-consistent (i.e., all in-transitions of a given node carry the same label) and that nodes are ordered by their in-transition letters, positions $1+\sum_{k=1}^{j-1}\|O_k\|$ for $j \in [\sigma]$ in I must necessarily correspond to the first edge of a node, hence they must contain a set bit. ◀

The co-domain of the function r can thus be restricted, and the function's signature can be redefined, as follows: $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$.

After describing this association of a WDFA $D \in \mathcal{D}_{n,m,\sigma}$ to a (unique) pair $r(D) = (O, I) \in \mathcal{R}_{n,m,\sigma}$, we will argue that function r is indeed a bijection from $\mathcal{D}_{n,m,\sigma}$ to $\mathcal{R}_{n,m,\sigma}$. It will follow that one way of generating elements from $\mathcal{D}_{n,m,\sigma}$ is to generate elements from $\mathcal{R}_{n,m,\sigma}$: this will lead us to an efficient algorithm to uniformly sample WDFAs from $\mathcal{D}_{n,m,\sigma}$, as well to a formula for the cardinality of $\mathcal{D}_{n,m,\sigma}$.

3.1 The Basic WDFA Sampler

Our overall approach is to (1) uniformly sample a matrix O from $\mathcal{O}_{n,\sigma,m}$ using Algorithm 2, then (2) uniformly sample a vector I from $\mathcal{I}_O$ using Algorithm 3 with input O, and finally (3) build a WDFA D using O and I as input via Algorithm 4. We summarize this procedure in Algorithm 1. A crucial point in our correctness analysis (Section 4) will be to show that uniformly sampling from $\mathcal{O}_{n,\sigma,m}$ and $\mathcal{I}_O$ does indeed lead to a uniform WDFA from $\mathcal{D}_{n,m,\sigma}$ (besides the bijectivity of r, intuitively, this is because $|\mathcal{I}_O| = |\mathcal{I}_{O'}|$ for any $O, O' \in \mathcal{O}_{n,\sigma,m}$).

As source of randomness, our algorithm uses a black-box *shuffler* algorithm: given a bit-vector $B \in \{0,1\}^*$, function `shuffle`(B) returns a random permutation of B. To improve readability, in this subsection we start by describing a preliminary simplified version of our algorithm which does not assume any particular representation for the matrix-bit-vector pair $(O, I) \in \mathcal{R}_{n,m,\sigma}$, nor a particular shuffling algorithm (for now, we only require the shuffling algorithm to permute uniformly its input). By employing a particular *sequential shuffler*, in Subsection 3.2 we then show that we can generate a sparse representation of O and I on-the-fly, thereby achieving *constant* working space and linear expected running time.

Algorithm 1 `sample_D`(n, m, σ).

```
1 $O :=$ sample_O$(n, m, \sigma)$
2 $I :=$ sample_I$(O)$
3 $D :=$ build_D$(O, I)$
4 return $D$
```

Out-transition Matrix. In order to sample the matrix O from $\mathcal{O}_{n,\sigma,m}$, in addition to function `shuffle` we assume a function $\texttt{reshape}_{k,\ell}$ that takes a vector x of dimension $k \cdot \ell$ and outputs a matrix A of dimension $k \times \ell$ with the j'th column A_j being the portion $x_{(j-1)\cdot k+1}, \dots, x_{j\cdot k}$ of x. The algorithm to uniformly generate O from $\mathcal{O}_{n,\sigma,m}$ then simply samples a bit vector of length $n\sigma$ with exactly m 1-bits, shuffles it uniformly, reshapes it to be a matrix of dimension $n \times \sigma$ and repeats these steps until a matrix is found with at least one 1-bit in each column (rejection sampling).

Algorithm 2 `sample_O`(n, m, σ).

```
1 repeat
2     $O := \texttt{reshape}_{n,\sigma}(\texttt{shuffle}(1^m 0^{n\sigma-m}))$
3 until $\|O_j\| \geq 1$ for all $j \in [\sigma]$
4 return $O$
```

Looking at the running example of Figures 1 and 2, the shuffler is called as $\texttt{shuffle}(1^6 0^4)$. In this particular example, this bit-sequence is permuted as 0100110111 by function `shuffle`. Function $\texttt{reshape}_{n,\sigma}$ converts this bit-sequence into the matrix O depicted in Figure 2, left.

In-transition Vector. In order to generate the vector I from $\mathcal{I}_{m,n}$, we proceed as follows. The algorithm takes O as input and generates a uniform random element from the set $\mathcal{I}_O$ by first creating a "mask" that is a vector of the correct length m and contains σ 1-bits at the points $1 + \sum_{k=1}^{j-1} \|O_k\|$ for $j \in [\sigma]$. These are the points in I where the character of the corresponding transition changes and hence, by the input-consistency condition, also the state has to change. The remaining $m - \sigma$ positions in the mask are filled with the wildcard character $\#$. We then give this mask vector as the first argument to a function `fill` that replaces the $m - \sigma$ positions that contain the wildcard character $\#$ with the characters in the second argument (in order). Formally, the function `fill` takes two vectors as arguments a and b with the condition that a contains $|b|$ times the $\#$ character and $|a| - |b|$ times a 1-bit. The function then returns a vector c that satisfies $c_i = 1$ whenever $a_i = 1$ and $c_i = b_{i-\text{rank}(a,i)}$ otherwise, i.e., when $a_i = \#$.

Algorithm 3 `sample_I`(O).

```
1 extract $n, m, \sigma$ from $O$
2 mask $:= 1\#^{\|O_1\|-1}1\#^{\|O_2\|-1}\dots 1\#^{\|O_\sigma\|-1}$
3 $I := \texttt{fill}(\text{mask}, \texttt{shuffle}(1^{n-\sigma-1}0^{m-n+1}))$
4 return $I$
```

Going back to our running example of Figures 1 and 2, we have mask = **1**#**1**### (that is, all bits but the bold ones in the right part of Figure 2 are masked with a wildcard). The shuffler is called as $\texttt{shuffle}(1100)$ and, in this particular example, returns the shuffled bit-vector 0101. Finally, function `fill` is called as `fill`(**1**#**1**###,0101) and returns the bit-vector I =101101 depicted in the right part of Figure 2.

Building the WDFA. After sampling O and I, the remaining step is to build the output DFA D. This is formalized in Algorithm 4. By iterating over all non-zero elements in O, we construct the transition function δ: the i'th non-zero entry in O corresponds to an in-transition at state rank$(I, i) + 1$ (we keep a counter named v corresponding to this rank). The origin state of the transition is the row in which we find the i'th 1 in O when reading O column-wise. The column itself corresponds to the label of this transition.

Algorithm 4 `build_D`(O, I).

```
1 extract $n, m, \sigma$ from $O$, $Q := [n]$, $\Sigma := [\sigma]$
2 $\delta := \emptyset$, $i := 1$, $v := 1$
3 for $j = 1, 2, \ldots, \sigma$ do
4     for $u = 1, 2, \ldots, n$ do
5         if $O_{u,j} = 1$ then
6             if $I_i = 1$ then  $v := v + 1$
7             $\delta := \delta \cup \{((u, j), v))\}$, $i := i + 1$
8 return $D = (Q, \Sigma, \delta)$
```

3.2 Constant-Space WDFA Sampler

Notice that our Algorithm 4 accesses the matrix O and the bit-vector I in a sequential fashion: O is accessed column-wise and I from its first to last position. Based on this observation, we now show how our WDFA sampler can be modified to use *constant* working space. The high-level idea is to generate on-the-fly the positions of non-zero entries of O and I in increasing order.

In order to achieve this, we employ the *sequential shuffler* described by Shekelyan and Cormode [17]. Given two integers N and n, the function `init_sequential_shuffler`(N, n) returns an iterator S that can be used (with a stack-like interface) to extract n uniform integers without replacement from $[N]$, in *ascending order* and using a *constant* number of words of working space (that is, the random integers are generated on-the-fly upon request, from the smallest to the largest). More specifically, function S.`pop`() returns the next sampled integer, while S.`empty`() returns true if and only if all n integers have been extracted. The sequential shuffling algorithm is essentially a clever modification of Knuth's shuffle [14] (also referred to as Fisher-Yates shuffler). Knuth's shuffler, after going through the arbitrarily ordered set $[N]$, and in the i'th iteration (for i from 1 to n) swapping the i'th item with the item at a random position $[i, N]$, returns the first n items in the resulting permutation. Knuth's shuffler requires working space proportional to n as we need to remember which elements have been swapped from lower positions (i.e., index $\leq n$) into higher positions (i.e., index $> n$). The idea behind the sequential shuffler of Shekelyan and Cormode is to first sample just the cardinality H of the set of items in higher positions that Knuth's shuffler would swap into lower position. Then, in a second step the algorithm samples H actual items from higher positions with replacement, resulting in $h \leq H$ elements. Finally, in a third step, $n - h$ items are sampled from lower positions. We note that the distribution in the first step is chosen such that the sampling in the second step can be done with replacement – sampling duplicates simply increases the number of items sampled from lower positions. We refer the reader to the article by Shekelyan and Cormode [17] for further details.

Algorithm Description. We now describe Algorithm 5. We recall the mask employed in Algorithm 3: Algorithm 5 iterates, using variable i, over the ranks (i.e., i-th occurrences) of characters # (wildcards) in the mask. Variable i', on the other hand, stores the rank of the next wildcard # that is replaced with a set bit by the shuffler; the values of i' are extracted

Algorithm 5 `sample_D_constant_space`(n, m, σ).

```
1  $i := 1$                                  /* Current position in the subsequence of #'s of the mask */
2  $v := 1$                                  /* Destination state of current transition */
3  $S_O :=$ init_sequential_shuffler$(n\sigma, m)$
4  $S_I :=$ init_sequential_shuffler$(m - \sigma, n - \sigma - 1)$
5  $i' := S_I$.pop()                         /* next nonzero position in sequence of #'s in the mask */
6  $j := 0$                                  /* current column in $O$ */
7  $prev\_j := 0$                            /* previously-visited column in $O$ */
8  while not $S_O$.empty() do
9      $t := S_O$.pop()
10     $(u, j) := \Big(\big((t-1) \mod n\big) + 1, \big((t-1) \text{ div } n\big) + 1\Big)$   /* Nonzero coordinate in $O$ */
11     if $j > prev\_j + 1$ then
12         clear output stream and goto line 1   /* Rejection: $\|O_{prev\_j+1}\| = 0$ */
13     if $j = prev\_j + 1$                  /* Column of $O$ changes */
14      then
15         $v := v + 1$
16         $prev\_j := j$
17     else
18         if $i = i'$ then
19             $v := v + 1$
20             $i' := S_I$.pop()             /* next nonzero position in sequence of #'s in the mask */
21         $i := i + 1$
22     output $((u, j), v)$                  /* Stream transition to output */
23 if $j \neq \sigma$ then
24     clear output stream and goto line 1   /* Rejection: $\|O_\sigma\| = 0$ */
```

from the shuffler S_I. Now, whenever $i = i'$, we are looking at a bit set in bit-vector I (which here is not stored explicitly, unlike in Algorithm 4) and thus we have to move to the next destination state v. This procedure exactly simulates Lines 6 and 7 of Algorithm 4.

The iteration (column-wise) over all non-zero entries of matrix O is simulated by the extraction of values from the shuffler I_O (one value per iteration of the while loop at Line 8): each such value t extracted at Line 9 is converted to a pair (u, j) at Line 10. Variables j and $prev\_j$ store the columns of the current and previously-extracted non-zero entries of O, respectively. If $j > prev\_j + 1$, then column number $prev\_j + 1$ has been skipped by the shuffler, i.e., O_{prev_j+1} does not contain non-zero entries. In this case, we reject and start the sampler from scratch (Line 12; note that we need to clear the output stream before re-initializing the algorithm). If, on the other hand, $j = prev\_j + 1$ (Line 13), then the current non-zero entry of O belongs to the next column with respect to the previously-extracted non-zero entry; this means that the character labeling incoming transitions changes and we therefore move to the next destination node by increasing $v := v + 1$ (Line 15). In this case we do not increment i, since the new destination node v is the first having incoming label j and thus it does not correspond to a character # in the mask. Variable i gets incremented only if $j = prev\_j$: this happens at Line 21. The other case in which we need to move to the next destination node ($v := v + 1$) is when $j = prev\_j$ and $i = i'$ (Line 19). In such a case, in addition to incrementing v we also need to extract from the shuffler S_I the rank i' of the next mask character # that is replaced with a set bit (Line 20). After all these operations, we write the current transition $((u, j), v)$ to the output stream (Line 22). The last two lines of Algorithm 5 check if the last visited column of matrix O is indeed O_σ. If not, $\|O_\sigma\| = 0$ and we need to reject and re-start the algorithm.

The remaining components of Algorithm 5 are devoted to simulate Algorithm 1, using as input the two sequences of random pairs/integers extracted from S_O and S_I, respectively. As a matter of fact, the two loops in Algorithm 4 correspond precisely to extracting the pairs (u, j) from S_O, and the check at Line 6 of Algorithm 4, together with the increment of i at Line 7, corresponds to extracting the integers i' from S_I. The rejection sampling mechanism (repeat-until loop in Algorithm 2) is simulated in Algorithm 5 by re-starting the algorithm whenever the column j of the current pair (u, j) is either larger by more than one unit than the column j_prev of the previously-extracted pair (i.e., $\|O_{j\_prev+1}\| = 0$, Line 12), or if the last pair extracted from S_O is such that j is not the σ-th column (i.e., $\|O_\sigma\| = 0$, Line 24).

Running Example. To understand how the sequential shuffler is used in Algorithm 5, refer again to the running example of Figures 1 and 2. In Algorithm 5 at Line 3, the sequential shuffler S_O is initialized as $S_O :=$ `init_sequential_shuffler`$(n\sigma = 10, m = 6)$, i.e. the iterator S_O returns 6 uniform integers without replacement from the set $\{1, 2, \dots, 10\}$. In this particular example, function `pop`() called on iterator S_O returns the following integers, in this order: 2,5,6,8,9,10. Using the formula at Line 10 of Algorithm 5, these integers are converted to the matrix coordinates $(2, 1), (5, 1), (1, 2), (3, 2), (4, 2), (5, 2)$, i.e., precisely the nonzero coordinates of matrix O in Figure 2, sorted first by column and then by row.

Using the same running example, the sequential shuffler S_I is initialized in Line 4 of Algorithm 5 as $S_I :=$ `init_sequential_shuffler`$(m - \sigma = 4, n - \sigma - 1 = 2)$, i.e. the iterator S_I returns two uniform integers without replacement from the set $\{1, 2, 3, 4\}$. In this particular example, function `pop`() called on iterator S_I returns the following integers, in this order: 2,4. Using the notation of the previous subsection, this sequence has the following interpretation: the 2-nd and 4-th occurrences of # of our mask 1#1### used in Algorithm 3 have to be replaced with a bit 1, while the others with a bit 0. After this replacement, the mask becomes 101101, i.e. precisely bit-vector I of Figure 2.

4 Analysis

4.1 Correctness, Completeness and Uniformity

Being Algorithm 5 functionally equivalent to Algorithm 1 (the only relevant difference between the two being the employed data structures to represent matrix O and bit-vector I), for ease of explanation in this section we focus on analyzing the correctness (the algorithm generates only elements from $\mathcal{D}_{n,m,\sigma}$), completeness (any element from $\mathcal{D}_{n,m,\sigma}$ can be generated by the algorithm) and uniformity (all $D \in \mathcal{D}_{n,m,\sigma}$ have the same probability to be generated by the algorithm) of Algorithm 1. These properties then automatically hold on Algorithm 5 as well.

We start with a simple lemma. The lemma says the following: Assume that $r(D) = (O, I)$ and O contains a 1 in position (u, j), meaning that there is a transition leaving state u, labeled with letter j. Then this out-transition is the $i = \text{rank}(O, (u, j))$ bit that is set to 1 in O and hence the entry in I corresponding to this transition can be found at $I[i]$. The state to which this transition is in-going is exactly the number of 1s in I up to this point, i.e., $\text{rank}(I, i)$, plus one (the offset is due to the source having no in-transitions).

▶ **Lemma 12.** *Let $D \in \mathcal{D}_{n,m,\sigma}$ and let $r(D) = (O, I)$. If $O_{u,j} = 1$ and* $\text{rank}(O, (u, j)) = i$*, then* $\delta(u, j) = \text{rank}(I, i) + 1$.

Proof. First, notice that since $O_{u,j} = 1$, it is clear that there is an outgoing transition from u labeled j. Furthermore, since $\text{rank}(O, (u, j)) = i$, we know that this transition corresponds to the i-th entry in I. Now, by the definition of I, it follows that the destination state of the considered transition is $v = \text{rank}(I, i) + 1$. ◀

Algorithm 4 is a deterministic algorithm and thus describes a function, say f, from the set of its possible inputs to the set of its possible outputs. The set of its possible inputs, i.e., the domain of f, is exactly $\mathcal{R}_{n,m,\sigma}$. The algorithm's output is certainly a finite automaton, i.e., the co-domain of f is $\mathcal{A}_{n,m,\sigma}$. We will in fact show that the range of f is exactly $\mathcal{D}_{n,m,\sigma}$. We will do so by showing that f is actually an inverse of r, more precisely we show that (1) r is surjective and (2) f is a left-inverse of r (and thus r is injective).

Surjectivity of r. We start with proving that r is surjective.

▶ **Lemma 13.** *It holds that $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is surjective.*

Proof. Fix an element $(O, I) \in \mathcal{R}_{n,m,\sigma}$, i.e., an $O \in \mathcal{O}_{n,\sigma,m}$ and $I \in \mathcal{I}_O$. We now construct an automaton $D = (Q, \Sigma, \delta)$ and then show that $r(D) = (O, I)$. We let $Q = [n]$, $\Sigma = [\sigma]$, and

$$\delta = \{((u, j), v) : O_{u,j} = 1 \text{ and } v = \text{rank}(I, \text{rank}(O, (u, j))) + 1\}.$$

Let $r(D) = (O', I')$ and let us proceed by showing that $O = O'$ and $I = I'$. Recall the definition of r, see Equations (1) and (2). It is immediate that $O' = O$ given the definition of O' and δ. In order to show that $I' = I$, first note that $I' = \prod_{i=2}^{n} 10^{|\delta^{in}(i)|-1}$. Then, consider the following relation between I and $\delta^{in}(i)$ for any state $i \in [n]$, which uses the definition of δ and Lemma 12:

$$\begin{aligned}
|\delta^{in}(i)| &= |\{(u, j) \in [n] \times [\sigma] : O_{u,j} = 1 \text{ and } \text{rank}(I, \text{rank}(O, (u, j))) + 1 = i\}| \\
&= |\{k \in [m] : k = \text{rank}(O, (u, j)) \text{ for some } (u, j) \in [n] \times [\sigma] \text{ with } O_{u,j} = 1 \\
&\qquad\qquad \text{and } \text{rank}(I, k) + 1 = i\}| \\
&= \max\{k \in [m] : \text{rank}(I, k) = i - 1\} - \max\{k \in [m] : \text{rank}(I, k) = i - 2\}.
\end{aligned}$$

Now, recall that $I \in \mathcal{I}_O$, hence the first bit in I is 1. Using the previous equality, it now follows that the second 1-bit in I is at position $|\delta^{in}(2)| + 1$. By using this equality another $n - 3$ times, we obtain that the first $\sum_{i=2}^{n-1} |\delta^{in}(i)| + 1$ positions of I are equal to $(10^{|\delta^{in}(2)|-1}10^{|\delta^{in}(3)|-1} \ldots 10^{|\delta^{in}(n-1)|-1})1$ and thus agree with I'. It remains to observe that this portion of I already contains $n - 1$ bits that are equal to 1 and thus the remaining bits have to be zero-bits as $I \in \mathcal{I}$. Hence, $I = I'$ and this completes the proof. ◀

Injectivity of r via Left-inverse f. In order to establish that f is the inverse of r, it remains to prove that f is a left-inverse of r (which implies that r is injective).

▶ **Lemma 14.** *The function f is a left-inverse of r, i.e., $f(r(D)) = D$ for any $D \in \mathcal{D}_{n,m,\sigma}$.*

Proof. Let $D = (Q, \Sigma, \delta) \in \mathcal{D}_{n,m,\sigma}$ and let $(O, I) = r(D)$. We have to show that $D' = f(O, I)$, i.e., the automaton $D' = (Q', \Sigma', \delta')$ output by Algorithm 4 on input (n, m, σ, O, I) is equal to D. Notice that clearly $Q = Q' = [n]$ and $\Sigma = [\sigma]$. It remains to show that $\delta = \delta'$. It is clear that Algorithm 4 adds m transitions to δ', one in each of the $m = \|O\|$ iterations. It thus remains to prove that each such transition $((u, j), v)$ added in some iteration i is contained in δ. Firstly, as $O_{u,j} = 1$ it is clear that D has an outgoing transition at state u with letter j, second it is clear that the algorithm maintains the property that $v = \text{rank}(I, i) + 1$ and thus due to Lemma 12 it holds that $\delta(u, j) = v$ and thus this transition is also contained in δ. ◀

We can thus denote the function f with r^{-1}.

▶ **Corollary 15.** *Function $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is bijective.*

The above lemma has several consequences. First, it shows that the output of Algorithm 4 is always a WDFA. Second, as the function r is bijective, this means that generating uniform pairs from the range of r results in a uniform distribution of WDFAs from $\mathcal{D}_{n,m,\sigma}$.

▶ **Lemma 16.** *Algorithm 1 on input n, m, σ generates uniformly distributed WDFAs from $\mathcal{D}_{n,m,\sigma}$.*

Proof. In the light of r being a bijection and Algorithm 4 implementing the function r^{-1}, it remains to argue that the statements $O :=$ `sample_O`(n, m, σ) and $I :=$ `sample_I`(O) from Algorithm 1 in fact generate uniformly distributed pairs from the domain of r^{-1}, i.e., from $\mathcal{R}_{n,m,\sigma}$. It is clear that `sample_O`(n, m, σ) results in a uniformly distributed element O from $\mathcal{O}_{n,\sigma,m}$ and that `sample_I`(O) results in a uniformly distributed element I from $\mathcal{I}_O$. It thus remains to observe that $|\mathcal{I}_O|$ is identical for all $O \in \mathcal{O}_{n,\sigma,m}$, namely $|\mathcal{I}_O| = \binom{m-\sigma}{n-\sigma-1}$ for all $O \in \mathcal{O}_{n,\sigma,m}$. This completes the proof. ◀

4.2 Run-time and Space

We now analyze the number of iterations of Algorithm 2, that is, the expected number of rejections before extracting a bit-matrix O with $\|O_j\| > 0$ for all $j \in [\sigma]$. Algorithm 5 is clearly equivalent to Algorithm 1 also under this aspect, since at Lines 12 and 24 we re-start the algorithm whenever we generate a column O_j without non-zero entries. We prove:

▶ **Lemma 17.** *Assume that $m \geq \sigma \ln(e \cdot \sigma)$. The expected number of iterations of Algorithm 2 (equivalently, rejections of Algorithm 5) is at most 1.6. Furthermore, the algorithm terminates after $O(\log m)$ iterations with probability at least $1 - m^{-c}$ for any constant $c > 0$.*

We refer the reader to the full version of this article [4] for the proof of the above lemma. Now assume that $\sigma \leq m/\ln m$. This implies that $e \cdot \sigma \leq m$ (for m larger than a constant), which together with the initial assumption implies that $\sigma \ln(e \cdot \sigma) \leq \sigma \ln m \leq m$. This is exactly the condition in Lemma 17. Hence, if $\sigma \leq m/\ln m$ then the expected number of rejections of Algorithm 5 is $O(1)$ (or $O(\log m)$ with high probability). Our main Theorem 1 follows from the fact that the sequential shuffler of [17] uses constant space, its functions `pop`() and `empty`() run in constant time, and the while loop at Line 8 of Algorithm 5 runs for at most m iterations (less only in case of rejection) every time Algorithm 5 is executed.

5 Counting Wheeler DFAs

In this section, we use the WDFA characterization of Section 3.1 to give an exact formula for the number $|\mathcal{D}_{n,m,\sigma}|$ of WDFAs with n nodes and m edges on effective alphabet $[\sigma]$ with Wheeler order $1 < 2 < \cdots < n$. All proofs of this section can be found in the full version of the article [4]. From our previous results, all we need to do is to compute the cardinalities of $\mathcal{O}_{n,m,\sigma}$ and $\mathcal{I}_O$.

▶ **Lemma 18.** $|\mathcal{O}_{n,m,\sigma}| = \sum_{j=0}^{\sigma}(-1)^j\binom{\sigma}{j}\binom{n(\sigma-j)}{m}$.

This lemma is obtained via an inclusion-exclusion argument. From Algorithm 3, it is immediate that $|\mathcal{I}_O| = \binom{m-\sigma}{n-\sigma-1}$ for all $O \in \mathcal{O}_{n,m,\sigma}$ (see also the proof of Lemma 16). Since $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is bijective (Corollary 15), we obtain an exact formula for $|\mathcal{D}_{n,m,\sigma}|$:

▶ **Theorem 19.** *The number $|\mathcal{D}_{n,m,\sigma}|$ of WDFAs with set of nodes $[n]$ and m transitions labeled from the effective alphabet $[\sigma]$, for which $1 < 2 < \cdots < n$ is a Wheeler order is*

$$|\mathcal{D}_{n,m,\sigma}| = \binom{m-\sigma}{n-\sigma-1}\sum_{j=0}^{\sigma}(-1)^j\binom{\sigma}{j}\binom{n(\sigma-j)}{m}.$$

Using similar techniques, in the case where σ is not arbitrarily close to n, i.e., $\sigma \le (1-\varepsilon)\cdot n$ for some constant ε, we moreover obtain a tight formula for the logarithm of the cardinality of $\mathcal{D}_{n,\sigma} = \bigcup_m \mathcal{D}_{n,m,\sigma}$, the set of all Wheeler DFAs with n states over effective alphabet $[\sigma]$ and Wheeler order $1 < 2 < \cdots < n$:

▶ **Theorem 20.** *The following bounds hold:*
1. $\log|\mathcal{D}_{n,\sigma}| \ge n\sigma + (n-\sigma)\log\sigma - (n + \log\sigma)$*, for any n and $\sigma \le n-1$, and*
2. $\log|\mathcal{D}_{n,\sigma}| \le n\sigma + (n-\sigma)\log\sigma + O(n)$*, for any $n \ge 2/\varepsilon$ and $\sigma \le (1-\varepsilon)\cdot n$, where ε is any desired constant such that $\varepsilon \in (0, 1/2]$.*

Note that $\log|\mathcal{D}_{n,\sigma}|$ is the information-theoretic worst-case number of bits necessary (and sufficient) to encode a WDFA from $\mathcal{D}_{n,\sigma}$. Our Theorem 20 states that, up to an additive $\Theta(n)$ number of bits, this value is of $n\sigma + (n-\sigma)\log\sigma$ bits. As a matter of fact, our encoding $r(D) = (O, I)$ of Section 3, opportunely represented using succinct bitvectors [16], achieves this bound up to additive lower-order terms and supports efficient navigation of the transition relation.

6 Implementation

We implemented our uniform WDFA sampler in `C++`.[1] We tested our implementation by generating WDFAs with a broad range of parameters: $n \in \{10^6 \cdot 2^i : i = 0, \ldots, 6\}$, $m \in \{n \cdot 2^i - 1 : i = 0, \ldots, 7\}$ and $\sigma = 128$. To analyze the impact of streaming to disk on the running time, we tested two versions of our code: (1) We stream the resulting WDFA to disk (SSD). (2) We stream the WDFA to a pre-allocated vector residing in internal memory. Note that constant working space is achieved only in case (1). Our experiments were run on a server with Intel(R) Xeon(R) W-2245 CPU @ 3.90GHz with 8 cores, 128 gigabytes of RAM, 512 gigabytes of SSD, running Ubuntu 18.04 LTS 64-bit. Working space was measured with `/usr/bin/time` (Resident set size).

Figure 3 shows the running time of both variants (left: (1) streaming to SSD; right: (2) streaming to RAM). Both versions exhibit a linear running time behavior, albeit with a different multiplicative constant. The algorithm storing the WDFA in internal memory is between 1.2 and 1.7 times faster than the version streaming the WDFA to the disk (the relatively small difference is due to the fact that we used an SSD). We measured a throughput of at least 5.466.897 and 7.525.794 edges per second for the two variants, respectively. In our experiments we never observed a rejection: this is due to the fact that $\sigma \ll m$, making it extremely likely to generate bit-matrices O containing at least one set bit in each column.

[1] Implementation available at `https://github.com/regindex/Wheeler-DFA-generation`.

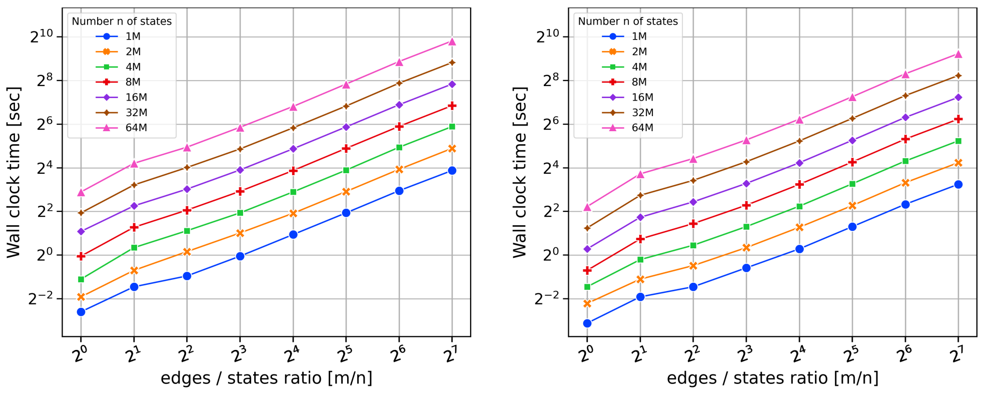


Figure 3 Wall clock time for generating random WDFAs using Algorithm 5. Left: running time for the algorithm in case (1), i.e., streaming the resulting WDFAs to disk. Right: running time in case (2), i.e., storing WDFAs in internal memory.

As far as space usage is concerned, version (1), i.e., streaming the WDFA to disk, always used about 4 MB of internal memory, independently from the input size (this memory is always required to load the `C++` libraries). This confirms the constant space usage of our algorithm, also experimentally. As expected, the space usage of version (2) is linear with the input's size. Nevertheless, both algorithms are extremely fast in practice: in these experiments, the largest automaton consisting of 64 million states and more than 8 billion edges was generated in about 15 and 10 minutes with the first and second variant, respectively.

References

1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular Languages Meet Prefix Sorting. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, pages 911–930, USA, 2020. Society for Industrial and Applied Mathematics.

2 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021. URL: `https://www.sciencedirect.com/science/article/pii/S0890540121001504`.

3 Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. In *2019 Data Compression Conference (DCC)*, pages 122–131. IEEE, 2019.

4 Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Riccardo Maso, and Nicola Prezza. Random wheeler automata. *CoRR*, abs/2307.07267, 2023. `doi:10.48550/arXiv.2307.07267`.

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

6 Kuan-Hao Chao, Pei-Wei Chen, Sanjit A Seshia, and Ben Langmead. WGT: Tools and algorithms for recognizing, visualizing and generating Wheeler graphs. *bioRxiv*, pages 2022–10, 2022.

7 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on wheeler dfas. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. `doi:10.1109/DCC55655.2023.00023`.

8 Giovanna D'Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages and automata: Complexity. *Theoretical Computer Science*, 949:113709, 2023. URL: `https://www.sciencedirect.com/science/article/pii/S0304397523000221`.

9 Lavinia Egidi, Felipe A Louza, and Giovanni Manzini. Space efficient merging of de Bruijn graphs and Wheeler graphs. *Algorithmica*, 84(3):639–669, 2022.

10 Travis Gagie. On Representing the Degree Sequences of Sublogarithmic-Degree Wheeler Graphs. In *String Processing and Information Retrieval: 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8–10, 2022, Proceedings*, pages 250–256. Springer, 2022.

11 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. `doi:10.1016/j.tcs.2017.06.016`.

12 Daniel Gibney and Sharma V Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022.

13 Adrián Goga and Andrej Baláž. Prefix-Free Parsing for Building Large Tunnelled Wheeler Graphs. In *22nd International Workshop on Algorithms in Bioinformatics*, 2022.

14 Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998. URL: `https://www.worldcat.org/oclc/312898417`.

15 Cyril Nicaud. Random Deterministic Automata. In *Proceedings of the 39th International Symposium on Mathematical Foundation of Computer Science (MFCS)*, pages 5–23, 2014.

16 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, USA, 2002. Society for Industrial and Applied Mathematics.

17 Michael Shekelyan and Graham Cormode. Sequential random sampling revisited: Hidden shuffle method. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 3628–3636. PMLR, April 2021. URL: `https://proceedings.mlr.press/v130/shekelyan21a.html`.

Connecting de Bruijn Graphs

Giulia Bernardini
University of Trieste, Trieste, Italy

Huiping Chen
University of Birmingham, Birmingham, UK

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Christoffer Krogh
Technical University of Denmark, Lyngby, Denmark

Grigorios Loukides
King's College London, London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Leen Stougie
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Michelle Sweering
CWI, Amsterdam, The Netherlands

Abstract

We study the problem of making a *de Bruijn graph* (dBG), constructed from a collection of strings, *weakly connected* while minimizing the total cost of edge additions. The input graph is a dBG that can be made weakly connected by adding edges (along with extra nodes if needed) from the underlying complete dBG. The problem arises from genome reconstruction, where the dBG is constructed from a set of sequences generated from a genome sample by a sequencing experiment. Due to sequencing errors, the dBG is never Eulerian in practice and is often not even weakly connected. We show the following results for a dBG $G(V, E)$ of order k consisting of d weakly connected components:

1. Making G weakly connected by adding a set of edges of minimal total cost is NP-hard.
2. No PTAS exists for making G weakly connected by adding a set of edges of minimal total cost (unless the *unique games conjecture* fails). We complement this result by showing that there does exist a polynomial-time $(2 - 2/d)$-approximation algorithm for the problem.
3. We consider a restricted version of the above problem, where we are asked to make G weakly connected by *only adding directed paths between pairs of components*. We show that making G weakly connected by adding $d-1$ such paths of minimal total cost can be done in $\mathcal{O}(k|V|\alpha(|V|) + |E|)$ time, where $\alpha(\cdot)$ is the inverse Ackermann function. This improves on the $\mathcal{O}(k|V|\log(|V|) + |E|)$-time algorithm proposed by Bernardini et al. [CPM 2022] for the same restricted problem.
4. An ILP formulation of polynomial size for making G Eulerian with minimal total cost.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithm, graph algorithm, de Bruijn graph, Eulerian graph

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.6

Funding *Giulia Bernardini*: Supported by the MUR - FSE REACT EU - PON R&I 2014-2020.
Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Solon P. Pissis: Supported by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.
Leen Stougie: Supported by the PANGAIA and ALPACA projects (grant agreements No 872539 and 956229) and by the Netherlands Organisation for Scientific Research (NWO) under projects OCENW.GROOT.2019.015 "Optimization for and with Machine Learning (OPTIMAL)" and Gravitation-grant NETWORKS-024.002.003.
Michelle Sweering: Supported by the Netherlands Organisation for Scientific Research (NWO) under the Gravitation-grant NETWORKS-024.002.003.

1 Introduction

Let us start with some basic definitions and notation following [5]. An *alphabet* Σ is a finite set of elements called *letters*. We consider an integer alphabet $\Sigma = [0, \sigma)$. Let $x = x[0] \dots x[n-1]$ be a *string* of length $n = |x|$ over Σ. By Σ^k we denote the set of all strings of length $k > 0$. For two indices i and $j \geq i$ of x, $x[i \mathinner{.\,.} j]$ is the *fragment* of x starting at position i and ending at position j. The fragment $x[i \mathinner{.\,.} j]$ is an *occurrence* of the underlying *substring* $p = x[i] \dots x[j]$; we say that p occurs (or starts) at *position* i in x. A *prefix* of x is a substring of the form $x[0 \mathinner{.\,.} j]$ and a *suffix* of x is a substring of the form $x[i \mathinner{.\,.} n-1]$. By xy or $x \cdot y$ we denote the *concatenation* of strings x and y: $xy = x[0] \dots x[|x|-1]y[0] \dots y[|y|-1]$. Given strings x and y, a *suffix/prefix overlap* of x and y is a suffix of x that is a prefix of y.

Let S be a collection of strings. The *order-k de Bruijn graph* (dBG) of S is a directed multigraph, denoted by $G_{S,k}(V, E)$, such that V is the set of length-$(k-1)$ substrings of the strings in S and $G_{S,k}$ contains an edge (u, v) with multiplicity $m_{u,v}$ if and only if the string $u[0] \cdot v$ is equal to the string $u \cdot v[k-2]$ and this string occurs exactly $m_{u,v}$ times in total in the strings in S. For instance, suppose that S is generated from a genome sample by a sequencing experiment: then any Eulerian circuit[1] of $G_{S,k}(V, E)$ corresponds to a single genome reconstruction [26, 23]. In this model, due to sequencing errors, $G_{S,k}$ would never be Eulerian in practice [24]; and it would not even be weakly connected. One could, however, try to make $G_{S,k}$ Eulerian by duplicating some of its *existing* edges [22] or introducing *new ones* when the former do not suffice to make $G_{S,k}$ Eulerian [5]. A natural optimization goal in either case would be to minimize the total cost of edge additions.

In this paper, we study the problem of *making any arbitrary $G_{S,k}$ weakly connected* by introducing a set of new edges of minimal total cost (as well as the underlying set of new nodes when they do not exist in $G_{S,k}$). Finding a cheapest way for making $G_{S,k}$ weakly connected is important because one can subsequently apply the linear-time algorithm of Bernardini et al. [5] to balance it by adding a set of edges of minimal total cost, and thus making the graph Eulerian. Since making the dBG *directly* Eulerian by adding a set of new edges of minimal total cost is NP-hard (from the *shortest common superstring* problem [11]), the *connect-and-balance* approach, generally, serves as a good-performing heuristic [5]. Our work falls into a broader line of research that is concerned with algorithmic problems on strings that can be formulated as problems on dBGs [7, 6, 8, 30, 28, 29, 31, 3, 4].

By $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$, we denote the *complete* dBG of order k over alphabet Σ with $|V_{\Sigma,k}| = \sigma^{k-1}$ and $|E_{\Sigma,k}| = \sigma^k$. Any two nodes u and $v \neq u$ in $V_{\Sigma,k}$ can be connected by a *super-edge* whose weight $w_{u,v}$ is in $[1, k)$: this is the shortest path of $w_{u,v}$ unit-cost edges in $G_{\Sigma,k}$. For example, for edge $(aabc, bcac)$ with $aabc, bcac \in V_{\Sigma,k}$ and $k = 5$, we have $w_{aabc,bcac} = 2$ corresponding to the following two unit-cost edges: $aabc \to abca \to bcac$.

We next formally define the main problem in scope; see Figure 1 for an example.

Connecting de Bruijn Graphs with Edges (Connect-DBG-E)
Input: A de Bruijn graph $G(V, E)$ of order k over alphabet $\Sigma = [0, \sigma), \sigma \leq (k-1)|V|$.
Output: A set $A \subseteq E_{\Sigma,k}$ of edges and a set $B \subseteq V_{\Sigma,k}$ of nodes such that $G(V \cup B, E \cup A)$ is weakly connected and A is of minimum size.

Let us remark that Connect-DBG-E allows for connecting two components C_i, C_j of G by a path directed from C_i to C_j but this needs not be the case in general; see Figure 1.

[1] An Eulerian circuit is a graph cycle using each graph edge exactly once. Such a graph is called Eulerian.

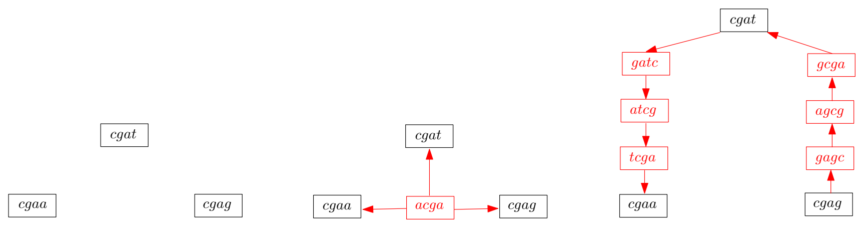


Figure 1 An input dBG of order $k = 5$ with $d = 3$ weakly connected components (left); a solution to Connect-DBG-E with cost 3 (middle); a solution to Connect-DBG-P with cost 8 (right). The Connect-DBG-P problem is a restricted version of the Connect-DBG-E problem allowing to connect the graph *only by means of directed paths* whose endpoints are two components. In fact, the graph on the right also shows an optimal solution to making the graph on the left semi-Eulerian.

We fix throughout a dBG $G(V, E)$ of order k over the integer alphabet $\Sigma = [0, \sigma), \sigma \leq (k-1)|V|$, consisting of d weakly connected components. We show the following results:

1. Connect-DBG-E is NP-hard. We show this via a somewhat surprising and highly non-trivial reduction from the Minimum Vertex Cover problem [15]. See Section 2.
2. No *polynomial-time approximation scheme* (PTAS) exists for Connect-DBG-E unless the *unique games conjecture* [16] fails. We complement this result with a polynomial-time $(2-2/d)$-approximation algorithm for Connect-DBG-E. Our strategy relies on an existing $(2-2/d)$-approximation algorithm for the Minimum Steiner Tree problem [18], where d is the number of terminals of the *Steiner tree.*[2] See Section 3.
3. Making G weakly connected by adding a set of $d-1$ directed paths (between components) of *minimal total cost* can be done in $\mathcal{O}(k|V|\alpha(|V|) + |E|)$ time, where $\alpha(\cdot)$ is the inverse Ackermann function. We call this the Connect-DBG-P problem; see Figure 1 for an example. Our algorithm improves the $\mathcal{O}(k|V|\log(|V|)+|E|)$-time algorithm by Bernardini et al. [5]. We make use of an augmented static version of the Aho-Corasick machine [1] to select the shortest possible paths, while keeping track of the connected components as they are dynamically merged by using a *union-find* data structure [10]. See Section 4.
4. An *integer linear program* (ILP) formulation of polynomial size for making G Eulerian with *minimal total cost.* This is a flow-based formulation inspired by the above relaxation of connecting the d components with $d-1$ paths (Connect-DBG-P). Since the graph must also be balanced (the in- and out-degree for every node is the same), *an optimal solution can always be decomposed into such paths.* We complement our ILP with proof-of-concept experiments on real data showing that problem instances of around 900 nodes and edges can be solved using an off-the-shelf ILP solver within 10 hours. See Section 5.

2 Hardness of Connect-DBG-E

In this section, we prove that Connect-DBG-E is NP-hard via a reduction from Minimum Vertex Cover [15]. Recall that Minimum Vertex Cover asks, given an undirected graph $G(V, E)$, to find a smallest subset C of V such that every edge in E has at least one endpoint in C. In this section, we use the term *vertex* instead of *node* for obvious reasons.

▶ **Theorem 1.** *Connect-DBG-E is NP-hard.*

[2] The Steiner tree of some subset of the nodes of a graph G is a minimum-weight connected subgraph of G that includes all the nodes.

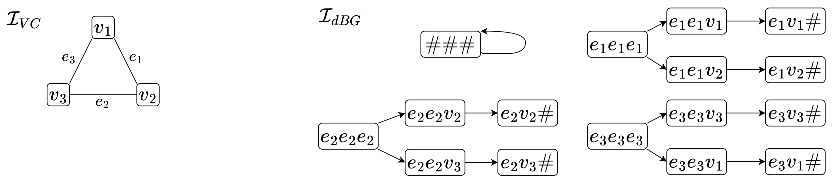


Figure 2 An instance of MINIMUM VERTEX COVER (left) and the instance of CONNECT-DBG-E (right) implied by the reduction of Theorem 1.

Proof. Let $\mathcal{I}_{VC} = G(V, E)$ be an instance of MINIMUM VERTEX COVER. We reduce it to an instance $\mathcal{I}_{dBG}$ of CONNECT-DBG-E, consisting of a dBG $\tilde{G}$ of order 4 over an alphabet Σ of size $\sigma = |V| + |E| + 1$. $\tilde{G}$ consists of $|E|$ edge gadgets, plus a vertex $v_\# = \#\#\#$ and an edge $(v_\#, v_\#)$. The edge gadget for $e_i = (u, v) \in E$ has the following vertices and edges: $V_i = \{v_{1i}, v_{2i}, v_{3i}, v_{4i}, v_{5i}\}$, with $v_{1i} = e_ie_ie_i$; $v_{2i} = e_ie_iu$; $v_{3i} = e_ie_iv$; $v_{4i} = e_iu\#$; $v_{5i} = e_iv\#$; and $E_i = \{(v_{1i}, v_{2i}), (v_{1i}, v_{3i}), (v_{2i}, v_{4i}), (v_{3i}, v_{5i})\}$. We call v_{4i} and v_{5i} the *terminal* vertices of the ith component; the remaining vertices are called *non-terminal*. The reduction requires polynomial time: an example is illustrated in Figure 2. Let $OPT(\mathcal{I}_{VC})$ and $OPT(\mathcal{I}_{dBG})$ denote the size of an optimal solution to $\mathcal{I}_{VC}$ and $\mathcal{I}_{dBG}$, respectively.

▷ Claim 2. A solution to $\mathcal{I}_{VC}$ of size α implies a solution to $\mathcal{I}_{dBG}$ of size $\alpha + |E|$.

Proof. Let C be a cover of G of size α. For each $v \in C$, we add to $\tilde{G}$ a new vertex $v\#\#$; we then connect all the new vertices to $\#\#\#$ using α edges in total. Since C is a vertex cover for G, by construction, one of the two terminal vertices of each edge gadget in $\tilde{G}$ corresponds to a vertex in C and it can thus be connected with a single edge to one of the newly added vertices, using another $|E|$ edges in total. We can thus make $\tilde{G}$ weakly connected by adding α vertices and $\alpha + |E|$ edges. ◁

▷ Claim 3. A solution to $\mathcal{I}_{dBG}$ of size $\beta + |E|$ implies a solution to $\mathcal{I}_{VC}$ of size at most β.

Proof. We observe that any solution to $\mathcal{I}_{dBG}$ must add new vertices, as by construction, no two gadgets can be connected with a single edge, nor can they be connected to $\#\#\#$ with a single edge. Moreover, any solution that adds γ new vertices must add at least $\gamma + |E|$ new edges, as this is the minimum possible number to connect $|E| + \gamma + 1$ components (the $|E|$ edge gadgets, the vertex $\#\#\#$, and the γ new vertices).

We further observe that the only way two distinct edge gadgets can be connected using two edges is by adding an edge from one of the terminal vertices of each gadget to a newly added vertex of the form $v\#\lambda$, where λ is any letter from the alphabet and v is a letter corresponding to a vertex of $\mathcal{I}_{VC}$ that is an endpoint of both the edges corresponding to the two gadgets. This is because any two vertices of two distinct gadgets have no suffix/prefix overlap, thus no path of length two can connect them; and any two vertices of two distinct gadgets have no common prefix, thus there cannot be two edges out of a new vertex that reach two distinct gadgets. On the other hand, the terminal vertices of two distinct gadgets can have the same suffix $v\#$ for some v and thus can be both connected to a vertex of the form $v\#\lambda$ – note that when $\lambda = \#$ these vertices can be, in turn, connected to $\#\#\#$.

Now consider a solution to $\mathcal{I}_{dBG}$ that adds β new vertices. We construct a cover for $\mathcal{I}_{VC}$ as follows. For every newly added vertex $u\#\lambda$ that is adjacent to more than one gadget, add the corresponding vertex u to the cover: this covers all the edges corresponding to the adjacent gadgets. For the edge gadgets that are connected to some new vertex which is not adjacent to any other gadget, add one of the endpoints of the corresponding edge of E to the vertex cover: this covers the remaining edges of $\mathcal{I}_{VC}$. The cover is thus of size at most β. ◁

Let us now prove that $OPT(\mathcal{I}_{VC}) = \ell \iff OPT(\mathcal{I}_{dBG}) = |E| + \ell$.

$\Rightarrow$) By Claim 2, an optimal solution to $\mathcal{I}_{VC}$ of size ℓ implies a solution to $\mathcal{I}_{dBG}$ of size $\ell + |E|$. Suppose for a contradiction that this solution is not optimal, i.e., there exists another solution to $\mathcal{I}_{dBG}$ of size $\ell' + |E|$ with $\ell' < \ell$ new vertices. By Claim 3, this would imply a cover for $\mathcal{I}_{VC}$ of size at most $\ell' < \ell$, a contradiction.

$\Leftarrow$) By Claim 3, an optimal solution to $\mathcal{I}_{dBG}$ of size $|E| + \ell$ implies a solution to $\mathcal{I}_{VC}$ of size at most ℓ. Suppose for a contradiction that $OPT(\mathcal{I}_{VC}) = \ell' < \ell$: by Claim 2, this would imply a solution to $\mathcal{I}_{dBG}$ of size $\ell' + |E|$, a contradiction. ◀

The above reduction is not approximation preserving (because $OPT(\mathcal{I}_{VC}) = \ell \iff OPT(\mathcal{I}_{dBG}) = |E| + \ell$), which would have allowed us to directly obtain a constant-factor approximation algorithm for Connect-DBG-E from Minimum Vertex Cover [14], and to prove its inapproximability from the inapproximability of Minimum Vertex Cover [25].

3 Approximating Connect-DBG-E

We start by proving that the existence of a PTAS for Connect-DBG-E is excluded under the *unique games conjecture* [16]. To achieve this, we restrict to a specific class of graphs.

▶ **Theorem 4.** *There exists no PTAS for Connect-DBG-E unless the unique games conjecture fails.*

Proof. Consider the same reduction as in the proof of Theorem 1. The sizes of the solutions to the two problem instances $\mathcal{I}_{VC}$ and $\mathcal{I}_{dBG}$ always differ by a term of exactly $|E|$, implying that the reduction preserves the inapproximability of Connect-DBG-E in the case where the size of the minimum vertex cover is $\Omega(|E|)$. Indeed, suppose for a contradiction that there exists a PTAS for Connect-DBG-E. Then given any instance $\mathcal{I}_{VC}$, we could obtain a solution of size d by reducing it to $\mathcal{I}_{dBG}$, running the PTAS, and subtracting $|E|$ from the result. Let $d_{OPT} + |E|$ denote the size of an optimal solution to $\mathcal{I}_{dBG}$ (thus d_{OPT} is the size of an optimal solution to $\mathcal{I}_{VC}$), and $d + |E|$ the solution returned by the PTAS. We have that $d + |E| \leq (1+\epsilon)(d_{OPT} + |E|)$, for some input parameter $\epsilon > 0$, and thus

$$d \leq (1+\epsilon)(d_{OPT} + |E|) - |E| = (1+\epsilon)d_{OPT} + \epsilon|E|. \tag{1}$$

When $d_{OPT} = \Omega(|E|)$, let $c > 1$ be a constant such that $\frac{|E|}{c} \leq d_{OPT} \leq |E|$ (as the size of any vertex cover is bounded by $|E|$). From Equation 1, we obtain that $d \leq (1 + (1+c)\epsilon)d_{OPT}$, which contradicts the inapproximability of Minimum Vertex Cover. An example of graphs for which the size of the minimum vertex cover is $\Omega(|E|)$ are bounded-degree graphs: indeed, they have at most $|V| \cdot \Delta/2$ edges and a minimum vertex cover of size at least $|V|/(\Delta+1) = \Omega(|E|)$, where Δ is the bounded maximum degree. Minimum Vertex Cover is hard to approximate (unless the unique games conjecture fails) on bounded degree graphs to within a factor $2 - (2 + o_\Delta(1))\frac{\log\log\Delta}{\log\Delta}$ for a sufficiently large integer Δ [2].

This implies that there is no PTAS for Connect-DBG-E (conditioned on the unique games conjecture) when restricted to the very specific instances obtained, via the reduction of Theorem 1, from instances of bounded-degree Minimum Vertex Cover. We can thus conclude that, in general, there exists no PTAS for Connect-DBG-E conditioned on the unique games conjecture. ◀

Motivated by Theorem 4, we next present a $(2 - 2/d)$-approximation algorithm for Connect-DBG-E. Our strategy relies on an existing $(2 - 2/d)$-approximation algorithm for the Minimum Steiner Tree problem, where d is the number of terminals. Recall that Minimum Steiner Tree asks, given a graph $G'(V', E')$ with non-negative edge weights and a subset of *terminal* nodes, to compute a tree of minimum weight that contains all terminals.

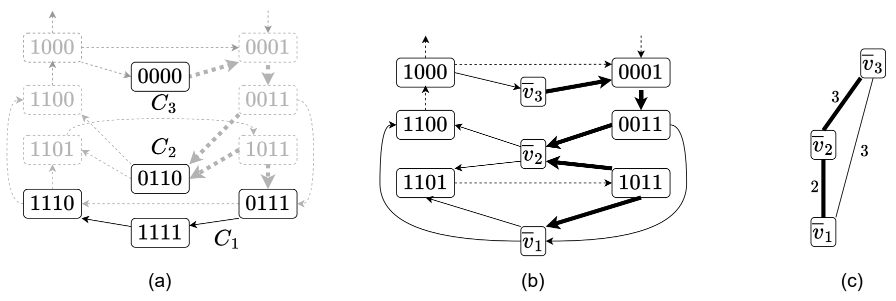


Figure 3 Construction of Theorem 6. The input G consists of 3 weakly connected components shown in **(a)** with black solid lines; grey dashed lines represent nodes and edges of the underlying complete dBG (only the portion directly connected to nodes of G is depicted); grey thick dashed edges form a solution to Connect-DBG-E, which in this case would be returned by the approximation algorithm. G' is shown in **(b)**: solid edges are in $\overline{E}$, thick edges represent the same solution as in **(a)**. The metric closure of G' is shown in **(c)**: thick edges represent the same solution as in **(a)**.

Given a dBG $G(V, E)$ of order k consisting of d weakly connected components $C_1, \ldots, C_d$, let $G'(V', E')$ be the graph obtained from the complete dBG $G_{\Sigma,k}$ collapsing each component C_i into one super-node $\overline{v}_i$: an example is in Figure 3. More formally, $V' = (V_{\Sigma,k} \setminus V) \cup \overline{V}$, where $\overline{V} = \{\overline{v}_1, \ldots, \overline{v}_d\}$ is a set of unlabeled nodes s.t. $\overline{v}_i \notin V_{\Sigma,k}$ corresponds to C_i for all $i \in [1, d]$; and $E' = (E_{\Sigma,k} \cap ((V_{\Sigma,k} \setminus V) \times (V_{\Sigma,k} \setminus V))) \cup \overline{E}$, where $E_{\Sigma,k} \cap ((V_{\Sigma,k} \setminus V) \times (V_{\Sigma,k} \setminus V))$ are simply the edges of the complete dBG connecting pairs of nodes that are both not in G; the edges in $\overline{E}$ are s.t. there is an edge from a super-node $\overline{v}_i$ to a node $v \in (V_{\Sigma,k} \setminus V)$ if and only if at least one of the nodes of C_i would be connected to v by an edge in the complete dBG; and likewise for edges $(v, \overline{v}_i)$. Two super-nodes are connected by an edge if and only if two nodes in the respective components would be connected by an edge in the complete dBG. Formally, $\overline{E} = \overline{E}_1 \cup \overline{E}_2 \cup \overline{E}_3$, where

$$\overline{E}_1 = \{(\overline{v}_i, v) \mid \exists u \in C_i \textit{ and } v \in (V_{\Sigma,k} \setminus V) \textit{ s.t. } (u, v) \in E_{\Sigma,k}\},$$
$$\overline{E}_2 = \{(v, \overline{v}_i) \mid \exists u \in C_i \textit{ and } v \in (V_{\Sigma,k} \setminus V) \textit{ s.t. } (v, u) \in E_{\Sigma,k}\},$$
$$\overline{E}_3 = \{(\overline{v}_i, \overline{v}_j) \mid \exists u \in C_i \textit{ and } v \in C_j \textit{ s.t. } (v, u) \in E_{\Sigma,k}\}.$$

Inspect Figure 3(b): the edge $(1011, \overline{v}_1)$ belongs to set $\overline{E}_2$; the edge $(\overline{v}_3, 0001)$ belongs to $\overline{E}_1$; no edges belong to $\overline{E}_3$ in this example.

It is easy to see that solving Connect-DBG-E for G is equivalent to solving an instance of Minimum Steiner Tree on G' with $\overline{v}_1, \ldots, \overline{v}_d$ as terminals. Any polynomial-time approximation algorithm for Minimum Steiner Tree can therefore be applied to solve Connect-DBG-E with the same approximation ratio. Unfortunately, when applied naively, this strategy does not give a polynomial-time algorithm for Connect-DBG-E, because G' has an exponential size and thus constructing it requires, in general, exponential time.

To overcome this issue, we focus on a specific approximation algorithm for the Minimum Steiner Tree problem which does not require computing the whole graph G' but rather only its *metric closure*, defined as a weighted complete graph on the set of terminals $\overline{v}_1, \ldots, \overline{v}_d$ such that the weight on edge $(\overline{v}_i, \overline{v}_j)$ is the length of the shortest *undirected* path between $\overline{v}_i$ and $\overline{v}_j$ in G'. An example of the metric closure of G' is in Figure 3**(c)**. Note, in particular, that the length of the shortest undirected path between two nodes (i.e., a sequence of edges that form a path if their direction is ignored) is smaller or equal to the length of the shortest *directed* path: for instance, the shortest undirected path between 0110 and 0111 in Figure 3 is of length 2 (through node 1011), while the shortest directed path is of length 3 (through nodes 1101 and 1011). In contrast to explicitly constructing the whole G', computing only its metric closure can be done in polynomial time, as stated by the following lemma.

▶ **Lemma 5.** *For any dBG $G(V,E)$ of order k, computing the metric closure of G' can be done in $\mathcal{O}(k|V|^2)$ time.*

Proof. Let G consist of the weakly connected components $C_1,\dots,C_d$. By the definition of G', computing its metric closure requires computing the length of the shortest undirected path in $G_{\Sigma,k}$ between any pair of nodes that lie in two different components of G. An algorithm to compute shortest undirected paths in dBGs in $\mathcal{O}(k)$ time per pair has been proposed in [20]: this algorithm only relies on computing common substrings for each pair of nodes and it does not require to construct G'.

The weight of an edge $(\overline{v}_i,\overline{v}_j)$ in the metric closure of G' is thus obtained by computing the length of the shortest undirected path between every pair of nodes $u \in C_i$, $v \in C_j$ and taking the minimum over such values. Over all edges $(\overline{v}_i,\overline{v}_j)$, this requires time $\mathcal{O}(k\sum_{i,j\in[1,d]}|C_i||C_j|) = \mathcal{O}(k|V|^2)$. ◀

▶ **Theorem 6.** *For any dBG $G(V,E)$ of order k consisting of d weakly connected components, there exists an $\mathcal{O}(k|V|^2)$-time $(2-2/d)$-approximation algorithm for* Connect-DBG-E*.*

Proof. The algorithm, which is an adaptation of [18] to dBGs, consists of three steps:
- **(i)** Construct the metric closure of G'.
- **(ii)** Compute a minimum spanning tree of the metric closure.
- **(iii)** Convert the minimum spanning tree into a set of nodes and a set of edges to be added to G to make it weakly connected.

The correctness follows directly from the fact that a minimum spanning tree for the metric closure of G' is a $(2-2/d)$-approximation for the minimum Steiner tree [18], where d is the number of terminals and thus the number of weakly connected components of G.

Step (i) requires $\mathcal{O}(k|V|^2)$ time as per Lemma 5. Step (ii) can be done in $\mathcal{O}(d^2)$ time by applying, e.g., Prim's algorithm [27]. Finally, Step (iii) can be done by applying again the algorithm from [20] to compute the shortest undirected path between every pair $\overline{v}_i,\overline{v}_j$ such that the edge $(\overline{v}_i,\overline{v}_j)$ is in the minimum spanning tree of the metric closure of G' and taking the union of the nodes and edges in such paths. This requires $\mathcal{O}(k|V|^2)$ total time. ◀

4 Connecting de Bruijn Graphs with Paths in Essentially Optimal Time

In this section, we present an exact algorithm for a restricted version of Connect-DBG-E, in which we are asked to make a dBG $G(V,E)$ of order k weakly connected by adding a set of *directed paths* (between components) of minimum total length. This problem was already considered and solved in (nearly optimal) polynomial time in [5]; here we propose a much simpler and essentially time-optimal algorithm. To formally define the restricted problem we consider, we first need the following definition of a *condensed graph* of a dBG from [5].

▶ **Definition 7** (Condensed Graph)**.** *Given a dBG $G(V,E)$ of order k over an alphabet Σ with a set $\mathcal{C}$ of weakly connected components, its* condensed graph $\widehat{G}(\widehat{V},\widehat{E})$ *is a weighted directed multigraph whose nodes $\widehat{V}$ are in a bijection with $\mathcal{C}$. The edges have integer weights in $[1,k)$: there is an edge $(i,j)\in\widehat{E}$ for each pair of nodes $u_i \in C_i$, $u_j \in C_j$, with $C_i, C_j \in \mathcal{C}$, and its weight is the length of the shortest path from u_i to u_j in the complete dBG $G_{\Sigma,k}$.*

Connecting de Bruijn Graphs with Paths (Connect-DBG-P)
Input: A de Bruijn graph $G(V,E)$ of order k over alphabet $\Sigma = [0,\sigma), \sigma \le (k-1)|V|$.
Output: A minimum-weight spanning tree $\mathcal{T}$ of the condensed graph $\widehat{G}$ of G.

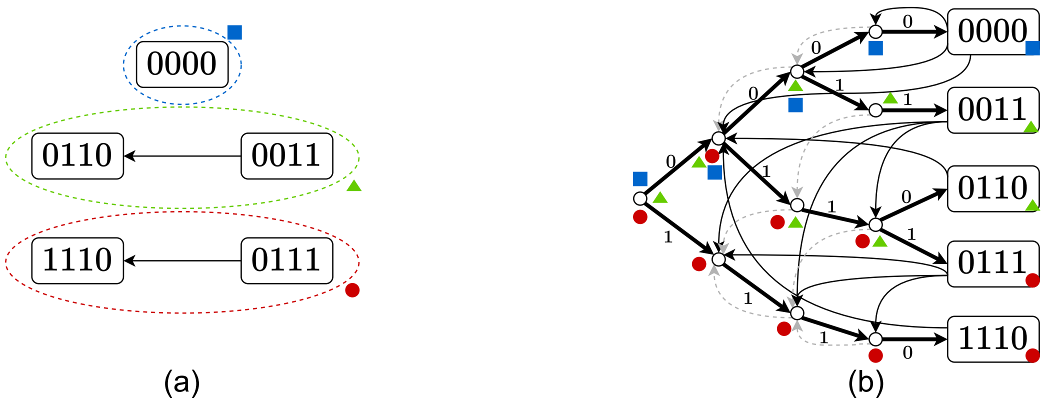


Figure 4 **(a)** An instance of CONNECT-DBG-P consisting of 3 components. **(b)** The modified AC machine built in the preprocessing phase of Algorithm 1. Dashed arrows are the backward edges of the original AC machine; solid, curved arrows are the backward edges of the modified AC machine (backward edges to the root are omitted). Symbols close to the states represent their lists of colors.

A solution $\mathcal{T}$ to CONNECT-DBG-P naturally corresponds to a set $\mathcal{P}$ of paths on $G_{\Sigma,k}$ that make G weakly connected: an edge (i, j) of $\mathcal{T}$ corresponds to the shortest path from some node $u_i \in C_i$ to some $u_j \in C_j$, and in turn, by the definition of dBG, such a path is determined by the longest suffix/prefix overlap of u_i and u_j.

The algorithm for CONNECT-DBG-P proposed in [5] makes use of a dynamic version of the Aho-Chorasick (AC) machine of the nodes of G to find the shortest connecting paths and to keep track of the connected components as they are progressively united by these paths. Here we will use an augmented but static version of the same AC machine to select the paths, and we will keep track of the connected components as they are dynamically merged by employing a union-find data structure.

Before describing our solution, let us recall that AC machines generalize the Knuth-Morris-Pratt [17] algorithm for a set of strings. Informally, AC machines are finite-state machines that resemble a trie with additional *backward edges* (also called *failure transitions*) between the states. There is exactly one failure transition $f(u) = v$ from each state u (except for the root state) to some state v. Backward edges encode suffix/prefix overlaps between the strings represented by the AC machine, as specified by the following lemma.

▶ **Lemma 8** (Aho-Corasick lemma [1]). *Let u and v be two strings representing two distinct states of the AC machine, and identify the states with such strings. Then, $f(u) = v$ if and only if v is the longest proper suffix of u that is also a prefix of some string in the machine.*

In a preprocessing step, we compute the d weakly connected components of G, choose a representative node for each component, and assign it a unique color: we will identify each color with the connected component and with the representative node it is associated with. To store the weakly connected components of G, we construct a union-find data structure [10]. Union-find data structures allow to efficiently perform any sequence of operations of the following two kinds on disjoint sets: $\mathsf{union}(A, B)$ merges sets A and B; and $\mathsf{find}(x)$ returns the representative element of the unique set containing x.

We then construct the AC machine of the nodes of G and preprocess it as follows; see Figure 4 for an example. We color each terminal state with the color of the connected component of the node of G it represents. Each internal state is assigned the union of the colors of its descendants. From each terminal state s, we follow the unique path of backward edges to the root and, for each state u on this path, we add to the machine a backward edge (s, u). We finally prune all the backward edges connecting two non-terminal states.

Once we are done with the preprocessing phase, we start performing a reverse BFS of the machine (which begins from the deepest internal states and proceeds level-by-level towards the root) and check whether the overlap encoded by the backward edges incoming to each of the visited states can be used to unite some components. This is because the deeper the state u reached by a backward edge (s, u), the longer the overlap encoded by the edge; and the longer the overlap, the shorter the path connecting s with all the nodes represented by the terminal states below u. The idea is to greedily select the backward edges encoding paths that connect two currently separate components, using the union-find data structure both to check which components are still separate and to unite them when we select a shortest path.

Implementation Details. We associate two lists to each state u: one for the colors; and one for the incoming backward edges. The colors of u are stored in a list LC_u of ordered pairs $< c, p_c >$, where c is a distinct color and p_c is a pointer to any terminal state of color c below u. The backward edges incoming to u are stored using a list LE_u of their tails (recall that all the tails are terminal states). We will need to keep track of the states of the AC machine visited during the execution of the algorithm, therefore we set up a visited/unvisited flag for each internal state, initially set to "unvisited".

When we visit a state u for the first time, we select the first backward edge (s, u) of the list LE_u (if any). Let c be the color of the terminal state s. For each color α in the list LC_u, we compare the representative of the current connected component of the node associated with color c and the representative of the current connected component of the node associated with α, that is, we compare the results of operations find(c) and $\mathsf{find}(\alpha)$. If they differ, it means that the components of the two nodes are still separate, thus we can unite them by adding the path linking s to the node pointed by p_α, and keep track of the fact that they now constitute a single connected component by performing $\mathsf{union}(c, \alpha)$ (recall that we identify colors and connected components). If $\mathsf{find}(c) = \mathsf{find}(\alpha)$, then the two components were united in a previous step, thus we just move on to the next color in LC_u. At the end of the scan of LC_u, c and all the colors of u will represent the same connected component.

We then select each subsequent backward edge in LE_u, and we compare just the color of its tail and the first color in LC_u, again by performing two find operations. We merge the two components and add the appropriate path if they differ, or move on to the next backward edge in LE_u (or to the next state, if LE_u is exhausted) if they are the same.

The whole procedure is summarized in Algorithm 1.

▶ **Theorem 9.** *Algorithm 1 solves* Connect-DBG-P *in* $\mathcal{O}(k|V|\alpha(|V|) + |E|)$ *time.*

Proof.

Correctness. Algorithm 1 is essentially Kruskal's algorithm [19] applied to the condensed graph $\hat{G}$. Indeed, the longest suffix/prefix overlap between any two nodes u_1, u_2 (which determines the weight of the corresponding edge in $\hat{G}$) is encoded in the AC machine by a path of backward edges starting from u_1 and ending at an ancestor of u_2 [33, Theorem 4]. Thus, by construction, the backward edges we add to the AC machine encode the edges of $\hat{G}$ (a single backward edge may correspond to multiple edges of $\hat{G}$), plus some shorter suffix/prefix overlaps that are discarded: they have no correspondence in $\hat{G}$. In particular, a backward edge (s, u) of the modified AC machine encodes overlaps of length $d(u)$, where $d(u)$ is the depth of state u, i.e., the length of the string it represents [33, Lemma 3].

Algorithm 1 always returns a feasible solution. Indeed, every time a state u is visited, all the components (i.e., nodes from $\hat{V}$) descending from u are connected (Lines 9-11); since all the components descend from the root, at the end the whole $\hat{G}$ is connected; and the union

Algorithm 1 Connect dBG with Paths.

```
1: Compute the $d$ weakly connected components of $G(V, E)$ and the union-find data structure over
   the components; identify each component with a distinct color and each color with a node of
   that component. Construct and preprocess the AC machine of $V$.
2: $\mathcal{P} \leftarrow \emptyset$; comp-count $\leftarrow d$;
3: while comp-count $> 1$ do
4:   $u \leftarrow$ next state of the AC machine in a reverse BFS order;
5:   for all $(s, u)$ in $\mathrm{LE}_u$ do
6:     $c \leftarrow$ color($s$);
7:     if $u$ is unvisited then
8:       Flag $u$ as visited;
9:       for all $< \alpha, p_\alpha >$ in $\mathrm{LC}_u$ do
10:        if find($c$) $\neq$ find($\alpha$) then
11:          union($c, \alpha$); comp-count $\leftarrow$ comp-count $- 1$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{$path from $s$ to $p_\alpha\}$;
12:    else
13:      $< \alpha, p_\alpha > \leftarrow$ first element of $\mathrm{LC}_u$;
14:      if find($c$) $\neq$ find($\alpha$) then
15:        union($c, \alpha$); comp-count $\leftarrow$ comp-count $- 1$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{$path from $s$ to $p_\alpha\}$;
16: return $\mathcal{P}$
```

and find operations ensure that no loop is created. Moreover, the algorithm only returns paths corresponding to backward edges that encode maximal suffix/prefix overlaps, thus edges of $\hat{G}$. Suppose for a contradiction that the algorithm uses an edge (s, u) to connect s with a descendant s' of u using an overlap of length $d(u)$, while the longest overlap between s and s' is of length $\ell > d(u)$. Then, by construction, there is a lower ancestor v of s' with $d(v) = \ell$ and another backward edge (s, v). Since the states are visited in a reverse BST order, v is visited before u, (s, v) is considered before (s, u) and it is used to connect s to s', thus uniting their components; when u is visited afterwards and (s, u) is considered, s and s' are already in the same component, so the shorter overlap is discarded, a contradiction.

Finally, optimality follows directly from the correctness of Kruskal's algorithm [19].

Complexity. Computing the connected components of G and assigning each a color $c \in [1, d]$ requires $\mathcal{O}(|V| + |E|)$ time, where $|E|$ is the number of *distinct* edges of G [13]. Building the AC machine of V takes $\mathcal{O}(k|V|)$ time because each string is of length $k - 1$ [1, 9]. Initializing a union-find data structure for the weakly connected components of G requires $\mathcal{O}(|V|)$ time [10].

During the execution of the algorithm, we perform exactly $d - 1 < |V|$ union operations; moreover, at each visited state u, we perform a number of find operations proportional to the sum of the number of colors of u and the number of backward edges incoming to u. The total size of lists LE_u and LC_u over all non-terminal states u is bounded by $\mathcal{O}(k|V|)$, because the color of each of the $|V|$ terminal states propagates to at most $k - 2$ non-terminal states (the depth of the AC machine is $k - 1$), and by construction, there are up to $k - 2$ backward edges from each terminal state.

Since the cost of each find and union operation amortizes to $\mathcal{O}(\alpha(|V|))$ [10], the total cost of this procedure is $\mathcal{O}(\alpha(|V|)(k|V| + d - 1)) = \mathcal{O}(k|V|\alpha(|V|))$. Since the preprocessing phase requires $\mathcal{O}(k|V| + |E|)$ time, the statement follows. ◀

5 Making a dBG Eulerian through ILP

Let us recall some basic definitions. An *Eulerian trail* is a trail in a finite graph that visits every edge exactly once allowing for revisiting nodes. An *Eulerian circuit* is an Eulerian trail that starts and ends on the same node. A graph with an Eulerian circuit is called *Eulerian*. A graph with an Eulerian trail but with no Eulerian circuit is called *semi-Eulerian*.[3]

Recall that, by $G_{\Sigma,k}(V_{\Sigma,k}, E_{\Sigma,k})$, we denote the complete dBG of order k over alphabet Σ. Here, we present an ILP formulation for making any arbitrary dBG $G(V,E)$ Eulerian (or semi-Eulerian) by adding a multiset of edges from $E_{\Sigma,k}$; this problem is NP-hard via a simple reduction from the shortest common superstring problem [11]. Instead of explicitly adding nodes, we assume that any two nodes can be connected with a (super-)edge whose cost is in $[1,k)$, and try to make G Eulerian by adding edges with a minimum total cost.

5.1 The ILP Formulation

Let $\mathcal{E}$ be the set of edges (u,v) between nodes $u,v \in V$ for which there is a path from u to v in $G_{\Sigma,k}$. This is possible for every pair of nodes in V. We define the set $\mathcal{V}^-(u)$ of in-neighbors of node $u \in V$ as $\mathcal{V}^-(u) = \{v \in V \mid (v,u) \in \mathcal{E}\}$. Similarly, we define the set $\mathcal{V}^+(u)$ of out-neighbors of node $u \in V$ as $\mathcal{V}^+(u) = \{v \in V \mid (u,v) \in \mathcal{E}\}$. We also define a weight function $W : \mathcal{E} \to \mathbb{Z}_{\geq 0}$, which assigns a weight $w_{u,v}$ to an edge $(u,v) \in \mathcal{E}$ equal to the length of the shortest directed path from u to v in $G_{\Sigma,k}$. In particular, we have $w_{u,v} = k - 1 - \mathsf{MO}(u,v)$, where k is the order of G and $\mathsf{MO}(u,v)$ is the length of the longest overlap between a suffix of u and a prefix of v. For example, for edge $(aabc, bcac) \in \mathcal{E}$ with $aabc, bcac \in V$ and $k = 5$, we have $w_{aabc,bcac} = k - 1 - |bc| = 5 - 1 - 2 = 2$, corresponding to the following two unit-cost edges: $aabc \to abca \to bcac$. All $\Theta(|V|^2)$ weights of the $|V|$ nodes can be precomputed in the optimal $\mathcal{O}(k|V| + |V|^2)$ time [12, 21]. By Euler's theorem, making G Eulerian (resp. semi-Eulerian) reduces to finding a minimum weight multiset of edges $\mathcal{A}'$ from $\mathcal{E}$ such that $G' = (V, E \cup \mathcal{A}')$ is weakly connected and balanced (resp. semi-balanced).

To compute multiset $\mathcal{A}'$, we employ the ILP formulation presented in Figure 5. Each edge (u,v) is associated to a non-negative integer variable $a_{u,v}$ whose value corresponds to the increase of the multiplicity $m_{u,v}$ of (u,v) (where $m_{u,v} = 0$ for any $(u,v) \in \mathcal{E} \setminus E$). Thus, $a_{u,v} + m_{u,v}$ denotes the actual multiplicity of (u,v) in G'. Each node v is associated to binary variables x_v and y_v which determine if v is a source and/or a target node in an Eulerian trail of G', respectively. Specifically, if $x_v = 1, y_v = 0$, then v is a source node and not a target node; if $x_v = 0, y_v = 1$ then v is a target node but not a source; and if $x_v = y_v = 0$ then v is either (i) both a source and a target node in an Eulerian trail; or (ii) neither a source nor a target node in an Eulerian trail. Due to the constraint in Equation (2e), it cannot be that $x_v = y_v = 1$, and due to the constraints in Equation (2c) and Equation (2d) there is up to one source and one target node in G'.

Since the existence of a single component in G' is necessary for G' to be Eulerian or semi-Eulerian, we introduce a non-negative integer variable $b_{u,v}$ for each edge $(u,v) \in \mathcal{E}$ to check the connectivity of G', through constraints that will be explained in detail later. Let us provide the main idea behind modeling connectedness. Suppose there are $d = r + 1$ components in G. We select one node from each component of G arbitrarily, such that we have a set $\mathcal{S} = \{s_1, \ldots, s_r\}$ of start nodes and one destination node which we denote by dn. Let C_i be the component containing s_i, where $i \in [1, r]$, and C_{dn} be the component

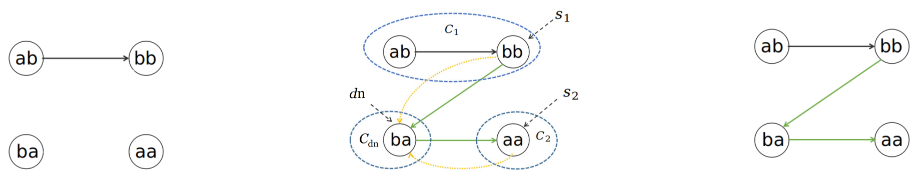
[3] Note that both Eulerian and semi-Eulerian graphs are required to be weakly connected.

$$
\begin{array}{llrr}
\text{minimize} & \sum_{(u,v)\in\mathcal{E}} a_{u,v}\cdot w_{u,v} & & \text{(2a)}\\
\text{subject to} & \displaystyle\sum_{u\in\mathcal{V}^-(v)} (a_{u,v}+m_{u,v}) - \sum_{u\in\mathcal{V}^+(v)} (a_{v,u}+m_{v,u}) + x_v - y_v = 0, \quad v\in V & & \text{(2b)}\\
& \displaystyle 0 \le \sum_{v\in V} x_v \le 1, & & \text{(2c)}\\
& \displaystyle \sum_{v\in V} x_v = \sum_{v\in V} y_v, & & \text{(2d)}\\
& x_v + y_v \le 1, & v\in V & \text{(2e)}\\
& b_{u,v} \le r\cdot(a_{u,v}+m_{u,v}+a_{v,u}+m_{v,u}), & (u,v)\in\mathcal{E} & \text{(2f)}\\
& \displaystyle \sum_{v\in\mathcal{V}^+(s_i)} b_{s_i,v} - \sum_{u\in\mathcal{V}^-(s_i)} b_{u,s_i} = 1 & \forall i\in[1,r] & \text{(2g)}\\
& \displaystyle \sum_{u\in\mathcal{V}^-(dn)} b_{u,dn} - \sum_{v\in\mathcal{V}^+(dn)} b_{dn,v} = r & & \text{(2h)}\\
& \displaystyle \sum_{u\in\mathcal{V}^-(v)} b_{u,v} = \sum_{u\in\mathcal{V}^+(v)} b_{v,u} & v\in V\setminus(\mathcal{S}\cup\{dn\}) & \text{(2i)}\\
& b_{u,v}\in\mathbb{Z}_{\ge 0}, & (u,v)\in\mathcal{E} & \text{(2j)}\\
& a_{u,v}\in\mathbb{Z}_{\ge 0}, & (u,v)\in\mathcal{E} & \text{(2k)}\\
& x_v\in\{0,1\}, & v\in V & \text{(2l)}\\
& y_v\in\{0,1\}, & v\in V & \text{(2m)}
\end{array}
$$

Figure 5 The complete ILP formulation for making a dBG Eulerian or semi-Eulerian.

containing dn. C_i and C_{dn} are connected if there exists a (positive) flow from s_i to dn. Assume that each connection between C_i and C_{dn} provides one unit of flow. There are r start nodes in G, so dn must absorb r units of flow in total from all start nodes.

Equation (2a) seeks to minimize the cost of multiset $\mathcal{A}'$ (the sum of weights for all edges added to G to make it Eulerian or semi-Eulerian). All other equations seek to guarantee that the graph G' is Eulerian or semi-Eulerian by ensuring that all its nodes are balanced (Equation (2b) to Equation (2e)) and that all its nodes with nonzero degree belong to a single strongly connected component (Equation (2f) to Equation (2i)). Let $\delta^-(v)$ and $\delta^+(v)$ denote the in- and out-degree of node v, respectively. Recall that a weakly connected graph is Eulerian if $\delta^-(v)=\delta^+(v)$ for each $v\in V$, and semi-Eulerian if $\delta^-(s)=\delta^+(s)-1$, $\delta^-(t)=\delta^+(t)+1$, and $\delta^-(v)=\delta^+(v)$ for each $v\in V\setminus\{s,t\}$, where s and t are the source and target nodes, respectively. Equation (2b) enforces that G' is Eulerian by requiring $x_v=0, y_v=0$ such that $\delta^-(v)=\delta^+(v)$ for each $v\in V$, or that G' is semi-Eulerian by requiring $x_s=1, y_s=0$ for source node, $x_t=0, y_t=1$ for target node and $x_v=0, y_v=0$ for all other nodes, respectively. Equation (2c), Equation (2d) and Equation (2e) enforce that there exists at most only one source node and one target node in G'. Equation (2f) bounds the value of $b_{u,v}$. In particular, if nodes u and v are not connected in G' (i.e., $a_{u,v}+m_{u,v}+a_{v,u}+m_{v,u}=0$), then Equation (2f) together with Equation (2j) ensure that both $b_{u,v}=0$ and $b_{v,u}=0$; otherwise, $b_{u,v}\ge 0$ and $b_{v,u}\ge 0$. Equation (2g) enforces that each s_i provides one unit of flow; and Equation (2h) enforces that dn absorbs r units of flow from all s_i's together. Last, Equation (2i) enforces that the amount of flow that enters each node that is not in $S\cup\{dn\}$ is equal to the amount of flow that exits this node.



(a) Input graph $G(V, E)$. **(b)** Computing the flow units. **(c)** Output graph G'.

Figure 6 An example of making the dBG on the left semi-Eulerian. The units of flow are depicted with yellow edges and the edges we add to make the graph semi-Eulerian are colored green.

▶ **Example 10.** Consider the subgraph $G(V, E)$ of the complete order-3 dBG shown in Figure 6a, where $V = \{aa, ab, ba, bb\}$ and $E = \{(ab, bb)\}$. By adding edge (bb, ba) and edge (ba, aa) in G, we find an Eulerian trail where the source node is ab and the target node is aa. The weights of the added edges are both 1, i.e., $w_{bb,ba} = w_{ba,aa} = 1$, and the total cost of this solution is 2, which is minimal (since anyway we must connect $d = 3$ connected components).

We show that the solution we found satisfies the constraints of the ILP from Figure 5. For the source node ab, we have $x_{ab} = 1$, $y_{ab} = 0$, and it is semi-balanced since $\delta^-(ab) - \delta^+(ab) + x_{ab} - y_{ab} = 0 - 1 + 1 - 0 = 0$. Similarly, for the target node aa, we have $x_{aa} = 0$, $y_{aa} = 1$, and it is semi-balanced. For all other nodes, $v \in \{bb, ba\}$, $\delta^-(v) = \delta^+(v) = 1$ and $x_v = y_v = 0$. Also, the values of x_v and y_v, $\forall v \in V$, satisfy Equation (2c) to Equation (2e).

Next, we show the connectivity of G'. There are $d = 3$ components in G (namely, C_1, C_2 and C_{dn}). We select one node from each component arbitrarily, such that we have $\mathcal{S} = \{bb, aa\}$ and $dn = ba$. The destination node dn needs to absorb two units of flow from C_1 and C_2. Since $a_{ba,aa} = 1 > 0$, $b_{aa,ba} \geq 0$, there exists a flow starting at node aa and ending at node dn (Equation (2f)). Similarly, node dn absorbs another unit of flow from bb; the two units of flow are represented with yellow lines in Figure 6b. Thus, the output graph G' is semi-Eulerian: the source node is ab and the target node is aa; see Figure 6c.

5.2 Proof-of-concept Experiments

We implemented our ILP formulation in `C++` using Gurobi 9.5.2. We will refer to this algorithm as ILP. Our code is available at `https://bitbucket.org/eulerian-ext/cpm2024/`.

We present proof-of-concept experiments using ILP on small samples of the *Staphylococcus aureus* (STA) whole-genome shotgun benchmark dataset. This dataset is available from `http://gage.cbcb.umd.edu/data/index.html`. The number of reads in the STA dataset is 1, 294, 104 (Library 1), the average read length is 101 base pairs (bp) and the insert length is 180bp. All experiments ran on an AMD EPYC 7702 CPU with 256GB RAM.

We first applied ILP on dBGs of varying order k. We constructed these order-k dBGs, one for each k value, using the first 10 reads of STA. To compute the weight $w_{u,v}$ of each edge given as input to ILP, we used the implementation of [32] for computing $\mathsf{MO}(u, v)$; and then set $w_{u,v} = k - 1 - \mathsf{MO}(u, v)$. Observe in Table 1a that, as expected, increasing k slightly reduced the number of edges $|E|$ of the input dBG and that it also generally increased the number of connected components. As can be seen, making a graph with more connected components Eulerian incurred a larger total cost and required more time. For example, for $k = 8$, there are 2 components extended to an Eulerian graph with a cost of 47 in less than 30 minutes, while for $k = 13$, there are 10 components extended to an Eulerian graph with a cost of 92 in about 6.5 hours. This is because when r increases, the number of distinct possible combinations of values of the variables $b_{u,v}$ increases exponentially with $d = r + 1$,

Table 1 Runtime of ILP on dBGs with varying k and $|E|$ on the STA dataset. Note that the time to compute all constants in ILP is not included in the reported runtimes.

(a) Runtime of ILP on dBGs with varying k constructed from the first 10 reads of the STA dataset.

k	$\|V\|$	$\|E\|$	d	Cost	Time (s)
8	709	724	2	47	1,766
9	720	714	7	56	9,260
10	713	704	9	65	24,378
11	704	694	10	78	26,982
12	694	684	10	87	34,989
13	684	674	10	92	23,119

(b) Runtime of ILP on dBGs with varying number $|E|$ of edges constructed from the first 8, 9, 10, 11, and 12 reads of the STA dataset.

k	$\|V\|$	$\|E\|$	d	Cost	Time (s)
9	562	554	8	43	13,779
9	641	634	7	49	9,205
9	720	714	7	56	11,748
9	798	794	7	61	13,183
9	876	875	6	65	19,751

as each possible combination corresponds to a different weighted spanning tree of the d components; and there are exponentially many possible spanning trees. In other words, there are exponentially many ways to form the sums in Equations 2g, 2h, and 2i.

We then applied ILP on dBGs of fixed order $k = 9$ and varying number $|E|$ of edges. We started with a dBG G_1, constructed as explained before from the first 8 reads of STA with $k = 9$. G_1 corresponds to the first row in Table 1b. Then, we constructed dBGs G_2, G_3, G_4, and G_5, with a larger number of edges than G_1, by adding into G_1 nodes and edges corresponding to the next 1, 2, 3, and 4 reads in STA, respectively. That is, G_2 is constructed from the first 9 reads of STA with $k = 9$. Observe in Table 1b that, as expected, increasing $|E|$ also increases $|V|$ and generally reduces the number of components. As expected, making a graph with more edges Eulerian, while keeping the number of components the same, incurred a larger total cost and required more time. Indeed, the main factor that determines the runtime is the number of components. For example, it took 50% more time to make the dBG in the first row of Table 1b Eulerian compared to the time in the second row of the same table because the former has more components, although it has fewer edges and nodes.

These results show that despite the NP-hardness of the problem, ILP can be used to obtain optimal solutions for small graphs within a reasonable amount of time. These graphs may be specific subgraphs of a much larger graph that need to be made Eulerian.

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 Per Austrin, Subhash Khot, and Muli Safra. Inapproximability of vertex cover and independent set in bounded degree graphs. In *2009 24th Annual IEEE Conference on Computational Complexity*, pages 74–80, 2009. `doi:10.1109/CCC.2009.38`.

3 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe data structures for text indexing. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 199–213. SIAM, 2020. `doi:10.1137/1.9781611976007.16`.

4 Giulia Bernardini, Huiping Chen, Gabriele Fici, Grigorios Loukides, and Solon P. Pissis. Reverse-safe text indexing. *ACM J. Exp. Algorithmics*, 26:1.10:1–1.10:26, 2021. `doi:10.1145/3461698`.

5 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Making de Bruijn graphs Eulerian. In *33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CPM.2022.12`.

6 Giulia Bernardini, Alessio Conte, Estéban Gabory, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, Giulia Punzi, and Michelle Sweering. On strings having the same length-k substrings. In *33rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 223 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.CPM.2022.16`.

7 Giulia Bernardini, Alberto Marchetti-Spaccamela, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Constructing strings avoiding forbidden substrings. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 191 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.CPM.2021.9`.

8 Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome biology*, 22:1–24, 2021. `doi:10.1186/s13059-021-02297-z`.

9 Shiri Dori and Gad M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006. `doi:10.1016/j.ipl.2005.11.019`.

10 Zvi Galil and Giuseppe F Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991. `doi:10.1145/116873.116878`.

11 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. `doi:10.1016/0022-0000(80)90004-5`.

12 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992. `doi:10.1016/0020-0190(92)90176-V`.

13 John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973. `doi:10.1145/362248.362272`.

14 George Karakostas. A better approximation ratio for the vertex cover problem. *ACM Trans. Algorithms*, 5(4):41:1–41:8, 2009. `doi:10.1145/1597036.1597045`.

15 Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computation*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. `doi:10.1007/978-1-4684-2001-2_9`.

16 Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 767–775. ACM, 2002. `doi:10.1145/509907.510017`.

17 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`.

18 Lawrence T. Kou, George Markowsky, and Leonard Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981. `doi:10.1007/BF00288961`.

19 Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. `doi:10.1090/S0002-9939-1956-0078686-7`.

20 Zhen Liu. Optimal routing in the De Bruijn networks. Research Report RR-1130, INRIA, 1990. URL: `https://hal.inria.fr/inria-00075429`.

21 Grigorios Loukides and Solon P. Pissis. All-pairs suffix/prefix in optimal time using Aho-Corasick space. *Inf. Process. Lett.*, 178:106275, 2022. `doi:10.1016/J.IPL.2022.106275`.

22 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *7th WABI*, volume 4645 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 2007. `doi:10.1007/978-3-540-74126-8_27`.

23 Paul Medvedev and Mihai Pop. What do Eulerian and Hamiltonian cycles have to do with genome assembly? *PLOS Computational Biology*, 17(5):1–5, May 2021. `doi:10.1371/journal.pcbi.1008928`.

24 Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010. `doi:10.1016/j.ygeno.2010.03.001`.

25 Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991. doi:10.1016/0022-0000(91)90023-X.

26 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.

27 Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957. doi:10.1002/j.1538-7305.1957.tb01515.x.

28 Amatur Rahman and Paul Medevedev. Representation of k-mer sets using spectrum-preserving string sets. *J. Comput. Biol.*, 28(4):381–394, 2021. doi:10.1089/CMB.2020.0431.

29 Sebastian Schmidt, Shahbaz Khan, Jarno N Alanko, Giulio E Pibiri, and Alexandru I Tomescu. Matchtigs: minimum plain text representation of k-mer sets. *Genome Biology*, 24(1):136, 2023. doi:10.1186/s13059-023-02968-z.

30 Sebastian S. Schmidt and Jarno N. Alanko. Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. *Algorithms Mol. Biol.*, 18(1):5, 2023. doi:10.1186/S13015-023-00227-1.

31 Ondřej Sladkỳ, Pavel Veselỳ, and Karel Břinda. Masked superstrings as a unified framework for textual k-mer set representations. *bioRxiv*, pages 2023–02, 2023.

32 William H.A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix–prefix problem. *Journal of Discrete Algorithms*, 37:34–43, 2016. 2015 London Stringology Days and London Algorithmic Workshop (LSD & LAW). doi:10.1016/j.jda.2016.04.002.

33 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(3):313–323, 1990. doi:10.1007/BF01840391.

A Class of Heuristics for Reducing the Number of BWT-Runs in the String Ordering Problem

Gianmarco Bertola
Department of Computer Science, University of Pisa, Italy

Anthony J. Cox
Independent Researcher, Cambridge, UK

Veronica Guerrini[1]
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone[2]
Department of Computer Science, University of Pisa, Italy

Abstract

The Burrows-Wheeler transform (BWT) is a famous text transformation that rearranges the symbols of the input strings so that occurrences of a same symbol tend to occur in runs. The number of runs is an important parameter in the BWT output string, historically associated with its high compressibility and more recently used as a measure for the space complexity of efficient data structures. It is a known fact that reordering the strings in the input collection $\mathcal{S}$ affects the number of runs in the output string $bwt(\mathcal{S})$ produced by applying the BWT to the string collection. In this paper, we define a class of transformed strings where symbols in particular blocks of the $bwt(\mathcal{S})$ can be reordered according to a different adaptive alphabet order. Then, we introduce new heuristics to reduce the number of runs in the BWT output of a string collection that improve on the two existing heuristics introduced in Cox et al. [7]. These new heuristics are computed when applying the BWT to a string collection assuming no *a priori* order on the input strings and without requiring any pre- and/or post- processing of the collection $\mathcal{S}$ or of the BWT string. In this paper, we also face the problem of reconstructing the input collection $\mathcal{S}$ from the string $\mathsf{bwt}(\mathcal{S})$ together with the string permutation realized when applying an alphabetical reordering of symbols during the construction of $\mathsf{bwt}(\mathcal{S})$.

2012 ACM Subject Classification Theory of computation → Data compression; Applied computing → Bioinformatics

Keywords and phrases Burrows-Wheeler Transform, SAP-interval, repetitive text, string compression

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.7

Funding Work partially supported by PNRR - M4C2 - Investimento 1.5, Ecosistema dell'Innovazione ECS00000017 - "THE - Tuscany Health Ecosystem" - Spoke 6 "Precision medicine & personalized healthcare", funded by the European Commission under the NextGeneration EU programme, by the MUR PRIN 2022YRB97K PINC and by INdAM - GNCS Project, codice CUP_E53C23001670001.

1 Introduction

The Burrows–Wheeler transform (BWT), introduced by M. Burrows and D. Wheeler in the 1990s [3] as a method for compressing a single input text, has since evolved into a versatile tool with many applications well beyond its original purpose [23]. Just as examples, the BWT has been used as the building block for compact text indexing [8, 16, 17, 10], and for bioinformatics applications, *e.g.*, for sequence alignment [20], phylogenetic analysis [12], genome assembly [24] as well as for sequencing data compression [13].

[1] corresponding author
[2] corresponding author

Informally, the BWT is a text transformation that rearranges the symbols of an input string S into a string $\mathsf{bwt}(S)$, which is obtained by concatenating the symbols that precede the cyclic rotations of S once the rotations have been sorted into lexicographic order. An equivalent and faster way to build $\mathsf{bwt}(S)$ [3] is to sort the suffixes of a related string obtained by appending an end-marker symbol (usually \$) that is lexicographically smaller than any of the symbols in S but does not appear in S itself. Both ways have two important properties: *reversibility* and *clustering effect.*

The *reversibility* permits to invert the transformed string by reconstructing S, and allows to search patterns in S very efficiently. While the *clustering effect* describes the inner property of the BWT to carry occurrences of a given symbol to runs of equal consecutive symbols.

The more symbols can be grouped into runs of the same symbol, the better is the performance of compression techniques such as, for instance, run-length encoding (RLE) where a string is coded as a concatenation of pairs formed by the symbol c and the number of times c is repeated. The total number of runs of a same symbol in the BWT-string is usually referred to as r. Recently, the parameter r is increasingly appearing not only for data compression, but also for measuring the space requirement of BWT-based text indexing data structures (see for instance [16, 17, 10]). Therefore, a text containing a few long runs is easier to compress or index than a text having the same characters but organized into a greater number of shorter runs. An interested reader can find theoretical studies and applications about the clustering effect in [19, 21, 22, 5] and references therein. Due to the ever-increasing volume and repetitive nature of data, developing new techniques that reduce or minimize the number of runs produced by the BWT is paramount for managing big data in applications.

Just as for a single string, the BWT of a collection of strings can be constructed either by sorting their cyclic rotations[3] as in [18] or sorting their suffixes [1]. In the latter case, a distinct end-marker symbol is appended to each string, making the collection ordered according to the order established among the end-marker symbols. Moreover, it is known that given a string collection $\mathcal{S}$, the two strings $\mathsf{bwt}(\mathcal{S})$ and $\mathsf{bwt}(\mathcal{S}')$ can only differ within particular intervals, if $\mathcal{S}'$ is a string permutation of $\mathcal{S}$ [7, 5].

Our contributions. In this paper, we define a class of transformed strings obtained by applying the BWT to a string collection $\mathcal{S}$ in which the symbols in particular blocks of the $\mathsf{bwt}(\mathcal{S})$ can permute according to a different adaptive alphabet ordering, while maintaining the reversibility property. Some known strategies falling into this class have already been introduced in the literature [7, 15, 2, 4]; and we recall them in Section 3.

Then, we introduce new heuristics for reducing the number of runs while computing the BWT-string; these heuristics improve on the number of runs of both the BWT-string obtained from the input-ordered collection and the two previously-introduced heuristics [7]. We show experimentally that the new heuristics tend to minimize the number of runs.

In this paper, the BWT output string is obtained by sorting all the suffixes of the input strings assuming that each string ends with a different end-marker symbol, but no *a priori* ordering of the end-marker symbols is given. Among the state-of-the-art approaches to compute the BWT for a string collection, we employ the algorithm BCR described in [1] to ensure that the order between any two end-marker symbols is determined during the construction of the BWT and not *a priori.* The interested reader can refer to [5] for a survey on the different output strings obtained by different tools.

[3] In this case, one needs to use the ω-order defined in [18].

We also address the problem of inverting the $\mathsf{bwt}(\mathcal{S})$ preserving the input order in $\mathcal{S}$ in case a symbol reordering has been applied during its construction. This property allows to reconstruct only a single string or groups of strings of the input collection and it might be useful in some applications, where only specific groups of strings are to be decoded (*e.g.*, in short-reads collections). In fact, without knowing the string reordering applied to $\mathsf{bwt}(\mathcal{S})$, the inverse transform of $\mathsf{bwt}(\mathcal{S})$ is no longer lossless in terms of string order.

1.1 Related works

In the literature, the problem of reducing the number of runs in the BWT-string has been approached from two perspectives. Indeed, on the one hand, the number of runs is affected by the order of the symbols in the considered alphabet; on the other hand, it is also impacted by the order of the strings in the collection.

Alphabet order. Chapin and Tate [6] show experimentally that ordering symbols by their ASCII code does not always give the best compression and discuss several heuristics for varying the alphabet order. For instance, they propose a scheme in which rotations are sorted in a manner inspired by reflected Gray codes. In [14], the authors introduce the Alternating BWT (ABWT) that is defined as the BWT by using a different order of the cyclic rotations, where one needs to alternate the standard and reverse orderings in odd and even positions. In [11], the authors describe a class of BWT string transformations based on context adaptive alphabet orderings, where in the rotation sorting phase, the alphabet orderings depend on the context (i.e., the longest common prefix of the rotations being compared). Moreover, they consider the problem of determining the BWT variant that minimizes the number of runs in the transformed string. Recently, Bentley et al. [2] derived the computational complexity of minimizing the number of runs in the BWT via alphabet ordering. They prove that the problem of deciding whether there exists an ordering of the alphabet symbols such that the number of runs in the BWT is at most equal to a given integer is NP-complete and its corresponding minimization problem is APX-hard.

String order. When the BWT is applied to a string collection by sorting the suffixes of its strings, one needs to append a different end-marker symbol to each string and to establish a order among them. In this case, the problem of minimizing the number of runs also needs to consider the different orderings of the input strings, since the ordering of the input strings depends on the reciprocal ordering of the end-marker symbols appended to each string. The authors in [7] provide the first experimental study showing: *i*) one can permute symbols within the $\mathsf{bwt}(\mathcal{S})$ associated to particular blocks, named "SAP-intervals" (SAP standing for "*same-as-previous*"), without destroying the string reversibility; *ii*) one can obtain a reduced number of runs in the $\mathsf{bwt}(\mathcal{S})$ while permuting symbols in SAP-intervals (see also [5]).

The problem of minimizing the number of runs in the $\mathsf{bwt}(\mathcal{S})$ via string ordering has been tackled as a closely related problem by Bentley et al. [2]. Indeed, finding a string order that minimizes the number of runs in the $\mathsf{bwt}(\mathcal{S})$ is equivalent to finding an order for the end-marker symbols that results in the minimum number of runs in the $\mathsf{bwt}(\mathcal{S})$. They show that given the $\mathsf{bwt}(\mathcal{S})$, the problem of minimizing its runs via string order can be solved in linear time by reducing such problem to a tuple sorting problem (more details in [2]). In [4], the authors provide the first implementation that computes the $\mathsf{bwt}(\mathcal{S})$ with the fewest number of runs using the post-processing strategy described in [2] combined with the SAP-array [7].

2 Preliminary and Materials

Let $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$ be a finite ordered alphabet Σ with $c_1 < c_2 < \dots < c_\sigma$, where $<$ denotes the standard lexicographic order. Let S be a string of length n on Σ. We denote the i-th symbol of S by $S[i]$. A *substring* of S is denoted as $S[i, j] = S[i] \cdot S[i+1] \cdots S[j]$, with $\cdot$ being the concatenation operator.

Let $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ be a collection of m strings on the alphabet Σ. We assume that each string $S_i \in \mathcal{S}$ has length $n_i + 1$, since we append a special end-marker symbol $\$_i$ to each S_i, *i.e.* $S_i[n_i + 1] = \$_i$, such that each $\$_i$ does not belong to Σ and it is lexicographically smaller than any other symbol in Σ. Let us denote by $N = \sum_{i=1}^{m} n_i + m$ the number of symbols of all strings in $\mathcal{S}$ (including their end-marker symbols).

The *suffix* of a string S_i starting at position k is $S_i[k, n_i + 1]$ and we define the j-suffix of S_i as the suffix starting at position $n_i + 1 - j$ of S_i, *i.e.* $S_i[n_i + 1 - j, n_i + 1]$, which has length $j + 1$ (including the end-marker symbol $\$_i$). Note that the *0-suffix* of S_i is just $\$_i$.

A *run* in a string S is a maximal substring consisting of repetitions of only one character.

The BWT is a reversible text transformation that, given as input a string $S\$$ (with $\$$ not appearing in S), produces an output string $\mathsf{bwt}(S\$)$ such that $\mathsf{bwt}[i]$ is the symbol preceding the i-th lexicographically smallest suffix of the string $S\$$. In the seminal paper by Burrows and Wheeler [3], two important properties that establish a correlation between the string $\mathsf{bwt}(S\$) = L$ and the string F, formed by lexicographically sorting the symbols of $S\$$, have been shown[4]:

- For all $i = 1 \dots n + 1$, the symbol $F[i]$ circularly follows the symbol $L[i]$ in the string $S\$$;
- For each alphabet symbol c, the h-th occurrence of c in L corresponds to the h-th occurrence of c in F. In particular, if $L[i]$ is any occurrence of c in L, the position of its corresponding occurrence in F is given by $C[L[i]] + \text{rank}(L[i], i)$, where $C[c]$ is the total number of symbols in $S\$$ that are smaller than c and $\text{rank}(c, i)$ is the number of occurrences of c in the substring $L[1, i]$.

The above function that maps symbol occurrences in L to their corresponding symbol occurrences in F is known as *LF-mapping* [9].

2.1 The BWT applied to a string collection

A way for applying the BWT to a string collection consists in appending to each string an end-marker symbol and then concatenating the resulting strings to form a unique larger string. Nevertheless, it is also built without concatenating the input strings by using two approaches: *i)* sorting cyclic rotations of the input strings [18]; *ii)* sorting suffixes of the input strings [1]. The former approach uses a special order to sort the cyclic rotations which is not affected by the order of the input strings; while the sorting performed by the latter approach deeply depends on the order defined on the end-marker symbols. For this reason, in this paper, we focus on the latter approach and we follow the strategy introduced in [1] to handle the list of sorted suffixes. Note that in [1], the suffixes of the strings in $\mathcal{S}$ are sorted assuming that each string S_i ends with a distinct end-marker symbol $\$_i$ such that $\$_i < \$_j$, if $i < j$ in $\mathcal{S}$. See Table 1, sixth column (inputBWT), for an example of the BWT of a string collection $\mathcal{S}$ obtained by concatenating the symbols preceding the sorted suffixes of the strings in $\mathcal{S}$ (last column). The authors of [1] provide two related methods for computing such a BWT for large collections of strings making use of sequential reading and writing of

[4] The same properties hold for the BWT of a string collection [18, 1] - see Section 2.1.

files from disk: the first variant, BCR, is a semi external memory approach (see Section 3.1) that requires more RAM than the second variant, BEETL-BCRext, which uses negligible RAM at the expense of a larger amount of disk I/O.

2.2 SAP-array, SAP-interval and BWT by SAP

The authors of [7] showed that compression of the BWT output string can be improved by reordering the strings in the input collection, and that an "implicit sorting" strategy can be applied while computing the BWT. Such a strategy is based on the observation that in some particular blocks of the $\mathsf{bwt}(\mathcal{S})$, the order of the symbols is entirely determined by the order established among the associated j-suffixes that are equal up to the end-marker symbols [5]. In order to keep track of these blocks, we recall the notion of SAP-array and of SAP-interval.

▶ **Definition 1** ([7]). *The* SAP-array *(for 'same-as-previous'-array) of a collection $\mathcal{S}$ is a binary vector of the same length as the $\mathsf{bwt}(\mathcal{S})$ string such that $\mathsf{SAP}[i] = 1$ if and only if the suffix corresponding to the symbol $\mathsf{bwt}(\mathcal{S})[i]$ is* same as *the* previous *suffix in the list of sorted suffixes (their end-marker symbols excluded). A* SAP-interval *$\mathsf{bwt}(\mathcal{S})[b, e]$ is a maximal block of consecutive symbols such that $\mathsf{SAP}[i] = 1$, for all $b < i \leq e$.*

In other words, any run of 1's preceded by a 0 in the SAP-array corresponds to a block of equal j-suffixes, with $j \geq 0$, that differ only for their end-marker symbols.

Therefore, given two collections $\mathcal{S}$ and $\mathcal{S}'$ having the same strings but in different order, the following results hold (see [7, 5]).

▶ **Observation 2.** *The BWTs of $\mathcal{S}$ and $\mathcal{S}'$ have identical SAP-arrays and can only differ within SAP-intervals that contain more than one distinct symbol.*

▶ **Observation 3.** *Within a SAP-interval containing more than one distinct symbol, the reordering of the characters implicitly involves permuting the strings in the collection.*

3 A class of heuristics based on SAP-intervals

In this section, we describe a class of BWT transformed strings that reduce the number of runs based on the notion of SAP-intervals and the two key observations above (Observations 2 and 3). Moreover, we introduce new heuristics that apply an implicit string reordering during the construction of the BWT output string allowing a reduction of the number of runs with respect to the original input order.

We define the following class of transformed strings associated with a string collection $\mathcal{S}$:

▶ **Definition 4.** *Given a string collection $\mathcal{S}$, the class $\mathfrak{S}_{\mathcal{S}}$ comprises all the strings obtained from $\mathsf{bwt}(\mathcal{S})$ by possibly sorting the symbols of each SAP-interval according to a different adaptive alphabet ordering.*

The following existing variants of the BWT of $\mathcal{S}$ belong to the class $\mathfrak{S}_{\mathcal{S}}$:

1. **rloBWT** (or **colexBWT**), which is obtained by using the lexicographic alphabet ordering for each SAP-interval [15, 7] - see rloBWT column in Figure 1. Note that the rloBWT corresponds to sorting the input collection in reverse lexicographic order (RLO), or co-lexicographic order, and it can be computed not only by pre-processing the strings, but also on-the-fly during the construction of the $\mathsf{bwt}(\mathcal{S})$ itself (more details in [15]).

[5] Such a key observation is also stated in Bentley et al. [2], where such blocks are modeled as tuples, and in Cenzato et al. [5] through the notion of "interesting intervals".

2. **sapBWT**, which is obtained by sorting the symbols in those SAP-intervals whose number of distinct symbols is smaller than the SAP-interval's length, *i.e.* only in SAP-intervals in which it is possible to decrease the number of runs of the SAP-interval. The alphabet order used in any of such SAP-intervals, $\mathsf{bwt}[b, e]$, is given by setting $\mathsf{bwt}[b]$ as the smallest alphabet symbol and using the lexicographic order for all the other symbols. Note that the sapBWT is obtained on-the-fly during the construction of the $\mathsf{bwt}(\mathcal{S})$ (through BEETL-BCRext) where the SAP-array information is implicitly taken into account by computing a SAP status (more details in [7]) - see sapBWT column in Figure 1.
3. **optBWT**, which is obtained by using an alphabet order designed *ad hoc* for each SAP-interval containing more than one distinct symbol that minimizes the number of mismatches at the boundaries of the SAP-intervals. The *ad hoc* alphabet order for each SAP-interval is established in a backward fashion while scanning the $\mathsf{bwt}(\mathcal{S})$ and its SAP-array (both pre-computed) and using a stack to manage consecutive SAP-intervals (more details in [4]). In our running example (Figure 1), the optBWT is $TTTTTTT\$\$GGGG\$\$GGGGGCAAGC\$\$\$CCCAAAA$. Note that the number of runs in the optBWT is the minimum possible [2, 4].

Now, we focus on a particular subclass of $\mathfrak{S}_{\mathcal{S}}$ in which the adaptive alphabet order used in SAP-intervals is selected on-the-fly while building the BWT string itself. The sapBWT and rloBWT belong to this subclass, differently from the optBWT that is obtained as post-processing.

Therefore, we do not assume that the strings in $\mathcal{S}$ are ordered: we define a string order for $\mathcal{S}$ while building the BWT string, on the basis of the alphabet order choices performed in the SAP-intervals.

In Section 3.2, we define new heuristics belonging to $\mathfrak{S}_{\mathcal{S}}$ that reduce the number of runs on-the-fly during the construction of the BWT output string. To this end, we adopt the construction method introduced in [1] (see Section 3.1), which does not concatenate the input strings, but incrementally builds the $\mathsf{bwt}(\mathcal{S})$ by parsing the suffixes of the same length through a right-to-left scanning of all the strings at the same time.

3.1 BCR Construction and Data Structure Design

In this section we recall how the BCR algorithm works without describing the previous work in full detail, rather summarizing the explanation and data structures employed. For the space and time complexities of BCR we refer to the original article [1, Table 1].

BCR proceeds incrementally in k steps, where k is the length of the longest string in the collection plus one for the appended end-marker symbol. At the end of step j, BCR has built a partial BWT, $\mathsf{bwt}_j(\mathcal{S})$, corresponding to the concatenation of the symbols preceding the lexicographically sorted suffixes of length less than or equal to j.

In order to compute $\mathsf{bwt}_j(\mathcal{S})$, BCR needs an array A of m elements, that uses $O(m \log(m + |\Sigma| + |\mathsf{bwt}(\mathcal{S})|))$ bits of workspace, which is updated at each iteration j, for $j = 0, 1, 2, \ldots, k$. We denote by $A^{(j)}$ the array A at the j-th iteration, and by q any index in $[1, m]$, then:

- $A^{(j)}[q].seq$ stores the index of the string in $\mathcal{S}$ whose j-suffix is ranked q after lexicographically sorting all j-suffixes, *i.e.*, $A^{(j)}.seq$ gives the lexicographic order of all j-suffixes;
- $A^{(j)}[q].sym$ stores the symbol circularly preceding the j-suffix of the string with index $A^{(j)}[q].seq$, *i.e.*, a symbol to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$;
- $A^{(j)}[q].pos$ stores in which position symbol $A^{(j)}[q].sym$ must be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$.

Table 1 The SAP-array and the different SAP-ordering heuristics applied to the collection $\mathcal{S} = \{CGAT, GGAT, CGCT, AGCT, AGAT, GGAT, GGCT\}$. The SAP-intervals are colored and the sorted suffixes related to $\mathcal{S}$ are listed in the last column. The number of runs is computed considering the end-marker symbols as the same symbol \$.

	Different heuristics string order						*Sorted suffixes in*
SAP-ARRAY	altBWT	plusBWT	randBWT	sapBWT	rloBWT	InputBWT	*input collection*
0	T	T	T	T	T	T	$\$_1$
1	T	T	T	T	T	T	$\$_2$
1	T	T	T	T	T	T	$\$_3$
1	T	T	T	T	T	T	$\$_4$
1	T	T	T	T	T	T	$\$_5$
1	T	T	T	T	T	T	$\$_6$
1	T	T	T	T	T	T	$\$_7$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$A\ G\ A\ T\ \$_5$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$A\ G\ C\ T\ \$_4$
0	G	G	G	G	G	G	$A\ T\ \$_1$
1	G	G	G	G	G	G	$A\ T\ \$_2$
1	G	G	G	G	G	G	$A\ T\ \$_5$
1	G	G	G	G	G	G	$A\ T\ \$_6$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$C\ G\ A\ T\ \$_1$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$C\ G\ C\ T\ \$_3$
0	G	G	G	G	G	G	$C\ T\ \$_3$
1	G	G	G	G	G	G	$C\ T\ \$_4$
1	G	G	G	G	G	G	$C\ T\ \$_7$
0	**A**	**G**	**C**	**C**	**A**	**C**	$G\ A\ T\ \$_1$
1	**C**	**G**	**A**	**A**	**C**	**G**	$G\ A\ T\ \$_2$
1	**G**	**A**	**G**	**G**	**G**	**A**	$G\ A\ T\ \$_5$
1	**G**	**C**	**G**	**G**	**G**	**G**	$G\ A\ T\ \$_6$
0	**G**	**C**	**G**	**C**	**A**	**C**	$G\ C\ T\ \$_3$
1	**C**	**G**	**A**	**A**	**C**	**A**	$G\ C\ T\ \$_4$
1	**A**	**A**	**C**	**G**	**G**	**G**	$G\ C\ T\ \$_7$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$G\ G\ A\ T\ \$_2$
1	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$G\ G\ A\ T\ \$_6$
0	$\$$	$\$$	$\$$	$\$$	$\$$	$\$$	$G\ G\ C\ T\ \$_7$
0	A	A	C	A	A	A	$T\ \$_1$
1	A	A	C	A	A	A	$T\ \$_2$
1	A	A	C	A	A	C	$T\ \$_3$
1	A	A	A	A	A	C	$T\ \$_4$
1	C	C	A	C	C	A	$T\ \$_5$
1	C	C	A	C	C	A	$T\ \$_6$
1	C	C	A	C	C	C	$T\ \$_7$
Number of runs	13	12	13	14	14	17	

A trivial "iteration 0" sets the initial partial BWT, $\mathsf{bwt}_0(\mathcal{S})$, by simulating the insertion of the end-marker symbols in the sorted list of suffixes. Thus, we set $A^{(0)}[q].seq = q$, $A^{(0)}[q].sym = S_q[n_q]$ and $A^{(0)}[q].pos = q$, for $q = 1 \dots m$. In the original version of BCR, $\mathsf{bwt}_0(\mathcal{S}) = A^{(0)}[1].sym \cdots A^{(0)}[m].sym$, *i.e.*, $\mathsf{bwt}_0(\mathcal{S})$ is the concatenation of the symbols preceding the end-marker symbols assuming that $\$_i < \$_j$, if $i < j$.

For each iteration $j = 1, 2, \dots, k$, BCR computes $\mathsf{bwt}_j(\mathcal{S})$ by inserting the symbols preceding all the j-suffixes of $\mathcal{S}$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$, through the following three phases:

1. BCR computes $A^{(j)}$ from $A^{(j-1)}$. For any q, let $x = A^{(j-1)}[q].seq$, $c = A^{(j-1)}[q].sym$ and $p = A^{(j-1)}[q].pos$. The value $A^{(j)}[q].pos$ is set by reading $\mathsf{bwt}_{j-1}(\mathcal{S})$ and by using the LF-mapping, *i.e.*, $A^{(j)}[q].pos = C[c] + rank(c, p)$ (we omit details for space reasons). While, $A^{(j)}[q].sym$ is updated with the symbol preceding the j-suffix of S_x.
2. BCR sorts the array $A^{(j)}$ by using $A^{(j)}.pos$ as sorting key.
3. For each q, BCR inserts the symbol $A^{(j)}[q].sym$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$ at position $A^{(j)}[q].pos$.

At the end of iteration j, after inserting all the symbols preceding the j-suffixes into $\mathsf{bwt}_{j-1}(\mathcal{S})$, we get $\mathsf{bwt}_j(\mathcal{S})$ available for the next iteration. Whenever the first symbol of a string S_x has been inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$, the symbol $\$_x$ must be inserted at the next iteration and then no other symbol of the string S_x will be inserted.

After the last iteration k, all end-marker symbols have been inserted in their correct positions into $\mathsf{bwt}_{k-1}(\mathcal{S})$ and the BWT of the collection is completed.

Note that, the BCR implementation inserts the same end-marker symbol, \$, for all strings (*i.e.*, $\$_i = \$$ for all $i = 1, \dots, m$) so as not to increase the size of the alphabet. However, one can store in a separate file the values in $A^{(j)}.seq$ to which each end-marker symbol is associated.

Actually, in BCR as well as in our implementation, the partial BWT is split into σ segments $B_j(z)$ formed by the symbols preceding suffixes starting with the symbol z (more details in [1]). Hence, in the array $A^{(j)}$, for each value $A^{(j)}[q].pos$, one needs to store two pieces of information: the symbol z and the position in $B_j(z)$ – see also [15].

3.2 Improved SAP-heuristics

Here we introduce three heuristics whose associated BWT strings belong to the class $\mathfrak{S}_{\mathcal{S}}$ of Definition 4. These heuristics are such that:

- they improve the number of runs of the BWT output string with respect to the input order (inputBWT) and the two existing heuristics sapBWT and rloBWT;
- symbols in SAP-intervals are sorted during an incremental construction of the BWT string that parses the suffixes of the same length through a simultaneous right-to-left scanning of all the strings, like BCR does. At the jth-iteration, the sorting takes into account symbols already stored in $\mathsf{bwt}_{j-1}(\mathcal{S})$ or that will be inserted into the $\mathsf{bwt}_j(\mathcal{S})$.

Let j be any BCR iteration, for $j = 1, \dots, k$.

The first heuristic, called **altBWT**, uses an alternating lexicographic order to sort symbols in consecutive SAP-intervals which are to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$. Thus, it differs from the rloBWT, as it alternates the lexicographic order and its inverse when inserting consecutive SAP-intervals - see altBWT column in Table 1.

The second heuristic, called **plusBWT**, designs an *ad hoc* alphabet order for each SAP-interval to be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$ on the basis of the symbols already in it. In particular, let p be the position in which the first symbol of the SAP-interval must be inserted into $\mathsf{bwt}_{j-1}(\mathcal{S})$. We modify the alphabet order by setting the smallest symbol as $\mathsf{bwt}_{j-1}[p-1]$ (if it exists) and the greatest one as $\mathsf{bwt}_{j-1}[p]$ (if it exists), and by keeping the alphabet order among all the other symbols - see plusBWT column in Table 1.

The third heuristic, called **randBWT**, applies a random alphabet order for each SAP-interval inserted into bwt_{j-1}. Note that all these heuristics correspond to a string reordering that cannot be obtained *a-priori* unless having the associated string permutation.

In order to sort symbols according to any of the above strategies, the array $A^{(j)}$ defined in Section 3.1 is augmented with a binary value $A^{(j)}[q].sap$ that stores, for any $q = 1, \dots, m$, the SAP-status of the associated symbol $A^{(j)}[q].sym$[6]. More precisely, $A^{(j)}[q].sap$ encodes whether or not the j-suffix of the string S_x, where $x = A^{(j)}[q].seq$, is same as the j-suffix of the string S_y, where $y = A^{(j)}[q-1].seq$, up to the end-marker symbol[7].

[6] Differently from [7], we compute the SAP-status and the SAP-intervals for the current iteration.
[7] Similar strategies have been used in [7, 15] and in the BCR-implementation of the tool `optimalBWT` introduced in [4] for explicitly computing the SAP-array.

Hence, if we have $A^{(j)}[q'].sap = 0$ and $A^{(j)}[q' + i].sap = 1$, for some q' and all i with $1 \leq i < \ell$, then there exist ℓ j-suffixes in $\mathcal{S}$ that are equal up to the end-marker symbols and that belong to the strings with indices $A^{(j)}[q' + i].seq$, for $0 \leq i < \ell$. For this reason, the symbols $A^{(j)}[q' + i].sym$, for $0 \leq i < \ell$, form a SAP-interval and they are inserted in consecutive positions into $\mathsf{bwt}_{j-1}(\mathcal{S})$.

Now, we describe how to modify BCR to compute any of the above heuristics.

At “iteration 0”, the array A is initialized as described in Section 3.1. Moreover, since we simulate the insertion of the 0-suffixes of $\mathcal{S}$, we set $A^{(0)}[1].sap = 0$ and $A^{(0)}[q].sap = 1$, for all $1 < q \leq m$. Differently from the original BCR, before storing in $\mathsf{bwt}_0(\mathcal{S})$ the symbols $A^{(0)}[q].sym$, for $q = 1 \dots m$, we perform a sorting on $A^{(0)}$ with respect to $A^{(0)}[\cdot].sym$ as sorting key. Supposing $\sigma < m$, we perform a linear sorting on $A^{(0)}$ on the basis of a special alphabet order, which is for both altBWT and plusBWT the alphabetic order, and for randBWT a random order on Σ. Then, by setting $\mathsf{bwt}_0(\mathcal{S}) = A^{(0)}[1].sym \dots A^{(0)}[m].sym$, we have that the number of runs of $\mathsf{bwt}_0(\mathcal{S})$ is minimized. Note that the sorting is possible since we assume there is no fixed order among the end-marker symbols, (*i.e.*, it is no longer true that $\$_i < \$_j$ if $i < j$). The sorting of the symbols depends on the selected alphabet order rather than on the string ordering.

At each iteration $j = 1, 2, \dots, k$, BCR updates the partial $\mathsf{bwt}_{j-1}(\mathcal{S})$ by inserting the symbols preceding the j-suffixes of $\mathcal{S}$ by using the three phases described in Section 3.1. We modify both phase 1 and phase 3 in order to update $A^{(j)}[q].sap$ values and to permute symbols in SAP-intervals while building $\mathsf{bwt}_j(\mathcal{S})$.

In particular, during phase 1, when computing $A^{(j)}$ from $A^{(j-1)}$, we propagate the SAP-status from iteration $j-1$ to iteration j. For each maximal interval $[b, e]$ in $A^{(j-1)}$ such that $A^{(j-1)}[i].sap = 1$ for all $b < i \leq e$, we set $A^{(j)}[q].sap = 0$ if $A^{(j-1)}[q].sym \neq A^{(j-1)}[q-1].sym$, for any $b < q \leq e$, and keep $A^{(j)}[q].sap = A^{(j-1)}[q].sap$, otherwise. Intuitively, let c and c' be the symbols preceding the two equal j-suffixes of S_x and S_y, where $x = A^{(j)}[q].seq$ and $y = A^{(j)}[q-1].seq$. Both c and c' are in the same SAP-interval, but being $c \neq c'$, the $(j+1)$-suffixes of S_x and S_y are no longer equal and thus their preceding symbols are no longer in the same SAP-interval. During phase 2, BCR sorts the array $A^{(j)}$ by using $A^{(j)}[q].pos$ as sorting key. No modifications need to be performed at this phase, but we can make a key observation relevant for phase 3: for each maximal interval $[b, e]$ in $A^{(j)}$ such that $A^{(j)}[i].sap = 1$ for $b < i \leq e$, the symbols $A^{(j)}[q].sym$ need to be inserted in consecutive positions into $\mathsf{bwt}_{j-1}(\mathcal{S})$ starting from position $p = A^{(j)}[b].pos$ (that is $A^{(j)}[b+i].pos = p+i$, for all $i = 1, \dots, e-b$). During phase 3, for each maximal interval $[b, e]$ in $A^{(j)}$ such that $A^{(j)}[i].sap = 1$ for all $b < i \leq e$, we first linearly sort the sub-array $A^{(j)}[b, e]$ by using a specific alphabet order on the key $A^{(j)}.sym$, and then for all $b \leq q \leq e$, we write the symbol $A^{(j)}[q].sym$ into $\mathsf{bwt}_{j-1}(\mathcal{S})$ in consecutive positions starting from p.

At the end, BCR has built a BWT string for the collection $\mathcal{S}$ in which the string order is not given *a priori*, but it has implicitly established during the BWT construction itself according to the alphabet order used within SAP-intervals.

The additional space required with respect to the original BCR is given by both the space for storing the SAP status in m bits and the space used for linearly sorting the elements in the SAP-intervals, which is $O(\sigma \log m)$ bits to store the number of symbol occurrences in a SAP-interval and $O(m \log(\sigma + m))$ bits to linearly sort at most m symbols carrying the indices of the strings to which they belong. The time complexity of each iteration increases by $O(m)$, since first the array $A^{(j)}$ is scanned to find any maximal interval $[b, e]$ such that $A^{(j)}[i].sap = 1$ (for $b < i \leq e$) and for each of them the elements in $A^{(j)}[b, e]$ are linearly sorted according to $A^{(j)}.sym$. Thus, the overall space and time complexity remains as in [1, Table 1].

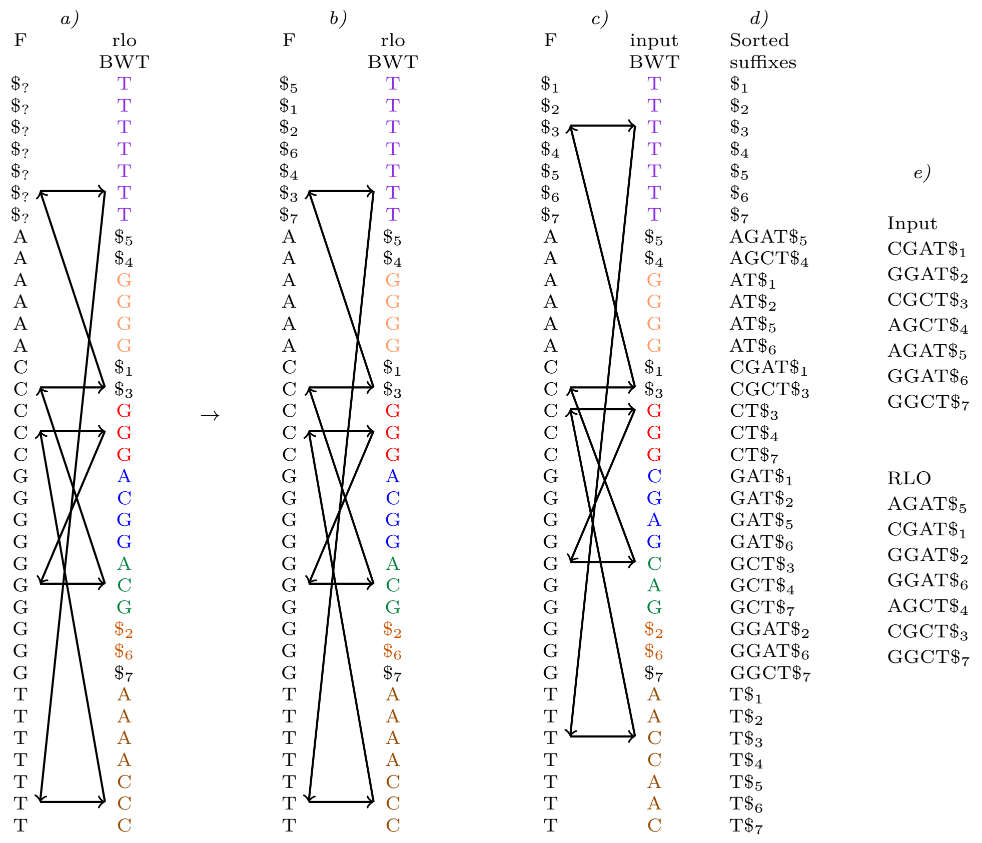


Figure 1 Considering the string collection of Table 1: *a)* and *b)* show the decoding of a string when the strings are sorted by using the reverse lexicographic order (RLO). In a), the indices of the end-markers in column F cannot be assigned, if we do not know the string permutation performed during the encoding. In b), these indices are assigned since the encoding transformation outputted the string permutation. While *c)* shows the decoding of a string when no string permutation is performed. In *d)*, we list the sorted suffixes according to the input order, and in *e)* the two considerd re-orderings of our string collection.

4 Inverting the BWT and input order-preserving

In this section, we address the problem of inverting the BWT transform when a symbol re-ordering in the SAP-intervals has been applied.

For ease of description, Figures 1*a)*-*b)* show the columns F and L of the collection of our running example where the symbols of any SAP-interval have been sorted lexicographically (rloBWT). However, what we show in the following holds for all the other SAP-ordering heuristics. Figure 1*c)* shows the LF mapping applied to $S_3 = CGCT\$_3$ in the BWT string with input order of the collection, whose associated list of sorted suffixes is in Figure 1*d)*.

Note that by applying the LF-mapping to the rloBWT, starting from the first m symbols in L, we retrieve the m strings of the collection but permuted according to the order determined by the local alphabet order used within the SAP-intervals (RLO in Figure 1*e)*). However, the input string permutation can be recovered: in fact, when applying LF-mapping starting from the q-th symbol in L (e.g., in Figure 1*a)* $q = 6$), we end up with the end-marker $\$_k$ (e.g., in Figure 1*a)* $\$_3$) which means that the string S_k (S_3) has been placed at the q-th position (6-th position) in the permuted string collection. Since the indices of the end-marker symbols in L can be stored in a dedicated file, any \$ in L can be associated with the correct string to which it belongs even if all end-marker symbols appearing in L are equal to \$. In this way, since the LF-mapping starting from the q-th symbol in L ends in $\$_k$, we can label the q-th end-marker symbol $\$_?$ in F with the index k (e.g., in Figure 1*a)* $F[6] = \$_3$).

The example in Figure 1 also shows that, although the operation of symbol swapping is not visible in any SAP-interval with a run of a same symbol, the symbols are (implicitly) swapped with respect to the inputBWT (e.g., the red G-symbols in the third SAP-interval of Figure 1*c)*, since $\$_3 > \$_4$ in rloBWT).

We point out that by using LF-mapping we can decode the entire collection, but decoding one single string or a specific group of strings in $\mathcal{S}$ is not possible. Indeed, in Figure 1*a)*, we do not know how to decode the sixth string of the input collection, since starting from $L[6]$ we end up decoding the third string. In addition, it is not possible to start from $\$_6$ in L and to apply the LF-mapping, since we are not able to map $\$_6$ in L to the corresponding $\$_?$ in F. Therefore, the crucial property of decoding only specific groups of strings that the BCR algorithm guarantees is compromised.

We address this issue by designing a strategy so that BCR can output the permutation of the string indices at the end of the BWT construction phase. In this way, it is possible to assign the correct index to any end-marker symbol in F, and decoding groups of strings without decoding the entire collection (Figure 1*b)*).

Let π be an array of length m storing the permutation of the string indices of the input collection. For instance, in Figure 1*b)*, where rloBWT is computed, the permutation $\pi = [5\ 1\ 2\ 6\ 4\ 3\ 7]$. Whereas, at the last iteration, the array $A^{(4)}.seq$ contains the indices $[5\ 4\ 1\ 3\ 2\ 6\ 7]$, which correspond to the indices of the end-marker symbols in L. Indeed, at the last iteration, $A^{(k)}.seq$ contains the indices of the strings according to their lexicographic order, regardless of the SAP-ordering heuristics used for building the BWT string.

Therefore, during the BWT construction, we need to keep track of the symbol swapping performed. We modify the BCR data structure so that some entries of the array A point to indices of π. More precisely, for any iteration j, we have a pointer $A^{(j)}[q].pi$ to a position in π whenever a symbol swapping may affect the entries of $A^{(j)}$ from position q, *i.e.*, $A^{(j)}[q].sap = 0$ and $A^{(j)}[q+1].sap = 1$. That allows to report in π any string index swapping due to a symbol swapping within a SAP-interval. In fact, the array $A^{(j)}$ is designed to assign to each symbol $A^{(j)}[q].sym$ the string index to which it belongs (*i.e.*, $A^{(j)}[q].seq$).

After "iteration 0", we have $A^{(0)}[1].pi$ points to $\pi[1]$ and we initialize $\pi[q]$ with the value $A^{(0)}[q].seq$, for all $1 \le q \le m$. At each iteration j, we update $A^{(j)}[q].pi$ during phase 1 at the same time as $A^{(j)}[q].sap$. In particular, if $A^{(j)}[q].sap$ is set to 0, for some q, then $A^{(j)}[q].pi$ points to $\pi[x]$, where x is obtained by moving the position pointed by $A^{(j)}[q'].pi$ (with q' the rightmost index preceding q such that $A^{(j)}[q'].sap = 0$) by the offset $q - q'$. During phase 3, we need to update π when a symbol swapping is performed. Thus, if $[b, e]$ is a maximal interval in $A^{(j)}$ such that $A^{(j)}[q].sap = 1$ for all $b < q \le e$, and $\pi[x]$ is the entry of π pointed by $A^{(j)}[b].pi$, then we copy in $\pi[x, x + b - e]$ the values $A^{(j)}[b, e].seq$ after the symbol swapping in that interval. At the end of the BWT construction, π corresponds to the list of the indices of the end-marker symbols in F.

5 Experimental Results

In this section, we assess the performance of the introduced heuristics that we have integrated into BCR tool[8] implemented in C++ working in semi-external memory. To evaluate the performance, we have designed a series of tests on real-life datasets (see Table 2).

[8] Source code: `https://github.com/giovannarosone/BCR_LCP_GSA`.

Table 2 Real-life datasets together with the BWT length, the maximum string length and the number of strings. The column optBWT reports the minimum number of runs for each dataset, the column ρ stores the ratio between the sum of the lengths of all SAP-intervals and the number of runs in them and the column τ stores the ratio between the number of runs in the SAP-intervals and the number of different symbols in them (higher values in bold).

	Dataset	Description	BWT length	Max len.	Number of sequences	optBWT	ρ	τ
1	SRR7494928–30	*Epstein Barr Virus*	984,191,064	101	9,648,932	40,700,607	3.32	**35.51**
2	ERR732065–70	*HIV-virus*	1,345,713,812	150	8,912,012	11,539,661	10.98	15.57
3	SRR12038540	*SARS-CoV-2 RBD*	1,690,229,250	50	33,141,750	14,864,523	7.08	**25.46**
4	ERR022075_1	*E. Coli str. K-12*	2,294,730,100	100	22,720,100	71,203,469	1.46	11.38
5	SRR059298	*Deformed wing virus*	2,455,299,082	72	33,634,234	48,376,632	8.09	17.62
6	SRR065389–90	*C. Elegans*	14,095,870,474	100	139,563,074	921,561,895	1.56	8.15
7	SRR2990914_1	*Sindibis virus*	15,957,722,119	36	431,289,787	105,250,120	3.16	**129.84**
8	ERR1019034	*H. Sapiens*	123,506,926,658	100	1,222,840,858	10,860,229,434	1.82	7.58
9	*pdb_seqres*	*proteins*	241,121,574	16,181	865,773	16,829,629	5.51	5.20

For each dataset, we computed two parameters: ρ and τ. The former parameter is given by the ratio between the sum of the lengths of all SAP-intervals in the BWT string and the total number of runs in the SAP-intervals of the inputBWT and it can be considered as a repetitiveness measure in SAP-intervals. The latter parameter is given by the ratio between the total number of runs in the SAP-intervals of the inputBWT and the sum of the number of distinct symbols in each SAP-interval. The higher τ, the more the heuristics can reduce the number of runs in the SAP-intervals, since the alphabet ordering applied to any SAP-interval reduces its number of runs to the number of distinct symbols.

All tests were done on a DELL PowerEdge R750 machine, 24-core machine with 2 Intel(R) Xeon(R) Gold 5318Y 24C/48T CPUs at 2.10 GHz, with 960 GB. The system is Ubuntu 22.04.2 LTS.

In Table 3, we report the number of runs for the new heuristics plusBWT, altBWT and randBWT, and show they improve on BWT-string with input order (inputBWT), and the two previously-introduced heuristics rloBWT and sapBWT[9].

Recall that the sapBWT heuristic is built using BEETL-BCRext [1], which uses negligible RAM at the expense of a larger amount of disk I/O. Therefore, its computation requires more time than the other heuristics. In fact, all other BWT-strings are obtained by the BCR-based tool that works in semi-external memory by sequential reading and writing files on disk and requires more RAM (to store the array A) than BEETL-BCRext.

The heuristics altBWT, plusBWT, randBWT and rloBWT have similar performances: on the largest dataset of about 123 Gb containing more than a billion sequences, they required a time construction of about 17 hours and an internal memory usage of about 20GB. On the contrary, the inputBWT required a time construction of about 15 hours and a similar internal memory usage (about 20GB). Note that the optBWT is computed as post-processing [4] by taking about 19 hours due to the fact that it needs to explicitly compute the SAP-array.

The experimental results show that plusBWT is the heuristic that gives the fewest runs, improving on the number of runs in inputBWT by up to 97% and giving at least a 50% reduction in runs for all eight of the DNA sequence datasets (strings from the alphabet $\{A, C, G, N, T\}$ of the same length). For the last dataset, containing proteins (on an alphabet of 26 symbols of variable length), we observe that the reduction in the number of runs

[9] Note that the implementation of sapBWT requires strings of the same length – see `https://github.com/BEETL/BEETL`.

Table 3 Number of runs in the BWT-string without symbol reordering (inputBWT) compared to the number of runs for any heuristic being in the class $\mathfrak{S}_{\mathcal{S}}$ for each dataset in Table 2.

		Different heuristics string order				
	inputBWT	rloBWT	sapBWT	plusBWT	altBWT	randBWT
1	254,663,327	41,730,649	65,040,263	**41,372,530**	41,592,394	41,599,327
2	48,727,709	11,941,093	17,662,811	**11,766,827**	11,858,536	11,872,578
3	209,136,502	17,026,009	17,949,348	**15,226,766**	16,014,506	16,626,930
4	259,821,570	75,846,202	92,304,201	**74,529,428**	75,239,739	75,332,300
5	249,873,376	50,495,777	75,142,244	**49,619,150**	50,207,432	50,302,961
6	2,251,887,226	968,098,124	1,066,534,827	**954,489,749**	960,811,214	963,741,035
7	3,313,966,937	109,772,697	188,817,402	**108,466,351**	109,365,518	109,599,875
8	23,084,021,291	11,312,737,256	12,151,830,264	**11,179,873,104**	11,250,843,471	11,273,506,405
9	17,971,532	16,862,960	–	**16,848,496**	16,861,264	16,861,897

is smaller compared to the one obtained for larger datasets, since the overall number of SAP-intervals is smaller. In addition, for this particular dataset, only $96,814$ of its $24,055,929$ SAP-intervals have at least two distinct symbols[10], and reordering symbols in SAP-intervals can have an impact on the number of runs only if SAP-intervals have at least two distinct symbols.

Finally, the additional overhead for the computation of any BWT-string in the class $\mathfrak{S}_{\mathcal{S}}$ is negligible compared to the number of runs reduction obtained with respect to the inputBWT.

6 Conclusions and further work

In this paper, we defined from a theoretical viewpoint a class $\mathfrak{S}_{\mathcal{S}}$ of transformed strings obtained by applying the BWT to a string collection $\mathcal{S}$ in which the symbols in particular blocks (SAP-intervals) permute according to a different adaptive alphabet ordering. We showed that the symbol swapping is important to reduce the number of runs in the BWT-string with respect to the one computed using the string input order, and it can be performed while maintaining the reversibility property of the BWT.

From a practical viewpoint, we introduced some heuristics belonging to $\mathfrak{S}_{\mathcal{S}}$ that reduce the number of runs, while computing the BWT-string itself. These heuristics improve on both the BWT-string obtained from the input-ordered collection and the two previously-introduced heuristics in [7].

In the experiments, the heuristics in the class $\mathfrak{S}_{\mathcal{S}}$ showed a considerable reduction in the number of runs. For instance, for all datasets (apart from *pdb_seqres* dataset), plusBWT obtained a reduction in the number of runs of about 50%-96% with respect to the inputBWT. Such reordering strategies can be very useful for data compression and for data structures whose properties have a favourable dependence on a small number of runs. Furthermore, the experiments showed that good results in terms of number of runs can be obtained using a random alphabet order for any SAP-interval (*i.e.*, randBWT). That heuristic performs better than the rloBWT heuristic that establishes the lexicographic alphabet order for each SAP-interval. This is an intriguing fact that shows that picking random symbols to place at the borders of a SAP-interval can be better than always choosing the lexicographic order to sort them. Experimentally, the best results are obtained when the alphabet order choice in

[10] These SAP-intervals with at least two distinct symbols are associated with the interesting intervals introduced in [5].

SAP-intervals keeps track of the symbols immediately preceding/succeeding, as done in the plusBWT heuristic. In addition, we observe that a pre-processing reordering of the input strings in $\mathcal{S}$ can only be applied if the string reordering is known *a priori*, such as for the reverse lexicographic order; nevertheless, this condition does not universally apply.

From Observations 2 and 3, we can conclude that the size of the introduced class $\mathfrak{S}_\mathcal{S}$ is at most $m!$. However, strings that are equal keep their original order in $\mathcal{S}$ and not all permutations may be possible. As future work, we intend to study further the permutations in the class $\mathfrak{S}_\mathcal{S}$ taking into account also the permutation study related to the rloBWT in [5].

Finally, an interesting direction for further studies involves to determine how the other data structures related to BWT are affected by the symbol swapping, considering that the LCP-array is not affected, as well as the SAP-array.

References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013. Source code: `https://github.com/BEETL/BEETL`.

2 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *ESA 2020*, volume 173 of *LIPIcs*, pages 15:1–15:13, 2020. `doi:10.4230/LIPIcs.ESA.2020.15`.

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

4 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *Data Compression Conference, DCC 2023*, pages 71–80. IEEE, 2023. Source code: `https://github.com/davidecenzato/optimalBWT`. `doi:10.1109/DCC55655.2023.00015`.

5 Davide Cenzato and Zsuzsanna Lipták. A theoretical and experimental analysis of BWT variants for string collections. In *CPM 2022*, volume 223 of *LIPIcs*, pages 25:1–25:18, 2022. `doi:10.4230/LIPIcs.CPM.2022.25`.

6 Brenton Chapin and Stephen R. Tate. Higher compression from the Burrows-Wheeler transform by modified sorting. In *DCC*, page 532, Washington, DC, USA, 1998. IEEE Computer Society.

7 Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinform.*, 28(11):1415–1419, 2012. Availability: Code is part of the BEETL library, available as a github repository at `https://github.com/BEETL/BEETL`. `doi:10.1093/bioinformatics/bts173`.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. `doi:10.1109/SFCS.2000.892127`.

9 Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001. `doi:10.1016/S0020-0255(01)00098-6`.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), 2020. `doi:10.1145/3375890`.

11 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. A new class of string transformations for compressed text indexing. *Information and Computation*, 294:105068, 2023. `doi:10.1016/j.ic.2023.105068`.

12 Veronica Guerrini, Alessio Conte, Roberto Grossi, Gianni Liti, Giovanna Rosone, and Lorenzo Tattini. phyBWT2: phylogeny reconstruction via eBWT positional clustering. *Algorithms Mol. Biol.*, 18(1):11, 2023. `doi:10.1186/S13015-023-00232-4`.

13 Veronica Guerrini, Felipe A. Louza, and Giovanna Rosone. Parallel lossy compression for large FASTQ files. In *Biomedical Engineering Systems and Technologies*, pages 97–120, Cham, 2023. Springer Nature Switzerland. `doi:10.1007/978-3-031-38854-5_6`.

14 Christophe Reutenauer Ira M. Gessel, Antonio Restivo. A bijection between words and multisets of necklaces. *Eur. J. Combin.*, 33(7):1537–1546, 2012.

15 Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014. Source code: `https://github.com/lh3/ropebwt2`. `doi:10.1093/bioinformatics/btu541`.

16 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.

17 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

18 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. `doi:10.1016/j.tcs.2007.07.014`.

19 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino, and Luca Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017.

20 Joong Chae Na, Hyunjoon Kim, Seunghwan Min, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. Fm-index of alignment with gaps. *Theor. Comput. Sci.*, 710:148–157, 2018.

21 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2021.

22 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2021.

23 Giovanna Rosone and Marinella Sciortino. The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words. In *CiE*, volume 7921 LNCS of *LNCS*, pages 353–364. Springer, 2013. `doi:10.1007/978-3-642-39053-1_42`.

24 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinform.*, 26(12):367–373, 2010.

Faster Sliding Window String Indexing in Streams

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Simon R. Tarnow
Technical University of Denmark, Lyngby, Denmark

Abstract

The classical string indexing problem asks to preprocess the input string S for efficient pattern matching queries. Bille, Fischer, Gørtz, Pedersen, and Stordalen [CPM 2023] generalized this to the streaming sliding window string indexing problem, where the input string S arrives as a stream, and we are asked to maintain an index of the last w characters, called the window. Further, at any point in time, a pattern P might appear, again given as a stream, and all occurrences of P in the current window must be output. We require that the time to process each character of the text or the pattern is worst-case. It appears that standard string indexing structures, such as suffix trees, do not provide an efficient solution in such a setting, as to obtain a good worst-case bound, they necessarily need to work right-to-left, and we cannot reverse the pattern while keeping a worst-case guarantee on the time to process each of its characters. Nevertheless, it is possible to obtain a bound of $\mathcal{O}(\log w)$ (with high probability) by maintaining a hierarchical structure of multiple suffix trees.

We significantly improve this upper bound by designing a black-box reduction to maintain a suffix tree under prepending characters to the current text. By plugging in the known results, this allows us to obtain a bound of $\mathcal{O}(\log\log w + \log\log \sigma)$ (with high probability), where σ is the size of the alphabet. Further, we introduce an even more general problem, called the streaming dynamic window string indexing, where the goal is to maintain the current text under adding and deleting characters at either end and design a similar black-box reduction.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases data structures, pattern matching, string indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.8

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).

1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. Bille, Fischer, Gørtz, Pedersen, and Stordalen [6] introduced a variant of the string indexing problem, called the *streaming sliding window string indexing (SSWSI) problem*, where S arrives as a stream one character at a time. Here, we want to maintain an index of a *window* of the last w character for a specified parameter w. At any point in time, a pattern matching query for a pattern P may arrive also streamed one character at a time, and we need to report the occurrences of P within the current window.

We measure the complexity of the algorithm by the worst-case time it processes a single character of the text or pattern. The goal is to compactly maintain the index while processing the characters arriving from either S or a pattern query efficiently. The SSWSI problem captures scenarios where we want to index recent data in an incoming stream (the window) while supporting fast pattern matching queries. For instance, monitoring a high-speed data stream, where we cannot afford to index the entire stream but still want to support fast queries.

As discussed in Bille, Fischer, Gørtz, Pedersen, and Stordalen [6], the standard string indexing structures, such as *sliding suffix tree* [8,12,20–22] and *online suffix tree* [1–3,7,14,17–19] constructions, do not provide an efficient solution to the SSWSI problem. For instance, efficient online suffix tree constructions require that we process the string (and hence also the pattern) in right-to-left order. In our setting we cannot afford to reverse pattern while keeping a worst-case guarantee on the time to process each of its characters. Bille, Fischer, Gørtz, Pedersen, and Stordalen [6] showed how achieve $\mathcal{O}(\log w)$ (with high probability) time per character by maintaining a hierarchical structure of multiple suffix trees.

In this paper, we present a new black-box reduction to online suffix tree construction algorithms, i.e., algorithms that maintain suffix trees while prepending one character at a time to the current text. By plugging in known results, we obtain solutions using either $\mathcal{O}(\log\log w + \log\log\sigma)$ time (with high probability) or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ (deterministic) time per character. Here, σ is the size of the alphabet. We also consider a generalized version of this problem, called the *streaming dynamic window string indexing (SDWSI) problem.* Here, the window is a dynamic string that can be updated by adding or deleting characters at either end of the string, and we have to support streamed pattern matching queries as above. We show how to extend our reduction and results for this problem, and obtain similar bounds.

1.1 Setup

We now formally define the streaming dynamic window string indexing and streaming sliding window string indexing problems and our main results.

Streaming Dynamic Window String Indexing. Let S be a dynamic string over an alphabet Σ. The *streaming dynamic window string indexing (SDWSI) problem* is to maintain a data structure on S that supports the following operations:

- AddRight(a): add the character a to the right end of S.
- AddLeft(a): add the character a to the left end of S.
- RemoveRight(): remove the last character from S.
- RemoveLeft(): remove the first character from S.
- Report(P) report all the occurrences of P in S.

In the Report(P) query, the pattern string P is streamed one character at a time from left-to-right and the goal is to begin reporting occurrences immediately after receiving the last character. We do not assume that we know the length P before the arrival of its last character.

Streaming Sliding Window String Indexing. Given an integer parameter $w \geq 1$, we define the *streaming sliding window string indexing (SSWSI) problem* as above, except that we support a restricted set of operations:

- Report(P) report all the occurrences of P in S.
- Update(a): AddRight(a). If $|S|$ is now greater than w also perform a RemoveLeft().

Thus, except for the first w Update operations, the window always has size w and changes only by "sliding" one character to the right. This is also called the *timely streaming sliding window string indexing problem* in Bille et al. [6].

Online Suffix Trees and Dynamic Dictionaries. Our main results use online suffix tree construction algorithms and dynamic dictionaries as a black box. We define the precise requirements for these. Let R be a string of length r over an alphabet of size σ and T_i be the suffix tree of $R[i..r]$. An *online suffix tree construction algorithm* processes R from right to left such that at the ith step, the algorithm explicitly constructs T_i and returns a pointer to the new leaf ℓ corresponding to suffix $R[i..r]$, the parent of ℓ, and the edge between ℓ and the parent. We will use the currently best known algorithms for online suffix tree construction due to Kopelowitz [17] and Fischer and Gawrychowski [14].

▶ **Lemma 1** ([14, 17]). *Given a string R of length r over an alphabet of size σ, we can solve online suffix tree in linear space using either $\mathcal{O}(\log\log r + \log\log\sigma)$ time with high probability*[1] *or $\mathcal{O}(\log\log r + \frac{(\log\log\sigma)^2}{\log\log\log\sigma})$ (deterministic) time per character, respectively.*

Let X be a set of x integers from a universe of size u. A *dynamic dictionary structure* on X supports membership (i.e., determine if a given integer is in X or not), insert, and delete on X. We use the following results.

▶ **Lemma 2.** *A set $X \subseteq [U]$ can be maintained in a linear space dynamic dictionary structure that uses either $\mathcal{O}(1)$ time with high probability or $\mathcal{O}(\frac{(\log\log U)^2}{\log\log\log U})$ (deterministic) time.*

Proof. The first bound is obtained by using a dynamic hash table [10]. The second bound follows from a result of Andersson and Thorup [4]. ◀

We will maintain a dynamic dictionary structure $D(v)$ for every explicit node v of the current suffix tree T_i. $D(v)$ maps the first character on an edge to the edge, which allows us to navigate down in T_i to find the (implicit or explicit) node corresponding to $P[1..i]$, for $i = 1, 2, \ldots, m$, in either $\mathcal{O}(1)$ time with high probability or $\mathcal{O}\left(\frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ (deterministic) time per character of P.

1.2 Results

We can now define our main results. Let $t_{\text{suff}}(r, \sigma)$ denote the time per character of a linear space online suffix tree construction algorithm on a string of length r over an alphabet σ. Also, let $t_{\text{dict}}(x, u)$ denote the time per operation of a linear space dynamic dictionary structure. We show the following result for streaming sliding window string indexing.

▶ **Theorem 3.** *Let S be a string over an alphabet of size σ. Given an integer parameter $w \geq 1$ we can solve the streaming sliding window string indexing problem on S for a window of size w with an $\mathcal{O}(w)$ space data structure that supports* Update *and* Report *in $\mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

Plugging in Lemmas 1 and 2 in Theorem 3 we obtain the following bounds:

[1] Kopelowitz [17] claims only worst-case expected time, but the expectation is due to hash tables, so one can plug in e.g. the construction of Dietzfelbinger and auf der Heide [10].

▶ **Corollary 4.** *Let S be a string over an alphabet of size σ. Given an integer parameter $w \geq 1$ we can solve the streaming sliding window string indexing problem on S for a window of size w with an $\mathcal{O}(w)$ space data structure that supports* Update *and* Report *in either $\mathcal{O}(\log\log w + \log\log\sigma)$ time per character with high probability or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ deterministic time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

For the streaming dynamic window string indexing we show the following result.

▶ **Theorem 5.** *Let S be a dynamic string of length w over an alphabet σ. We can solve the streaming dynamic window string indexing problem on S with an $\mathcal{O}(w \cdot t_{\text{suff}}(w,\sigma))$ space data structure that supports* AddRight, AddLeft, RemoveRight, RemoveLeft *and* Report *in $\mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

Again, plugging in Lemmas 1 and 2 in Theorem 5 we obtain the following bounds:

▶ **Corollary 6.** *Let S be a dynamic string of length w over an alphabet σ. We can solve the streaming dynamic window string indexing problem on S with an $\mathcal{O}(w \cdot t_{\text{suff}}(w,\sigma))$ space data structure that supports* AddRight, AddLeft, RemoveRight, RemoveLeft *and* Report *in either $\mathcal{O}(\log\log w + \log\log\sigma)$ time per character with high probability or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ deterministic time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

1.3 Techniques

We first show how to use an online suffix tree construction algorithm to solve the version where the string only changes by appending characters to the right. As noted, existing fast algorithms for this problem work from right to left by prepending characters and then reverse patterns to do a pattern matching query. We cannot do this efficiently in our scenario since we want fast per character processing and we do not know the length of the pattern ahead of time. To overcome this, we construct the online suffix tree over the reverse string and then answer a pattern matching query P by prepending the characters of P to the string as we receive them. Thus after receiving all of P the suffix tree contains all suffixes of $\mathsf{rev}(P)\$\mathsf{rev}(S)$. When we receive the last character from the pattern, we determine if there is an occurrence by checking if the edge in the suffix tree created by prepending the last character starts with a \$. Finally, we return the state of the online suffix tree before the query. To quickly return the state of the online suffix tree to the state before the query, we use techniques from persistent data structures.

To solve the streaming sliding window string indexing problem, we use the above data structure over the last part of the window and a static suffix tree with a range maximum query data structure over the first part of the window. The two data structures always overlap by $w/3$. We can then answer pattern matching queries by querying each of these structures. To find occurrences of long patterns not covered by any of the structures, we give an algorithm that can report all occurrences of a pattern P in a text of length $\mathcal{O}(|P|)$ in constant time per streamed character in P.

We solve the streaming dynamic window string indexing problem by representing S as a concatenation of two shorter strings $S = S_1 \cdot S_2$. We then use online suffix tree data structures on S_1 and S_2. The one on S_1 supports only the updates AddLeft and RemoveLeft, and the one on S_2 supports only the updates AddRight and RemoveRight. To obtain the final result, we use a classic technique to implement a deque with two stacks (see e.g., Hoogerwood [15] combined with a deamortization scheme from Chuang and Goldberg [9].

1.4 Overview

In Section 2 we introduce some notation. In Section 3 we give our algorithm for finding occurrences of a streamed pattern P in strings of length $\mathcal{O}(|P|)$. In Section 4 we show how to solve the version where the string S only grows by appending characters to the right. Section 5 contains our new improved solution to the streaming sliding window string indexing (SSWSI) problem. Finally, in Section 6 we give our solution to the streaming dynamic window string indexing (SDWSI) problem.

2 Preliminaries

Given a string S of length n over an alphabet Σ, the ith character is denoted $S[i]$, and the substring starting at $S[i]$ and ending at $S[j]$ is denoted $S[i..j]$. The substrings of the form $S[i..n]$ are the *suffixes* of S. The reverse of a string S is the string $\mathsf{rev}(S) = S[n]S[n-1]\cdots S[1]$.

The *suffix tree* [23] T over a string $S[1..n]$ is the compact trie of all suffixes of $S\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of S, and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $\mathcal{O}(n)$ space by implicitly representing the string associated with each edge using two indices into S. Farach-Colton, Ferragina, and Muthukrishnan [11] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ. The *suffix array* L of a string S is the array where $L[i]$ is the starting position of the ith lexicographically smallest suffix of S. Note that $L[i]$ corresponds to the ith leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string spelled out by the root-to-v path. The descendant leaves of v exactly correspond to the suffixes of S that start with s_v, and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v.

▶ **Definition 7** (Periods). *We say that a positive integer p is a* period *of a string S if $S[i] = S[i+p]$ for all $i = 1, \ldots, |S| - p$. A string S is* periodic *if its smallest period is at most $|S|/2$.*

For a periodic pattern P with the smallest period p, we say $a_1, \ldots, a_k$ form a *chain* of occurrences of P in S if $P = S[a_i..a_i + |P| - 1]$ for $i = 1, \ldots, k$ and $a_i - a_{i-1} = p$ for $i = 2, \ldots, k$. The following (known) lemma is an easy consequence of the periodicity lemma [13].

▶ **Lemma 8.** *Let $a_1 < a_2 < \ldots < a_k$ be all occurrences of P in S, where $|S| \leq 2|P|$. If $k \geq 3$ then $a_1, a_2, \ldots, a_k$ form a chain.*

3 Matching Long Streaming Patterns

In this section we describe an algorithm that can find and compactly report all occurrences of a streamed pattern P in a string S of length $\mathcal{O}(|P|)$ in worst-case constant time per streamed character.

We are given an integer m and a string S of length $\mathcal{O}(m)$ supporting random access in constant time. We now receive a streamed pattern P of length between m and $3m$. We want to find all occurrences of P in S using constant time per streamed character. Since we do not know the precise length of P before we receive the last character our algorithm must be able to report all occurrences of the current P in constant time after receiving the first m

characters. Note that, since $|S| = \mathcal{O}(m)$ there is a constant number of occurrences of P in S unless P is periodic. If P is periodic then, by Lemma 8, the occurrences can be described by a constant number of chains.

Algorithm

The algorithm works in three phases. The ith phase starts after $i \cdot \lfloor m/4 \rfloor$ of characters of P have been streamed for $i \in \{1, 2, 3\}$. The third phase has two variants based on the periodicity of $P[0..\lfloor m/4 \rfloor - 1]$. Throughout the phases, we store the streamed characters of P.

Phase 1. Starts after the first $\lfloor m/4 \rfloor$ characters have arrived. We build a KMP automaton [16] of $P[0..\lfloor m/4 \rfloor - 1]$. When we have built the KMP automaton, we check if $P[0..\lfloor m/4 \rfloor - 1]$ is periodic and find its smallest period p.

Phase 2. Starts after $2\lfloor m/4 \rfloor$ characters have arrived. Find all occurrences of $P[0..\lfloor m/4 \rfloor - 1]$ in S using the KMP automaton. If $P[0..\lfloor m/4 \rfloor - 1]$ is not periodic then there is a constant number of occurrences and they are maintained explicitly. If $P[0..\lfloor m/4 \rfloor - 1]$ is periodic then, by Lemma 8, the occurrences can be described by a constant number of chains. In more detail, let $P = S[a_i..a_i + |P| - 1]$ be the previous occurrence and $P = S[a_{i+1}..a_{i+1} + |P| - 1]$ be the next occurrence. If $a_i + p = a_{i+1}$ then we extend the last chain by a_{i+1}, and otherwise we create a new chain initially consisting of only a_{i+1}.

Phase 3. Starts after $3\lfloor m/4 \rfloor$ characters have arrived. There are two cases depending on whether $P[0..\lfloor m/4 \rfloor - 1]$ is periodic or not.

Non-periodic. Extend the match of each occurrence from phase 2 simultaneously by explicitly matching each character of $P[\lfloor m/4 \rfloor..|P|]$. For each streamed character of P, match 4 characters until we have caught up to the stream. When the stream ends, report all occurrences.

Periodic. We match in each chain simultaneously as follows. Let $a_1, \ldots, a_k$ be a chain. We match against one occurrence in the chain, matching 4 characters at a time for each streamed character of P as in the non-periodic case. Let the current occurrence we are checking be occurrence j. Initially, $j = k$ and $i = \lfloor m/4 \rfloor$. As long as $i < |P|$ we do the following: Compare $P[i]$ and $S[a_j + i]$. If we match we set $i = i + 1$ and continue. Otherwise, there are two cases. If the mismatched character $P[i]$ is a continuation of the period and $j > 1$, we set $j = j - 1$, $i = i + 1$, and continue. Otherwise, we stop matching in this chain. While matching, we also check if P still has period p.
When the stream ends there are two cases. If P does not have period p (P is non-periodic or its smallest period is greater than p), then return the occurrence a_j. If P has period p we return the chain $a_1, \ldots, a_j$.

Analysis

Constructing a KMP automaton of $P[0..\lfloor m/4 \rfloor - 1]$ takes $\mathcal{O}(m)$ time. We can find the periodicity through the KMP in $\mathcal{O}(m)$ time. The number of characters streamed in phase 1 is $2\lfloor m/4 \rfloor - \lfloor m/4 \rfloor$, and thus we spent $\mathcal{O}(m)/\lfloor m/4 \rfloor = O(1)$ time per character in phase 1.

In phase 2, we match the KMP automaton of $P[0..\lfloor m/4 \rfloor - 1]$ against S. Since the length of S is $\mathcal{O}(m)$ and the number of characters streamed in phase 2 is $3\lfloor m/4 \rfloor - 2\lfloor m/4 \rfloor$, we spent $\mathcal{O}(m)/\lfloor m/4 \rfloor = \mathcal{O}(1)$ time per character in phase 2.

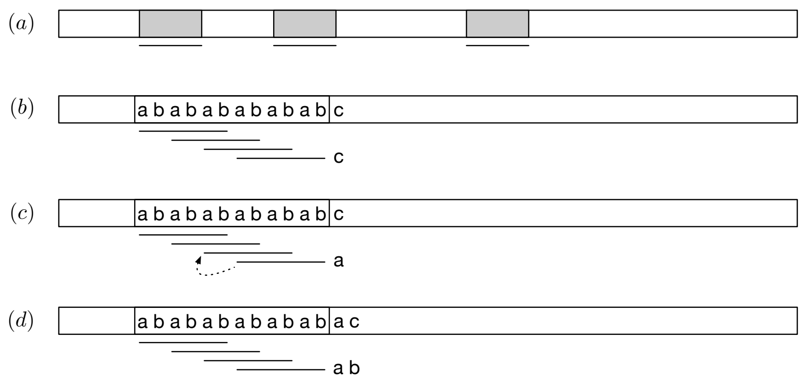

(a)
(b)
a b a b a b a b a b a b c
c
(c)
a b a b a b a b a b a b c
a
(d)
a b a b a b a b a b a b a c
a b

Figure 1 Phase 3. In (a) the pattern is non-periodic and we continue matching from each occurrence of $P[0..\lfloor m/4\rfloor - 1]$ (marked with gray). (b)-(d) show different cases of the periodic case. Here $P[0..\lfloor m/4\rfloor - 1] = ababab$. In (b) we match $P[i] = c$ and thus continue matching from this position. Since $P[i] = c$ is not a continuation of the period, we will never shift back to the previous occurrence. In (c) we mismatch and $P[i] = a$ is a continuation of the period, so we shift to the previous occurrence in the chain and keep matching. In (d) we mismatch and the and $P[i] = c$ is not a continuation of the period, so we stop.

In phase 3, we extend each match from phase 2. Since these are occurrences of a pattern of length at least $\lfloor m/4\rfloor$ and S has length $\mathcal{O}(m)$, by Lemma 8 the number of occurrences or chains from phase 2 is constant. Since we match at most 3 characters at a time per occurrence or chain, we use $\mathcal{O}(1)$ time per streamed character in phase 3.

The space of the KMP automaton is linear, and in addition to that, we only need space for the strings S and P and a constant number of positions (set of possible occurrences) in S. Thus the space is $\mathcal{O}(m)$.

Correctness. For correctness, assume that while matching against chain $a_1, \ldots, a_k$ we have a mismatch $S[a_j + i] \neq P[i]$. If $P[0..i]$ has period p, then since $S[a_j..a_j + i - 1] = P[0..i-1]$ and $S[a_{j'}..a_{j'} + i - 1 + p] = P[0..p] \cdot S[a_{j'+1}..a_{j'+1} + i - 1]$ for $1 \leq j' < j$, then $a_{j'}$ is a starting position of $P[0..i]$. Otherwise, by the same argument, $P[0..i]$ has no starting position in the chain. Let $P[0..\ell]$ be the longest prefix of P such that $P[0..\ell]$ has period p and let $a_1, \ldots, a_j$ be all the starting positions of $P[0..\ell]$ in a chain. If $\ell \neq |P| - 1$, then the remaining part of P is non-periodic, and by the same argument as before, only a_j can be an occurrence of P, which we match explicitly against.

When we enter phase 3 we have matched against $P[0..\lfloor m/4\rfloor - 1]$ and $3\lfloor m/4\rfloor$ characters have been streamed. The earliest time the stream can terminate is after m characters. Since we match up to 4 characters at a time, by the time the stream can terminate, we could have matched $\lfloor m/4\rfloor + 4(m - 3\lfloor m/4\rfloor) > m$ characters, and thus we catch up to the stream before it can terminate.

In summary, we have shown the following.

▶ **Lemma 9.** *We are given an integer m and a string S of length $\mathcal{O}(m)$ supporting random access in constant time and a streamed pattern P of length between m and $3m$. For each arriving character of P we use constant time, and when the stream ends we output all the occurrences of P in S in constant time. If P is periodic, we output the occurrences as the $\mathcal{O}(1)$ chains describing all occurrences. The algorithm uses $\mathcal{O}(m)$ space.*

Note that we can replace KMP with any real-time pattern matching algorithm.

4 Matching Streaming Patterns with Append

In this section, we show how to maintain the string S under appending characters (adding characters at the right end using $\mathsf{AddRight}(a)$) while supporting $\mathsf{Report}(P)$.

Data structure

The data structure consists of a suffix tree over the reverse string of S, i.e., $\mathsf{rev}(S)$. We utilize the online suffix tree algorithm to maintain the suffix tree. To perform $\mathsf{AddRight}(a)$, we use the online suffix tree algorithm to insert the new suffix $a \cdot \mathsf{rev}(S)$ in the suffix tree by adding a new edge (v, w). Furthermore, if the edge (v, w) splits an edge in the suffix tree, i.e., the node v is a new node, then we store a pointer to the leaf w in node v.

Rollback. When we answer $\mathsf{Report}(P)$ queries, we will modify the data structure, but to quickly return the data structure to the state it had before the query, we do the following. We store three values for each memory cell c used by the data structure:

- The value v_c that cell c has when we are not processing a query.
- The value q_c that cell c has when we are processing a query.
- The timestamp t_c of the last query that modified the cell c. Initially, $t_c = -1$.

When we process the t'th query, if we access cell c, we first check if q_c is outdated by checking t_c. If $t_c < t$ then we access v_c. Otherwise, we access q_c. Whenever we modify a cell c during the query, we set $t_c = t$ and update q_c.

Query

To perform a query $\mathsf{Report}(P)$, we do the following. We prepend $\mathsf{rev}(S)$ with "\$" and then prepend each streamed character from P when we receive it. When we prepend the last character of P, we get the edge (v, w) that is added to insert the new suffix $\mathsf{rev}(P)\$\mathsf{rev}(S)$ in the suffix tree. If the first character of the string on edge (v, w) is "\$" then all other children of v are occurrences of P in S. Otherwise, there are no occurrences of P in S.

To report all occurrences in worst-case constant time per reported occurrence, we do the following. We do a depth first traversal of the subtree rooted at v, visiting four nodes at each time step. We get an occurrence for each node we visit, either by a leaf we visit, or the pointer to a leaf stored in an internal node visited. To avoid reporting the same occurrence twice, we keep an array of size w, storing which indices have been reported in the current $\mathsf{Report}(P)$ query. If we find multiple new occurrences in a single time step, we output one and store the remaining in a buffer. If we do not find any new occurrences in a time step we output an occurrence from the buffer.

Analysis

For each $\mathsf{AddRight}(a)$, we use the online suffix tree algorithm to find the new edge (v, w) in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time. We can identify if v is a new node in constant time by checking the number of children of v and update its stored pointer in constant time. Thus, each $\mathsf{AddRight}(a)$ takes $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time.

For each $\mathsf{Report}(P)$, we spend $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ per character in P that we prepend to the string. When we traverse the subtree rooted at v, we visit at most four nodes per reported occurrence. Thus, each $\mathsf{Report}(P)$ query uses $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ per character in P and an additional $\mathcal{O}(1)$ time per reported occurrence.

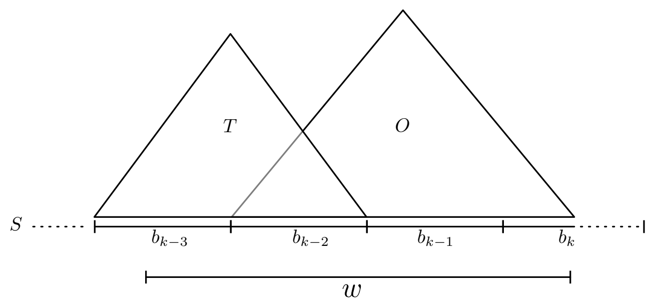


Figure 2 The data structures over the window.

The online suffix tree algorithm uses linear space and both the buffer and the array storing reported indices have size $\mathcal{O}(w)$. Thus the total space is $\mathcal{O}(w)$.

Correctness. Let (v, w) be the edge that is added to insert the new suffix $\mathsf{rev}(P)\$\mathsf{rev}(S)$ in the suffix tree. If the first character on the edge (v, w) is "\$", then the string on the path from the root of the suffix tree to v is $\mathsf{rev}(P)$. Since the suffix tree is built on the string $\mathsf{rev}(P)\$\mathsf{rev}(S)$, then all descendant leaves of v besides w, are occurrences of P in the string S.

Let t be the current time step since we started reporting occurrences. After the t^{th} time step, we have visited at least $2t$ nodes (or all the nodes in the subtree if the size of the subtree is less than $2t$). Since each leaf is stored in at most one internal node and the number of nodes in the subtree rooted at v is $2occ - 1$, then after the t^{th} time step, the number of unique occurrences found is at least t. Thus, at each time step, we either find an occurrence or an occurrence is stored in the buffer.

▶ **Lemma 10.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of streaming dynamic window string indexing operations on S: AddRight and Report in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time per character in $\mathcal{O}(w)$ space. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

5 Streaming Sliding Window String Indexing

In this section, we show Theorem 3. In the SSWSI problem, we can see it as we have a string S that is being streamed, and the string S' that we maintain corresponds to the window, i.e., after the i^{th} update $S' = S[i - w + 1..i]$. We partition the streamed string S into consecutive blocks $b_0, b_1, b_2, \ldots$ of length $B = \lceil w/3 \rceil$ (except possibly the last one), i.e., $b_j = S[j \cdot B..(j + 1) \cdot B - 1]$. Let b_k be the block containing the last streamed character $S[i]$, that is $k = \lfloor i/B \rfloor$. Thus the whole window is contained in $b_{k-3} \cdot b_{k-2} \cdot b_{k-1} \cdot b_k$.

Our data structure for the SSWSI problem consists of two structures. A static data structure T for $b_{k-3} \cdot b_{k-2}$ and an online data structure O for the string $b_{k-2} \cdot b_{k-1} \cdot b_k$. We utilize Lemma 10 for the online data structure O. See Figure 2. Furthermore, we store S' in a rotated array.

Static data structure

The static data structure consists of the following.

- A suffix tree.
- A suffix array A containing the leaves of the suffix tree in left to right order.
- A range maximum query data structure on the array A.
- Furthermore, each node in the suffix tree stores the range of its descendant leaves in the array A.

This is the same as the structure used in Bille et al. [6]. For the suffix tree, we use the same online suffix tree algorithm as used for the online data structure to build it. For the range maximum query, we use a data structure using linear space and preprocessing time and constant query time [5]. To perform a query P in the static data structure, we do the following. We search for P in the suffix tree, reading one character at a time. Let $[\ell, r]$ be the range of leaves stored in the locus of P. We perform a range maximum query in $A[\ell..r]$ to find the rightmost occurrence x. If x is not in the window, then there are no occurrences starting in $b_{k-3}b_{k-2}$ that are in the window. Otherwise, we report x and recurse on $A[\ell..x-1]$ and $A[x+1..r]$. To use worst-case constant time per reported occurrence we do three range maximum queries in one time step and keep a buffer of found but unreported occurrences as in Section 4. It then follows from a similar argument that we can report each occurrence in worst-case constant time.

The combined query

To find all occurrences of a streamed pattern P in S' we do the following. Assume that P has length $m \leq B$. We later show how to handle the case where $m > B$. We query T and O in parallel with the streamed characters of P. When P has arrived, we first report occurrences from the online data structure O. While we report occurrences from the online data structure, we prepare the static data structure to report occurrences that begin in b_{k-3} and are in the window, since occurrences that begin in b_{k-2} are also reported by the online structure. The static data structure does not report the occurrences from the right (even though the first reported occurrence is the rightmost). However, we can modify its reporting procedure so that all the occurrences in b_{k-2} will be reported before b_{k-3} as follows. If the currently considered occurrence falls within b_{k-1} we report it and recurse. Otherwise, we pause the recursion and add the current occurrence to a list. Next, we iterate over the list of paused recursive calls and resume each of them one-by-one. For each reported occurrence in the online data structure, we also process an occurrence in the static data structure until we get to the occurrences in b_{k-3}. When we have finished reporting occurrences in the online data structure O, we resume reporting the occurrences in b_{k-3}.

If $m > B$, we use Lemma 9 to find occurrences of P in S'. We note that, because of how these occurrences are reported (as either a constant number of explicitly given positions or a constant number of arithmetical progressions), it is trivial to filter out the occurrences that are anyway reported by the static or the online data structure. In summary, we have shown that we can report all occurrences of P in S'.

Rebuilding

We rebuild the structures in the background to keep the static and online data structure partially and completely inside the window, respectively. When block b_k begins, we start building the static data structure T' for $b_{k-2}b_{k-1}$ and the online data structure O' for $b_{k-1}b_k$. Since block b_{k-2} and b_{k-1} have been completed, we can use the online suffix tree algorithm to construct a suffix tree for $b_{k-2}b_{k-1}$. We construct T' and O' at a pace such that when the b_{k+1} begins, we have completed the structures and can swap T with T' and O with O'.

Analysis

We update the online data structure for each Update operation in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time. We rebuild the static and online suffix trees once per block. It takes $\mathcal{O}(t_{\text{suff}}(w, \sigma) \cdot w)$ time to rebuild the static and online suffix trees. To build the range maximum query data structure and add ranges to the suffix tree, we spent an additional $\mathcal{O}(w)$ time. We augment the suffix tree with a dictionary data structure in each node over the first character of the labels of the outgoing edges. This takes $\mathcal{O}(t_{\text{dict}}(w, \sigma) \cdot w)$ time. Thus we use $\mathcal{O}((t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma)) \cdot w)/B = \mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ time on rebuilding for each Update.

For a query P, we spend $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time in the online data structure for each character in P by Lemma 10. In the static data structure, we traverse the suffix tree, using $\mathcal{O}(t_{\text{dict}}(w, \sigma))$ time for each character in P. By Lemma 9, we spend $\mathcal{O}(1)$ time per streamed character of P to report long patterns. Thus, in total, we spend $\mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ time per character plus additional constant time per reported occurrence.

Both the online and the static data structures use linear space. We have a constant number of such data structures at a time each over a substring of $b_{k-3}b_{k-2}b_{k-1}b_k$. Since $|b_{k-3}b_{k-2}b_{k-1}b_k| = \mathcal{O}(w)$, this use $\mathcal{O}(w)$ space in total. All other components, i.e. the buffer and the algorithm for the long patterns, use $\mathcal{O}(w)$ space. Thus the total space is $\mathcal{O}(w)$.

Correctness. The static and online data structures cover the entirety of S' and overlap by B characters, and thus they report all occurrences of pattern P in S' if the length of P is no more than B. When we report, we need time to discard the occurrences in b_{k-2} reported by the static data structure. Since the number of occurrences in b_{k-2} is no more than the occurrences in $b_{k-2}b_{k-1}b_k$, we have time to discard the occurrences in the static data structure if we report the occurrences in the online data structure first. If the pattern is longer than B, then the algorithm of Lemma 9 is ready to report occurrences since the sliding window has size $O(w)$ and P has a length between $B = \lceil w/3 \rceil$ and w.

6 Streaming Dynamic Window String Indexing

In this section, we consider the more general case, where we want to maintain the text S under adding and removing characters at either end, while still supporting Report(P) queries. We first show how to maintain S under only prepending/appending characters, and then extend this to the general case.

Directly from our definition of an online suffix tree construction algorithm we have:

▶ **Lemma 11.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of streaming dynamic window string indexing operations on S: AddLeft and Report in $\mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ time per character in $\mathcal{O}(w)$ space. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

To get reporting in worst-case constant time per reported occurrence we do as in Section 4.

6.1 Prepend and Append

The current text S is represented as a concatenation $S = S_1S_2$. We store the characters of S_1 and S_2 on a doubly-linked list L_1 and L_2, respectively, and maintain the structure from Lemma 11 for S_1, and the structure from Lemma 10 for S_2. Initially, S_1 and S_2 are empty. The operation S.AddLeft(a) prepends a to L_1 and calls S_1.AddLeft(a), while S.AddRight(a) appends a to L_2 and calls S_2.AddRight(a). The operation S.Report(P) needs to consider occurrences of P in S_1, S_2, and straddling between S_1 and S_2. The first and the second case is implemented by running S_1.Report(P) and S_2.Report(P) in parallel. The third case is implemented by proceeding in phases $0, 1, 2, \ldots$. Phase k corresponds to $m \in (3^{k-1}, 3^k]$. In each phase, we maintain two instances of the procedure from Lemma 9. We maintain an invariant that the text S available to the first instance is $S_1[(|S_1|-3^k+1)..|S_1|]S_2[1..3^k]$ (length-3^k suffix of S_1 concatenated with length-3^k prefix of S_2), and after having read $P[i]$, where $i \in (3^{k-1}, 3^k]$, the pattern fed to the first instance is simply the whole $P[1..i]$. Thus, for any $m \in (3^{k-1}, 3^k]$ the first instance allows us to report all occurrences of P that straddle between S_1 and S_2. Meanwhile, we maintain the following invariant concerning the second instance. While reading $P[i]$, for $i \in (3^{k-1}, 2 \cdot 3^{k-1}]$, we create an array storing $S_1[(|S_1|-3^{k+1}+1)..|S_1|]S_2[1..3^{k+1}]$ (length-3^{k+1} suffix of S_1 concatenated with length-3^{k+1} prefix of S_2). This can be done by traversing L_1 from the last element and L_2 from the first element, spending constant time for every such i. Thus, after reaching $i = 2 \cdot 3^{k-1}$, the array stores the text that we will need in the next phase. Then, while reading $P[i]$, for $i \in (2 \cdot 3^{k-1}, 3^k]$, we send three characters of the pattern to the second instance for every new character of the pattern. More precisely, after receiving $P[i]$ we send $P[3(i-2\cdot 3^{k-1})-2], P[3(i-2\cdot 3^{k-1})-1]$, and $P[3(i-2\cdot 3^{k-1})]$ to the second instance. Thus, after reaching $i = 3^k$, the second instance has received the whole current pattern $P[1..3^k]$, so we can swap the instances, reset the second instance, and proceed to the next phase.

6.2 General case

We first explain how to extend Lemma 10 to support both AddLeft(a) and RemoveLeft(), and similarly how to extend Lemma 11 to support both AddRight(a) and RemoveRight(). In both cases, we use the same simple idea.

▶ **Lemma 12.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of SDWSI operations on S:*

- *either AddLeft, RemoveLeft, and Report in $O(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ per character in $\mathcal{O}(w \cdot (t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma)))$space,*
- *or AddRight, RemoveRight, and Report in $O(t_{\text{suff}}(w,\sigma))$ per character, in $\mathcal{O}(w \cdot t_{\text{suff}}(w,\sigma))$ space.*

Furthermore, Report uses additional worst-case constant time per reported occurrence.

Proof. Supporting AddLeft and RemoveLeft or AddRight and RemoveRight can be seen as providing the possibility of undoing the most recent updates. We consider a structure that can be modified with an Update operation and denote the empty structure by $\perp$. Then, the current structure will be always $S = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_k$. We want to either modify it to obtain $S' = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_k.\mathsf{Update}_{k+1}$, or (if $k \geq 1$) undo the most recent update to obtain $S' = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_{k-1}$. This can be implemented as follows. We maintain a stack consisting of k records. The i-th record stores (in e.g. a linked list) all modifications made when executing Update_i on $\perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots, \mathsf{Update}_{i-1}$. Each modification is described by specifying the address of a memory cell, and its value before the update. Let $u(w,\sigma)$ be the time for an update. Because any update modifies only $u(w,\sigma)$ memory cells, the space usage is $\mathcal{O}(w \cdot u(w,\sigma))$. Then, to undo the most recent update we retrieve the top record and revert all memory cells modified by the most recent update to their original values. This takes $\mathcal{O}(u(w,\sigma))$ time. During an update, we push a new record onto the stack and store all modified memory cells there. This also takes $\mathcal{O}(u(w,\sigma))$ time. The results now follow from plugging in the update times from Lemma 10 and Lemma 11. ◀

We are now ready to describe the general case. We use the well-known idea of implementing a deque with two stacks, see e.g. Hoogerwoord [15]. We briefly describe this idea. The current deque is represented as $S = \mathsf{rev}(S_1).S_2$, where S_1 and S_2 are stacks, rev denotes the reversal, and . the concatenation. Then, prepending an element is implemented by pushing it onto S_1 while appending an element is implemented by pushing it onto S_2. Removing the first element is implemented by popping it from S_1, while removing the last element is implemented by popping it from S_2. This works as long as both S_1 and S_2 are non-empty. As soon as one of them, say S_1, becomes empty, we rebuild the structure by distributing the elements stored on S_2 evenly between S_1 and S_2. It is easy to prove that the amortized cost of the rebuilding is constant, by defining the potential of the structure as $||S_1| - |S_2||$.

However, we need a worst-case efficient version. Chuang and Goldberg [9] provide a particularly clean description of how to modify the amortized version to obtain an implementation where every operation takes worst-case constant time. We refer the reader to their original description and only describe what is stored in their implementation. The current deque is represented as $S = \mathsf{rev}(S').\mathsf{rev}(S).B.B'$, where S', S, B and B' are stacks. Additionally, the structure maintains additional stacks $auxS, auxB$, and $extraS, newS, newB, extraB$. The rebuilding is done incrementally, and while this is being done every prepended element is pushed onto both S' and $extraS$, while every appended element is pushed onto both B' and $extraB$. Then, after the rebuilding has finished, the current deque is represented as $S = \mathsf{rev}(extraS).\mathsf{rev}(newS).newB.extraB$.

We built on the worst-case efficient implementation of Chuang and Goldberg [9] to prove Theorem 5. We maintain the current string S in a deque, and represent it as $S = \mathsf{rev}(S').\mathsf{rev}(S).B.B'$. Additionally, for each of the stacks S', S, B, B' and similarly $extraS, newS, newB, extraB$, we maintain a doubly-linked list storing its elements. Note that this would not be allowed in a purely functional implementation, which is the model assumed by Chuang and Goldberg [9], but we are not making any such assumption. Next, for each of the stacks $S', S, extraS, newS$ we maintain an instance of Lemma 12 with $\mathsf{AddLeft}$ and $\mathsf{RemoveLeft}$ while for each of the stacks $B, B', newB, extraB$ we maintain an instance of Lemma 12 with $\mathsf{AddRight}$ and $\mathsf{RemoveRight}$. This makes the update time $\mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ and space $\mathcal{O}(w \cdot (t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma)))$. To implement $\mathsf{Report}(P)$, we separately consider occurrences of P inside $\mathsf{rev}(S'), \mathsf{rev}(S), B$ and B' by running $\mathsf{Report}(P)$ in parallel for each of the maintained instances. It remains to consider occurrences of P that straddle between $\mathsf{rev}(S')$ and $\mathsf{rev}(S).B.B'$, or between $\mathsf{rev}(S').\mathsf{rev}(S)$ and $B.B'$. or $\mathsf{rev}(S').\mathsf{rev}(S).B$ and B'. Each of these cases is solved as described in Section 6.1 by observing that we can provide access to the corresponding doubly-linked lists by (temporarily) concatenating some of the maintained doubly-linked lists. All three instances are run in parallel, so the overall additional time per character of P is constant.

References

1 Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014. `doi:10.1137/110836377`.

2 Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005. `doi:10.1007/11575832_9`.

3 Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008. URL: `http://dl.acm.org/citation.cfm?id=1347082.1347201`.

4 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

5 Michael A. Bender and Martín Farach-Colton. The lca problem revisited. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 88–94, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

6 Philip Bille, Johannes Fischer, Inge Li Gørtz, Max Rishøj Pedersen, and Tord Joakim Stordalen. Sliding window string indexing in streams. In *Proc. 34th CPM*, pages 4:1–4:18, 2023. doi:10.4230/LIPICS.CPM.2023.4.

7 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. doi:10.1016/j.jda.2012.07.003.

8 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. doi:10.3390/a11080118.

9 Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead thring machines, and purely functional programming. In *Proc. 6th FPCA*, pages 289–298, 1993.

10 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th ICALP*, pages 6–19, 1990.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

12 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.

13 N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16(1):109–114, 1965.

14 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th CPM*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

15 Rob R. Hoogerwoord. A symmetric set of efficient list operations. *J. Funct. Program.*, 2(4):505–513, 1992.

16 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

17 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd FOCS*, pages 283–292, 2012. doi:10.1109/FOCS.2012.79.

18 S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994. doi:10.1145/195058.195170.

19 Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017. doi:10.1007/s00453-016-0199-7.

20 N. Jesper Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Lund University, Sweden, 1999. URL: http://lup.lub.lu.se/record/19255.

21 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.

22 Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008. doi:10.1007/978-3-540-89097-3_12.

23 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.

Tight Bounds for Compressing Substring Samples

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Christian Mikkelsen Fuglsang
Technical University of Denmark, Lyngby, Denmark

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Abstract

We consider the problem of compressing a set of substrings sampled from a string and analyzing the size of the compression. Given a string S of length n, and integers d and m where $n \geq m \geq 2d > 0$, let $\mathrm{SCS}(S, m, d)$ be the string obtained by sequentially concatenating substrings of length m sampled regularly at intervals of d starting at position 1 in S. We consider the size of the LZ77 parsing of $\mathrm{SCS}(S, m, d)$, in relation to the size of the LZ77 parsing of S. This is motivated by genome sequencing, where the mentioned sampling process is an idealization of the short-read DNA sequencing. We show the following upper bound:

$$|\mathrm{LZ77}(\mathrm{SCS}(S, m, d))| \leq |\mathrm{LZ77}(S)| + 2\frac{n-m}{d}.$$

We also give a lower bound showing that this is tight. This improves previous results by Badkobeh et al. [ICTCS 2022], and closes the open problem of whether their bound can be improved.

Another natural question is whether assuming that all letters in S are part of a sample, it is always the case that $|\mathrm{LZ77}(S)| \leq |\mathrm{LZ77}(\mathrm{SCS}(S, m, d))|$. Surprisingly, we show that there is a family of strings such that $|\mathrm{LZ77}(\mathrm{SCS}(S, m, d))| = |\mathrm{LZ77}(S)| - 1$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compression, Algorithms, Lempel-Ziv

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.9

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).

1 Introduction

The recent revolution in short-read sequencing technologies has made the acquisition of large genome sequences significantly cheaper and faster. This has led to a drastic increase in the amount of genome data and by extension the need to compress these vast datasets. Numerous ambitious sequencing projects (and existing databases) are currently underway, such as the recent Earth BioGenome Project [30, 31], the 10K Vertebrate Genomes Project [33], and The International Genome Sample Resource (IGSR) [9] built on the foundation of the 1,000 Genomes Project, among many others. These projects aim to create large databases of strings (genomes) that vary only slightly from each other and as a result, contain large repetitions of data.

The vast amount of genome data in these databases makes the importance of compression and fast random access especially apparent. There are several tools that are popular for compressing genome data, some of which are the standard `gzip` and `7zip` compressors. The

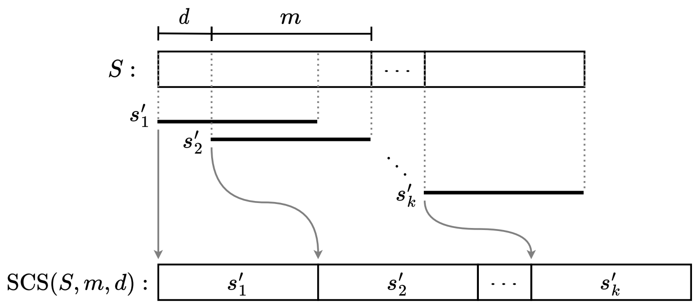


Figure 1 The sampling process for constructing $\mathrm{SCS}(S, m, d)$.

basis of these is Lempel-Ziv (LZ77) parsing [29, 43], typically followed by an entropy encoding. Additionally, there are several read-set specific compressors [1, 5, 7, 16], at least one of which is also a Lempel-Ziv type compressor.

A natural question is to consider the effects the sampling process has on the compressibility of the resulting data. This question was originally posed by Badkobeh et al. [3], and to our knowledge, this is the only instance before us where analysis in this area has been undertaken. In particular, we consider the problem of compressing a set of substrings sampled from a string and analyzing the size of the compression. Specifically, we will analyze the size of the LZ77 parsing (which we define later) as it and variations thereof are widely used in many relevant compressors.

The analysis is achieved by defining an idealized model, which mimics the genome sequencing process. The idealization occurs since we do not consider errors, i.e., insertions, deletions, and substitutions of letters, which are introduced by short-read sequencing technologies. Moreover, we do not consider variations in the distance between or the length of samples across the genome. In practice, these fluctuate for a variety of reasons [8]. However, analysis of these technologies, even in an idealized setting, can give much insight into their effects on compressibility.

We now define the idealized model, as described by Badkobeh et al. [3], which we use throughout the remainder of the paper. Given a string $S = S[1, n] = S[1]S[2] \cdots S[n]$ of length n, and integers d and m where $n \geq m \geq 2d > 0$, let $\mathrm{SCS}(S, m, d)$ be the string obtained by sequentially concatenating substrings of length m sampled regularly at intervals of d starting at position 1 in S. The sampling process and construction of $\mathrm{SCS}(S, m, d)$ is shown in Figure 1.

Formally, we have a total of $k = 1 + \lfloor (n - m)/d \rfloor$ samples, where $\lfloor x \rfloor$ is the largest integer which is smaller than or equal to x. The jth sample s'_j consists of the m consecutive letters in S starting at position $(j-1)d + 1$. Notice that the last $(n - m \mod d)$ letters in S are not part of any sample due to rounding. The samples are concatenated sequentially to form the string $\mathrm{SCS}(S, m, d) = s'_1 s'_2 \cdots s'_k$.

The connection between this model and genome sequencing is very well described by Badkobeh et al. [3]. In short, $\mathrm{SCS}(S, m, d)$ corresponds to a file of the short-read sequences, which is the typical output of a sequencing experiment (e.g. the FASTQ format). Here, m corresponds to the *read length* and m/d to the *coverage*, i.e., the average number of samples that cover a position in S. The assumption that $m \geq 2d$ corresponds to a coverage of at least 2, which is a relevant case for DNA sequencing.

In this paper, we consider the size of the LZ77 parsing of $\mathrm{SCS}(S, m, d)$, in relation to the size of the LZ77 parsing of the original string S. In the remainder of this section we define the LZ77 parsing of a string, briefly present previous work, and state our results and techniques.

1.1 The LZ77 Parsing

The Lempel-Ziv (LZ77) parsing [29, 43] (also known as the LZ77 factorization) of a string is a fundamental part of data compression [10, 22, 14, 21], and for string processing such as detecting the periodicities of a string [2, 19].

The LZ77 parsing of S partitions S into a sequence of z_S substrings $\mathrm{LZ77}(S) := f_1, f_2, \ldots, f_{z_S}$ called phrases. The size of the LZ77 parsing of S is $|\mathrm{LZ77}(S)| = z_S$. The phrases are constructed greedily from left to right using the following rules. The ith phrase f_i with starting position p_i is encoded either as (a) the first occurrence of a letter in S, or (b) the longest substring with an occurrence in S before p_i. The compression of S occurs since the phrases of type (b) are encoded as a pair (r_i, l_i), where $r_i > 0$ is the distance from p_i to the beginning of the previous occurrence of f_i, and l_i is the length of f_i. This is the LZ77-variant given by Storer and Szymanski [40], whereas the original definition [43] always added an extra letter to the end of these phrases. Furthermore, we consider the variant of LZ77, where a previous occurrence referenced by a phrase is allowed to overlap with the corresponding phrase.

Naturally, we say that a phrase f_i *covers* an interval $[a, b]$ if and only if every element in $[a, b]$ is contained within the interval $[p_i, p_i + l_i - 1]$, corresponding to the range of f_i in S. Similarly, we say that the phrase *overlaps* the interval when the range contains at least one element in $[a, b]$.

Computing the LZ77 parsing is a very well-studied problem, and there are many algorithms solving it with various trade-offs. The simplest way to construct the LZ77 parsing is to build an index on the input string (e.g. a suffix tree or suffix array), and greedily from left to right find the longest prefix of the current suffix with an occurrence to the left of the current position. There has been lots of previous (and ongoing) research on LZ77 leading to practical and space-efficient computation [6, 13, 15, 23, 24, 18, 26, 20, 34, 37], parallel [38] and external computation [25], online parsing [35, 36, 39, 41], and more [12]. Other practical solutions include the sliding window LZ77 parsing [11, 27, 4], where the previous occurrence of a phrase is restricted to start no more than w letters away from the start position of the phrase, with w as a parameter.

Often these articles include performance metrics obtained experimentally by compressing collections of strings, such as DNA sequences (e.g. [25, 22, 21, 15, 23, 24, 34, 38, 35]), to emphasize the benefits of the corresponding compressor. This demonstrates that an important motivation for improving upon LZ77 factorization is among others to improve the storage of genome databases.

1.2 Previous Work

We will now consider previous results and the techniques that have been used. As mentioned earlier, Badkobeh et al. [3] originally posed the question of the effects of the sampling process on compressibility. They have shown that $|\mathrm{LZ77}(\mathrm{SCS}(S, m, d))| \leq m - d + 2|\mathrm{LZ77}(S)| + (2n - m)/d$.

The techniques they employ in their proof involve partitioning $\mathrm{SCS}(S, m, d)$ into several smaller intervals and examining the number of phrases incurred by each individually. More precisely, they consider the intervals given by the first $m - d$ letters followed by the last

d letters in each sample. In the former, they show that the first $m-d$ letters in the first sample trivially incur at most $m-d$ phrases and that the first $m-d$ letters in the remaining samples incur at most one phrase each. The last d letters in each sample are analyzed by defining a rather involved projection of the individual phrases in $\mathrm{LZ77}(S)$ onto $\mathrm{SCS}(S,m,d)$, and showing a bound on how many phrases are incurred by the projected intervals.

They conclude their paper by posing the open question of whether their upper bound can be improved.

1.3 Our Results

In this paper, we answer the question asked by Badkobeh et al. [3] in the affirmative and give tight upper and lower bounds improving upon theirs. More precisely, we show the following upper bound.

▶ **Theorem 1.** *Let S be a string of length n, then for all integers d and m where $n \geq m \geq 2d > 0$:*

$$|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| \leq |\mathrm{LZ77}(S)| + 2\frac{n-m}{d} .$$

Intuitively, the upper bound given in Theorem 1 states that there is no overhead for the first sample and that the remaining samples have an overhead of two phrases each. This bound is strictly better than that given by Badkobeh et al. [3]. In particular, consider what happens when $m = n$, i.e., when we only have a single sample. In this case, their upper bound is $(n - d + 2|\mathrm{LZ77}(S)| + n/d)$, whereas ours is $|\mathrm{LZ77}(S)|$. The latter is tight, since $\mathrm{SCS}(S,n,d) = S$ regardless of the choice of d.

The techniques we use in the proof of Theorem 1 are similar to those used by Badkobeh et al. [3], in the sense that we also partition $\mathrm{SCS}(S,m,d)$ into several smaller intervals which we consider individually. The primary differences are that we tightly analyze the phrases incurred by the entire first sample, and we do not define a projection for analyzing the last d letters of the remaining samples. Instead, we categorize the substrings of phrases incurred by dividing S during the sampling process. We show that these substrings incur at most one phrase each in $\mathrm{LZ77}(\mathrm{SCS}(S,m,d))$, and use this to give a bound on the total number of phrases. This new approach allowed us to significantly improve the upper bound.

Furthermore, we show that our upper bound is tight for any choice of $d \geq 3$.

▶ **Theorem 2.** *Let d and m be integers, where $d \geq 3$ and $m \geq 2d$. Then for all integers $n \geq m$ there exists a string S of length n such that:*

$$|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| = |\mathrm{LZ77}(S)| + 2\left\lfloor\frac{n-m}{d}\right\rfloor .$$

The primary difference between this and the upper bound is the floor after division. This is necessary here since we are referring to an exact number of phrases. We obtain Theorem 2 by constructing a string S of arbitrary length $n \geq m$ based on parameters d and m, and analyzing the compressibility of $\mathrm{SCS}(S,m,d)$ and the constructed string.

Another natural question is whether $|\mathrm{LZ77}(S)| \leq |\mathrm{LZ77}(\mathrm{SCS}(S,m,d))|$ is always the case, assuming that all letters in S are part of a sample, i.e., $n \geq m$ and $n \equiv m \pmod{d}$. The latter assumption is important since it is otherwise trivial to disprove (e.g. let S be m repetitions of a followed by a single b). According to our knowledge, this question has not been considered in detail until now. We show that there exists a family of strings where this is not the case, leading to the following surprising result.

▶ **Theorem 3.** *Let d and m be integers, where $d \geq 2$, $m \geq 2d$, and $m \equiv 0 \pmod{d}$. Then, there exists a string S of length $n \geq m$ where $n \equiv m \pmod{d}$ such that:*

$$|\mathrm{LZ77}(\mathrm{SCS}(S, m, d))| = |\mathrm{LZ77}(S)| - 1 .$$

We obtain Theorem 3 by constructing a string S of length $n = 3m - d$, and analysing the compressibility after sampling, in a similar fashion to the proof of Theorem 2.

In the following sections, we provide proof of our results. We prove the upper bound in Section 2, the corresponding lower bound showing that this is tight in Section 3, and the theorem on improved compressibility in Section 4. Finally, we finish the paper with concluding remarks and future work in Section 5.

2 Upper Bound

Let S be the given string of length n and let $S' := \mathrm{SCS}(S, m, d)$. We assume w.l.o.g. that every letter in S is part of some sample, i.e., $n \geq m$ and $n \equiv m \pmod{d}$, and partition each sample into two substrings such that $s'_j = u_j s_j$, where $|u_j| = m - d$ and $|s_j| = d$. Thus, $S' = u_1 s_1 u_2 s_2 \cdots u_k s_k$. In order to prove Theorem 1, we partition S' by considering the following three cases separately:

1. the first sample $s'_1 = u_1 s_1$,
2. the first $m - d$ letters u_j in every sample s'_j for $2 \leq j \leq k$, and
3. the last d letters s_j in every sample s'_j for $2 \leq j \leq k$.

This partitions S' into non-overlapping intervals. Similarly, we use this interpretation to write the given string equivalently as $S = u_1 s_1 s_2 \cdots s_k$, and define $z_S := |\mathrm{LZ77}(S)|$ and $z_{S'} := |\mathrm{LZ77}(S')|$ which will be useful when showing these cases.

Any LZ77 parsing contains exactly the same number of phrase starting positions as phrases. Therefore, by bounding the number of starting positions of phrases in the above intervals, we bound the total number of phrases in the LZ77 parsing of S'. As a property of LZ77 there are several substrings which incur at most one starting position of a phrase in the compression of S'. These substrings are categorized in Lemma 4 and Lemma 5, and we use these several times throughout the proof. This strategy is similar to that used by Badkobeh et al. [3], but we show a tighter bound.

▶ **Lemma 4.** *Let T be a string, and P be a substring of T, i.e., $P = T[i, j]$, where $1 \leq i \leq j \leq |T|$. If P has a previous occurrence in T starting at position $i' < i$, then the LZ77 parsing of T contains at most one starting position of a phrase in the interval $[i, j]$.*

Proof. Assume the LZ77 parsing contains more than one starting position in the interval $[i, j]$. Then the first of these phrases has a starting position p and length l, where $i \leq p \leq p+l-1 < j$. Since P has an occurrence in T at position $i' < i$, the substring $T[p, j]$ with length $j-p+1 > l$ also has an occurrence at $i' + p - i < p$. This contradicts the property that every phrase starting at position p covers the *longest* substring with an occurrence in T before p. ◀

▶ **Lemma 5.** *Let T be a string, and $f_1 f_2 \cdots f_{z_T}$ be the phrases in the LZ77 parsing of T. Then for every phrase f_i with starting position p_i which is not the first occurrence of a letter, and for every pair of integers (v, w) where $1 \leq v \leq w \leq l_i$, substring $f_i[v, w]$ has an occurrence in T before $p_i + v$.*

Proof. Consider phrase f_i. This is either the first occurrence of a letter, in which case the lemma trivially holds, or it has an occurrence in T at position $p_i - r_i$. Therefore, the substring $f_i[v, w]$ for any pair (v, w) where $1 \leq v \leq w \leq l_i$ must also have an occurrence in T at position $p_i - r_i + v < p_i + v$. ◀

As mentioned, we partition S' into several intervals. We consider these cases in the following paragraphs, whereafter we collect the results to give the final bounds.

Case 1. Consider the first sample s'_1. By definition, this is the substring sampled by the letters in the interval $[1, m]$ in S. We denote the interval as X and the number of phrases overlapping X in the LZ77 parsing of S as z_X. Intuitively, the LZ77 parsing of S' contains exactly z_X starting positions of phrases in that same interval. This is illustrated in Figure 2.

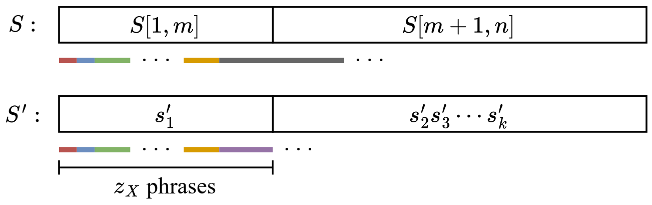


Figure 2 Intuition of the first z_X phrases in LZ77(S') compared to those in LZ77(S).

Formally, we consider the phrases f_i for $1 \leq i < z_X$ in LZ77(S). Since f_i only depends on the previous letters and $S[1, p_i + l_i - 1] = S'[1, p_i + l_i - 1]$, this phrase is exactly the same as the ith phrase in LZ77(S'). This is the case for every phrase except f_{z_X}. However, by Lemma 5 the substring $S[p_{z_X}, m] = S'[p_{z_X}, m]$ has a previous occurrence, and by Lemma 4 this incurs at most one starting position of a phrase, and the proof follows.

Case 2. Consider the first $m - d$ letters u_j in samples s'_j for every j, where $2 \leq j \leq k$. By definition of the jth sample, u_j is a repeat of the last $m - d$ letters in s'_{j-1}. This is illustrated in Figure 3.

Following directly from Lemma 4, these repeating intervals contain at most one starting position of a phrase and therefore contribute at most $k - 1 = (n - m)/d$ to the total number of phrases.

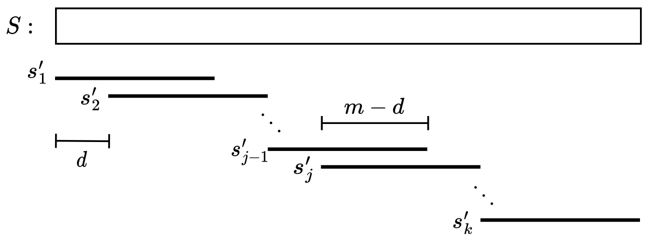


Figure 3 Illustration of the overlap between samples.

Case 3. Finally, consider the last d letters s_j in samples s'_j for every j, where $2 \leq j \leq k$. The positions of these in S' are illustrated in Figure 4.

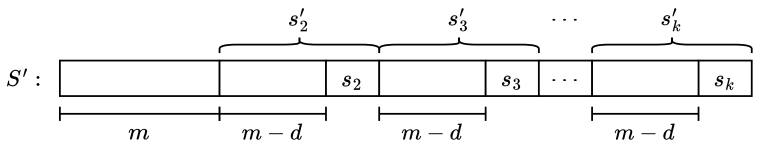


Figure 4 Illustration of the last d letters s_j in sample s'_j, for each j where $2 \leq j \leq k$.

Consider the phrases in LZ77(S) which encode these substrings, i.e., the phrases overlapping the interval $[m+1, n]$. There are at most $(z_S - z_X + 1)$ such phrases, where z_X is the number of phrases overlapping the interval $[1, m]$. This is trivially shown since phrase intervals are disjoint, and at most one phrase f_{z_X} might overlap both intervals in S. This is also illustrated in Figure 5.

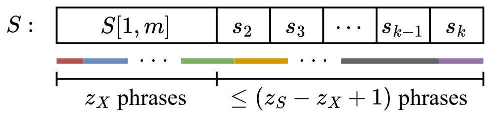


Figure 5 The number of phrases in LZ77(S) overlapping the interval $[m+1, n]$.

We denote these phrases as $f_{z_X}, f_{z_X+1}, \ldots, f_{z_S}$, and consider their overlap with each sample s_j. In particular, each phrase f_i which overlaps s_j either (i) has zero endpoints in s_j, (ii) ends in s_j, (iii) starts in s_j, or (iv) has both endpoints in s_j.

These cases are illustrated in Figure 6.

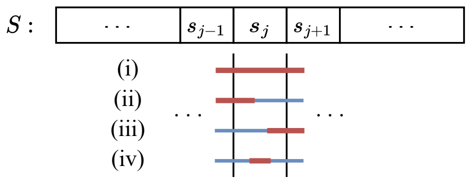


Figure 6 All cases where a phrase overlaps substring s_j. The phrase considered is marked in red.

We partition each phrase f_i into the largest substring for each s_j which does not cross the border from s_{j-1} to s_j, or from s_j to s_{j+1}. I.e., the substring in S given by the intersection between the intervals of f_i and s_j. This ensures that the substrings exist in S'. These substrings are collectively denoted as *phrase parts*, and exactly partition the last d letters of every sample except s'_1. This is illustrated in Figure 7. Notice that by definition the length of each phrase part is at most d.

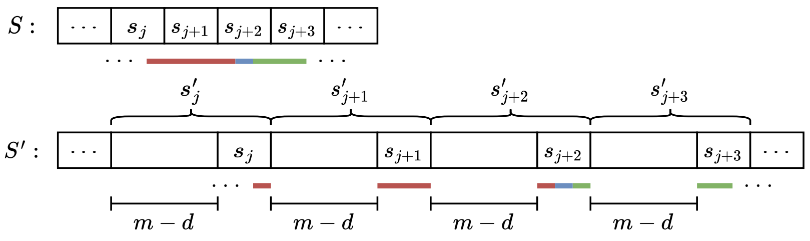


Figure 7 Example of phrase parts constructed from phrases.

▶ **Lemma 6.** *A phrase part incurs at most one phrase starting position in the LZ77 parsing of S'.*

Proof. Consider phrase part P constructed from phrase f_i and substring s_j. The lemma trivially holds if f_i is the first occurrence of a letter. Therefore, we assume w.l.o.g. that f_i has a previous occurrence in S. It then follows directly from Lemma 5 that P also has a previous occurrence in S. We will show that a previous occurrence of P also exists in S' and that by Lemma 4 this incurs at most one phrase starting position. There are three cases that should be considered for the previous occurrence of P in S. It either starts in (a) u_1, (b) some $s_{j'}$ where $j' < j$, or (c) s_j. In case (a) the occurrence must either end in u_1 or s_1, since the length is at most d. This occurrence is therefore contained within s'_1, and must also be present in S'. Similarly, in case (b) the occurrence crosses at most one border from $s_{j'}$ to $s_{j'+1}$ due to the length being at most d. Since the length of each sample is at least $2d$, the sample $s'_{j'+1}$ must end with the substring $s_{j'}s_{j'+1}$, and since $j' + 1 \leq j$ a previous occurrence of P must also exist in S'. Lastly, in case (c), the occurrence must also end in s_j since the phrase part would otherwise overlap with the border from s_j to s_{j+1}. Such a previous occurrence clearly also exists in S'. Thus, a phrase part always has a previous occurrence in S', and the proof follows. ◀

In order to determine a bound on the number of phrase starting positions, we therefore only have to count how many phrase parts are present in S in the interval $[m + 1, n]$.

There can be at most one phrase part of type (i) or (ii) for each substring s_j. This is shown by a simple contradiction. Assume there is more than one phrase part of either type in s_j. This would imply that more than one phrase starts before s_j and crosses the border from s_{j-1} to s_j. However, this contradicts the requirement that phrases do not overlap. Thus, there are at most $k - 1 = (n - m)/d$ such parts. Similarly, we will at most have $(z_S - z_X)$ phrase parts of type (iii) or (iv). There cannot be any more, since phrase f_{z_X} does not have its starting position in the interval $[m + 1, n]$ and a phrase would otherwise need to have two starting positions which is not possible. Therefore, there are at most $(z_S - z_X + (n - m)/d)$ phrase parts partitioning the interval $[m + 1, n]$ in S, and as shown in Lemma 6 these incur at most one phrase each in the LZ77 parsing of S'.

Putting it together

In summary, we have shown that case 1 incurs at most z_X phrases, case 2 at most $(n - m)/d$, and finally case 3 at most $(z_S - z_X + (n - m)/d)$. Therefore:

$$z_{S'} \leq z_X + \frac{n-m}{d} + z_S - z_X + \frac{n-m}{d} = z_S + 2\frac{n-m}{d}.$$

This concludes the proof of Theorem 1.

3 Lower Bound

Given integers $d \geq 3$, $m \geq 2d$, and $k > 0$, we construct the string S of length $m + (k - 1)d$ over the alphabet $\Sigma = \{\lambda_0, \lambda_1, ..., \lambda_k\}$. In particular, we construct S from several substrings such that $S := u_1 s_1 s_2 \cdots s_k$, where $|u_1| = m - d$, and $|s_j| = d$ for all $1 \leq j \leq k$. This is exactly the interpretation used during the proof of Theorem 1. We define u_1 as $m - d$ consecutive repetitions of letter λ_0, such that $u_1 := \lambda_0 \cdots \lambda_0$. Similarly, we define each s_j as a single letter λ_{j-1} followed by $d - 1$ repetitions of letter λ_j, such that $s_j := \lambda_{j-1}\lambda_j \cdots \lambda_j$. This construction is shown in Figure 8:

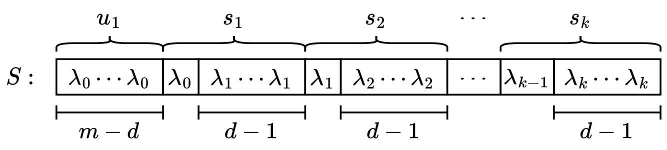

u_1
s_1
s_2
$\cdots$
s_k
$S:$
$\lambda_0 \cdots \lambda_0$
λ_0
$\lambda_1 \cdots \lambda_1$
λ_1
$\lambda_2 \cdots \lambda_2$
$\cdots$
λ_{k-1}
$\lambda_k \cdots \lambda_k$
$m - d$
$d - 1$
$d - 1$
$d - 1$

Figure 8 Construction of S in the proof of Theorem 2.

As seen above $S = \lambda_0^{m-d+1}\lambda_1^d \cdots \lambda_{k-1}^d \lambda_k^{d-1}$. Due to the constraints on m and d, the length of each of the repetitions is at least 2 and therefore requires exactly two phrases to encode. Thus, the total number of phrases in the LZ77 parsing of S is $z_S = 2k + 2$.

We now consider how many phrases are required to encode $S' := \mathrm{SCS}(S, m, d)$. We prove this by induction showing that every sample is encoded by exactly four phrases, and these phrases never cross the border between two samples. We consider the first sample s'_1 in Figure 9.

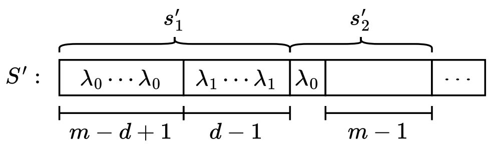


Figure 9 The structure of the first sample in S'.

The first sample s'_1 consists of $m - d + 1$ repetitions of λ_0 followed by $d - 1$ repetitions of λ_1. Since the repetitions have length at least 2, these are encoded by exactly four phrases in total. Therefore, we only have to argue that the last phrase does not overlap s'_2. The first letter in s'_2 is always λ_0 since the start position of this sample is $d + 1$, which is within the first $m - d + 1$ repetitions of λ_0, since $m \geq 2d$. Hence, this is the first time λ_0 follows λ_1, and thus the last phrase in s'_1 does not overlap s'_2.

We now assume the hypothesis for every sample prior to the jth sample, seen in Figure 10.

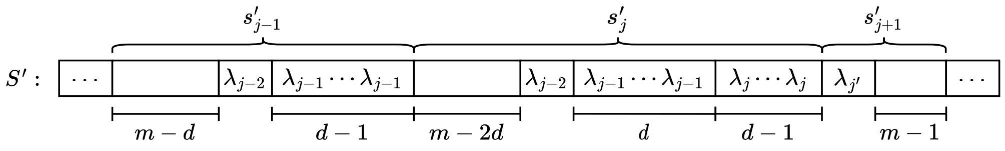


Figure 10 The structure of the jth sample in S'.

By the induction hypothesis, the last phrase encoding s'_{j-1} does not overlap any letters in s'_j. Therefore, the first phrase encoding s'_j starts with the first letter in s'_j. We show that s'_j is encoded by exactly four phrases which also do not cross the border to s'_{j+1}.

(i) The first phrase covers exactly the first $m - d$ letters in s'_j.
(ii) The second phrase has length 1, covering the last occurrence of λ_{j-1} in s'_j.
(iii) The third phrase has length 1, covering the first occurrence of λ_j.
(iv) The fourth phrase covers exactly the last $d - 2$ repetitions of λ_j in s'_j.

As mentioned during the proof of Theorem 1, the first $m - d$ letters u_j in sample s'_j where $j > 1$, is a repeat of the last $m - d$ letters in s'_{j-1}. By the induction hypothesis and Lemma 4, this implies that there can be at most one phrase overlapping this interval. Since every sample has length $m \geq 2d$, the jth sample ends with the substring $s_{j-1}s_j$ as seen in Figure 10. Therefore, if the first phrase covered more than $m - d$ letters, it would also have to cover d repetitions of λ_{j-1} following letter λ_{j-2}. However, this is the first time we encounter d repetitions of λ_{j-1} in a row. This shows (i). The last occurrence of λ_{j-1} is also covered by exactly one phrase since the phrase would otherwise also have to cover the first occurrence of λ_j which is not possible. This shows (ii). It is trivial to show (iii) and (iv), since it is the first occurrence of λ_j. In the latter case, the phrase does not overlap s'_{j+1}, since that sample begins with some $\lambda_{j'}$ where $j' \neq j$.

The induction proof does not consider the last sample, however, in this case, the last phrase clearly ends at the end of s'_k. Therefore, we have shown that each sample is encoded by exactly four phrases. Thus, the total number of phrases in the LZ77 parsing of S' is $z_{S'} = 4k = z_S + 2(k-1)$.

Theorem 2 is therefore shown for some parameter k. However it is also possible to construct a string S of any length $n \geq m$ by letting $k := \lfloor (n-m)/d \rfloor + 1$ and padding $(n-m \mod d)$ repetitions of λ_k to the end of S in the definition stated in Figure 8. This yields exactly the same number of phrases for encoding S, and since the last $(n-m \mod d)$ letters are not part of any sample, S' remains unchanged. Thus, we have shown the equality:

$$|\text{LZ77}(\text{SCS}(S,m,d))| = |\text{LZ77}(S)| + 2(k-1) = |\text{LZ77}(S)| + 2\left\lfloor \frac{n-m}{d} \right\rfloor.$$

This concludes the proof of Theorem 2.

4 Improved Compressibility

Given integers $d \geq 2$ and $m \geq 2d$, where $m \equiv 0 \pmod d$, we construct a string S of length $n = 3m - d$ over alphabet Σ consisting of exactly two letters a and b, i.e., $\Sigma = \{a, b\}$. Notice that $n \geq m$ and the required property of $n \equiv m \pmod d$ holds, since $m \equiv 0 \pmod d$ implies that $3m - d \equiv m \pmod d$. We construct S by concatenating several repetitions of letters from Σ, as shown in Figure 11. The phrases in the LZ77 parsing of S are shown in Figure 12.

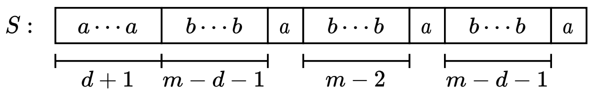


Figure 11 Construction of S in the proof of Theorem 3.

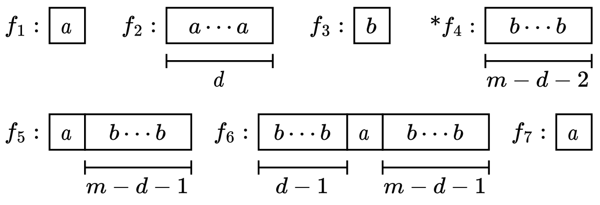


Figure 12 The phrases in the LZ77 parsing of S. Phrase $*f_4$ is only present when $m > d + 2$.

Notably, the first repetition of letter b in Figure 11 has two cases, and requires either one or two phrases to encode depending on whether $m = d + 2$ or $m > d + 2$, respectively. This is only relevant when $d = 2$ and $m = 4$, since we otherwise always fulfill $m > d + 2$. Therefore, the number of phrases in the LZ77 parsing of S following this construction is:

$$|\text{LZ77}(S)| = \begin{cases} 6 & \text{if } d = 2 \text{ and } m = 4, \\ 7 & \text{otherwise.} \end{cases}$$

We now consider the number of phrases in the LZ77 parsing of $S' := \text{SCS}(S,m,d)$. There is a total of $k = \lfloor (3m - d - m)/d \rfloor + 1 = 2m/d$ samples contributing to S'. The first sample s'_1 consists of $d+1$ repetitions of letter a followed by $m - d - 1$ repetitions of letter b, since these are the first m letters in S. This sample is shown in Figure 13.

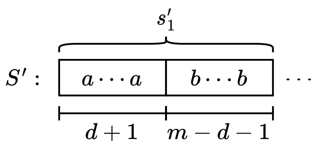


Figure 13 The first sample of S'.

The following $m/d-1$ samples together follow a pattern of a single a followed by $m-d-1$ repetitions of b. This pattern occurs since the jth sample ends with $(j-1)d-1$ repetitions of letter b, and the $(j+1)$th sample starts with $m-jd$ repetitions of letter b, resulting in a total of $m-d-1$ repetitions of b. This repeating pattern is illustrated in Figure 14.

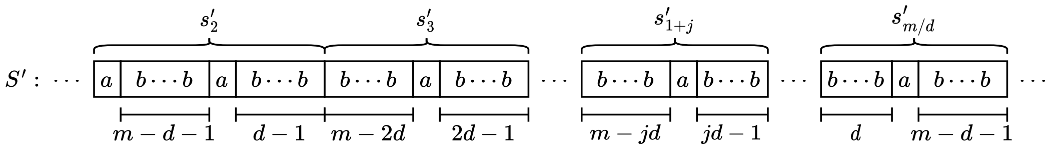


Figure 14 The 2nd to the (m/d)th samples of S'. These form a repeating pattern of a single letter a followed by $m-d-1$ repetitions of letter b.

The $(m/d+1)$th sample is shown in Figure 15. This is the sample starting at position m in S and also starts with the pattern of a single a followed by $m-d-1$ repetitions of b. The number of times this pattern occurs is therefore once for the first sample, m/d times for the following $m/d-1$ samples, and once for the $(m/d+1)$th sample, totaling $m/d+2$ times.

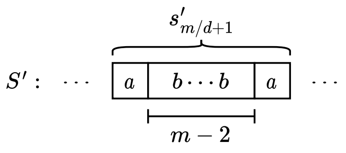


Figure 15 The $(m/d+1)$th samples of S'.

Finally, the remaining $m/d-1$ samples consist of a similar repeating pattern with $m-d-1$ repetitions of b followed by a single a. The pattern in this case is repeated only m/d times, i.e., two times less than previously. This is illustrated in Figure 16.

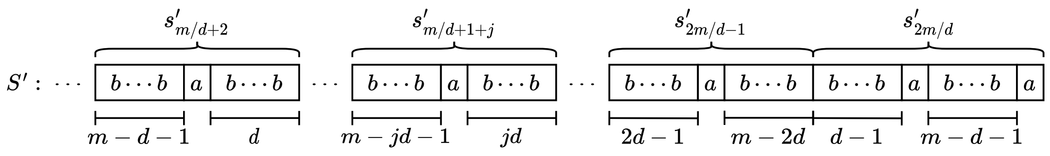


Figure 16 The $(m/d+2)$th to the $(2m/d)$th samples of S'. These form a repeating pattern of $m-d-1$ repetitions of letter b followed by a single letter a.

The complete structure of S' is shown in Figure 17. We have adjusted the second pattern slightly, to make it more similar to the first pattern. The phrases in the LZ77 parsing of S' are shown in Figure 18.

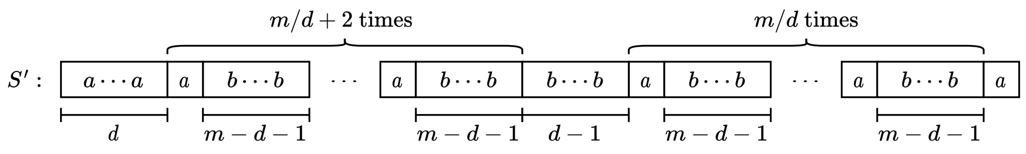


Figure 17 The structure of S' in the proof of Theorem 3.

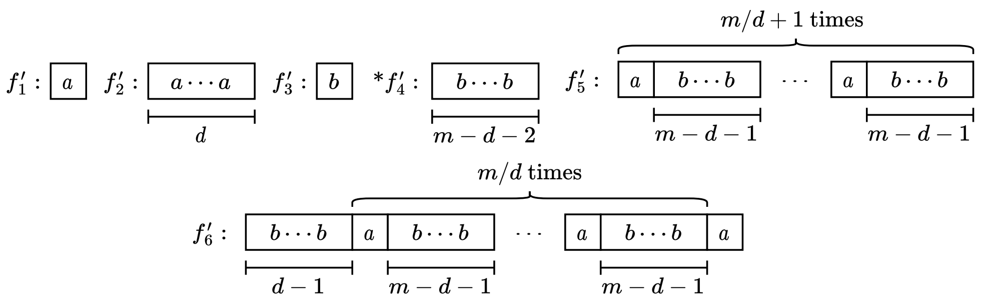


Figure 18 The phrases in the LZ77 parsing of S'. Phrase $*f'_4$ is only present when $m > d+2$.

Notably, the LZ77 parsing of S' takes advantage of the pattern where a single letter a is followed by a repetition of letter b. This is especially relevant for phrases f'_5 and f'_6. Again, the first repetition of the letter b has the same two cases exactly as described previously, requiring either one or two phrases. Therefore, the size of the LZ77 parsing of S' is:

$$|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| = \begin{cases} 5 & \text{if } d=2 \text{ and } m=4, \\ 6 & \text{otherwise.} \end{cases}$$

This is always exactly one phrase less than the number of phrases in the LZ77 parsing of S. Thus, we have shown the following equality:

$$|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| = |\mathrm{LZ77}(S)| - 1$$

This concludes the proof of Theorem 3.

5 Concluding Remarks

We have considered the problem of compressing a set of substrings sampled from a string and analyzing the size of the compression. We have shown that $|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| \leq |\mathrm{LZ77}(S)| + 2(n-m)/d$ and that this upper bound is tight. Likewise, we have shown that there exists a family of strings where the compressibility after the sampling process improves by exactly one phrase, i.e., where $|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| = |\mathrm{LZ77}(S)| - 1$.

There are several directions that future work could take. A natural question is to derive bounds for other compression algorithms, such as Relative Lempel-Ziv [17, 22], LZ78 [44], Re-Pair [28], or other well-known context-free grammar compressors [32, 42]. As an extension of the original motivation it is also relevant to consider what happens when we change some of the idealizations made to the model (e.g. errors during sampling as introduced by short-read sequencing technologies, or a generalization of sample lengths and/or positions). Finally, an interesting question is whether there exists a string where the size of the LZ77 parsing of the concatenated samples improves by more than one phrase, i.e., is the best you can do the equality in Theorem 3, or does there exist an instance where $|\mathrm{LZ77}(\mathrm{SCS}(S,m,d))| < |\mathrm{LZ77}(S)| - 1$?

References

1 Sultan Al Yami and Chun-Hsi Huang. LFastqC: A lossless non-reference-based FASTQ compressor. *PLoS One*, 14(11):e0224806, 2019.

2 Golnaz Badkobeh, Maxime Crochemore, and Chalita Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *Proc. SPIRE*, pages 61–72, 2012.

3 Golnaz Badkobeh, Sara Giuliani, Zsuzsanna Lipták, and Simon J. Puglisi. On compressing collections of substring samples. In *Proc. 23rd ICTCS*, pages 136–147, 2022.

4 Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. Lempel-Ziv compression in a sliding window. In *Proc. 28th CPM*, volume 78, pages 15:1–15:11, 2017.

5 Shubham Chandak, Kedar Tatwawadi, Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, 35(15):2674–2676, 2018.

6 Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Proc. DCC*, pages 482–488, 2008.

7 Sebastian Deorowicz. FQSqueezer: k-mer-based compression of sequencing data. *Sci. Rep.*, 10(1):578, 2020.

8 Robert Ekblom, Linnéa Smeds, and Hans Ellegren. Patterns of sequencing coverage bias revealed by ultra-deep sequencing of vertebrate mitochondria. *BMC Genomics*, 15:467, 2014.

9 Susan Fairley, Ernesto Lowy-Gallego, Emily Perry, and Paul Flicek. The International Genome Sample Resource (IGSR) collection of open human genomic variation resources. *Nucleic Acids Res.*, 48(D1):D941–D947, 2019.

10 Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proc. WSDM*, pages 391–400, 2010.

11 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

12 Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. ESA*, pages 533–544, 2015.

13 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Proc. CPM*, pages 172–184, 2015.

14 Travis Gagie and Paweł Gawrychowski. Grammar-based compression in a streaming model. In *Proc. LATA*, pages 273–284, 2010.

15 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proc. DCC*, pages 163–172, 2014.

16 Christopher Hoobin, Trey Kind, Christina Boucher, and Simon J. Puglisi. Fast and efficient compression of high-throughput sequencing reads. In *Proc. 6th ACM-BCB*, pages 325–334, 2015.

17 Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endow.*, 5(3):265–273, 2011.

18 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. ALENEX*, pages 103–112, 2013.

19 Roman Kolpakov and Gregory Kucherov. Finding approximate repetitions under Hamming distance. *Theor. Comput. Sci.*, 303(1):135–156, 2003.

20 Dmitry Kosolobov. Faster lightweight Lempel-Ziv parsing. In *Proc. MFCS*, pages 432–444, 2015.

21 Sebastian Kreft and Gonzalo Navarro. Lz77-like compression with fast random access. In *Proc. DCC*, pages 239–248, 2010.

22 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. SPIRE*, pages 201–206, 2010.

23 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. SEA*, pages 139–150, 2013.

24 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, pages 189–200, 2013.

25 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Proc. DCC*, pages 153–162, 2014.

26 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12, 2016.

27 Niklas Jesper Larsson. Extended application of suffix trees to data compression. In *Proc. DCC*, pages 190–199, 1996.

28 Niklas Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

29 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inform. Theory*, 22(1):75–81, 1976.

30 Harris A. Lewin et al. Earth BioGenome Project: Sequencing life for the future of life. *Proc. Natl. Acad. Sci. U.S.A*, 115(17):4325–4333, 2018.

31 Harris A. Lewin et al. The earth biogenome project 2020: Starting the clock. *Proc. Natl. Acad. Sci. U.S.A*, 119(4), 2022.

32 Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *JAIR*, 7:67–82, 1997.

33 Genome 10K Community of Scientists. Genome 10K: A Proposal to Obtain Whole-Genome Sequence for 10 000 Vertebrate Species. *J. Hered.*, 100(6):659–674, 2009.

34 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Proc. CPM*, pages 15–26, 2011.

35 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA*, pages 696–707, 2008.

36 Alberto Policriti and Nicola Prezza. Fast online Lempel-Ziv factorization in compressed space. In *Proc. SPIRE*, pages 13–20, 2015.

37 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In *Proc. DCC*, pages 23–32, 2016.

38 Julian Shun and Fuyao Zhao. Practical parallel Lempel-Ziv factorization. In *Proc. DCC*, pages 123–132, 2013.

39 Tatiana Starikovskaya. Computing lempel-ziv factorization online. In *Proc. MFCS*, pages 789–799, 2012.

40 James Andrew Storer and Thomas Gregory Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

41 Jun'ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In *Proc. 31st STACS*, volume 25, pages 675–686, 2014.

42 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. Inform. Theory*, 46(3):755–777, 2000.

43 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, 1977.

44 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

Searching 2D-Strings for Matching Frames

Itai Boneh
Reichman University, Herzliya, Israel
University of Haifa, Israel

Dvir Fried
Bar Ilan University, Ramat Gan, Israel

Shay Golan
Reichman University, Herzliya, Israel
University of Haifa, Israel

Matan Kraus
Bar Ilan University, Ramat Gan, Israel

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

Arseny Shur
Bar Ilan University, Ramat Gan, Israel

Abstract

We study a natural type of repetitions in 2-dimensional strings. Such a repetition, called a matching frame, is a rectangular substring of size at least 2×2 with equal marginal rows and equal marginal columns. Matching frames first appeared in literature in the context of Wang tiles.

We present two algorithms finding a matching frame with the maximum perimeter in a given $n \times m$ input string. The first algorithm solves the problem exactly in $\tilde{O}(n^{2.5})$ time (assuming $n \geq m$). The second algorithm finds a $(1-\varepsilon)$-approximate solution in $\tilde{O}(\frac{nm}{\varepsilon^4})$ time, which is near linear in the size of the input for constant ε. In particular, by setting $\varepsilon = O(1)$ the second algorithm decides the existence of a matching frame in a given string in $\tilde{O}(nm)$ time. Some technical elements and structural properties used in these algorithms can be of independent interest.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases 2D string, matching frame, LCP, multidimensional range query

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.10

Related Version *Full Version*: `https://arxiv.org/abs/2310.02670` [9]

Funding *Itai Boneh*: supported by Israel Science Foundation grant 810/21.
Dvir Fried: supported by ISF grant no. 1926/19, by a BSF grant 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Shay Golan: supported by Israel Science Foundation grant 810/21.
Matan Kraus: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Adrian Miclăuş: supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-0253, within PNCDI III.
Arseny Shur: supported by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064) and by the State of Israel through the Center for Absorption in Science of the Ministry of Aliyah and Immigration.

1 Introduction

Throughout the years, a variety of notions for repetitive structures in strings have been explored; see, e.g., [18, 31, 27, 42, 29]. Even recently, new efficient algorithms regarding palindromes [10, 22, 37], squares [17], runs [6, 16, 33], and powers [4] have been introduced. In the studies on 2-dimensional strings (aka 2*d-strings* or *matrices*), periodic and palindromic structures also attracted definite interest [2, 3, 5, 13, 21, 30, 19, 38].

Matching frame is a natural repetition in 2d-strings, first considered by Wang [40] when introducing Wang tiles. Given a 2d-string M over an alphabet Σ, a *frame* in M is a rectangle defined by a tuple (u, d, ℓ, r) such that $u < d$ and $\ell < r$. This rectangle covers the submatrix $M[u..d][\ell..r]$ and is *matching* if this submatrix has equal marginal rows and equal marginal columns. Formally, (u, d, ℓ, r) is a matching frame if $M[u][\ell..r] = M[d][\ell..r]$ and $M[u..d][\ell] = M[u..d][r]$ (see Figure 1). Wang's *fundamental conjecture*, later disproved by Berger [8], said "a set of tiles is solvable (= tiles the plane) if and only if it admits a cyclic rectangle (= matching frame)". Note that a fast algorithm to find matching frames would simplify a huge computation conducted by Jeandel and Rao [24] to prove that their aperiodic set of tiles is minimal.

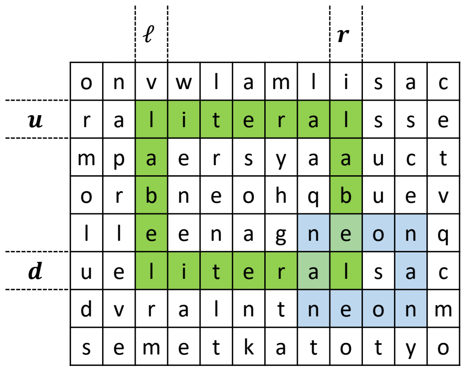


Figure 1 An example of a matching frame $(u, d, \ell, r) = (2, 6, 3, 9)$. The strings on the top and bottom sides of the frame are equal, and the strings on the left and right sides are also equal. The perimeter of the frame is $2 \cdot (6 - 2 + 9 - 3) = 20$. The matrix also contains a smaller matching frame.

Matching frames indicate "potential" periodicity in two dimensions. Namely, if a 2d-string M is built according to some local rule, then any matching frame in M can be extended to a periodic tiling of the plane, *respecting this local rule*. Well-known examples of such local rules are given, in particular, by self-assembly models such as aTAM [36] or 2HAM [11]. Note that matching frame is an *avoidable* repetition: as was first observed by Wang [41], there exist infinite *binary* 2d-strings without matching frames. Avoidable repetitions are interesting, in particular, due to a nontrivial decision problem.

Overall, there is a clear motivation to design efficient algorithms searching for matching frames. Let us specify the exact problem studied in this paper. The *perimeter* of a frame $F = (u, d, \ell, r)$ is the total number of cells in its marginal rows and columns, i.e. $\mathsf{per}(F) = 2(d - u + r - \ell)$. By *maximum* frame (in a set of frames) we mean the frame with the maximal perimeter in this set. In the *maximum matching frame problem*, the goal is to find a maximum matching frame in a given matrix or report that no matching frame exists. We also consider the $(1 - \varepsilon)$-approximation version of this problem, in which the goal is to find a matching frame with a perimeter within the factor $(1 - \varepsilon)$ from the maximum possible.

Our Results. We present $\tilde{O}(nm)$-space algorithms that establish the following bounds on the complexity of the maximum matching frame problem and its approximation version.

▶ **Theorem 1** (Maximum Matching Frame). *The time complexity of the maximum matching frame problem for an $n \times m$ matrix M is $\tilde{O}(n^{2.5})$ in the case $m = \Theta(n)$. In the general case, the complexity is $\tilde{O}(ab \min\{a, \sqrt{b}\})$, where $a = \min\{n, m\}$ and $b = \max\{n, m\}$.*[1]

▶ **Theorem 2** ($(1-\varepsilon)$-Approximation). *The time complexity of the $(1-\varepsilon)$-approximation maximum matching frame problem for an $n \times m$ matrix M is $\tilde{O}(\frac{nm}{\varepsilon^4})$.*

▶ **Corollary 3** (Deciding Matching Frame). *There is an algorithm deciding whether an $n \times m$ matrix M contains a matching frame in $\tilde{O}(nm)$ time and space.*

We remark that our exact and approximation algorithms can be straightforwardly adapted to find matching frames with the maximum area / the minimum perimeter / the minimum area instead of matching frames with the maximum perimeter.

High-Level Overview

Maximum Matching Frame. The algorithm for finding a maximum matching frame follows a heavy-light approach. The parameter used to distinguish between heavy and light frames is the *shorter side* of the frame. A frame $F = (u, d, \ell, r)$ has *height* $d - u$ and *width* $r - \ell$. We assume that there is a maximum matching frame having its height smaller than or equal to its width. (Either the input matrix or its transpose satisfies this assumption and we can apply our algorithm to both matrices and return the better of two results.) For some integer threshold x, we say that a frame with $d - u \le x$ is *short* (or light); otherwise, it is *tall* (or heavy). We provide two algorithms, one that returns a maximum *short* matching frame in M and another returns a maximum *tall* matching frame in M. The largest of the two answers is the maximum matching frame in M.

The algorithm for short frames iterates over all pairs of rows with distance at most x from each other. Note that there are $O(nx)$ such pairs. Moreover, under the assumption that some matching frame $F = (u, d, \ell, r)$ is short, the rows u and d used by F are processed as a pair. When processing a pair, the algorithm decomposes its rows into maximal equal segments. Every segment is processed in linear time to obtain a maximum matching frame that uses a portion of the segment as top and bottom rows (see Section 5.1). The accumulated size of the segments is bounded by m, so the algorithm runs in $\tilde{O}(n \cdot m \cdot x)$ time.

The algorithm for tall frames (see Section 5.2) first guesses a range $[H/2..H]$ for the height and a range $[W/2..W]$ for the width of a maximum matching frame. As we consider tall frames, the ranges are sufficiently large, so it is easy to find a small set of positions $\mathcal{P}$ in the matrix M such that every frame with the height and width from the given ranges contains a position from $\mathcal{P}$. The algorithm employs a subroutine that, given H, W, and a position (i, j), computes a maximum matching frame among the frames that contain (i, j), have the height in $[H/2..H]$ and the width in $[W/2..W]$. The implementation of this subroutine is the main technical part of the algorithm. This is done by maintaining and querying a range data structure (see Section 4) that allows one to process pairs of columns and pairs of rows with the position (i, j) between them. There are $O(W^2)$ pairs of columns and $O(H^2)$ pairs of rows to be processed, which we do in $\tilde{O}(H^2 + W^2) = \tilde{O}(W^2)$ total time. We also show that $|\mathcal{P}| = O(\frac{nm}{HW})$, and therefore the running time for one pair of ranges is $\tilde{O}(nm\frac{W}{H})$. We

[1] Throughout the paper, $\tilde{O}(f(n)) = O(f(n) \cdot \mathsf{polylog} n)$

further observe that the sum of values $\frac{W}{H}$ over all guessed ranges is $O(\frac{W'}{H'})$ for some single guessed pair (W', H'). Since $x \le H' \le W' \le \max\{n, m\}$, we obtain the running time of $\tilde{O}(nm\frac{\max(n,m)}{x})$.

Finally, the algorithm selects the threshold $x = \sqrt{\max\{n, m\}}$ and applies the algorithms for both the short and the tall case to obtain a running time of $\tilde{O}(nm\sqrt{\max\{n, m\}})$. Alternatively, one can run the algorithm for short frames alone, setting $x = \min(n, m)$. Taking the better of these two options proves Theorem 1.

Approximation Algorithm. As a preliminary step in our approach for finding a $(1-\varepsilon)$-approximation to the maximum matching frame, we apply a two-dimensional variant of the so-called *standard trick* [15, 12] from certain one-dimensional pattern matching problems. In pattern matching, we are given a text $T[1..n]$ and a pattern $P[1..m]$ and the goal is to find all the indices $i \in [n-m+1]$ such that $T[i..i{+}m{-}1]$ "matches" P. The standard trick refers to partitioning T into $O(n/m)$ overlapping fragments of size $\Theta(m)$, such that every match of P is contained in a fragment. In general, the trick allows one to assume that the length of the text is within a small factor from the length of the pattern. Our two-dimensional variant of this trick (Lemma 15) allows us to assume that both dimensions of the maximum matching frame are within a $\mathsf{poly}(1-\varepsilon)$ factor of the vertical and the horizontal lengths of M.

This assumption allows us to focus on matching frames with sides that are "close" to the boundaries of M; we call such frames *large*. The algorithm uses a carefully selected threshold for being close to the boundaries, guaranteeing that (1) the maximum matching frame is large and (2) the perimeter of every large frame approximates the perimeter of the maximum matching frame. With that, the problem boils down to determine whether there exists a large matching frame. The main technical novelty of the approximation algorithm is solving this decision problem in near-linear time.

The algorithm for the above decision problem consists of two main components. The first component (see Section 6.3) is an $\tilde{O}(1)$ time subroutine that, given a triplet (u, d, ℓ), decides if there is an integer r such that (u, d, ℓ, r) is a large matching frame. However, applying this subroutine to every triplet would cost $\Omega(n^2 m)$ time. The second component (see Section 6.2) of the algorithm is the retrieval of a set of $\tilde{O}(nm)$ triplets such that if some large matching frame exists, there must also be a large matching frame derived from one of these triplets.

We conclude by presenting the combinatorial structure that allows us to consider $\tilde{O}(nm)$ triplets in the second component. Consider a triplet (u, d, ℓ) and let k be the largest integer such that $M[u][\ell..k] = M[d][\ell..k]$ (let S denote this string). Assuming there exists an index r such that (u, d, ℓ, r) is a large matching frame, one has $r \le k$. Observe that if there is an index $d' < d$ that is close to the bottom boundary of M such that $M[d'][\ell..k] = S$, then (u, d', ℓ, r) is also a large matching frame. Therefore, the triplet (u, d, ℓ) can be removed from the set of triplets that have to be processed. We say that a triplet that is not eliminated due to this reasoning is *interesting*. Surprisingly, the number of interesting triplets is bounded by $O(nm \log n)$ (see Section 6.1). This combinatorial observation is the main novelty of the approximation algorithm.

2 Preliminaries

We use range notation for integers and strings. We write $[i..j]$ and $[i..j)$ for the sets $\{i, \dots, j\}$ and $\{i, \dots, j-1\}$ respectively (assuming $i \le j$). Further, we abbreviate $[1..n]$ to $[n]$. A string $S[1..n] = S[1]S[2]\cdots S[n]$ is a sequence of characters from an alphabet Σ. We also write $\overleftarrow{S[1..n]} = S[n]S[n{-}1]\cdots S[1]$. For every $i \le j \in [n]$, $S[i..j] = S[i]S[i+1]\cdots S[j]$ is a *substring* of S. The substring is called a *prefix* (resp., a *suffix*) of S if $i = 1$ (resp., $j = n$). We assume Σ to be linearly ordered, inducing a *lexicographic order* (*lex-order*) on strings.

An $n \times m$ *matrix* (or $2d$*-string*) M is a 2-dimensional array of symbols from Σ. We refer to the number of cells in M as the *size* of M, writing $|M| = nm$. We denote a horizontal substring of M as $M[i][j_1..j_2] = M[i][j_1]M[i][j_1+1]\dots M[i][j_2]$. Similarly, we denote a vertical substring as $M[i_1..i_2][j] = M[i_1][j]M[i_1+1][j]\dots M[i_2][j]$.

2.1 Suffix Arrays, Longest Common Prefixes

For a tuple of strings $\mathcal{S} = (S_1, S_2, \dots, S_n)$, the *lexicographically sorted array* $\mathsf{LSA}_{\mathcal{S}}$ is an array of length n that stores the lex-order of the strings in $\mathcal{S}$. Formally, $\mathsf{LSA}_{\mathcal{S}}[i] = j$ if S_j is the ith string in $\mathcal{S}$ according to the lex-order (ties are broken arbitrarily). For a string $S[1..n]$, the *suffix array* SA_S of S is the LSA of all suffixes of S. Formally, for every $i \in [n]$ let $S_i = S[i..n]$ and let $\mathcal{S}_S = (S_1, S_2, \dots, S_n)$; then $\mathsf{SA}_S = \mathsf{LSA}_{\mathcal{S}_S}$. The suffix arrays were introduced by Manber and Myers [32] and became ubiquitous in string algorithms. The array can be constructed in near-linear time and space by many algorithms [25, 26, 28, 34, 35, 42, 39].

▶ **Lemma 4.** *Given a string $S[1..n]$, the suffix array of S can be constructed in $O(n \log n)$ time and space.*

An important computational primitive is a data structure for computing the length of the *longest common prefix* of two strings $S[1..n]$ and $T[1..m]$, given as $\mathsf{LCP}(S,T) = \max\{\ell \in [\min\{n,m\}] \mid S[1..\ell] = T[1..\ell]\}$. An LCP data structure $\mathsf{LCP}_{\mathcal{S}}$ for a set of strings $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ supports queries in the form "given two indices $i, j \in [n]$, report $\mathsf{LCP}(S_i, S_j)$". We denote by $\mathsf{LCP}(S)$ the LCP data structure for the set of suffixes of a given string $S[1..n]$. It is known that the following can be obtained by applying the lowest common ancestor data structure of [23] to the suffix tree of [42].

▶ **Lemma 5.** *There is an* LCP *data structure with $O(n \log n)$ construction time and $O(1)$ query time. The data structure uses $O(n)$ space.*

The following facts are easy. For their proofs, see the full version [9] of this paper.

▶ **Fact 6.** *Given three strings S_1, S_2 and S_3, the condition* $\mathsf{LCP}(S_1, S_2) > \mathsf{LCP}(S_1, S_3)$ *implies* $\mathsf{LCP}(S_1, S_3) = \mathsf{LCP}(S_2, S_3)$.

▶ **Fact 7.** *Let $\mathcal{S} = (S_1, S_2, \dots, S_n)$ be a tuple of strings and let $P[1..m]$ be a string. The set* $\mathsf{Occ}(\mathcal{S}, P) = \{k \mid S_k[1..m] = P\}$ *coincides with the range* $\mathsf{LSA}_{\mathcal{S}}[i..j]$ *for some $i, j \in [n]$.*

Furthermore, there is an $O(\log n)$ time algorithm that given k, m, $\mathsf{LSA}_{\mathcal{S}}$*, and* $\mathsf{LCP}_{\mathcal{S}}$ *computes i and j such that* $\mathsf{Occ}(\mathcal{S}, S_k[1..m]) = \mathsf{LSA}_{\mathcal{S}}[i..j]$.

▶ **Definition 8** (Fingerprint)**.** *For a tuple $\mathcal{S}$ and a string $P = S_k[1..m]$, the* fingerprint *of P in $\mathcal{S}$ is the tuple (i, j, m) such that i and j are the indices specified in Fact 7.*

2.2 Orthogonal Range Queries

Our algorithms use data structures for *orthogonal range queries.* Such a data structure stores, for some positive integer dimension d, a set $\mathcal{P} \subseteq \mathbb{R}^d$ of d-dimensional points. Each point $p \in \mathcal{P}$ has an associated value $v(p) \in \mathbb{R}$. The data structure supports the queries regarding an input d-dimensional orthogonal range $R = [a_1..b_1] \times [a_2..b_2] \times \dots \times [a_d..b_d]$. For a point $p = (x_1, x_2, \dots, x_d)$ one has $p \in R$ if $x_i \in [a_i..b_i]$ for every $i \in [1..d]$. We need the queries $\mathsf{Maximum}(R) = \mathsf{argmax}_{v(p)}(p \in R \cap \mathcal{P})$ and $\mathsf{Minimum}(R) = \mathsf{argmin}_{v(p)}(p \in R \cap \mathcal{P})$. For this, we use the data structure [43, 14] with the following running times.

▶ **Lemma 9.** *For any integer d, a set of n points in $\mathbb{R}^d$ can be preprocessed in $O(n \log^{d-1} n)$ time and space to support* Maximum *and* Minimum *range queries in $O(\log^{d-1} n)$ time.*

In Section 6.3, we use a very particular type of 2-dimensional Maximum/Minimum queries, where $v(p)$ is one of the coordinates of p. Though faster data structures are known in this case [7, 20], using these data structures cannot improve the asymptotics of our results.

3 Data Structures

When looking for matching frames in an $n \times m$ matrix M, we make use of the following data structures, which all our algorithms create during their preprocessing phase.

- For each column $\ell \in [m]$ we use
  1. a lex-sorted array $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ of the strings $\{M[i][\ell..m] \mid i \in [n]\}$ (see Figure 2a);
  2. an LCP structure $\mathsf{LCP}^{\ell}_{\mathsf{rows}}$ over $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$;
  3. a range query structure D^{ℓ}_{rows}, containing all pairs $\{(i, I^{i,\ell}_{\mathsf{rows}}) \mid i \in [n]\}$, where $I^{i,\ell}_{\mathsf{rows}}$ is the index of the string $M[i][\ell..m]$ in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ (see Figure 2b).

  In addition, we build the same three structures for the set of all strings of the form $\overleftarrow{M[i][1..\ell]}$, denoted as $\mathsf{LSA}^{\ell}_{\overleftarrow{\mathsf{rows}}}$, $\mathsf{LCP}^{\ell}_{\overleftarrow{\mathsf{rows}}}$ and $D^{\ell}_{\overleftarrow{\mathsf{rows}}}$.
- Symmetrically, for each row $u \in [n]$ we use
  1. a lex-sorted array $\mathsf{LSA}^{u}_{\mathsf{columns}}$ of the strings $\{M[u..n][i] \mid i \in [m]\}$;
  2. an LCP structure $\mathsf{LCP}^{u}_{\mathsf{columns}}$ over $\mathsf{LSA}^{u}_{\mathsf{columns}}$;
  3. a range query structure D^{u}_{columns}, containing all pairs $\{(i, I^{u,i}_{\mathsf{columns}}) \mid i \in [m]\}$, where $I^{u,i}_{\mathsf{columns}}$ is the index of the string $M[u \dots n][i]$ in $\mathsf{LSA}^{u}_{\mathsf{columns}}$.

  In addition, we build the same three structures for the set of all strings of the form $\overleftarrow{M[1..u][i]}$, denoted as $\mathsf{LSA}^{u}_{\overleftarrow{\mathsf{columns}}}$, $\mathsf{LCP}^{u}_{\overleftarrow{\mathsf{columns}}}$ and $D^{u}_{\overleftarrow{\mathsf{columns}}}$.

In the full version [9] of this paper we show how to construct all these structures in $O(nm \log(nm))$ time and space.

4 The Segment Compatibility Data Structure

In this section we present the *segment compatibility data structure* (SCDS), which is at the core of our maximum matching frame algorithm (see Section 5.2). We start with technical definitions.

Segment, aligned pair, compatible pairs. A horizontal (resp. vertical) *segment* is a triplet (i, j_1, j_2) (resp. (i_1, i_2, j)) with $j_1 < j_2$ (resp. $i_1 < i_2$). It represents the horizontal (resp. vertical) segment in the plane connecting the points (i, j_1) and (i, j_2) (resp. (i_1, j) and (i_2, j)). A pair (s_1, s_2) of horizontal segments is *aligned* if $s_1 = (i_1, j_1, j_2)$ and $s_2 = (i_2, j_1, j_2)$ for some $i_1 < i_2, j_1 < j_2 \in \mathbb{N}$. Such a pair has *distance* $|i_2 - i_1|$. Symmetrically, a pair of vertical segments (s_1, s_2) is aligned if $s_1 = (i_1, i_2, j_1)$ and $s_2 = (i_1, i_2, j_2)$ for some $i_1 < i_2, j_1 < j_2 \in \mathbb{N}$. Such a pair has *distance* $|j_2 - j_1|$.

An aligned pair of horizontal segments (i_1, j_1, j_2) and (i_2, j_1, j_2) and an aligned pair of vertical segments (a_1, a_2, b_1) and (a_1, a_2, b_2) are *compatible* if and only if $a_1 \le i_1 \le i_2 \le a_2$, and $j_1 \le b_1 \le b_2 \le j_2$.

The SCDS stores a set of aligned pairs of vertical segments and supports the query

- MaxCompatible(h_1, h_2): given an aligned pair (h_1, h_2) of horizontal segments, return a pair (v_1, v_2) with the maximum distance among the stored pairs compatible with (h_1, h_2), or return null if no stored pair is compatible with (h_1, h_2).

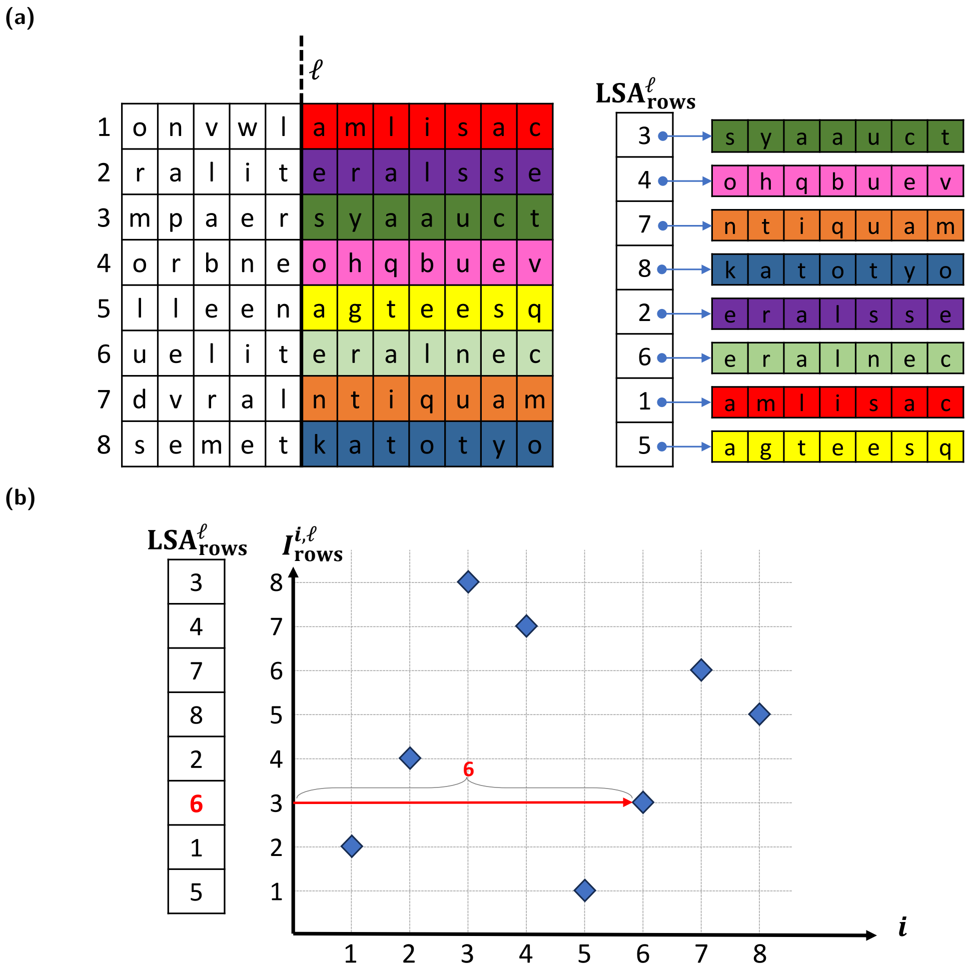

(a)
ℓ
LSA$^{\ell}_{\text{rows}}$
1 o n v w l a m l i s a c
2 r a l i t e r a l s s e
3 m p a e r s y a a u c t
4 o r b n e o h q b u e v
5 l l e e n a g t e e s q
6 u e l i t e r a l n e c
7 d v r a l n t i q u a m
8 s e m e t k a t o t y o
3 s y a a u c t
4 o h q b u e v
7 n t i q u a m
8 k a t o t y o
2 e r a l s s e
6 e r a l n e c
1 a m l i s a c
5 a g t e e s q
(b)
LSA$^{\ell}_{\text{rows}}$
$I^{i,\ell}_{\text{rows}}$
3 4 7 8 2 6 1 5
6
i

Figure 2 (a) An example of $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$. Every cell in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$ contains an index corresponding to a horizontal word in the matrix starting in column ℓ. The (indices representing the) words appear bottom-up in lex-order.
(b) A visualization of the points stored in D^{ℓ}_{rows}. Every point corresponds to a horizontal word. The height of every point corresponds to the location of the corresponding word in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$. The horizontal location of a point represents the index of its appearance in the string.

▶ **Lemma 10.** *Given a set T of t aligned pairs of vertical segments, the* SCDS *with $O(\log^3 t)$ query time can be built in $O(t \log^3 t)$ time.*

Proof. For each aligned pair $P = \big((a_1, a_2, b_1), (a_1, a_2, b_2)\big)$, we define a 4-dimensional point $\mathsf{point}(P) = (a_1, a_2, b_1, b_2)$ with the value $v(\mathsf{point}(P)) = b_2 - b_1$. Then we build, for the set of points $\{\mathsf{point}(P) \mid P \in T\}$, a 4-dimensional range data structure D with $\mathsf{Maximum}$ queries.

Let $(h_1, h_2) = \big((i_1, j_1, j_2), (i_2, j_1, j_2)\big)$ be a pair of aligned horizontal segments and let $R = ([-\infty, i_1], [i_2, \infty], [j_1, j_2], [j_1, j_2])$. It is clear that a pair P is compatible with (h_1, h_2) if and only if $\mathsf{point}(P) \in R$. Hence, to perform the query $\mathsf{MaxCompatible}(h_1, h_2)$, we query D with $\mathsf{Maximum}(R)$ and return the output.

Due to Lemma 9, the construction time and the query time are as required. ◀

5 Maximum Matching Frame

In this section we prove Theorem 1, describing an algorithm with the announced time complexity. We assume that the input matrix M contains a maximum matching frame (u, d, ℓ, r) whose *height* $d - u$ is smaller than or equal to its *width* $r - \ell$. To cover the complementary case, the algorithm is applied both to the original matrix M and to its transpose $M^\top$ and then the maximum result is reported.

Our algorithm chooses a parameter x and distinguishes between *short* frames of height at most x and *tall* frames with height larger than x. It processes the two types of frames separately and returns the maximum between two solutions.

5.1 Algorithm for Short Frames

In this section we prove the following lemma:

▶ **Lemma 11.** *There is an algorithm that for a given $x \in [n]$ finds, in $\tilde{O}(n \cdot m \cdot x)$ time and $O(n)$ additional space, a maximum matching frame of height at most x.*

Proof. For every two rows $u', d' \in [n]$ such that $d' \in [u' + 1..u' + x]$ the algorithm works as follows. First, the algorithm finds all maximal ranges $[a..b]$ such that $M[u'][a..b] = M[d'][a..b]$. By “maximal” we mean that a range can not be extended to the right or to the left while keeping equality. Note that all maximal ranges are disjoint. For $k \in [m]$ we denote the vertical string $M[u'..d'][k]$ by S_k.

Let $[a..b]$ be a maximal range. For every vertical string S_k with $k \in [a..b]$ we find its leftmost and rightmost occurrences in the range $[a..b]$. This is achieved by initializing an empty dictionary $D_{a,b}$ and scanning the range $[a..b]$ left to right. For each $k \in [a..b]$ the algorithm computes the fingerprint f in $\mathsf{LSA}^{u'}_{\mathsf{columns}}$ of the string S_k (see Definition 8). If f is not in $D_{a,b}$, we add f to $D_{a,b}$ and update both the leftmost and rightmost occurrence of S_k to be k. If f is already in $D_{a,b}$, we update the rightmost occurrence of S_k to be k.

After completing the scan, the algorithm finds a vertical string S_k such that the distance between the leftmost occurrence ℓ' and the rightmost occurrence r' of S_k is maximal. If $\ell' < r'$, we call the frame (u', d', ℓ', r') the (a, b)-range candidate of (u', d'); otherwise, there is no such candidate. Among all maximal ranges $[a..b]$, an (a, b)-range candidate with the maximal perimeter is the (u', d')-candidate (if there are no (a, b)-range candidates for (u', d'), there is no (u', d') candidate). The algorithm outputs a (u', d')-candidate with the maximal perimeter over all pairs of rows (u', d') or returns null if there are no such candidates.

Correctness. Let $F' = (u', d', \ell', r')$ be the frame returned by the algorithm. Then F' is the (a, b)-range candidate of (u', d') for some range $[a..b]$ such that $a \le \ell' < r' \le b$. Then, the equality $M[u'][a..b] = M[d'][a..b]$ implies $M[u'][\ell'..r'] = M[d'][\ell'..r']$, while $M[u'..d'][\ell'] = M[u'..d'][r']$ by the choice of ℓ', r'. Hence, F' is matching.

Let $F = (u, d, \ell, r)$ be a maximum matching frame among the frames of height at most x. When the algorithm iterates over the rows u, d, it identifies a range $[a..b]$ such that $a \le \ell < r \le b$. Let $\hat{F} = (u, d, \hat{\ell}, \hat{r})$ be the (a, b)-range candidate of (u, d). Since F is a valid choice for this candidate, the inequality $r - \ell \le \hat{r} - \hat{\ell}$ holds, implying $\mathsf{per}(F) \le \mathsf{per}(\hat{F}) \le \mathsf{per}(F')$.

Complexity. For a pair of rows (u', d'), identifying the maximal ranges takes $O(m)$ time. A maximal range $[a..b]$ requires $O(b - a)$ dictionary operations, each taking $O(\log n)$ time using, for example, an AVL tree [1]. Since all the maximal ranges of (u', d') are disjoint, their lengths sum to at most m, leading to the running time $\tilde{O}(m)$ for (u', d').

Since $d' \in [u'+1..u'+x]$, there are $O(n \cdot x)$ pairs of rows to process. Therefore, the total running time of the algorithm is $\tilde{O}(n \cdot m \cdot x)$. Since the algorithm considers every pair of rows (u', d') separately, the (additional) space usage of the algorithm is $O(n)$. ◀

5.2 Algorithm for Tall Frames

In this section, we prove the following lemma:

▶ **Lemma 12.** *There is an algorithm that for a given $x \in [n]$ finds, in $\tilde{O}(\frac{n \cdot m^2}{x})$ time and $\tilde{O}(m^2)$ additional space, a maximum matching frame of height at least x.*

Given a frame $F = (u, d, \ell, r)$ and a position $p = (i, j)$ such that $i \in [u..d]$ and $j \in [\ell..r]$, we say that p is *contained* in F and F *contains* p. We say that F is a (p, H, W)-frame if $d - u \in [H/2..H]$, $r - \ell \in [W/2..W]$, and F contains p. We introduce an algorithm that finds a maximum matching (p, H, W)-frame and use it as a subroutine of the algorithm finding the maximum matching tall frame.

▶ **Lemma 13.** *Given a position (i, j) in M and a pair of positive integers $(H, W) \in [n] \times [m]$, there is an algorithm finding a maximum matching $((i, j), H, W)$-frame in $\tilde{O}(H^2+W^2)$ time and $\tilde{O}(W^2)$ additional space.*

Proof. For every pair $(\ell, r) \in [m]^2$ such that $r - \ell \in [W/2..W]$ and $j \in [\ell..r]$, the algorithm finds the maximal aligned agreement between the columns ℓ and r intersecting the ith row by executing two LCP queries. First the algorithm queries $\mathsf{LCP}^i_{\mathsf{columns}}$ to obtain the maximal d' such that $M[i..d'][\ell] = M[i..d'][r]$. Similarly, the algorithm queries $\mathsf{LCP}^i_{\overleftarrow{\mathsf{columns}}}$ to obtain the minimal u' such that $M[u'..i][\ell] = M[u'..i][r]$. Then the algorithm stores the pair of segments $s_1 = (u', d', \ell)$ and $s_2 = (u', d', r)$. To conclude this part, the algorithm constructs an SCDS over all stored pairs.

Next, the algorithm iterates over all pairs $(u, d) \in [n]^2$ such that $d - u \in [H/2..H]$ and $i \in [u..d]$. For each such pair, the algorithm queries the data structures $\mathsf{LCP}^j_{\mathsf{rows}}$ and $\mathsf{LCP}^j_{\overleftarrow{\mathsf{rows}}}$ (similar to the above computation of vertical agreements), obtaining the minimal ℓ' and the maximal r' such that $M[u][\ell'..r'] = M[d][\ell'..r']$. The algorithm then constructs the horizontal aligned pair of segments $s^h_1 = (u, \ell', r')$ and $s^h_2 = (d, \ell', r')$. The algorithm queries SCDS for $(s^v_1, s^v_2) \leftarrow \mathsf{MaxCompatible}(s^h_1, s^h_2)$. Let $s^v_1 = (t_1, t_2, \ell)$ and $s^v_2 = (t_1, t_2, r)$. We call the frame (u, d, ℓ, r) the (u, d)-*optimal frame*. If the query $\mathsf{MaxCompatible}(s^h_1, s^h_2)$ returns null, there is no (u, d)-optimal frame. The algorithm reports the (u, d)-optimal frame with the maximum perimeter among all pairs (u, d), or returns null if no such frames were found.

Correctness. By construction, each frame (u, d, ℓ, r) identified by the algorithm is a (p, H, W)-frame. We proceed to show that it is a matching frame. Recall that (u, d, ℓ, r) was obtained from two compatible pairs of segments s^v_1, s^v_2 and s^h_1, s^h_2. Notice that for the pair $s^v_1 = (u_v, d_v, \ell)$ and $s^v_2 = (u_v, d_v, r)$ to be compatible with $s^h_1 = (u, \ell_h, r_h), s^h_2 = (d, \ell_h, r_h)$, the inequalities $u_v \leq u$ and $d_v \geq d$ must hold. By the construction of s^v_1 and s^v_2 we have $M[u_v..d_v][\ell] = M[u_v..d_v][r]$ and then $M[u..d][\ell] = M[u..d][r]$. In a similar way, one can prove $M[u][\ell..r] = M[d][\ell..r]$, showing that (u, d, ℓ, r) is a matching frame as required.

To conclude the correctness of our algorithm, we need to show that some maximum matching (p, H, W)-frame is (u, d)-optimal for some (u, d). Let (u_t, d_t, ℓ_t, r_t) be a maximum matching (p, H, W)-frame. For (u_t, d_t), the algorithm creates the horizontal aligned pair $s^h_1 = (u_t, \ell_h, r_h), s^h_2 = (d_t, \ell_h, r_h)$. Since $M[u_t][\ell_t..r_t] = M[d_t][\ell_t..r_t]$, we have $\ell_h \leq \ell_t$ and $r_h \geq r_t$. By a similar argument, when constructing the SCDS, the algorithm creates a vertical aligned pair $s^v_1 = (u_v, d_v, \ell_t)$, $s^v_2 = (u_v, d_v, r_t)$ with $u_v \leq u_t$ and $d_v \geq d_t$. Denote the output of $\mathsf{MaxCompatible}(s_1^h, s_2^h)$ by $\big((u', \ell', r'), (d', \ell', r')\big)$. One has $r' - \ell' \geq r_t - \ell_t$ since the pair (s_1^v, s_2^v) is compatible with (s_1^h, s_2^h). Then (u_t, d_t, ℓ', r') is a matching frame with perimeter $2(d_t - u_t + r' - \ell') \geq 2(d_t - u_t + r_t - \ell_t)$. Due to the maximality of the perimeter of (u_t, d_t, ℓ_t, r_t), we have that (u_t, d_t, ℓ', r') is a maximum matching (p, H, W)-frame.

Complexity. It can be easily shown that there are $O(W^2)$ pairs (ℓ, r) satisfying $r - \ell \leq W$ and $j \in [\ell..r]$. Similarly, there are $O(H^2)$ pairs (u, d) satisfying $d - u \leq H$ and $i \in [u..d]$. By Lemma 10, the construction of the SCDS takes $\tilde{O}(W^2)$ time. The algorithm then applies $O(H^2)$ queries to the SCDS and the overall complexity is $\tilde{O}(W^2 + H^2)$. The additional space usage of the algorithm is dominated by the SCDS data structure of size $\tilde{O}(W^2)$. ◀

Proof of Lemma 12. The algorithm iterates over all pairs $H, W \in \{x \cdot 2^k \mid k \geq 1\}$ such that $H \leq W < 2m$. For a pair (H, W), the algorithm runs the subroutine from Lemma 13 for every position $(i, j) \in [n] \times [m]$ such that $i \bmod H/2 = 0$ and $j \bmod W/2 = 0$. Finally, the algorithm reports the maximum matching frame among all outputs of this subroutine.

Correctness. Since every instance of the subroutine from Lemma 13 reports a matching frame or a null, the algorithm also reports a matching frame (or a null). Let $F = (u, d, \ell, r)$ be a maximum matching frame of height at least x. Let W (resp. H) be the smallest number in $\{x \cdot 2^k \mid k \geq 1\}$ which is at least $r - \ell$ (resp. $d - u$). Then there exist $i \in [u..d]$ and $j \in [\ell..r]$ such that $i \bmod H/2 = 0$ and $j \bmod W/2 = 0$. Hence the algorithm ran the subroutine for $((i, j), H, W)$-frames and got reported a matching frame F' with $\mathsf{per}(F') \geq \mathsf{per}(F)$. Therefore, the algorithm returns a maximum matching frame.

Complexity. For a given pair (H, W), the subroutine of Lemma 13 was called for $\lfloor \frac{2n}{H} \rfloor \cdot \lfloor \frac{2m}{W} \rfloor$ points (i, j). In total, these calls cost $\tilde{O}\big(\frac{nm}{HW}(W^2 + H^2)\big) = \tilde{O}\big(nm\frac{W}{H}\big)$ time. Therefore, the algorithm runs in $\tilde{O}(nm) \cdot \sum_{H,W} \frac{W}{H}$ time, where the summation is over all possible pairs. Let $t = \lceil \log \frac{m}{x} \rceil$. Since $x \leq H \leq W < 2m$, we have $\sum_{H,W} \frac{W}{H} = 2^t + 2 \cdot 2^{t-1} + 3 \cdot 2^{t-2} + \cdots \leq 4 \cdot 2^t = O(\frac{m}{x})$. The time bound from the lemma now follows. The additional space usage of the algorithm is dominated by the space of the largest instance of Lemma 13, which is $\tilde{O}(W^2)$ for some W. Since $W < 2m$, we have the required bound $\tilde{O}(m^2)$. ◀

5.3 Combining the Short and Tall Algorithms

In this section, we combine the results of Section 5.1 and Section 5.2 to prove Theorem 1.

Proof of Theorem 1. Applying the algorithm of Lemma 11 and the algorithm of Lemma 12 with the same threshold $x = \sqrt{m}$ and reporting the maximum frame between both outputs yields an algorithm with running time $\tilde{O}(nm \cdot \sqrt{m})$. We run the same scheme for the transposed matrix $M^\top$ and $x = \sqrt{n}$, which takes $\tilde{O}(nm \cdot \sqrt{n})$ time. In total, processing both M and $M^\top$ takes $\tilde{O}(nm \cdot \sqrt{\max\{n, m\}})$ time. The space usage of the algorithm is dominated by the preprocessed data, which takes $\tilde{O}(nm)$ space.

Notice that $d - u \leq r - \ell$ for all considered frames, yielding $d - u \leq \min\{n, m\}$. Therefore, applying Lemma 11 to both M and $M^\top$ with $x = \min\{n, m\}$ provides an alternative algorithm that outputs the maximum matching frame within $\tilde{O}(nm \cdot \min\{n, m\})$ time. Choosing the faster between the two above algorithms implies Theorem 1. ◀

6 Approximation Version

In the $(1-\varepsilon)$-approximation version of the problem, the goal is to find, given a matrix M with a maximum matching frame F, a matching frame F' in M with $\mathsf{per}(F') \geq (1-\varepsilon)\mathsf{per}(F)$. Our algorithm reduces the problem to multiple instances of a decision problem defined below. The reduction is shown in Lemma 15 below and the decision problem is solved in Section 6.3.

Decision problem. The input for this problem is a matrix M, and an *inner rectangle* $(u_\square, d_\square, \ell_\square, r_\square)$ in M. A frame (u, d, ℓ, r) in M is *surrounding* if $(u_\square, d_\square, \ell_\square, r_\square)$ is strictly inside it; formally, if $u < u_\square \leq d_\square < d$ and $\ell < \ell_\square \leq r_\square < r$. The goal in this version of the problem is to output a surrounding matching frame (u, d, ℓ, r) or report that no such frame exists in M. In Section 6.3, we show that this problem can be solved in near-linear time, by proving the following lemma.

▶ **Lemma 14.** *Given an $n \times m$ matrix M with an inner rectangle $(u_\square, d_\square, \ell_\square, r_\square)$, there is an algorithm that finds, in $\tilde{O}(nm)$ time and space, a surrounding matching frame in M or reports that no such frame exists.*

Via an application of a 2-dimensional variant of the so-called *standard trick* [15, 12], we obtain the following reduction.

▶ **Lemma 15.** *Let $a = 1 + \varepsilon/3$. For every $(h, w) \in [\log_a n] \times [\log_a m]$ such that $a^h, a^w \geq 2$, there is a set $\mathcal{M}_{h,w}$ of sub-matrices, each associated with an inner rectangle, such that the following properties are satisfied:*

1. $|\mathcal{M}_{h,w}| = O(\frac{nm}{\varepsilon^2 a^{h+w}})$.
2. *For every sub-matrix $M' \in \mathcal{M}_{h,w}$, $|M'| = O(a^{h+w})$.*
3. *For every frame (u, d, ℓ, r) with $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$ there is a sub-matrix $M' \in \mathcal{M}_{h,w}$ such that (u, d, ℓ, r) is a surrounding frame in M' with respect to its inner rectangle.*
4. *For every surrounding frame F in any $M' \in \mathcal{M}_{h,w}$, $\mathsf{per}(F) \geq (1-\varepsilon)\big(2(a^{w+1} + a^{h+1})\big)$.*

The inner rectangles and the corners of the sub-matrices in $\mathcal{M}_{h,w}$ can be obtained in $O(|\mathcal{M}_{h,w}|)$ time and space given h and w.

Proof. Fix $(h, w) \in [\log_a n] \times [\log_a m]$. We define several numeric values that are used repeatedly by our reduction, namely $\delta_w = \left\lfloor \frac{\varepsilon a^{w+1}}{3} \right\rfloor$, $\delta_h = \left\lfloor \frac{\varepsilon a^{h+1}}{3} \right\rfloor$, $W_w = \lceil a^{w+2} \rceil$, and $H_h = \lceil a^{h+2} \rceil$. For convenience, assume without loss of generality that both $\frac{n-H_h}{\delta_h}$ and $\frac{m-W_w}{\delta_w}$ are integers. Otherwise, the algorithm adds dummy rows and columns to the right and to the bottom sides of the matrix with distinct unique characters not in Σ until δ_h divides $n - H_h$ and δ_w divides $m - W_w$. The set $M_{h,w}$ of sub-matrices of M is defined as follows:

$$\mathcal{M}_{h,w} = \left\{ M[\alpha\delta_h + 1..\alpha\delta_h + H_h][\beta\delta_w + 1..\beta\delta_w + W_w] \mid \alpha \in [0..\tfrac{n-H_h}{\delta_h}] \text{ and } \beta \in [0..\tfrac{m-W_w}{\delta_w}] \right\}.$$

In words, those are all sub-matrices with width $W_w - 1$ and height $H_h - 1$, having their upper left corner in a cell (x', y') of M such that $x' \bmod \delta_h = y' \bmod \delta_w = 1$. Note that Properties 1 and 2 are trivially satisfied. Additionally, it is clear that the corners of each sub-matrix can be obtained in constant time.

Property 3 is obtained by combining the following two claims.

▷ Claim 16. Every frame (u, d, ℓ, r) with $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$ is contained in some $M' \in \mathcal{M}_{h,w}$.

Proof. Let x (resp. y) be the largest integer multiple of δ_h (resp. δ_w) that is smaller than u (resp. ℓ). By definition, $\mathcal{M}_{h,w}$ contains a sub-matrix $M' = M[x+1..x+H_h][y+1..y+W_w]$. In order to prove that (u,d,ℓ,r) is fully contained inside M', we need to show that (1) $x < u$, (2) $y < \ell$, (3) $x + H_h \geq d$ and (4) $y + W_w \geq r$. Conditions (1), (2) are immediate from the choice of x and y. Let us show (3). The choice of x also implies $x + \delta_h \geq u$. Therefore,

$$x + H_h \geq u - \delta_h + H_h = u - \left\lfloor \frac{\varepsilon a^{h+1}}{3} \right\rfloor + \lceil a^{h+2} \rceil$$
$$\geq u - \frac{\varepsilon a^{h+1}}{3} + a^{h+2} = u + a^{h+1}\left(a - \frac{\varepsilon}{3}\right) = u + a^{h+1}.$$

By conditions of the lemma, $d - u < a^{h+1}$, so we obtain $x + H_h > d$ as required. Condition (4) can be shown in the same way. ◁

For each sub-matrix $M' = M[x+1..x+H_h][y+1..y+W_w]$ we define the inner rectangle $R_\square = (u_\square, d_\square, \ell_\square, r_\square) = (x + H_h - \lceil a^h \rceil + 1, x + \lceil a^h \rceil - 1, y + W_w - \lceil a^w \rceil + 1, y + \lceil a^w \rceil - 1)$. As the further argument does not depend on x, y, we assume $x = y = 0$ for simplicity.

▷ Claim 17. If (u,d,ℓ,r) is a frame in M' with $r - \ell \in [a^w..a^{w+1}-1]$ and $d - u \in [a^h..a^{h+1}-1]$, then (u,d,ℓ,r) is a surrounding frame.

Proof. Since $d \leq H_h$ and $d - u \geq a^h$, one has $u \leq d - a^h \leq H_h - a^h < u_\square$, as required. Since $u \geq 1$, one also has $d \geq a^h + 1 > d_\square$ as required. The inequalities $\ell < \ell_\square$ and $r > r_\square$ are proved in the same way, so (u,d,ℓ,r) is surrounding by definition. ◁

To prove Property 4, we note that the perimeter of a surrounding frame in M' is at least $2((d_\square - u_\square + 2) + (r_\square - \ell_\square + 2))$. We show that $d_\square - u_\square + 2 \geq (1-\varepsilon) \cdot a^{h+1}$. It can be similarly argued that $r_\square - \ell_\square + 2 \geq (1-\varepsilon) \cdot a^{w+1}$; the two inequalities together yield Property 4. Recall that $u_\square = H_h - \lceil a^h \rceil + 1$, $d_\square = \lceil a^h \rceil - 1$, $H_h = \lceil a^{h+2} \rceil$. Then

$$d_\square - u_\square + 2 = \lceil a^h \rceil - 1 - H_h + \lceil a^h \rceil - 1 + 2 \geq 2a^h - a^{h+2} = a^{h+1}\left(\tfrac{2}{a} - a\right)$$

It remains to show that $\frac{2}{a} - a \geq 1 - \varepsilon$. Indeed,

$$\frac{2}{a} - a = \frac{2 - (1 + 2\varepsilon/3 + \varepsilon^2/9)}{1 + \varepsilon/3} = \frac{1 - 2\varepsilon/3 - \varepsilon^2/3 + 2\varepsilon^2/9}{1 + \varepsilon/3} = 1 - \varepsilon + \frac{2\varepsilon^2/9}{1 + \varepsilon/3} > 1 - \varepsilon,$$

as required. The lemma is proved. ◀

With Lemmas 14 and 15, we are ready to prove Theorem 2.

Proof of Theorem 2. The algorithm first processes frames of height 1 or width 1, applying the algorithm of Lemma 11 with $x = 1$ to both M and $M^\top$. After that, the algorithm proceeds as follows. For every pair $(h,w) \in [\log_a n] \times [\log_a m]$ such that $a^w, a^h \geq 2$, it creates the set $\mathcal{M}_{h,w}$ with the corresponding inner rectangles (see Lemma 15) and applies Lemma 14 on every $M' \in \mathcal{M}_{h,w}$ with its inner rectangle. The algorithm returns the maximum frame among the matching frames returned by algorithms of Lemma 11 and Lemma 14. If neither of these two algorithms reported a frame, then a "no frames" answer is reported.

Correctness. Let $F = (u, d, \ell, r)$ be a maximum matching frame in M. If $d = u + 1$ or $r = \ell + 1$, then F is found by the algorithm of Lemma 11. Otherwise, consider the pair $(h, w) \in [\log_a n] \times [\log_a m]$ such that $d - u \in [a^h..a^{h+1} - 1]$ and $r - \ell \in [a^w..a^{w+1} - 1]$. By Property 3 of Lemma 15, there is a sub-matrix $M' \in \mathcal{M}_{h,w}$ that contains F as a surrounding frame. The algorithm in Lemma 14 returns a surrounding matching frame F' in M', and by Property 4 of Lemma 15, $\mathsf{per}(F') \geq (1-\varepsilon)\big(2(a^{w+1} + a^{h+1})\big)$. Since $\mathsf{per}(F) < 2(a^{w+1} + a^{h+1})$, the approximation guarantee is fulfilled.

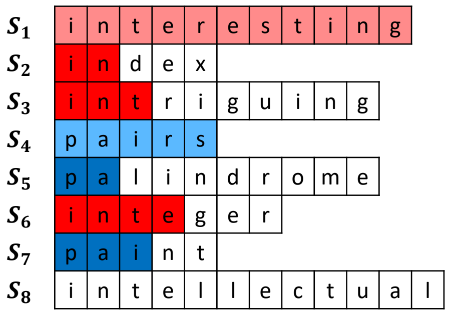
Complexity. Given h and w, the running time of the algorithm that obtains $\mathcal{M}_{h,w}$ and the suitable $R_\square$ is $O(|\mathcal{M}_{h,w}|) \subseteq O(nm/\varepsilon^2)$ by Property 1 of Lemma 15.

Due to Properties 1 and 2 of Lemma 15, the sum of the sizes of the matrices in $\mathcal{M}_{h,w}$ is $O\left(\frac{nm}{\varepsilon^2}\right)$. Hence, applying Lemma 14 on all $M' \in \mathcal{M}_{h,w}$ takes $\tilde{O}\left(\frac{nm}{\varepsilon^2}\right)$ time. Recall that there are $O(\log_{1+\varepsilon} n \cdot \log_{1+\varepsilon} m) = O(\frac{1}{\varepsilon^2} \log n \cdot \log m)$ values of h and w. Thus, the total running time of the algorithm is $\tilde{O}(\frac{nm}{\varepsilon^4})$. Each matrix in $\mathcal{M}_{h,w}$ is processed separately. The space complexity of processing a matrix is $\tilde{O}(a^{h+w}) = \tilde{O}(nm)$. The space is reused when each matrix is processed, so the overall space complexity of the algorithm is $\tilde{O}(nm)$. ◀

6.1 Interesting Pairs and Interesting Triplets

In order to prove Lemma 14, we introduce and study the following notion, illustrated by Figure 3.

▶ **Definition 18.** *Given a tuple $(S_1, \ldots, S_n)$ of strings, we call a pair (i, j)* interesting *if $i < j$ and for any ℓ such that $\ell \in [i + 1, j - 1]$ one has $\mathsf{LCP}(S_i, S_\ell) < \mathsf{LCP}(S_i, S_j)$.*



S_1 i n t e r e s t i n g
S_2 i n d e x
S_3 i n t r i g u i n g
S_4 p a i r s
S_5 p a l i n d r o m e
S_6 i n t e g e r
S_7 p a i n t
S_8 i n t e l l e c t u a l

Figure 3 An example of interesting pairs where the first component of the pair is S_1 or S_4. The rows beginning in red form interesting pairs with S_1 and the rows beginning in blue form interesting pairs with S_4. The color indicates the LCP of the components of the pair. Notice that (S_1, S_8) is not an interesting pair because of S_6.

Trivially, all pairs of the form $(i, i + 1)$ are interesting for any tuple. The next lemma bounds the number of interesting pairs. This bound is tight as shown in the full version [9] of this paper.

▶ **Lemma 19.** *For each n-tuple of strings, there are $O(n \log n)$ interesting pairs.*

Proof. For a given tuple $(S_1, \ldots, S_n)$, fix an integer $\ell \in [1..\lceil \log n \rceil]$ and consider the set $\mathcal{I}_\ell = \{(i, j) \mid (i, j)$ is interesting and $j - i \in [2^{\ell-1}..2^\ell - 1]\}$. We say that a pair $(i, j) \in \mathcal{I}_\ell$ is of the *first type* if $i = \max\{i' \mid (i', j) \in \mathcal{I}_\ell\}$ and of the *second type* otherwise. The following claim is crucial.

▷ Claim 20. All pairs of the first type from $\mathcal{I}_\ell$ have different second components; all pairs of the second type from $\mathcal{I}_\ell$ have different first components.

Proof. The first statement stems directly from the definition of the first type. Let us prove the second one. Assume by contradiction that $(i,j),(i,j') \in \mathcal{I}_\ell$ are pairs of the second type, with $j' < j$. As (i,j) is not of the first type, $\mathcal{I}_\ell$ contains a pair (i',j) with $i' > i$. We prove the following sequence of inequalities, leading to a contradiction.

$$\mathsf{LCP}(S_i,S_{i'}) \overset{(1)}{<} \mathsf{LCP}(S_i,S_{j'}) \overset{(2)}{=} \mathsf{LCP}(S_{j'},S_j) \overset{(3)}{=} \mathsf{LCP}(S_{i'},S_{j'}) \overset{(4)}{<} \mathsf{LCP}(S_{i'},S_j) \overset{(5)}{=} \mathsf{LCP}(S_i,S_{i'}),$$

Since $2^{\ell-1} \le j-i'$, $2^{\ell-1} \le j'-i$ and $j-i < 2^\ell \le j-i' + j'-i$, we have $i' < j'$. Since (i,j') is an interesting pair and $i' \in [i{+}1..j'{-}1]$, we obtain (1) by Definition 18. Since (i,j) is an interesting pair, every $k \in [i{+}1..j{-}1]$ satisfies $\mathsf{LCP}(S_i,S_k) < \mathsf{LCP}(S_i,S_j)$. Hence, by Fact 6 we have $\mathsf{LCP}(S_i,S_k) = \mathsf{LCP}(S_k,S_j)$. We obtain (2) and (5) by setting $k = j'$ and $k = i'$ respectively. Finally, (i',j) is an interesting pair, and $j' \in [i'+1..j-1]$. So, Definition 18 gives us (4) and then Fact 6 implies (3). ◁

Claim 20 says that $\mathcal{I}_\ell$ contains at most n pairs of the first type and at most n pairs of the second type. As ℓ takes $\lceil \log n \rceil$ values, the lemma follows. ◀

To relate interesting pairs to our decision problem we need one more notion.

▶ **Definition 21.** *Let M be an $n{\times}m$-matrix and $\ell \in [m]$. A triplet (u,d,ℓ) is called* interesting *if the pair (u,d) is interesting for the tuple $(M[1][\ell..m],\dots,M[n][\ell..m])$.*

6.2 Finding all interesting triplets

▶ **Lemma 22.** *All interesting triplets for an $n \times m$ matrix M can be found in $\tilde{O}(nm)$ time.*

We assume that the data structures described in Section 3 are constructed. We process each $\ell \in [m]$ independently, computing all interesting triplets of the form (u,d,ℓ). By Definition 21, such a triplet is interesting if the pair (u,d) is interesting for the tuple $\mathcal{S} = (S_1,\dots,S_n)$, where $S_i = M[i][\ell..m]$. Below we work with this fixed tuple $\mathcal{S}$. The algorithm scans $\mathcal{S}$ string by string; while processing S_i, the algorithm finds all the interesting pairs (i,j).

For $i < j \in [n]$, let $L(i,j)$ be the maximum LCP value between S_i and any S_k for $k \in [i+1\dots j]$. Let $I(i,j) = \min\{k \in [i+1\dots j] \mid \mathsf{LCP}(S_i,S_k) = L(i,j)\}$ be the minimum index k with this maximum LCP value. Using the function $I(i,j)$ we characterize the set of interesting pairs that share the first index i.

▶ **Lemma 23.** *For $i \in [n]$, let $j_1 > j_2 > \cdots > j_z$ be the second coordinates of all interesting pairs of the form (i,j). Then $j_1 = I(i,n)$ and $j_k = I(i,j_{k-1}-1)$ for every $k \in [2..z]$.*

Proof. First we need to prove that $(i,I(i,n))$ is interesting and that there is no interesting pair (i,j') with $j' > I(i,n)$. By the definitions of $L(i,n)$ and $I(i,n)$, for every $j' < I(i,n)$ we have $\mathsf{LCP}(S_i,S_{j'}) < L(i,n) = \mathsf{LCP}(S_i,S_{I(i,n)})$, so $(i,I(i,j))$ is interesting. Now consider a pair (i,j') with $j' > I(i,n)$. The same definitions imply $\mathsf{LCP}(S_i,S_{j'}) \le L(i,n) = \mathsf{LCP}(S_i,S_{I(i,n)})$, so the pair $(S_i,S_{j'})$ is not interesting and we have $j_1 = I(i,n)$ as required.

Let $k \in [2..z]$ and consider the second statement. Similar to the above, we argue that the pair $(i,I(i,j_{k-1}-1))$ is interesting and no pair (i,j') such that $I(i,j_{k-1}-1) < j' < j_{k-1}$ is interesting. Hence $I(i,j_{k-1}-1)$ follows j_{k-1} in the list of second coordinates of interesting pairs of the form (i,j), i.e., $j_k = I(i,j_{k-1}-1)$. ◀

We proceed to show how to compute $I(i,j)$ and $L(i,j)$ efficiently.

▶ **Lemma 24.** *Given i and j, $L(i,j)$ can be computed in $O(\log n)$ time.*

Proof. Note that if we lex-sort the tuple $(S_i, \dots, S_j)$, then the maximum LCP value with S_i would be reached by one of its neighbors $S_{j_{\mathsf{left}}}$ and $S_{j_{\mathsf{right}}}$ in the sorted tuple; we assume $S_{j_{\mathsf{left}}} < S_i < S_{j_{\mathsf{right}}}$ (one neighbor may absent). Thus, $L(i,j) = \max\{\mathsf{LCP}(S_i, S_{j_{\mathsf{left}}}), \mathsf{LCP}(S_i, S_{j_{\mathsf{right}}})\}$. The algorithm retrieves j_{left} and j_{right} using range queries on D^{ℓ}_{rows} as detailed below.

Recall that $I^{x,\ell}_{\mathsf{rows}}$ denotes the index of S_x in $\mathsf{LSA}^{\ell}_{\mathsf{rows}}$. Note that $I^{j_{\mathsf{right}},\ell}_{\mathsf{rows}}$ is the minimal index satisfying $I^{x,\ell}_{\mathsf{rows}} > I^{i,\ell}_{\mathsf{rows}}$ with $x \in [i+1..j]$. Hence, in order to get j_{right} one queries D^{ℓ}_{rows} for a point $(x, I^{x,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [I^{i,\ell}_{\mathsf{rows}}+1..\infty]$ that minimizes $I^{x,\ell}_{\mathsf{rows}}$; the first coordinate of this point is j_{right}. Symmetrically, in order to get j_{left} one queries D^{ℓ}_{rows} for a point $(x, I^{x,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [1..I^{i,\ell}_{\mathsf{rows}}-1]$ that maximizes $I^{x,\ell}_{\mathsf{rows}}$; the first coordinate of this point is j_{left}. After retrieving j_{right} and j_{left}, one queries the $\mathsf{LCP}^{\ell}_{\mathsf{rows}}$ structure for $\mathsf{LCP}(S_i, S_{j_{\mathsf{right}}})$ and $\mathsf{LCP}(S_i, S_{j_{\mathsf{left}}})$, and outputs the maximum as $L(i,j)$.

Two range queries take $O(\log n)$ time (Lemma 9 for $d = 2$) while two LCP queries take $O(1)$ time (Lemma 5). The lemma now follows. ◀

▶ **Lemma 25.** *Given i and j, $I(i,j)$ can be computed in $O(\log n)$ time.*

Proof. The algorithm starts by applying Lemma 24 to obtain $L(i,j)$ in $O(\log n)$ time. Let $P = S_i[1..L(i,j)]$ be the prefix of length $L(i,j)$ of S_i. Recall that by definition, $I(i,j)$ is the minimal index $k \in [i+1..j]$ such that $S_k[1..L(i,j)] = P$. Using Fact 7, the algorithm finds, in $O(\log n)$ time, a pair of indices i_P, j_P such that $S_z[1..L(i,j)] = P$ if and only if $I^{z,\ell}_{\mathsf{rows}} \in [i_P..j_P]$. After that, the algorithm retrieves $I(i,j)$ by querying D^{ℓ}_{rows} for the point $(k, I^{k,\ell}_{\mathsf{rows}})$ in the range $[i+1..j] \times [i_P..j_P]$ with the minimal first coordinate. This coordinate k is then reported as $I(i,j)$. As this query takes $O(\log n)$ time by Lemma 9 for $d = 2$, the lemma follows. ◀

Proof of Lemma 22. Let ℓ be fixed and $\mathcal{S} = \{S_1, \dots, S_n\}$ be defined as above. For each S_i, the algorithm finds $j_1 = I(i,n)$ using Lemma 25, reports (i, j_1) as an interesting pair (see Lemma 23), and then iterate. As long as $j_k \neq i+1$, the algorithm finds $j_{k+1} = I(i, j_k - 1)$ using Lemma 25 and reports the interesting pair (i, j_{k+1}). Note that the algorithm is guaranteed to finish the iteration, since the pair $(i, i+1)$ is interesting.

The algorithm spends $O(\log n)$ time per interesting pair by Lemma 22; the number of such pairs is $O(n \log n)$ by Lemma 19. Multiplying this by m choices for ℓ, we obtain the required time bound $\tilde{O}(nm)$. ◀

6.3 Algorithm for the Decision Variant

In this section we prove Lemma 14, presenting the required algorithm.

The algorithm starts by modifying M as follows. For every $(i,j) \in [u_\square \dots d_\square] \times [\ell_\square \dots r_\square]$, we set $M[i][j] = \$_{i,j}$ with $\$_{i,j}$ being a unique symbol not in Σ. Since neither of the changed symbols belongs to a marginal row/column of a surrounding frame, this modification preserves surrounding matching frames. The following claim clarifies the role of interesting triplets.

▶ **Lemma 26.** *If a matrix M with an inner rectangle $(u_\square, d_\square, \ell_\square, r_\square)$ contains a surrounding matching frame (u, d, ℓ, r), then it contains a surrounding matching frame (u', d', ℓ, r) such that (u', d', ℓ) is an interesting triplet.*

Proof. Let (u, d, ℓ, r) be a surrounding matching frame in M. We denote $S_h = M[u][\ell..r] = M[d][\ell..r]$. Let u' be the maximal index in $[u..u_\square - 1]$ such that $M[u'][\ell..r] = S_h$ and let d' be the minimal index in $[d_\square + 1..d]$ such that $M[d'][\ell..r] = S_h$. The frame (u', d', ℓ, r) is surrounding by definition and matching by construction (note that $M[u..d][\ell] = M[u..d][r]$ implies $M[u'..d'][\ell] = M[u'..d'][r]$). Finally, for arbitrary $d'' \in [u'+1..d'-1]$ one has $M[d''][\ell..r] \neq S_h$. If $d'' < u_\square$ or $d'' > d_\square$, this condition holds by the choice of u' and d' respectively. Otherwise the condition is guaranteed by uniqueness of the symbols of the inner rectangle. Hence $\mathsf{LCP}(M[u'][\ell..m], M[d''][\ell..m]) < |S_h| \leq \mathsf{LCP}(M[u'][\ell..m], M[d'][\ell..m])$, and the triplet (u', d', ℓ) is interesting by definition. ◀

The Algorithm. After setting $M[i][j] = \$_{i,j}$ for each $(i,j) \in [u_\square \ldots d_\square] \times [\ell_\square \ldots r_\square]$, the algorithm applies the preprocessing described in Section 3 and finds all interesting triplets in $O(nm\log^2 n)$ time by applying Lemma 22. The final ingredient we need is a mechanism verifying, given an interesting triplet (u, d, ℓ), if there is a surrounding matching frame (u, d, ℓ, r). For this purpose, we present the following lemma.

▶ **Lemma 27.** *There is an algorithm that, given an interesting triplet (u, d, ℓ) of M, outputs an integer r such that (u, d, ℓ, r) is a surrounding matching frame or reports* null *if no such r exists. The algorithm runs in $O(\log n)$ time.*

Proof. The algorithm reports null if $u \geq u_\square$, or $d \leq d_\square$, or $\ell \geq \ell_\square$. Otherwise, it seeks for a value r such that (i) $r \geq r_\square + 1$, (ii) $M[u][\ell..r] = M[d][\ell..r]$, and (iii) $M[u..d][r] = M[u..d][\ell]$.

The algorithm queries $\mathsf{LCP}^{\ell}_{\mathsf{rows}}$ for $L_{u,d} = \mathsf{LCP}(M[u][\ell..m], M[d][\ell..m])$. By definition of LCP, we have $M[u][\ell..r] = M[d][\ell..r]$ if and only if $r \leq \ell + L_{u,d} - 1$. Hence, conditions (i) and (ii) are satisfied if and only if $r \in [r_\square + 1 \ldots \ell + L_{u,d} - 1]$. To check (iii), let $S_v = M[u..d][\ell]$. Using Fact 7, the algorithm finds the pair of indices i_v, j_v such that $M[u..d][r] = S_v$ if and only if $r \in \mathsf{LSA}^{u}_{\mathsf{columns}}[i_v..j_v]$. Now the algorithm checks the existence of a value r satisfying (i)–(iii) by querying D^{u}_{columns} for a point within the range $[r_\square + 1..\ell + L_{u,d} - 1] \times [i_v..j_v]$. If the queried structure returns a point $(r, I^{u,r}_{\mathsf{columns}})$, the algorithm outputs r; otherwise, it reports null, as there is no value of r such that (u, d, ℓ, r) is a surrounding matching frame.

The algorithm performs a single LCP query ($O(1)$ time by Lemma 5), finds i_v and j_v ($O(\log n)$ time by Fact 7), queries D^{u}_{columns} ($O(\log n)$ time by Lemma 9), and compares a constant number of integers. The lemma follows. ◀

We are finally ready to prove Lemma 14.

Proof of Lemma 14. After finding all interesting triplets, the algorithm applies the subroutine from Lemma 27 to every interesting triplet (u, d, ℓ). If this subroutine outputs r, the algorithm outputs the surrounding matching frame (u, d, ℓ, r). If the subroutine outputs null for all interesting triplets, then, relying on Lemma 26, the algorithm reports that no surrounding matching frame exists.

The algorithm spends $O(nm\log^2(nm))$ for each of three tasks it performs: preprocessing (Section 3), finding interesting triplets (Lemma 22), and verifying interesting triplets (Lemma 19 and Lemma 27). Thus, its time (and therefore, space) complexity is $\tilde{O}(nm)$, as required. ◀

References

1 Georgii Maksimovich Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, 146:263–266, 1962.

2 Amihood Amir and Gary Benson. Two-dimensional periodicity and its applications. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 440–452, 1992.

3 Amihood Amir and Gary Benson. Two-dimensional periodicity in rectangular arrays. *SIAM Journal on Computing*, 27(1):90–106, 1998.

4 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA*, volume 144 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

5 Amihood Amir, Gad M Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. *Theoretical Computer Science*, 812:49–61, 2020.

6 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.

7 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.

8 Robert Berger. *The undecidability of the domino problem.* Amer. Math. Soc., 1966.

9 Itai Boneh, Dvir Fried, Shay Golan, Matan Kraus, Adrian Miclaus, and Arseny Shur. Searching 2d-strings for matching frames. *arXiv preprint arXiv:2310.02670*, 2023.

10 Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic length in linear time. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.23.

11 Sarah Cannon, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert T. Schweller, Scott M. Summers, and Andrew Winslow. Two hands are better than one (up to constant factors): Self-assembly in the 2HAM vs. aTAM. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 172–184. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPICS.STACS.2013.172.

12 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022*, volume 244 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ESA.2022.35.

13 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. The number of repetitions in 2d-strings. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA*, volume 173 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.32.

14 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.

15 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.

16 Jonas Ellert and Johannes Fischer. Linear time runs over general ordered alphabets. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.63.

17 Jonas Ellert, Pawel Gawrychowski, and Garance Gourdel. Optimal square detection over general alphabets. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, pages 5220–5242. SIAM, 2023. doi:10.1137/1.9781611977554.ch189.

18 Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

19 Guilhem Gamard, Gwénaël Richomme, Jeffrey O. Shallit, and Taylor J. Smith. Periodicity in rectangular arrays. *Inf. Process. Lett.*, 118:58–63, 2017. doi:10.1016/J.IPL.2016.09.011.

20 Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 54:1–54:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.54.

21 Pawel Gawrychowski, Samah Ghazawi, and Gad M. Landau. Lower bounds for the number of repetitions in 2d strings. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE*, volume 12944 of *Lecture Notes in Computer Science*, pages 179–192. Springer, 2021. doi:10.1007/978-3-030-86692-1_15.

22 Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. doi:10.1007/S00453-019-00591-8.

23 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

24 Emmanuel Jeandel and Michaël Rao. An aperiodic set of 11 Wang tiles. *CoRR*, abs/1506.06492, 2015. arXiv:1506.06492.

25 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.

26 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 2003. doi:10.1007/3-540-44888-8_14.

27 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.

28 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/j.jda.2004.08.002.

29 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.

30 Manasi S Kulkarni and Kalpana Mahalingam. Two-dimensional palindromes and their properties. In *International Conference on Language and Automata Theory and Applications*, pages 155–167. Springer, 2017.

31 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

32 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990*, pages 319–327. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

33 Oleg Merkurev and Arseny M. Shur. Searching runs in streams. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2019. doi:10.1007/978-3-030-32686-9_15.

34 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, *2009 Data Compression Conference (DCC 2009)*, pages 193–202. IEEE Computer Society, 2009.

35 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 54–67. Springer, 2009.

36 Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 459–468. ACM, 2000. doi:10.1145/335305.335358.

37 Mikhail Rubinchik and Arseny M. Shur. Palindromic k-factorization in pure linear time. In Javier Esparza and Daniel Kráľ, editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 81:1–81:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.MFCS.2020.81.

38 Taylor Smith. Properties of two-dimensional words. Master's thesis, University of Waterloo, 2017.

39 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

40 Hao Wang. Proving theorems by pattern recognition II. *Bell System Tech. J.*, 40:1–41, 1961.

41 Hao Wang. Games, logic and computers. *Scientific American*, 213(5):107, 1965.

42 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

43 Dan E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14(1):232–253, 1985. doi:10.1137/0214019.

Hairpin Completion Distance Lower Bound

Itai Boneh
Reichman University and University of Haifa, Israel

Dvir Fried
Bar Ilan University, Ramat Gan, Israel

Shay Golan
Reichman University and University of Haifa, Israel

Matan Kraus
Bar Ilan University, Ramat Gan, Israel

Abstract

Hairpin completion, derived from the hairpin formation observed in DNA biochemistry, is an operation applied to strings, particularly useful in DNA computing. Conceptually, a right hairpin completion operation transforms a string S into $S \cdot S'$ where S' is the reverse complement of a prefix of S. Similarly, a left hairpin completion operation transforms a string S into $S' \cdot S$ where S' is the reverse complement of a suffix of S. The hairpin completion distance from S to T is the minimum number of hairpin completion operations needed to transform S into T. Recently Boneh et al. [3] showed an $O(n^2)$ time algorithm for finding the hairpin completion distance between two strings of length at most n. In this paper we show that for any $\varepsilon > 0$ there is no $O(n^{2-\varepsilon})$-time algorithm for the hairpin completion distance problem unless the Strong Exponential Time Hypothesis (SETH) is false. Thus, under SETH, the time complexity of the hairpin completion distance problem is quadratic, up to sub-polynomial factors.

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

Keywords and phrases Fine-grained complexity, Hairpin completion, LCS

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.11

Related Version *Full Version*: `https://arxiv.org/abs/2404.11673` [2]

Funding *Itai Boneh*: supported by Israel Science Foundation grant 810/21.
Dvir Fried: supported by ISF grant no. 1926/19, by a BSF grant 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).
Shay Golan: supported by Israel Science Foundation grant 810/21.
Matan Kraus: supported by the ISF grant no. 1926/19, by the BSF grant 2018364, and by the ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

1 Introduction

Hairpin completion [6], derived from the hairpin formation observed in DNA biochemistry, is an operation applied to strings, particularly useful in DNA computing [10, 9, 8, 7]. Consider a sequences over an alphabet Σ with involution $\mathsf{Inv} : \Sigma \to \Sigma$ assigning for every $\sigma \in \Sigma$ an inverse symbol $\overline{\sigma}$. For a string $S \in \Sigma^*$, a left hairpin completion transforms S into $\overleftarrow{S'} \cdot S$, where S' is a suffix of S, and for any $X \in \Sigma^*$ we define $\overleftarrow{X} = \overline{X[|X|]} \cdot \overline{X[|X|-1]} \cdot \ldots \cdot \overline{X[1]}$. This operation can only be applied under the restriction that the suffix S' is preceded by the symbol $\overline{S[1]}$. Similarly, a right hairpin completion transforms S into $S \cdot \overleftarrow{S'}$ where S' is a prefix of S followed by $\overline{S[|S|]}$.

Several problems regarding hairpin completion were studied [11, 12, 13, 14, 3]. In this paper, we consider the hairpin completion distance problem. In this problem, we are given two strings x and y and our goal is to compute the minimum number of hairpin completion operations one has to apply on y to transform y into x, or to report that there is no sequence of hairpin completion operation can turn y into x. In 2009, Manea, Martín-Vide and Mitrana [12] proposed the problem and introduced a cubic time $O(n^3)$ algorithm (where $n = |x|$). Later Manea [11] introduced a faster algorithm that runs in $O(n^2 \log n)$ time. Recently, Boneh et al. [3] showed that the time complexity of the problem is $O(n^2)$. Moreover, Boneh et al. posed the following open problem.

▶ **Problem 1.** *Can one prove a lower bound for hairpin completion distance computation that matches the $O(n^2)$ upper bound?*

In this paper, we show that for every $\varepsilon > 0$, there is no $O(n^{2-\varepsilon})$ time algorithm for computing the hairpin completion distance from y to x, unless the Strong Exponential Time Hypothesis (SETH) [5] is false. Thus, we provide a conditional lower bound matching the upper bound of [3] up to sub-polynomial factors.

▶ **Theorem 2.** *Let $\varepsilon > 0$. If there is an algorithm that computes the hairpin completion distance from y to x in $O(|x|^{2-\varepsilon})$ time, then SETH is false. This holds even if the input strings are over an alphabet of size 4.*

We note that due to the relationship between hairpin operations and DNA biochemistry, a typical output for a hairpin-related problem is over the alphabet $\{A, C, G, T\}$ of size 4. Hence, our lower bound applies to a natural set of practical inputs.

Theorem 2 is proven by reducing Longest Common Subsequence (LCS) problem to the hairpin completion distance problem. Namely, for two ternary strings S and T, we show a linear time construction of a pair of strings x and y such that $\mathsf{LCS}(S, T)$ can be computed in linear time from the hairpin completion distance from y to x. The hardness of hairpin completion computation follows from the conditional lower bound on the LCS problem [4, 1]. We refer the reader to Section 3 where we introduce the reduction and to Section 3.1 where we provide a high-level discussion regarding the correctness of our construction.

2 Preliminaries

For $i, j \in \mathbb{N}$ let $[i..j] = \{k \in \mathbb{N} \mid i \leq k \leq j\}$. We denote $[i] = [1..i]$.

A string S over an alphabet Σ is a sequence of characters $S = S[1]S[2] \dots S[|S|]$. For $i, j \in [|S|]$, we call $S[i..j] = S[i]S[i+1] \dots S[j]$ a *substring* of S. If $i = 1$, $S[i..j]$ is a prefix of S, and if $j = |S|$, $S[i..j]$ is a suffix of S. Let x and y be two strings over an alphabet Σ. $x \cdot y$ is the concatenation of x and y. For strings $x_1, x_2, \dots x_m$, we denote as $\bigodot_{i=1}^{m} = x_1 \cdot x_2 \cdot \dots \cdot x_m$. For a string x and $k \in \mathbb{N}$ we write the concatenation of x to itself k times as x^k. For a symbol $\sigma \in \Sigma$, we denote as $\#_\sigma(x) = |\{i \in [|x|] \mid x[i] = \sigma\}|$ the number of occurrences of σ in x. We say that a string y occurs in x (or that x contains an occurrence of y) if there is an index $i \in [|x| - |y| + 1]$ such that $x[i..i + |y| - 1] = y$.

For two sets of strings $\mathcal{S}$ and $\mathcal{T}$, we define the set of strings $\mathcal{S} * \mathcal{T} = \{s \cdot x \cdot t \mid s \in \mathcal{S}, x \in \Sigma^*, t \in \mathcal{T}\}$. We use the notations $\mathcal{S}* = \mathcal{S} * \Sigma^*$ and $*\mathcal{S} = \Sigma^* * \mathcal{S}$. When using $*$ notation, we sometimes write $s \in \Sigma^*$ to denote the set $\{s\}$ (for example, $0*$ is the set of all strings starting with 0).

Hairpin Operations. Let $\mathsf{Inv} : \Sigma \to \Sigma$ be a permutation on Σ. We say that Inv is an *inverse function* on Σ if $\mathsf{Inv} = \mathsf{Inv}^{-1}$ and $\mathsf{Inv}(\sigma) \neq \sigma$ for every $\sigma \in \Sigma$. Throughout this paper, we discuss strings over alphabet $\Sigma = \{0, 1\}$ with $\mathsf{Inv}(\sigma) = 1 - \sigma$. For every symbol in σ, we denote $\overline{\sigma} = \mathsf{Inv}(\sigma)$. We further extend this notation to strings by denoting $\overline{x} = \overline{x[1]} \cdot \overline{x[2]} \cdot \dots \cdot \overline{x[|x|]}$. We denote $\overleftarrow{x} = \overline{x}[|x|] \cdot \overline{x}[|x| - 1] \cdot \dots \cdot \overline{x}[2] \cdot \overline{x}[1]$.

We define several types of hairpin operations that can be applied to a string over Σ with an inverse function on Σ. In [6], hairpin operations are defined as follows.

▶ **Definition 3** (Hairpin Operations). *Let $S \in \Sigma^*$. A right hairpin completion of length $\ell \in [|S|]$ transforms S into $S \cdot \overleftarrow{S[1..\ell]}$. A right hairpin completion operation of length ℓ can be applied on S only if $S[\ell+1] = \overline{S[|S|]}$. Similarly, a left hairpin completion of length ℓ transforms S into $\overleftarrow{S[|S|-\ell+1..|S|]} \cdot S$. A left hairpin completion of length ℓ can be applied to S only if $S[|S|-\ell] = \overline{S[1]}$. A right (resp. left) hairpin deletion operation of length $\ell \in [\left\lfloor \frac{|S|}{2} \right\rfloor]$ transforms a string S into a prefix (resp. suffix) S' of S such that S can be obtained from S' by a valid right (resp. left) hairpin completion of length ℓ.*

Throughout this paper, we use the following modified definition of hairpin operation, which removes the constraints regarding $S[1]$ and $S[|S|]$.

▶ **Definition 4** (Hairpin Operations, Modified definition). *Let $S \in \Sigma^*$. A right hairpin completion of length $\ell \in [|S|]$ transforms S into $S \cdot \overleftarrow{S[1..\ell]}$. Similarly, a left hairpin completion of length ℓ transforms S into $\overleftarrow{S[|S|-\ell+1..|S|]} \cdot S$. A right (resp. left) hairpin deletion of length $\ell \in [\left\lfloor \frac{|S|}{2} \right\rfloor]$ operation transforms a string S into a prefix (resp. suffix) S' of S such that S can be obtained from S' by a valid right (resp. left) hairpin completion of length ℓ.*

We highlight that the modified definition is *not* equivalent to the definition of [6]. Even though the paper is phrased in terms of the modified definition, we emphasize that Theorem 2 is correct with respect to both definitions. In the full version of this paper, we discuss the machinery required to make our hardness result applicable to Definition 3. The complete details for bridging this gap are developed in the full version of this paper [2].

Let x and y be two strings. We denote by $\mathsf{HDD}(x,y)$ (resp. $\mathsf{HCD}(x,y)$) the minimum number of hairpin deletion (resp. completion) operations required to transform x into y, counting both left and right operations. Note that $\mathsf{HDD}(x,y) = \mathsf{HCD}(y,x)$.

For the sake of analysis, we define the following graph.

▶ **Definition 5** (Hairpin Deletion Graph). *For a string x the* Hairpin Deletion Graph *$G_x = (V,E)$ is defined as follows. V is the set of all substrings of x, and $(u,v) \in E$ if v can be obtained from u in a single hairpin deletion operation.*

We define the distance between two vertices s and t in a graph G (denoted as $\mathsf{dist}_G(s,t)$) to be the minimal length (number of edges) of a path from s to t in G (of ∞ if there is no such path). Note that for a source string x and a destination string y, it holds that $\mathsf{HDD}(x,y) = \mathsf{dist}_{G_x}(x,y)$. We distinguish between two types of edges outgoing from $x[i..j]$. An edge of the form $x[i..j] \to x[i+\ell..j]$ for some $\ell \in \mathbb{N}$ is called a *left edge* and it corresponds to a left hairpin deletion operation of length ℓ. Similarly, an edge of the form $x[i..j] \to x[i..j-\ell]$ for some $\ell \in \mathbb{N}$ is called a *right edge* and it corresponds to a right hairpin deletion operation of length ℓ. When a path p in G_x traverses a left (resp. right) edge outgoing from v, we say that p applies a left (resp. right) hairpin deletion to v. For a path p we denote by $\mathsf{cost}(p)$ the length of p.

Hairpin Deletion. Since the paper makes intensive use of hairpin deletion notations, we introduce an alternative, more intuitive definition for hairpin deletion, equivalent to Definition 4. For a string S, if for some $\ell \in [\left\lfloor \frac{|S|}{2} \right\rfloor]$ we have $S[1..\ell] = \overleftarrow{S[|S|-\ell+1..|S|]}$ then a left (resp. right) hairpin deletion operation transforms S into $S[\ell+1..|S|]$ (resp. $S[1..|S|-\ell]$). In particular, if $S[1] \neq \overleftarrow{S[|S|]}$ then there is no valid hairpin deletion operation on S.

Longest Common Subsequence. A subsequence of a string S of length n is a string X of length ℓ such that there is an increasing sequence $1 \le i_1 < i_2 < \dots i_\ell \le n$ satisfying $X[k] = S[i_k]$ for every $k \in [\ell]$. For two strings S and T, a string X is a common subsequence of S and T if X is a subsequence of both S and T. The LCS problem is, given two strings S and T of length at most n, compute the maximum *length* of a common subsequence of S and T, denoted as $\mathsf{LCS}(S, T)$.

Bringmann and Künnemann [4] have shown the following.

▶ **Fact 6** (Hardness of LCS). *For every $\varepsilon > 0$, there is no $O(n^{2-\varepsilon})$-time algorithm that solves the* LCS *problem for ternary input strings unless* SETH *is false.*

Fibonacci sequence. The Fibonacci sequence is defined as follows. $\mathsf{Fib}(0) = 1$, $\mathsf{Fib}(1) = 1$ and for all integer $i > 1$ we have $\mathsf{Fib}(i) = \mathsf{Fib}(i-1) + \mathsf{Fib}(i-2)$. The inverse function $\mathsf{Fib}^{-1} : \mathbb{R} \to \mathbb{N}$ is defined as $\mathsf{Fib}^{-1}(x) = \min\{y \in \mathbb{N} \mid \mathsf{Fib}(y) \ge x\}$.

3 The Reduction

Here we introduce a reduction from the LCS problem on ternary strings. We also provide in Section 3.1 a high-level discussion of why the reduction should work.

We present a linear time algorithm such that given two strings $S, T \in \{0, 1, 2\}^*$, constructs two (binary) strings x and y with $|x| = O(|S| + |T|)$ and $|y| = O(1)$. The strings x and y have the property that $\mathsf{HDD}(x, y)$ can be used to infer $\mathsf{LCS}(S, T)$ in linear time. Thus, by Fact 6, we deduce that any algorithm computing $\mathsf{HDD}(x, y)$ cannot have running time $O(|x|^{2-\varepsilon})$ for any $\varepsilon > 0$ (assuming SETH).

We use several types of gadgets. Let:

- $I_{\mathsf{L}}(0) = (010^3)^{\mathsf{i}_0}$
- $I_{\mathsf{L}}(1) = (010^5)^{\mathsf{i}_1}$
- $I_{\mathsf{L}}(2) = (010^7)^{\mathsf{i}_2}$
- $P_{\mathsf{L}} = (010^9)^{\mathsf{p}}$
- $\mathsf{Sync}_{\mathsf{L}} = 01$
- $I_{\mathsf{R}}(0) = (\overline{0^3 10})^{\mathsf{i}_0} = \overleftarrow{I_{\mathsf{L}}(0)}$
- $I_{\mathsf{R}}(1) = (\overline{0^5 10})^{\mathsf{i}_1} = \overleftarrow{I_{\mathsf{L}}(1)}$
- $I_{\mathsf{R}}(2) = (\overline{0^7 10})^{\mathsf{i}_2} = \overleftarrow{I_{\mathsf{L}}(2)}$
- $P_{\mathsf{R}} = (\overline{0^9 10})^{\mathsf{p}} = \overleftarrow{P_{\mathsf{L}}}$
- $\mathsf{Sync}_{\mathsf{R}} = \overline{010}$

with $\mathsf{i}_0 = 55$, $\mathsf{i}_1 = 54$, $i_2 = 53$ and $\mathsf{p} = 144$. We call $I_{\mathsf{L}}(0)$, $I_{\mathsf{L}}(1)$ and $I_{\mathsf{L}}(2)$ left *information* gadgets and $I_{\mathsf{R}}(0)$, $I_{\mathsf{R}}(1)$ and $I_{\mathsf{R}}(2)$ right information gadgets. We say that $I_{\mathsf{L}}(\alpha)$ and $I_{\mathsf{R}}(\beta)$ *match* if $\alpha = \beta$ or *mismatch* otherwise. P_{L} and P_{R} are called left and right *protector* gadgets, respectively. $\mathsf{Sync}_{\mathsf{L}}$ and $\mathsf{Sync}_{\mathsf{R}}$ are called left and right *synchronizer* gadgets, respectively. We say that two gadgets g_1, g_2 are *symmetric* if $g_2 = \overleftarrow{g_1}$. Specifically, $(I_{\mathsf{L}}(0), I_{\mathsf{R}}(0))$, $(I_{\mathsf{L}}(1), I_{\mathsf{R}}(1))$, $(I_{\mathsf{L}}(2), I_{\mathsf{R}}(2))$ and $(P_{\mathsf{L}}, P_{\mathsf{R}})$ are the pairs of symmetric gadgets. We emphasize that $\mathsf{Sync}_{\mathsf{L}}$ and $\mathsf{Sync}_{\mathsf{R}}$ are *not* symmetric.

Using the gadgets above, we define 6 mega gadgets encoding characters from S and T. For $\alpha \in \{0, 1, 2\}$ we define

$$E_{\mathsf{L}}(\alpha) = P_{\mathsf{L}} \cdot \mathsf{Sync}_{\mathsf{L}} \cdot I_{\mathsf{L}}(\alpha) \cdot \mathsf{Sync}_{\mathsf{L}} \quad \text{and} \quad E_{\mathsf{R}}(\alpha) = \mathsf{Sync}_{\mathsf{R}} \cdot I_{\mathsf{R}}(\alpha) \cdot \mathsf{Sync}_{\mathsf{R}} \cdot P_{\mathsf{R}}.$$

Finally, we define $y = \boxed{P_{\mathsf{L}} \cdot \mathsf{Sync}_{\mathsf{L}} \cdot 01 \cdot 11\overline{1}\overline{1} \cdot \overline{1}\overline{0} \cdot \mathsf{Sync}_{\mathsf{R}} \cdot P_{\mathsf{R}}}$ and $x = \left(\bigodot_{i=1}^{|S|} E_{\mathsf{L}}(S[i])\right) \cdot y \cdot \left(\bigodot_{i=|T|}^{1} E_{\mathsf{R}}(T[i])\right)$. Note that on the suffix of x we concatenate the elements of T in *reverse* order.

In the remainder of this paper, we only use 'x' and 'y' to refer to the strings defined above. We define notations for indices in x which are endpoints of protector and information gadgets as follows. For $\ell \in [|S|+1]$, let $\mathsf{left}_\ell^P = 1+\sum_{j=1}^{\ell-1} |E_{\mathsf{L}}(S[j])|$ be the leftmost index of the ℓth P_{L} gadget (from the left) in x. Notice that $\mathsf{left}_{|S|+1}^P$ corresponds to the left P_{L} gadget contained in y. For $r \in [|T|+1]$, let $\mathsf{right}_r^P = |x| - \sum_{j=1}^{r-1} |E_{\mathsf{R}}(T[j])|$ be the rightmost index of the rth P_{R} gadget (from the right) in x. Notice that $\mathsf{right}_{|T|+1}^P$ corresponds to the right P_{R} gadget contained in y. For $\ell \in [|S|]$, let $\mathsf{left}_\ell^I = \mathsf{left}_\ell^P + |P_{\mathsf{L}}| + |\mathsf{Sync}_{\mathsf{L}}|$ be the leftmost index of the ℓth information gadget (from the left) in x. For $r \in [|T|]$, let $\mathsf{right}_r^I = \mathsf{right}_r^P - |P_{\mathsf{R}}| - |\mathsf{Sync}_{\mathsf{R}}|$ be the rightmost index of the rth information gadget (from the right) in x.

The rest of the paper is dedicated for proving the following property of x and y.

▶ **Lemma 7** (Reduction Correctness). *For some constants $D(0), D(1), D(2)$ and B we have:* $\mathsf{HDD}(x,y) = \sum_{\alpha\in\{0,1,2\}} D(\alpha)(\#_\alpha(S) + \#_\alpha(T)) - \mathsf{LCS}(S,T) \cdot B$.

Note that $\#_\alpha(S)$ and $\#_\alpha(T)$ can be easily computed for all values of α in linear time. Therefore, if $\mathsf{HDD}(x,y)$ can be computed in $O(n^{2-\varepsilon})$ time for some $\varepsilon > 0$, LCS can be computed in $O(n + n^{2-\varepsilon})$ (the values of the constants are fixed in the proof). Since $\mathsf{HDD}(x,y) = \mathsf{HCD}(y,x)$, hairpin deletion and hairpin completion distance are computationally equivalent. Recall that $\mathsf{HDD}(x,y)$ refers to the *modified* hairpin deletion distance (Definition 4). In order to bridge the gap to the original definition of hairpin deletion distance, we provide a linear time construction of strings x' and y' such that $\mathsf{HDD}'(x',y') = \mathsf{HDD}(x,y)$ in the full version of this paper [2]. Here, $\mathsf{HDD}'(x',y')$ denotes the hairpin deletion distance from x' to y' as defined in Definition 3. It clearly follows from this construction and the above discussion that $\mathsf{HDD}'(x',y')$ can not be computed in $O(n^{2-\varepsilon})$, unless SETH is false. Thus, proving Theorem 2.

3.1 Intuition for the Reduction Correctness

We provide some high-level discussion regarding the correctness of the construction. First, notice that y has a single occurrence in x. Therefore, a sequence of hairpin deletion operations transforming x into y has to delete all mega gadgets. Consider an intermediate step in a deletion sequence in which the substring $x[i..j]$ is obtained such that i is the leftmost index of some left gadget g_i and j is the rightmost index of some right gadget g_j.

If g_i and g_j are not symmetric, the next hairpin deletion would not be able to make much progress. This is due to the 1 symbols in g_i and the $\overline{1}$ symbols in g_j being separated by a different number of 0's and $\overline{0}$'s. For the goal of minimizing the number of deletions for removing all mega gadgets, this is a significant set-back, as either g_i or g_j would have to be removed using roughly $\#_1(g_i)$ (or $\#_{\overline{1}}(g_j)$) deletions.

Now consider the case in which g_i and g_j are symmetric to each other. In this case, either one of them can be deleted using a single hairpin deletion. However, note that greedily removing g_i will put us in the asymmetric scenario. Notice that there is another possible approach for deleting symmetric gadgets - a synchronized deletion. In this process, g_i and g_j are both deleted gradually. One can easily figure out a way to apply such synchronized deletion using roughly $\log(\#_1(g_i))$ steps.

Think of a scenario in which $i = \mathsf{left}_\ell^P$ and $j = \mathsf{right}_r^P$ for some ℓ and r, i.e. i and j are a leftmost index and a rightmost index of left and right mega gadgets m_ℓ and m_r, respectively. Initially, both i and j are in the beginning of protector gadgets p_ℓ and p_r. If m_ℓ and m_r are mega gadgets corresponding to the same symbol α, the protector gadgets and the information gadgets of m_ℓ and m_r are symmetric to each other. It is therefore very beneficial to remove the mega gadgets in a synchronized manner. The event in which the ℓ'th left mega gadget and the r'th right mega gadgets are deleted in a synchronized manner corresponds to $S[\ell]$ and $T[r]$ being matched by the longest common subsequence.

Now, consider the case in which m_ℓ and m_r do not match i.e. $S[\ell] \neq T[r]$. Deleting the protectors in a synchronized manner would not yield much benefit in this scenario, since the information gadgets are not symmetric, and therefore would have to be removed slowly. In this case, since an inefficient deletion of an information gadget is inevitable, it is more efficient to delete one of the protectors gadgets using a single deletion operation, and proceed to delete the following information gadget inefficiently. This would result in either the left mega gadget being deleted, or the right one. The event in which the ℓ'th left (resp. r'th right) mega gadget is deleted in a non synchronized manner corresponds to $S[\ell]$ (resp. $T[r]$) not being in the longest common subsequence. The gadgets are designed in a way such that deleting mega gadgets in a synchronized way is faster than deleting each mega gadget in a non-synchronized way. Furthermore, the cost reduction of a synchronized deletion over a non-synchronized deletion is a constant number B. Therefore, by selecting $D(\alpha)$ as the cost of deleting a mega gadget corresponding to the symbol α in a non-synchronized way, one obtains Lemma 7.

The above discussion makes an implicit assumption that the sequence of deletion is applied in *phases*. Each phase starts with $x[i..j]$ such that i and j are edge endpoints of mega gadgets and proceeds to either delete both in a synchronized manner or one in a non-synchronized manner. In order to show that $\mathsf{HDD}(x, y)$ is at most the term in Lemma 7, this is sufficient since we can choose a sequence of deletion with this structure as a witness. In order to show that $\mathsf{HDD}(x, y)$ is at least the expression in Lemma 7, one has to show that there is an optimal sequence of deletions with this structure. This is one of the main technical challenges in obtaining Theorem 2.

In a high level, the sync gadgets function as “anchors” that force any sequence of deletions to stop in their proximity. Another key property of our construction that enforces the “phases” structure is the large size of a protector relatively to the information. Intuitively, an optimal sequence would always avoid deleting a protector gadget inefficiently, so if a left protector is deleted using a right protector, left deletions would continue to occur until the next left protector is reached.

In Section 4, we provide the formal definition for a well-structured sequence and prove that there is an optimal sequence of deletions with this structure. In Section 5 we provide a precise analysis of every phase in a well-structured sequence. In Section 6 we put everything together and prove Lemma 7.

4 Well-Behaved Paths

We start by defining a well-behaved path.

▶ **Definition 8** (Well-Behaved Path). *A path p from x to y in G_x is well-behaved if for every $\ell \in [|S|+1]$ and $r \in [|T|+1]$, if p visits $x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$, one of the following vertices is also visited by p: $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$, $x[\mathsf{left}^P_\ell..\mathsf{right}^P_{r+1}]$, or $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$. If one of $\ell+1$ and $r+1$ is undefined, the condition is on the subset of defined vertices. If both are undefined, the condition is considered satisfied.*

This section is dedicated to proving the following lemma.

▶ **Lemma 9** (Optimal Well-Behaved Path). *There is a shortest path from x to y in G_x which is well-behaved.*

We start by proving properties regarding paths and shortest paths from x to y in G_x. Due to space constraints, the proofs for the lemmata in this section appear in the full version of this paper [2].

4.1 Properties of Paths in G_x

We start by observing that an x to y path in G_x never deletes symbols from y.

▶ **Observation 10** (Never Delete y). *The substring $x[\mathsf{left}^P_{|S|+1}..\mathsf{right}^P_{|T|+1}] = y$ is the unique occurrence of y in x. Let p be a path from x to y in G_x. For every vertex $x[i..j]$ in p, $[\mathsf{left}^P_{|S|+1}..\mathsf{right}^P_{|T|+1}] \subseteq [i..j]$.*

Since each hairpin deletion operation deletes a prefix (or a suffix) of a substring of x, we have the following observation and immediate corollary.

▶ **Observation 11.** *Let $x[i..j] \in 010^a1 * \overline{10^b10}$ for $a \neq b$. A single left hairpin deletion operation removes at most a single 1 character. Symmetrically, a right hairpin deletion operation removes at most a single $\overline{1}$ character.*

▶ **Corollary 12.** *A single left (resp. right) hairpin deletion operation on $x[i..j]$ can remove more than a single 1 (resp. $\overline{1}$) character only if i and j are in symmetric gadgets.*

The next lemma assures a restriction over the vertices along a path from x to y in G_x.

▶ **Lemma 13** (Always $01*$ or $*\overline{10}$). *Let p be a path from s to t in G_x such that $s, t \in 01 * \overline{10}$. For every vertex $x[i..j]$ visited by p, we have $x[i] = 0$ and $x[j] = \overline{0}$. Furthermore, $x[i+1] = 1$ or $x[j-1] = \overline{1}$.*

Due to the equivalence between a path in G_x and a sequence of hairpin deletions and due to the symmetry between hairpin deletion and hairpin completion, we obtain the following.

▶ **Corollary 14.** *Let $s, t \in 01 * \overline{10}$ and let $\mathcal{H}$ be a sequence of h hairpin deletion operations (or a sequence of hairpin completion operations) that transforms s into t. For $i \in [h]$, let S_i be the string obtained by applying the first i operations of $\mathcal{H}$ on s. For every $i \in [h]$, we have $S_i[1] = 0$ and $S_i[|S_i|] = \overline{0}$. Furthermore, either $S_i[2] = 1$ or $S_i[|S_i| - 1] = \overline{1}$.*

The following lemma discusses the situation in which p visits a vertex not in $01 * \overline{10}$. Essentially, the lemma claims that when p visits a substring $x[i..j]$ with a prefix 00, the next step would be $x[i+1..j]$, i.e., deleting a single zero from the left.

▶ **Lemma 15** (Return to $01 * \overline{10}$). *Let p be a path from x to y in G_x. If p visits a vertex $x[i..j]$ such that $x[i..j] = 01 * \overline{10^k}$ for some integer $k \geq 1$, then for every $k' \in [k-1]$ it must be that p visits $x[i..j-k']$ as well. Symmetrically, if p visits a vertex $x[i..j]$ such that $x[i..j] = 0^k1 * \overline{10}$ for some integer $k \geq 1$, then for every $k' \in [k-1]$ it must be that p visits $x[i+k'..j]$ as well.*

The following is a direct implication of Lemmata 13 and 15.

▶ **Observation 16.** *Let p be a path from x to y in G_x. If p applies a right hairpin deletion operation on v then $v \in 01*$. Symmetrically, if p applies a left hairpin deletion operation on v then $v \in *\overline{10}$.*

The following lemma establishes the importance of the synchronizer gadgets.

▶ **Lemma 17** (Synchronizer Lemma). *Let p be a path from x to y in G_x and let $s = x[i_s..j_s] =$ $\mathsf{Sync_L}$ be a left synchronizer which is not contained in y, p must visit a vertex $x[j_s + 1..k]$ for some integer k. Symmetrically, if $x[i_s..j_s] = \mathsf{Sync_R}$ is a right synchronizer which is not contained in y, p must visit a vertex $x[k..i_s - 1]$.*

Notice that the leftmost index of every left information and protector gadget is $j_s + 1$ for some left synchronizer $x[i_s..j_s]$ (excluding the leftmost protector gadget). A similar structure occurs with right gadgets. The following directly follows from Lemma 17.

▶ **Corollary 18.** *Let p be a path from x to y in G_x. Then, for each $\ell \in [|S|]$ the path p visits vertices $u = x[i..j]$ with $i = \mathsf{left}^P_\ell$ and $v = x[i..j]$ with $i = \mathsf{left}^I_\ell$. Symmetrically, for each $r \in [|T|]$ the path p visits vertices $u = x[i..j]$ with $j = \mathsf{right}^P_r$ and $v = x[i..j]$ with $j = \mathsf{right}^I_r$.*

4.2 Transitions Between Gadegets

In this section, we address the way shortest paths apply to vertices that transit from a gadget to the gadget afterward.

▶ **Lemma 19.** *Let p be a shortest path from x to y in G_x. For some $\ell \in |S|$, let $v = x[i_1..j_1]$ be the first vertex visited by p with $i_1 = \mathsf{left}^I_\ell$. Let $u = x[i_2..j_2]$ be the first vertex visited by p with $i_2 = \mathsf{left}^P_{\ell+1}$. Then, there is no occurrence of P_R in $x[j_2..j_1]$.*

Symmetrically, for some $r \in |T|$ let $v = x[i_1..j_1]$ be the first vertex visited by p with $j_1 = \mathsf{right}^I_r$. Let $u = x[i_2..j_2]$ be the first vertex visited by p with $j_2 = \mathsf{right}^P_{r+1}$. Then, there is no occurrence of P_L in $x[i_1..i_2]$.

Proof Sketch, Complete Proof in the full version of this paper [2]. The proof is by contradiction. If there is only a *single* P_R gadget that is contained in $x[j_2..j_1]$, by Corollary 12 the number of hairpin deletions that must happen just to remove this gadget is at least p. We introduce an alternative, shorter path: At the moment p reaches this P_R gadget, it first removes all the $I_\mathsf{L}(S[\ell])$ gadget, and then removes all the remaining characters on the right side greedily, until reaching $x[i_2..j_2]$. The reason why this alternative path is indeed shorter is since in this way the removal of the P_R gadget takes 1 operation, instead of p, and we may pay at most $\mathsf{i}_\alpha + \mathsf{i}_\beta \leq 2\mathsf{i}_0$ for some $\alpha, \beta \in \{0, 1, 2\}$, for removing one information gadget in the left side and the information gadget following the right protector gadget. Since p is much larger than i_0, the alternative path is shorter, contradicting the assumption that p is a shortest path. Notice that if there are more than one P_R gadgets in $x[j_2..j_1]$, the benefit from deleting $I_\mathsf{L}(S[\ell])$ first is even larger. ◀

The following lemma states that every right deletion on $x[i..j]$ with i being within a non $\mathsf{Sync_L}$ gadget can also be applied if i is the leftmost index of the gadget. A symmetric argument is stated as well.

▶ **Lemma 20.** *Let p be a path from x to y in G_x. Let $v = x[i..j]$ be a vertex visited by p such that $i \in [\mathsf{left}^P_\ell..\mathsf{left}^I_\ell - 1]$ for some $\ell \in [|S|]$. Let $v' = x[\mathsf{left}^P_\ell..j]$, let $u = x[i..j-k]$ and $u' = x[\mathsf{left}^P_\ell..j-k]$ for some k. If (v, u) is an edge in p, then (v', u') is an edge in G_x.*

The above statement considers the case in which v interacts with a P_L gadget, the following similar statements, regarding different gadgets hold as well:

- ***P_R:** Let $v = x[i..j]$ be a vertex visited by p such that $j \in [\mathsf{right}^I_r + 1..\mathsf{right}^P_r]$ for some $r \in [|T|]$. Let $v' = x[i..\mathsf{right}^P_r]$, let $u = x[i+k..j]$ for some k, and let $u' = x[i+k..\mathsf{right}^P_r]$ If (v, u) is an edge in p, then (v', u') is an edge in G_x.*

- $\boldsymbol{I_{\mathsf{L}}(\alpha)}$ ***for some*** $\boldsymbol{\alpha \in \{0,1,2\}}$***:*** *Let $v = x[i..j]$ be a vertex visited by p such that $i \in [\mathsf{left}^I_\ell..\mathsf{left}^P_{\ell+1} - 1]$ for some $\ell \in [|S|]$. Let $v' = x[\mathsf{left}^I_\ell..j]$, let $u = x[i..j-k]$ and $u' = x[\mathsf{left}^I_\ell..j-k]$ for some k. If (v,u) is an edge in p, then (v',u') is an edge in G_x.*
- $\boldsymbol{I_{\mathsf{R}}(\alpha)}$ ***for some*** $\boldsymbol{\alpha \in \{0,1,2\}}$***:*** *Let $v = x[i..j]$ be a vertex visited by p such that $j \in [\mathsf{right}^P_{r+1} + 1..\mathsf{right}^I_r]$ for some $r \in [|T|]$. Let $v' = x[i..\mathsf{right}^I_r]$, let $u = x[i+k..j]$ for some k, and let $u' = x[i+k..\mathsf{right}^I_r]$ If (v,u) is an edge in p, then (v',u') is an edge in G_x.*

Proof Sketch, Complete Proof in the full version of this paper [2]. We distinguish between two cases. If $x[i..i+1] \neq 01$, by Lemma 15 the hairpin deletion removes exactly one $\overline{0}$ character, and by $x[\mathsf{left}^P_\ell] = 0$ the edge (u',v') is in G_x. If $x[i..i+1] = 01$, we first prove (using Corollary 18) that $k \leq \mathsf{left}^I_\ell - i$. Moreover, since $i \in [\mathsf{left}^P_\ell..\mathsf{left}^I_\ell - 1]$ it must be the case that $i = \mathsf{left}^P_\ell + q \cdot 11$ for some q. Since $x[\mathsf{left}^P_\ell..\mathsf{left}^I_\ell - 1]$ is periodic with period 11. Thus, $x[\mathsf{left}^P_\ell..\mathsf{left}^P_\ell + k] = x[\mathsf{left}^P_\ell + q \cdot 11..\mathsf{left}^P_\ell + q \cdot 11 + k] = x[i..i+k]$ and the claim follows. ◀

We are now ready to prove Lemma 9.

▶ **Lemma 9** (Optimal Well-Behaved Path). *There is a shortest path from x to y in G_x which is well-behaved.*

Proof. We describe a method that converts a shortest path p from x to y that visits $u = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ into a shortest path p' from x to y that visits one of the following vertices: $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$, $x[\mathsf{left}^P_\ell..\mathsf{right}^P_{r+1}]$, or $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$. Moreover, the prefixes of p and p' from x to u are identical. Using this technique, it is straightforward to convert a shortest path from x to y in G_x into a well-behaved path of the same length.

Let $v_L = x[i_L..j_L]$ be the first vertex in p with $i_L = \mathsf{left}^P_{\ell+1}$ and let $v_R = x[i_R..j_R]$ be the first vertex in p with $j_R = \mathsf{right}^P_{r+1}$. By Corollary 18, v_L and v_R are well defined (unless $\ell = |S|+1$ or $r = |T|+1$, in such a case just one of the vertices is well defined and the claim follows trivially from Observation 10). We consider the case where v_L appears before v_R in p and show how to convert p. The other case is symmetric.

We distinguish between two cases:

Case 1: $\boldsymbol{j_L \in [\mathsf{right}^I_r + 1..\mathsf{right}^P_r]}$**.** Let q be the sub-path of p from u to v_L. We present a path q^* from u to v_L that is not longer than q and visits $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$. Recall that an edge of the form $x[i..j] \to x[i+k..j]$ is called a *left* edge, and an edge of the form $x[i..j] \to x[i..j-k]$ is called a *right* edge. Let cost_L be the number of left edges in q and cost_R be the number of right edges in q. We first show a path from $u = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$ of length cost_L. Let $e = x[i_1..j] \to x[i_2..j]$ be a left edge in q. It must be that $j \in [j_L..\mathsf{right}^P_r] \subseteq [\mathsf{right}^I_r + 1..\mathsf{right}^P_r]$. Hence, by Lemma 20, there exists an edge $e' = x[i_1..\mathsf{right}^P_r] \to x[i_2..\mathsf{right}^P_r]$. Let $e_1, e_2, \ldots, e_{\mathsf{cost}_L}$ be the subsequence of all left edges in q. The path $q^*_1 = e'_1, e'_2, \ldots, e'_{\mathsf{cost}_L}$ is a valid path of length cost_L from $u = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$.

If $j_L = \mathsf{right}^P_r$ then $q^* = q^*_1$ is a path that satisfies all the requirements. Otherwise, $\mathsf{cost}_R \geq 1$. We claim that there is an edge e_R from $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$ to $v_L = x[\mathsf{left}^P_{\ell+1}..j_L]$. This is true since $x[\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1} - 1] = P_{\mathsf{L}} \cdot \mathsf{Sync}_{\mathsf{L}} = \overleftarrow{\mathsf{Sync}_{\mathsf{R}}[2..|\mathsf{Sync}_{\mathsf{R}}|] \cdot P_{\mathsf{R}}} = \overleftarrow{x[\mathsf{right}^I_r + 2\ ..\ \mathsf{right}^P_r]}$ and $j_L \in [\mathsf{right}^I_r + 1..\mathsf{right}^P_r]$. We conclude q^* by appending e_R to the end of q^*_1. Finally, $\mathsf{cost}(q^*) = \mathsf{cost}_L + 1 \leq \mathsf{cost}_L + \mathsf{cost}_R = \mathsf{cost}(q)$, and q^* visits $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$.

Case 2: $j_L \in [\mathsf{right}^P_{r+1} - 1..\mathsf{right}^I_r]$. We first prove the following claim.

▷ Claim. $i_R \in [\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1} - 1]$.

Proof. Since v_R appears after v_L, we have $i_R \geq i_L = \mathsf{left}^P_{\ell+1}$. Assume to the contrary that $i_R \geq \mathsf{left}^I_{\ell+1}$. Let $v_f = x[i_f..j_f]$ be the first vertex in p with $j_f = \mathsf{right}^I_r$ (v_f exists according to Corollary 18). Note that v_f does not appear after v_L in p since $j_L =\leq \mathsf{right}^I_r = j_f$ Therefore, $i_f \leq i_L = \mathsf{left}^P_{\ell+1}$ and $[\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1}] \subseteq [i_f..i_R]$. Therefore, the occurrence of P_L starting in $\mathsf{left}^P_{\ell+1}$ is contained in $x[i_f..i_R]$. Since p is a shortest path, this is a contradiction to Lemma 19. ◁

Let q be the sub-path of p from v_L to v_R. Let cost_L be the number of left edges in q and cost_R be the number of right edges in q. We present a path q^* from v_L to v_R that is not longer than q and visits $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$. We first show a path q^*_1 from $v_L = x[\mathsf{left}^P_{\ell+1}..j_L]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$ of length cost_R. Let $e = x[i..j_1] \rightarrow x[i..j_2]$ be a right edge in q. It must be that $i \in [\mathsf{left}^P_{\ell+1}..i_R] \subseteq [\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1} - 1]$ due to the claim. Hence, by Lemma 20, there exists an edge $e' = x[\mathsf{left}^P_{\ell+1}..j_1] \rightarrow x[\mathsf{left}^P_{\ell+1}..j_2]$. Let $e_1, e_2, \ldots, e_{\mathsf{cost}_L}$ be the subsequence of all right edges in q. The path $q^*_1 = e'_1, e'_2, \ldots, e'_{\mathsf{cost}_L}$ is a valid path of length cost_R from $v_L = x[\mathsf{left}^P_{\ell+1}..j_L]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$.

If $i_R = \mathsf{left}^P_{\ell+1}$ then $q^* = q^*_1$ is a path that satisfies all the requirements. Otherwise, $\mathsf{cost}_L \geq 1$. We claim that there is an edge e_L from $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$ to $v_R = x[i_R..\mathsf{right}^P_{r+1}]$. This is true since $x[\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1}] = P_\mathsf{L} \cdot \mathsf{Sync}_\mathsf{L} \cdot 0 = \overleftarrow{\mathsf{Sync}_\mathsf{R} \cdot P_\mathsf{R}} = \overleftarrow{x[\mathsf{right}^I_{r+1} + 1 .. \mathsf{right}^P_{r+1}]}$ and $i_R \in [\mathsf{left}^P_{\ell+1}..\mathsf{left}^I_{\ell+1} - 1]$. We conclude q^* by appending e_L to the end of q^*_1. Finally, $\mathsf{cost}(q^*) = \mathsf{cost}_L + 1 \leq \mathsf{cost}_L + \mathsf{cost}_R = \mathsf{cost}(q)$, and q^* visits $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$. ◀

5 Cost of Well-Behaved Steps

In this section, we analyze the cost of each of the possible phases of a well-behaved path (Definition 8). We first consider the cost of deletion of a single mega-gadget.

▶ **Lemma 21** (Non Synchronized Deletion). *Let $v = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$, $u_1 = [\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$ and $u_2 = [\mathsf{left}^P_\ell..\mathsf{right}^P_{r+1}]$ for $\ell \in [|S|]$ and $r \in [|T|]$. Let $S[\ell] = \alpha$ and $T[r] = \beta$. It holds that $\mathsf{dist}_{G_x}(v, u_1) = \mathsf{i}_\alpha + 2$ and $\mathsf{dist}_{G_x}(v, u_2) = \mathsf{i}_\beta + 2$.*

Proof. We prove $\mathsf{dist}_{G_x}(v, u_1) = \mathsf{i}_\alpha + 2$. The proof for $\mathsf{dist}_{G_x}(v, u_2) = \mathsf{i}_\beta + 2$ is symmetrical. We prove the lemma by showing $\mathsf{dist}_{G_x}(v, u_1) \geq \mathsf{i}_\alpha + 2$ and $\mathsf{dist}_{G_x}(v, u_1) \leq \mathsf{i}_\alpha + 2$.

$\mathsf{dist}_{G_x}(v, u_1) \geq \mathsf{i}_\alpha + 2$. Let p be a v to u_1 path in G_x. Note that vertex $x[i..j]$ in p has $j = \mathsf{right}^P_r$. According to Corollary 18, p must visit $z = x[\mathsf{left}^I_\ell..\mathsf{right}^P_r]$. Since $z \neq v$, the sub-path of p from v to z induces a cost of at least 1 to p. Consider the sub-path q of p from z to u_1. According to Corollary 12, every left hairpin deletion step in q deletes at most a single '1' symbol. Due to $x[\mathsf{left}^I_\ell..\mathsf{left}^P_{\ell+1} - 1] = I_\mathsf{L}(\alpha) \cdot \mathsf{Sync}_\mathsf{L}$, the sub-path q consists of at least $\#_1(I_\mathsf{L}(\alpha)) + 1 = \mathsf{i}_\alpha + 1$ additional left hairpin deletions.

$\mathsf{dist}_{G_x}(v, u_1) \leq \mathsf{i}_\alpha + 2$. We present a path p with cost exactly $\mathsf{i}_\alpha + 2$ from v to u_1. Initially, p deletes a prefix of length $|P_\mathsf{L}| + |\mathsf{Sync}_\mathsf{L}|$ from v in one step. This is possible since v has a suffix $\overline{10} \cdot P_\mathsf{R}$. Then, p proceeds to delete $x[\mathsf{left}^I_\ell..\mathsf{left}^P_{\ell+1} - 1] = I_\mathsf{L}(\alpha) \cdot \mathsf{Sync}_\mathsf{L}$ a single '1' character at a time. Note that this is possible regardless of the value of α due to $x[\mathsf{right}^P_r - 8..\mathsf{right}^P_r] = \overline{0^7 10}$. The total cost of this path is $\mathsf{i}_\alpha + 2$ as required. ◀

In the following lemma, we show that the cost of deleting two disagreeing mega-gadgets is the same as deleting each one of them separately.

▶ **Lemma 22** (Synchronized Deletion of Disagreeing Mega Gadgets). *Let* $v = x[\mathsf{left}_\ell^P..\mathsf{right}_r^P]$, $u = [\mathsf{left}_{\ell+1}^P..\mathsf{right}_{r+1}^P]$ *for* $\ell \in [|S|]$ *and* $r \in [|T|]$ *with* $S[\ell] \neq T[r]$. *It holds that* $\mathsf{dist}_{G_x}(v,u) = \mathsf{i}_\alpha + \mathsf{i}_\beta + 4$ *with* $S[\ell] = \alpha$ *and* $T[r] = \beta$.

Proof. We prove the claim by showing $\mathsf{dist}_{G_x}(v,u) \geq \mathsf{i}_\alpha + \mathsf{i}_\beta + 4$ and $\mathsf{dist}_{G_x}(v,u) \leq \mathsf{i}_\alpha + \mathsf{i}_\beta + 4$.

$\mathbf{dist}_{G_x}(v,u) \geq \mathbf{i}_\alpha + \mathbf{i}_\beta + 4$. Let p be a path from v to u in G_x. According to Corollary 18, p visits vertices $z_1 = x[\mathsf{left}_\ell^I..j]$ and $z_2 = x[i..\mathsf{right}_r^I]$ for some i, j. The last left hairpin deletion in p before z_1 and the last right hairpin deletion in p before z_2 induce a cost of 2 to p. Consider a left hairpin deletion that is applied to a vertex $x[i'..j']$ after z_1 in p. Note that i' is either within an $I_{\mathsf{L}}(\alpha)$ gadget or within a $\mathsf{Sync}_{\mathsf{L}}$ gadget, and j' is either within an $I_{\mathsf{R}}(\beta)$ gadget, a $\mathsf{Sync}_{\mathsf{R}}$ gadget or a P_{R} gadget. In any of the above cases, Corollary 12 suggests that the deletion operation deletes at most a single '1' character. Therefore, there are at least $\mathsf{i}_\alpha + 1$ left deletions after z_1 in p. Due to similar reasoning, there are at least $\mathsf{i}_\beta + 1$ right hairpin deletions after z_2 in p. It follows that the total cost of p is at least $\mathsf{i}_\alpha + \mathsf{i}_\beta + 4$.

$\mathbf{dist}_{G_x}(v,u) \leq \mathbf{i}_\alpha + \mathbf{i}_\beta + 4$. Consider the path p that is composed of two sub-paths, the prefix p_1 is a shortest path from v to $w = x[\mathsf{left}_{\ell+1}^P..\mathsf{right}_r^P]$ and the suffix p_2 is a shortest path from w to u. By Lemma 21 we have $\mathsf{cost}(p_1) = \mathsf{i}_\alpha + 2$ and $\mathsf{cost}(p_2) = \mathsf{i}_\beta + 2$. Therefore $\mathsf{cost}(p) = \mathsf{cost}(p_1) + \mathsf{cost}(p_2) = \mathsf{i}_\alpha + 2 + \mathsf{i}_\beta + 2$. ◀

The last case we have to analyze is a synchronized deletion of agreeing mega gadgets. We first present the concept of *Fibonacci-regular numbers.*

▶ **Definition 23** (Fibonacci-regular number). *We say that* $a \in \mathbb{N}$ *is a* Fibonacci-regular *number if for all* $2 \leq k \leq a$ *it holds that* $\mathsf{Fib}^{-1}(a) \leq \mathsf{Fib}^{-1}(a/k) + k - 1$.

▶ **Fact 24.** $\mathsf{i}_2 = 53$, $\mathsf{i}_1 = 54$, $\mathsf{i}_0 = 55$ *and* $\mathsf{p} = 144$ *are Fibonacci-regular numbers.*

The following lemma, which proof is in the full version of this paper, provides the required machinery to analyze the cost of a synchronized deletion [2].

▶ **Lemma 25.** *Let* $\mathsf{per} = 010^{\mathsf{ext}}$ *and let* $q \in 010^{\mathsf{int}}01 * 11\overline{11} * \overline{100^{\mathsf{int}}10}$ *with* $\mathsf{int} \neq \mathsf{ext}$ *and* $\min\{\mathsf{int}, \mathsf{ext}\} \geq 3$. *For every Fibonacci-regular number* a, *we have* $\mathsf{HDD}(\mathsf{per}^a \cdot \mathsf{Sync}_{\mathsf{L}} \cdot q \cdot \mathsf{Sync}_{\mathsf{R}} \cdot \overleftarrow{\mathsf{per}}^a, q) = \mathsf{Fib}^{-1}(a) + \max(\mathsf{ext} - \mathsf{int} - 1, 0) + 3$.

Finally, we are ready to analyze the cost of synchronized deletion of agreeing mega gadgets.

▶ **Lemma 26** (Synchronized Deletion of Agreeing Mega Gadgets). *Let* $v = x[\mathsf{left}_\ell^P..\mathsf{right}_r^P]$, $u = [\mathsf{left}_{\ell+1}^P..\mathsf{right}_{r+1}^P]$ *for* $\ell \in [|S|]$ *and* $r \in [|T|]$ *with* $S[\ell] = T[r] = \alpha$. *Then* $\mathsf{dist}_{G_x}(v,u) = \mathsf{Fib}^{-1}(\mathsf{p}) + \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 11 - 2\alpha$.

Proof. Let $w = x[\mathsf{left}_\ell^I..\mathsf{right}_r^I]$. Consider the following path p' from u to v in G_x. The path p' consists of a prefix p_1' which is a shortest path from u to w and a suffix p_2' which is a shortest path from w to v. Since $\mathsf{i}_0, \mathsf{i}_1, \mathsf{i}_2$ and p are Fibonacci-regular numbers (Fact 24), according to Lemma 25 (with $\mathsf{ext} = 9$ and $\mathsf{int} = 3 + 2\alpha$) we have $\mathsf{cost}(p_1') = \mathsf{Fib}^{-1}(\mathsf{p}) + \max(9 - (3 + 2\alpha) - 1, 0) + 3 = \mathsf{Fib}^{-1}(\mathsf{p}) + 8 - 2\alpha$. Similarly, according to Lemma 25 (with $\mathsf{ext} = 3 + 2\alpha$

and $\mathsf{int} = 9$) we have $\mathsf{cost}(p'_2) = \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + \max(3 + 2\alpha - 9 - 1, 0) + 3 = \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 3$. In total we have $\mathsf{cost}(p') = \mathsf{cost}(p'_1) + \mathsf{cost}(p'_2) = \mathsf{Fib}^{-1}(\mathsf{p}) + \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 11 - 2\alpha$. Therefore $\mathsf{dist}_{G_x}(v, u) \leq \mathsf{Fib}^{-1}(\mathsf{p}) + \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 11 - 2\alpha = 31 - 2\alpha$.

We prove the following claim.

▷ Claim. There is a shortest path p from v to u that visits w.

Proof. Let $v_L = x[i_L..j_L]$ and $v_R = x[i_R..j_R]$ be the first vertices visited by p with $i_L = \mathsf{left}^I_\ell$ and $j_R = \mathsf{right}^I_r$. Assume without loss of generality that v_R occurs before v_L in p. We consider two cases regarding j_L.

Case 1: $j_L = \mathsf{right}^P_{r+1}$. Consider the suffix p_s of p from v_L to u. Let $v' = x[i'..j']$ be a vertex in p_s that is immediately followed by a left hairpin deletion operation in p_s. Since $i' \in [\mathsf{left}^I_\ell..\mathsf{left}^P_{\ell+1} - 1]$ is either within an $I_\mathsf{L}(\alpha)$ gadget or within a $\mathsf{Sync_L}$ gadget, and $j' = \mathsf{right}^P_{r+1}$ is in a P_R gadget, Corollary 12 suggests that the left hairpin deletion applied to v' deletes at most a single '1' character. It follows from the above analysis that the number of left hairpin deletions in p_s is at least $\#_1(I_\mathsf{L}(\alpha)) + 1 \geq \mathsf{i}_2 + 1 = 54$. Therefore, the cost of p is at least $55 > 31 \geq \mathsf{cost}(p')$, which contradicts the minimality of p.

Case 2: $j_L > \mathsf{right}^P_{r+1}$. Let q be the sub-path of p from v_R to v_L. Let cost_L be the number of left edges in q and cost_R be the number of right edges in q. We first show a path from v_R to $x[\mathsf{left}^I_\ell..\mathsf{right}^I_r]$ of length cost_L. Let $e = x[i_1..j] \to x[i_2..j]$ be a left edge in q. It must be that $j \in [j_L..\mathsf{right}^I_r] \subseteq [\mathsf{right}^P_{r+1} + 1..\mathsf{right}^I_r]$. Hence, by Lemma 20, there exists an edge $e' = x[i_1..\mathsf{right}^I_r] \to x[i_2..\mathsf{right}^I_r]$. Let $e_1, e_2, \ldots, e_{\mathsf{cost}_L}$ be the subsequence of all left edges in q. The path $q^*_1 = e'_1, e'_2, \ldots, e'_{\mathsf{cost}_L}$ is a valid path of length cost_L from v_R to $x[\mathsf{left}^I_\ell..\mathsf{right}^I_r]$.

If $j_L = \mathsf{right}^I_r$ then $q^* = q^*_1$ is a path that satisfies all the requirements. Otherwise, $\mathsf{cost}_R \geq 1$. We claim that there is an edge e_R from $x[\mathsf{left}^I_\ell..\mathsf{right}^I_r]$ to $v_L = x[\mathsf{left}^I_\ell..j_L]$. This is true since $x[\mathsf{left}^I_\ell..\mathsf{left}^P_{\ell+1} - 1] = I_\mathsf{L}(S[\ell]) \cdot \mathsf{Sync_L} = I_\mathsf{L}(T[r]) \cdot \mathsf{Sync_L} = \overleftarrow{\mathsf{Sync_R}[2..|\mathsf{Sync_R}|] \cdot I_\mathsf{R}(T[r])} = \overleftarrow{x[\mathsf{right}^P_{r+1} + 2\ ..\ \mathsf{right}^I_r]}$ and $j_L \in [\mathsf{right}^P_{r+1} + 1..\mathsf{right}^I_r]$. We conclude q^* by appending e_R to the end of q^*_1. Finally, $\mathsf{cost}(q^*) = \mathsf{cost}_L + 1 \leq \mathsf{cost}_L + \mathsf{cost}_R = \mathsf{cost}(q)$, and q^* visits $x[\mathsf{left}^I_\ell..\mathsf{right}^I_r]$. ◁

Let p be a shortest path from u to v in G_x. According to the claim, we can indeed assume that p consists of a shortest path p_1 from v to w and a shortest path p_2 from w to v. Therefore we have $\mathsf{cost}(p) = \mathsf{cost}(p') = \mathsf{Fib}^{-1}(\mathsf{p}) + \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 11 - 2\alpha$ as required. ◀

6 Correctness

Let $D(0) = 57$, $D(1) = 56$, $D(2) = 55$, $D_{\mathsf{sync}}(0) = 31$, $D_{\mathsf{sync}}(1) = 29$, $D_{\mathsf{sync}}(2) = 27$, and $B = 83$. The following lemma summarize Lemmata 21, 22, and 26.

▶ **Lemma 27.** *Let $\ell \in [|S|]$ and let $r \in [|T|]$ be two integers. Denote $S[\ell] = \alpha$ and $T[r] = \beta$. The following is satisfied.*

1. $\mathsf{dist}_{G_x}(x[\mathsf{left}^P_\ell..\mathsf{right}^P_r], x[\mathsf{left}^P_{\ell+1}, \mathsf{right}^P_r]) = D(\alpha)$
2. $\mathsf{dist}_{G_x}(x[\mathsf{left}^P_\ell..\mathsf{right}^P_r], x[\mathsf{left}^P_\ell, \mathsf{right}^P_{r+1}]) = D(\beta)$
3. $\mathsf{dist}_{G_x}(x[\mathsf{left}^P_\ell..\mathsf{right}^P_r], x[\mathsf{left}^P_{\ell+1}, \mathsf{right}^P_{r+1}]) = D(\alpha) + D(\beta)$ *if* $\alpha \neq \beta$
4. $\mathsf{dist}_{G_x}(x[\mathsf{left}^P_\ell..\mathsf{right}^P_r], x[\mathsf{left}^P_{\ell+1}, \mathsf{right}^P_{r+1}]) = D_{\mathsf{sync}}(\alpha)$ *if* $\alpha = \beta$
5. $2D(0) - D_{\mathsf{sync}}(0) = 2D(1) - D_{\mathsf{sync}}(1) = 2D(2) - D_{\mathsf{sync}}(2) = B$

Proof. According to Lemma 21, we have $D(\gamma) = \mathsf{i}_\gamma + 2$ for every $\gamma \in \{0,1,2\}$. It follows from Lemma 22 that if $\alpha \neq \beta$ we have $\mathsf{dist}_{G_x}(x[\mathsf{left}^P_\ell..\mathsf{right}^P_r], x[\mathsf{left}^P_{\ell+1}, \mathsf{right}^P_{r+1}]) = \mathsf{i}_\alpha + \mathsf{i}_\beta + 4 = \mathsf{i}_\alpha + 2 + \mathsf{i}_\beta + 2 = D(\alpha) + D(\beta)$. It follows from Lemma 26 that $D_{\mathsf{sync}}(\gamma) = \mathsf{Fib}^{-1}(\mathsf{p}) + \mathsf{Fib}^{-1}(\mathsf{i}_\alpha) + 11 - 2\gamma = 11 + 9 + 11 - 2\gamma = 31 - 2\gamma$ for every $\gamma \in \{0,1,2\}$.

Indeed, we have $2 \cdot D(0) - D_{\mathsf{sync}}(0) = 2 \cdot 57 - 31 = 83$, $2 \cdot D(1) - D_{\mathsf{sync}}(1) = 56 \cdot 2 - 29 = 83$ and $2 \cdot D(2) - D_{\mathsf{sync}}(2) = 55 \cdot 2 - 27 = 83$ as required. ◀

We are now ready to prove Lemma 7 which concludes the correctness of the reduction.

▶ **Lemma 7** (Reduction Correctness). *For some constants $D(0), D(1), D(2)$ and B we have:* $\mathsf{HDD}(x,y) = \sum_{\alpha \in \{0,1,2\}} D(\alpha)(\#_\alpha(S) + \#_\alpha(T)) - \mathsf{LCS}(S,T) \cdot B$.

Proof. We prove the equality claimed, by showing two sides of inequality.

$\mathsf{HDD}(x,y) \leq \sum_{\alpha \in \{0,1,2\}} D(\alpha)(\#_\alpha(S) + \#_\alpha(T)) - \mathsf{LCS}(S,T) \cdot B$. Denote $c = \mathsf{LCS}(S,T)$. Let $\mathcal{I} = i_1 < i_2 < i_3.., \ldots, .. < i_c \subseteq [|S|]$ and $\mathcal{J} = j_1 < j_2 < j_3 \ldots < j_c \subseteq [|T|]$ be two sequences of indices such that $S[i_k] = T[j_k]$ for every $k \in [c]$. Thus, $\mathcal{I}$ and $\mathcal{J}$ represent a maximal common subsequence of S and T.

We present a path p in G_x from x to y. The path p starts in $x = x[\mathsf{left}^P_1..\mathsf{right}^P_1]$, and consists of 3 types of subpaths.

1. Left deletion subpath: a shortest path from $x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_r]$ for some $\ell \in [|S|]$ and $r \in [|T|+1]$.
2. Right deletion subpath: a shortest path from $x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to $x[\mathsf{left}^P_\ell..\mathsf{right}^P_{r+1}]$ for some $\ell \in [|S|+1]$ and $r \in [|T|]$.
3. Match subpath: a shortest path from $x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to $x[\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$ for some $\ell \in [|S|]$ and $r \in [|T|]$.

Specifically, if p visits $x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$, then p proceeds in a left deletion subpath if $\ell \in [|S|] \setminus \mathcal{I}$. Otherwise, p proceeds in a right deletion subpath if $r \in [|T|] \setminus \mathcal{J}$. If both $\ell \in \mathcal{I}$ and $r \in \mathcal{J}$, the path p proceeds in a match subpath. Note that it is guaranteed that as long as $\ell \neq |S|+1$ or $r \neq |T|+1$, p continues to make progress until finally reaching $x[\mathsf{left}^P_{|S|+1}..\mathsf{right}^P_{|T|+1}] = y$.

We proceed to analyze the cost of p. For $\alpha \in \{0,1,2\}$ we introduce the following notation regarding $\mathcal{I}$ and $\mathcal{J}$. Let $u_L(\alpha) = |\{i \mid S[i] = \alpha \text{ and } i \notin \mathcal{I}\}|$, $u_R(\alpha) = |\{j \mid T[j] = \alpha \text{ and } j \notin \mathcal{J}\}|$. In addition, let $c(\alpha) = |\{k \mid k \in [c] \text{ and } S[i_k] = \alpha\}|$.

Clearly, by Lemma 27 every $k \in [c]$ induces a cost of $D_{\mathsf{sync}}(S[i_k])$ to p. Moreover, every $i \in [|S|] \setminus \mathcal{I}$, induces a cost of $D(S[i])$ to p, and every $j \in [|T|] \setminus \mathcal{J}$ induces a cost of $D(T[j])$ to p. Thus, we have

$$\begin{aligned}
\mathsf{cost}(p) &= \sum_{\alpha \in \{0,1,2\}} D(\alpha)(u_L(\alpha) + u_R(\alpha)) + \sum_{\alpha \in \{0,1,2\}} D_{\mathsf{sync}}(\alpha) \cdot c(\alpha) \\
&= \sum_{\alpha \in \{0,1,2\}} D(\alpha)(u_L(\alpha) + u_R(\alpha)) + \sum_{\alpha \in \{0,1,2\}} (2D(\alpha) - B) \cdot c(\alpha) \\
&= \sum_{\alpha \in \{0,1,2\}} D(\alpha)(u_L(\alpha) + u_R(\alpha) + 2c(\alpha)) - B \cdot \sum_{\alpha \in \{0,1,2\}} c(\alpha) \\
&= \sum_{\alpha \in \{0,1,2\}} D(\alpha)(\#_\alpha(S) + \#_\alpha(T)) - c \cdot B.
\end{aligned}$$

Where the first equality follows from $B = 2 \cdot D(\alpha) - D_{\mathsf{sync}}$ for every $\alpha \in \{0,1,2\}$, and the last equality is since for every $\alpha \in \{0,1,2\}$ we have $\#_\alpha(S) = u_L(\alpha) + c_\alpha$,$\#_\alpha(T) = u_R(\alpha) + c_\alpha$ and $c = c(0) + c(1) + c(2)$.

$\mathsf{HDD}(x, y) \geq \sum_{\alpha\in\{0,1,2\}} D(\alpha)(\#_\alpha(S) + \#_\alpha(T)) - \mathsf{LCS}(S, T) \cdot B$. Let p be a well-behaved shortest path from x to y in G_x. According to Lemma 9, such a path p exists.

Let $\mathcal{X} = \{v = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r] \mid p \textit{ visits } v\}$. Notice that the vertices of $\mathcal{X}$ are naturally ordered by the order of their occurrences in p, so we denote the ith vertex in $\mathcal{X}$ by x_i. For $i \in [|\mathcal{X}|-1]$, we classify the vertex $x_i = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ for some $\ell \in [|S|+1]$ and $r \in [|T|+1]$ into one of the following four disjoint types.

1. **Match vertex** : if $x_{i+1} = [\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$ and $S[\ell] = T[r]$.
2. **Mismatch vertex** : if $x_{i+1} = [\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r+1}]$ and $S[\ell] \neq T[r]$.
3. **Left deletion vertex** : if $x_{i+1} = [\mathsf{left}^P_{\ell+1}..\mathsf{right}^P_{r}]$.
4. **Right deletion vertex** : if $x_{i+1} = [\mathsf{left}^P_{\ell}..\mathsf{right}^P_{r+1}]$.

Notice that since p is well-behaved, x_i is classified into one of the four types.

We proceed to analyze the cost of the subpath of p from $x_i = x[\mathsf{left}^P_\ell..\mathsf{right}^P_r]$ to x_{i+1} using Lemma 27. If x_i is a match vertex, it induces a cost of $D_{\mathsf{sync}}(S[\ell])$. If x_i is a mismatch vertex, it induces a cost of $D(S[\ell]) + D(T[r])$. If x_i is a right (resp. left) deletion vertex, it induces a cost of $D(S[\ell])$ (resp. $D(T[r])$. For $\alpha, \beta \in \{0, 1, 2\}$ we present the following notations.

- $c_{\mathsf{match}}(\alpha) = |\{x \in \mathcal{X} \mid x \textit{ is a match vertex with } S[\ell] = \alpha\}|$
- $c_{\mathsf{mis}}(\{\alpha, \beta\}) = |\{x \in \mathcal{X} \mid x \textit{ is a mismatch vertex with } \{S[\ell], T[r]\} = \{\alpha, \beta\}\}|$
- $c_{\mathsf{mis}}(\alpha) = |\{x \in \mathcal{X} \mid x \textit{ is a mismatch vertex with } S[\ell] = \alpha \textit{ or } T[r] = \alpha\}|$
- $c_{\mathsf{left}}(\alpha) = |\{x \in \mathcal{X} \mid x \textit{ is a left deletion vertex with } S[\ell] = \alpha\}|$
- $c_{\mathsf{right}}(\alpha) = |\{x \in \mathcal{X} \mid x \textit{ is a right deletion vertex with } T[r] = \alpha\}|$

Note that since every super-gadget is deleted exactly once as a part of an x_i to x_{i+1} subpath. It follows that for every $\alpha \in \{0, 1, 2\}$ we have $\#_\alpha(S) + \#_\alpha(T) = c_{\mathsf{left}}(\alpha) + c_{\mathsf{right}}(\alpha) + c_{\mathsf{mis}}(\alpha) + 2c_{\mathsf{match}}(\alpha)$. Note that for $\alpha \in \{0, 1, 2\}$ we have $c_{\mathsf{mis}}(\alpha) = \sum_{\beta\neq\alpha} c_{\mathsf{mis}}(\{\alpha, \beta\})$. We denote $c_{\mathsf{match}} = c_{\mathsf{match}}(0) + c_{\mathsf{match}}(1) + c_{\mathsf{match}}(2)$. We make the following claim:

▷ Claim. $c_{\mathsf{match}} \leq \mathsf{LCS}(S, T)$.

Proof. We show that there is a common subsequence of S and T with length c_{match}. Let $\mathsf{Pairs} = \{(\ell, r) \mid x[\mathsf{left}^P_\ell..\mathsf{right}^P_r] \textit{ is a match vertex}\}$. Note that Pairs is naturally ordered by the order of occurrences of the corresponding vertices in p. We denote by (ℓ_i, r_i) the ith pair in Pairs according to this order. Note that for every $i \in [|\mathsf{Pairs}| - 1]$, we have $\ell_i < \ell_{i+1}$ and $r_i < r_{i+1}$ due to the definition of a match vertex. Furthermore, we have $S[\ell_i] = T[r_i]$ for every $i \in [|\mathsf{Pairs}|]$. It follows that the subsequence $S[\ell_1], S[\ell_2] \dots, S[\ell_{|\mathsf{Pairs}|}]$ equals to the subsequence $T[r_1], T[r_2], \dots, T[r_{|\mathsf{Pairs}|}]$. Therefore, S and T have a common subsequence of length $|\mathsf{Pairs}| = c_{\mathsf{match}}$. ◁

It follows from the above analysis that

$$\begin{aligned}
\mathsf{cost}(p) &= \sum_{\alpha\in\{0,1,2\}} D_{\mathsf{sync}}(\alpha)c_{\mathsf{match}}(\alpha) + \sum_{\alpha\neq\beta\in\{0,1,2\}} (D(\alpha)+D(\beta))\cdot c_{\mathsf{mis}}(\{\alpha,\beta\}) \\
&\qquad + \sum_{\alpha\in\{0,1,2\}} D(\alpha)(c_{\mathsf{left}}(\alpha)+c_{\mathsf{right}}(\alpha)) \\
&= \sum_{\alpha\in\{0,1,2\}} D_{\mathsf{sync}}(\alpha)c_{\mathsf{match}}(\alpha) + \sum_{\alpha\in\{0,1,2\}} D(\alpha)\sum_{\beta\neq\alpha} c_{\mathsf{mis}}(\{\alpha,\beta\}) \\
&\qquad + \sum_{\alpha\in\{0,1,2\}} D(\alpha)(c_{\mathsf{left}}(\alpha)+c_{\mathsf{right}}(\alpha)) \\
&= \sum_{\alpha\in\{0,1,2\}} D_{\mathsf{sync}}(\alpha)c_{\mathsf{match}}(\alpha) + \sum_{\alpha\in\{0,1,2\}} D(\alpha)\cdot c_{\mathsf{mis}}(\alpha) \\
&\qquad + \sum_{\alpha\in\{0,1,2\}} D(\alpha)(c_{\mathsf{left}}(\alpha)+c_{\mathsf{right}}(\alpha)) \\
&= \sum_{\alpha\in\{0,1,2\}} (2D(\alpha)-B)c_{\mathsf{match}}(\alpha) + \sum_{\alpha\in\{0,1,2\}} D(\alpha)(c_{\mathsf{left}}(\alpha)+c_{\mathsf{right}}(\alpha)+c_{\mathsf{mis}}(\alpha)) \\
&= \sum_{\alpha\in\{0,1,2\}} D(\alpha)(c_{\mathsf{left}}(\alpha)+c_{\mathsf{right}}(\alpha)+c_{\mathsf{mis}}(\alpha)+2c_{\mathsf{match}}(\alpha)) - B\sum_{\alpha\in\{0,1,2\}} c_{\mathsf{match}}(\alpha) \\
&= \sum_{\alpha\in\{0,1,2\}} D(\alpha)\cdot(\#_\alpha(S)+\#_\alpha(T)) - B\cdot c_{\mathsf{match}} \\
&\geq \sum_{\alpha\in\{0,1,2\}} D(\alpha)\cdot(\#_\alpha(S)+\#_\alpha(T)) - B\cdot \mathsf{LCS}(S,T).
\end{aligned}$$

Where the last inequality follows from the claim. ◀

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.14.

2 Itai Boneh, Dvir Fried, Shay Golan, and Matan Kraus. Hairpin completion distance lower bound, 2024. arXiv:2404.11673.

3 Itai Boneh, Dvir Fried, Adrian Miclaus, and Alexandru Popa. Faster algorithms for computing the hairpin completion distance and minimum ancestor. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.5.

4 Karl Bringman and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1216–1235. SIAM, 2018.

5 Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation*, pages 75–85, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

6 Daniela Cheptea, Carlos Martin-Vide, and Victor Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. *Transgressive Computing*, January 2006.

7 Lila Kari, Stavros Konstantinidis, Elena Losseva, Petr Sosík, and Gabriel Thierrin. Hairpin structures in DNA words. In Alessandra Carbone and Niles A. Pierce, editors, *DNA Computing*, pages 158–170, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

8 Lila Kari, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. On hairpin-free words and languages. In Clelia De Felice and Antonio Restivo, editors, *Developments in Language Theory*, pages 296–307, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

9 Lila Kari, Elena Losseva, Stavros Konstantinidis, Petr Sosík, and Gabriel Thierrin. A formal language analysis of DNA hairpin structures. *Fundamenta Informaticae*, 71(4):453–475, 2006.

10 Lila Kari, Kalpana Mahalingam, and Gabriel Thierrin. The syntactic monoid of hairpin-free languages. *Acta Informatica*, 44(3):153–166, June 2007.

11 Florin Manea. A series of algorithmic results related to the iterated hairpin completion. *Theoretical Computer Science*, 411(48):4162–4178, 2010.

12 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. On some algorithmic problems regarding the hairpin completion. *Discret. Appl. Math.*, 157(9):2143–2152, 2009. doi:10.1016/J.DAM.2007.09.022.

13 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. Hairpin lengthening. In Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes, editors, *Programs, Proofs, Processes*, pages 296–306, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

14 Florin Manea, Carlos Martín-Vide, and Victor Mitrana. On some algorithmic problems regarding the hairpin completion. *Discrete Applied Mathematics*, 157(9):2143–2152, 2009. Optimal Discrete Structures and Algorithms.

Solving the Minimal Positional Substring Cover Problem in Sublinear Space

Paola Bonizzoni
Department of Computer Science, University of Milano-Bicocca, Italy

Christina Boucher
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Davide Cozzi
Department of Computer Science, University of Milano-Bicocca, Italy

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Yuri Pirola
Department of Computer Science, University of Milano-Bicocca, Italy

Abstract

Within the field of haplotype analysis, the Positional Burrows-Wheeler Transform (PBWT) stands out as a key innovation, addressing numerous challenges in genomics. For example, Sanaullah et al. introduced a PBWT-based method that addresses the haplotype threading problem, which involves representing a query haplotype through a minimal set of substrings. To solve this problem using the PBWT data structure, they formulate the Minimal Positional Substring Cover (MPSC) problem, and then, subsequently present a solution for it. Additionally, they present and solve several variants of this problem: k-MPSC, leftmost MPSC, rightmost MPSC, and length-maximal MPSC. Yet, a full PBWT is required for each of their solutions, which yields a significant memory usage requirement. Here, we take advantage of the latest results on run-length encoding the PBWT, to solve the MPSC in a sublinear amount of space. Our methods involve demonstrating that k-Set Maximal Exact Matches (k-SMEMs) can be computed in a sublinear amount of space via efficient computation of k-Matching Statistics (k-MS). This leads to a solution that requires sublinear space for, not only the MPSC problem, but for all its variations proposed by Sanaullah et al. Most importantly, we present experimental results on haplotype panels from the 1000 Genomes Project data that show the utility of these theoretical results. We conclusively demonstrate that our approach markedly decreases the memory required to solve the MPSC problem, achieving a reduction of at least two orders of magnitude compared to the method proposed by Sanaullah et al. This efficiency allows us to solve the problem on large versions of the problem, where other methods are unable to scale to. In summary, the creation of μ-PBWT paves the way for new possibilities in conducting in-depth genetic research and analysis on a large scale. All source code is publicly available at `https://github.com/dlcgold/muPBWT/tree/k-smem`.

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

Keywords and phrases Positional Burrows–Wheeler Transform, r-index, minimal position substring cover, set-maximal exact matches

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.12

Supplementary Material *Software (Source code)*: `https://github.com/dlcgold/muPBWT/tree/k-smem` [2], archived at `swh:1:dir:d3467768a54423c8294abfc44f87f18705b3ed02`

Funding PB has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement ALPACA No. 956229. PB, DC, and YP have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539. PB, DC, and YP are also supported by the grant MIUR 2022YRB97K, PINC, Pangenome Informatics: from Theory to Applications. PB, DC, and YP are partially supported by the MUR under the grant "Dipartimenti di Eccellenza 2023-2027" of the Department of Informatics, Systems and Communication of the University of Milano-Bicocca, Italy. TG is funded by NSERC Discovery Grant RGPIN-07185-2020. CB is funded by NSF IIBR (Grant No. P0177268), NSF SCH (Grant No. P0212789) and NIH NIAID (Grant No. R01AI141810).

1 Introduction

In recent years, the Positional Burrows-Wheeler Transform (PBWT) has emerged as a fundamental data structure for solving various challenges in haplotype analysis. One challenge in haplotype analysis is *haplotype threading*, which aims to represent a query haplotype by using one or more substrings derived from at least k haplotypes within a reference panel. Sanaullah et al. [10, 11] showed that solutions to the Minimal Positional Substring Cover (MPSC) problem can be used to solve the *haplotype threading* problem in the context of the PBWT. This insight led to a PBWT-framework for solving haplotype threading that is an alternative to the classical Li and Stephens [7] model that is promised to be more scalable – as the original is unlikely to be efficient enough to be applied on large biobank datasets. In addition to the formulation and solution to the MPSC problem, Sanaullah et al. define several variants of this problem as well as algorithmic solutions. Although more space-efficient than the Li and Stephens model, each of the solutions presented by Sanaullah et al. require the construction and storage of the full PBWT. As it was previously mentioned by Durbin [4], storing the entire PBWT will scale linearly in the size of the input – more precisely, Durbin predicted it to be $13n$ bytes. For reasonably large biobank datasets – such as the ones offered by the 1000 Genomes Project data – this would quickly become unwieldy.

In this paper, we present a space- and time- efficient manner for solving MPSC as well as all variants suggested by Sanaullah et al: k-MPSC, leftmost MPSC, rightmost MPSC, and length-maximal MPSC. Our methods take advantage of the latest advancements in run-length encoding the PBWT, and exploit the fact that MPSC problem can be construed as computing all Set Maximal Exact Matches (SMEMs) in the PBWT, which are maximal matches that are common between a pattern and a panel that cannot be extended. More specifically, we show that we can efficiently compute k-Set Maximal Exact Matches (k-SMEMs), which have the additional constraint that any SMEM must occur in at least k rows in the reference panel. This efficient computation comes from extending the definition of Matching Statistics in the PBWT [3] to k-Matching Statistics (k-MS) in the PBWT. This allows for algorithms developed by Cozzi et al. [3] to compute Matching Statistics to be extended to computing k-MS, which yield k-SMEMs, and finally, a solution to the MPSC and the k-MPSC problem, i.e., find a MPSC in which each positional substring is covered by at least k rows in the reference panel. Hence, we show how k-SMEMs and k-MS naturally provide a framework for developing efficient solutions to the MPSC problem and its variants, proving that we can solve these problems in sublinear space with respect to the size of the panel.

Most importantly, we implement our approach and show that the sublinear bound on the space usage has a practical benefit in haplotype analysis. We compare our implementation to the current state-of-the-art on increasingly-larger autosome panels of the 1000 Genomes Project. Our findings on these datasets demonstrate that our methods substantially lowers memory consumption for solving the k-MPSC problem, achieving reductions of at least two orders of magnitude compared to the method presented by Sanaullah et al. Due to this memory usage, only our method was able to scale to the largest panel sizes in 1000 Genomes Project with reasonable memory constraints. Hence, we show our strategy is applicable of solving haplotype threading on extensive datasets found in modern biobanks.

2 Background

In this paper, we propose novel algorithms for solving the MPSC problem and its variants. Our algorithms are based on μ-PBWT, which is a run-length encoding of the PBWT. To lay the groundwork for describing the algorithms in this paper, we formally introduce the MPSC problems and its variants, and then introduce μ-PBWT and the concepts needed for the development of our methods.

2.1 The Minimal Positional Substring Cover Problems

Throughout this paper, we define a string X over a finite, ordered alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$ to be the concatenation of $|X| = n$ characters $X = X[1..n]$ of Σ. We denote the empty string as ε, the string spanning position i through j as $X[i..j]$ (with $X[i..j] = \varepsilon$ if $i > j$), the i-th prefix of X as $X[1..i]$, and the i-th suffix as $X[i..|X|]$.

A *positional substring* of a string X is a triplet (i, j, X) with $1 \leq i, j \leq |X|$ and we say that the substring corresponding to (i, j, X) is $X[i..j]$. Two positional substrings (i, j, X) and (k, l, Y) are *equal* iff $i = k$, $j = l$, and $X[i..j] = Y[k..l]$. A positional substring (i, j, X) is *contained* in a string Y iff $X[i..j] = Y[i..j]$.

Given a set S of strings of length w (i.e., panel), a *k-positional substring cover* of a w-length string P by S is a set C of positional substrings such that: *(i)* each position $l \in [1, w]$ of P is covered by a $(i, j, X) \in C$ (i.e., $i \leq l \leq j$), *(ii)* each $(i, j, X) \in C$ is contained in P, and *(iii)* each $(i, j, X) \in C$ is contained in at least k distinct strings of S. The size of the cover is the number of elements in C, which we denote as $|C|$.

▶ **Problem 1** (k-Minimal Positional Substring Cover problem, k-MPSC [10]). *Given a set S of h strings of length w and a string P of length w, find, if it exists, a k-positional substring cover of P by S with the smallest size over all k-positional substring covers of P by S.*

The MPSC problem is the k-MPSC problem where k is equal to 1 (i.e., each positional substring of the cover is contained in at least one string of the panel). It is easy to see that a solution to the problem exists iff for every i, with $1 \leq i \leq w$, the positional substrings (i, i, P) are contained in at least k distinct strings of S.

The best known algorithm for computing a k-MPSC requires $O(w)$ time [11] and it works column-wise from left to right by extending matches of the string P with at least k strings of the panel S. At any column, if the current match cannot be extended, then a new match starting at the current column is initiated. Optimality is ensured by the property of k-MPSC modularity [10, Lemma 2]. Matches of P with at least h strings in S are efficiently computed and extended using the Positional Burrows–Wheeler Transform (PBWT) of S. Hereon, we denote $h \cdot w$ as n, which will be used throughout this paper to bound the space and time complexity. The k-MPSC algorithm of Sanaullah et al. [10] requires $\mathcal{O}(n)$ space to ensure constant-time random access to the input panel and to the PBWT.

Next, we draw a relationship between positional substrings and a generalization of maximal exact matches, which we refer to as *k-Set Maximal Exact Match* (k-SMEM), by requiring the maximality of matches to be defined as follows.

▶ **Definition 2** (k-SMEM). *Let S be a set of h sequences of length w and let P be a string of length w. The pair (i, j) is a k-SMEM if the positional substring (i, j, P) is contained in at least k sequences of S and one of the following holds:*

- *$i = 1$ and $j = w$*
- *$i = 1$ and $(i, j + 1, P)$ is not contained in at least k strings of S*
- *$j = w$ and $(i - 1, j, P)$ is not contained in at least k strings of S*
- *$(i - 1, j, P)$ and $(i, j + 1, P)$ are not contained in at least k strings of S*

It is straightforward to observe that every positional substring (i, j, P), which is part of a k-MPSC of P by S, generates an interval that fits within a k-SMEM (i', j'), where $i' \leq i$ and $j' \geq j$. This is because (i, j) either directly constitutes a k-SMEM or can be extended either to the left, the right, or both.

▶ **Problem 3** (k-SMEM-finding)**.** *Given a set S of h sequences of length w, a string P of length w, and an integer k, such that $1 \leq k \leq h$, find all k-SMEMs between P and S.*

As previously mentioned, the k-MPSC problem does not admit a solution if there exists any positional substring (i, i, P) that is not included in at least k strings from S. On the contrary, the k-SMEM-finding problem always admits a solution–possibly not covering some columns.

Leftmost, Rightmost, and Length-maximal MPSC

Given a panel S and a string P there can exist several distinct k-MPSC of the same size (hence, several solutions to the problem). Since the returned solution might affect the results of downstream applications of k-MPSC, three problems have been identified [10, 11] to constrain (and, possibly, to uniquely identify) the returned solution. As in the original paper, we state the problems in terms of MPSC (hence, 1-MPSC), but they can be generalized to k-MPSC.

For the definition of the problems, given a MPSC C, the i-th positional substring of C, for $1 \leq i \leq |C|$, is the i-th positional substring in the enumeration of the positional substrings of C by increasing the starting positions, while the *length* of C is the sum of the lengths of its positional substrings.

- find a *leftmost MPSC C* of P by S, i.e. a MPSC of P by S such that any i-th substring in C starts at least as early as the i-th substring of every other MPSC of P by S
- find a *rightmost MPSC C* of P by S, i.e. a MPSC of P by S such that any i-th substring in C ends at least as late as the i-th substring of every other MPSC of P by S
- find a *length-maximal MPSC* of P by S, i.e. the MPSC that has the largest length out of all MPSCs of P by S

Given a string P of length w, a set S of h strings of length w, and the PBWT of S, Sanaullah et al. showed that all these problems can be solved in $\mathcal{O}(w)$ time and $\mathcal{O}(n)$ space [11].

2.2 Positional Burrows–Wheeler Transform

The PBWT [4] is a data structure that allows to efficiently perform pattern matching tasks on a set $S = \{S_1, \ldots, S_h\}$ of h binary sequences of length w.

The core data structure is composed of two arrays per each column j: the *prefix array* PA_j and the *divergence array* DA_j. In detail, PA_j stores the permutation of the set $\{1, \ldots, h\}$ induced by the co-lexicographic ordering of prefixes of S up to column $j-1$. More formally, $\mathsf{PA}_j[i]$ is equal to k iff $S_k[1..j-1]$ is the i-th element in co-lexicographical ordered list of prefixes $S_1[1..j-1], \ldots, S_h[1..j-1]$. We note that $\mathsf{PA}_1 = \{1, \ldots, h\}$. $\mathsf{DA}_j[i]$ stores the length of the longest common suffix between the sequence in position i and its predecessor in the co-lexicographic ordering of prefixes up to the $(j-1)$-th column, i.e., $S_{\mathsf{PA}_j[i]}[1..j-1]$ and $S_{\mathsf{PA}_j[i-1]}[1..j-1]$. We note that $\mathsf{DA}_1 = \{0, \ldots, 0\}$. Finally, the PBWT of S is a matrix $\mathsf{PBWT}[1..h][1..w]$ where each column j stores the bits contained in each position j of each input sequence reordered by the permutation induced by PA_j. More formally, if we consider the input set S as a matrix M and we denote the j-th column of a matrix A by $\mathsf{col}(A)_j$, we have $\mathsf{col}(\mathsf{PBWT})_j[i] = \mathsf{col}(\mathsf{M})_j[\mathsf{PA}_j[i]]$ for all $i = 1..h$ and $j = 1..w$. Durbin [4] prove that we can compute the entire set of PA arrays, the entire set of DA arrays, and the PBWT matrix in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

2.3 Run-length Encoded PBWT and μ-PBWT

In the seminal paper that first introduced the PBWT, Durbin [4] noted that run-length encoding can be adapted to the PBWT. Later Cozzi et al. [3] and Bonizzoni et al. [1] proposed various data structures to efficiently store and query a run-length encoded PBWT (RLPBWT). In the context of the PBWT, the number of *runs* is equal to the number of length-maximal substrings of equal symbols that appear in the columns of the PBWT. Given r_j as the number of runs in RLPBWT column j, we denote r as $\sum_{1\leq j\leq w} r_j$.

Here, we consider the RLPBWT implementation of Cozzi et al., which is referred to as μ-PBWT. Similar to the conventional BWT, the SMEMs-finding problem can be solved by computing Matching Statistics for a pattern P against a set of sequences S in a run-length manner.

▶ **Definition 4** (Matching Statistics in the PBWT). *Given a binary panel composed by h sequences $S = \{S_1, \dots, S_h\}$ of length w and a pattern $P[1..w]$, we define the Matching Statistics of P with respect to S as an array $MS[1..w]$ of $(\mathsf{row}, \mathsf{len})$ pairs such that, for each position $1 \leq j \leq w$:*

- $S_{MS[j].\mathsf{row}}[j - MS[j].\mathsf{len} + 1..j] = P[j - MS[j].\mathsf{len} + 1..j]$ *(having a match of length $MS[j].\mathsf{len}$ shared between P and $MS[j].\mathsf{row}$ that ends in j)*
- $P[j - MS[j].\mathsf{len}..j]$ *does not occur as a suffix ending in the j-th column in any sequences of S (left maximality)*
- $MS[j].\mathsf{row} = -$ *and* $MS[j].\mathsf{len} = 0$ *iff* $P[j]$ *does not occur in column j*

Observe that SMEM are computed from the Matching Statistics array. More precisely, a SMEM of length $MS[j].\mathsf{len}$ occurs between P and row $MS[j].\mathsf{row}$, starting from position $j - MS[j].\mathsf{len} + 1$ in P, if $MS[j].\mathsf{len} \neq 0$ and either $j = w$ or $MS[j].\mathsf{len} \geq MS[j+1].\mathsf{len}$. In fact, we cannot extend to the right the considered longest common suffix shared by P and any sequence in S, guaranteeing the right maximality.

As in [3], μ-PBWT computes the Matching Statistics array in $\mathcal{O}(r)$ space by storing only the following data:

- a *mapping structure* to support in constant time the FL (First-to-Last) function used to follow a row in the permutation induced by the PBWT from left to right. Note that μ-PBWT can perform in logarithmic time the reverse of this mapping, following a row from right to left;
- the PA *samples* at run boundary;
- the set of *thresholds*, that identify the positions of the first minimum DA value in the range of each run.

μ-PBWT also stores a small successor data structure, called Φ data structure, that is used to identify the location of the SMEMs in $\mathcal{O}(r)$ space. Using the Φ data structure, we retrieve the previous/next PA/DA value from given a PA/DA value in $\mathcal{O}(\log n/r)$ time [3]. We refer readers to Cozzi et al. paper and to Bonizzoni et al. paper to recall the methods used to compute the Matching Statistics array and the SMEMs.

3 Methods

In this section, we first present a method to extend the Matching Statistics computation to be able to detect k-SMEMs. We then present a novel approach using Matching Statistics to solve some variants of the MPSC problem. The principle underlying our approach is that addressing the k-SMEM problem is essential for dealing with the k-MPSC problem.

3.1 From MS and SMEMs to k-MS and k-SMEMs

In 2023, Tatarnikov et al. [12] extended MONI [9] (an efficient RLBWT [8] and r-index [5, 6] implementation) to demonstrate its capability in computing k-MEMs – which are defined as maximal exact matches of a pattern against a text T that occur at least k time in T.

We recall that FL is the function used to trace from left to right the position of a given row when it is changed by the reordering of rows induced by the PBWT, i.e., computing the position of the bit corresponding to a certain row in a column in the PBWT from the previous one.

▶ **Definition 5** (k-support values)**.** *Given an index column j, a run endpoint index b in the $(j-1)$-th column and the corresponding k-interval $\mathsf{DA}_j[\mathsf{FL}(b)-k+2..\mathsf{FL}(b)+k-1]$, we define $(\mathsf{off}_b, \mathsf{L}_b)$ as its k-support values, where:*

- off_b *stores the offset from $\mathsf{FL}(b)$ to get the beginning of the sub-interval of size $k-1$ of the k-interval that maximizes the minimum divergence array value d across all the possible sub-intervals of size $k-1$. If this interval starts in $\mathsf{FL}(b)+1$, we have $\mathsf{off}_b = -1$*
- $\mathsf{L}_b = d$

In other words, k-support values determine, for a run endpoint b, the sub-interval of size $k-1$ of $\mathsf{DA}_j[\mathsf{FL}(b)-k+2..\mathsf{FL}(b)+k-1]$ which has the longest possible common suffix (of length d) shared by all the $k-1$ rows (plus the previous one by DA array definition) stored in the same sub-interval in PA_j. In addition, due to the definition of DA_j, we can include the row that precedes the interval in this set of rows. In Figure 1 we illustrate an example of k-interval and k-support values.

We can compute the k-support values or at indexing time, accessing the entire PA/DA in constant time, or at querying time using the Φ data structure.

▶ **Theorem 6.** *Given k and $\mathsf{DA}_j[1..h]$ as an array with random access in constant time, we can compute the k-support values in $\mathcal{O}(k)$ time. Instead, if we consider the μ-PBWT Φ functions to access DA_j, these values can be computed in $\mathcal{O}(k \log n/r)$ time.*

The following result shows that adding the computation of the k-support to μ-PBWT does not change the space complexity.

▶ **Lemma 7.** *Given k and the μ-PBWT for a panel of size $n = hw$, we can store all the k-support values in a $\mathcal{O}(r)$ space, having two additional integers at each run boundary.*

Next, to efficiently compute k-SMEMs in a run-length manner, we generalize the definition of Matching Statistics to the k-MS.

▶ **Definition 8** (k-Matching Statistics in the PBWT)**.** *Given a binary panel composed by h sequences $S = \{S_1, \dots, S_h\}$ of length w, a pattern $P[1..w]$ and a value k, we define the k-Matching Statistics of P with respect to S as an array k-$MS[1..w]$ of $(\mathsf{row}, \mathsf{len})$ pairs such that, for each position $1 \leq j \leq w$:*

	DA_6	1	2	3	4	5	$col(\mathsf{PBWT})_6$
1	0	0	0	0	1	0	0
2	3	0	1	0	1	0	1
3	5	0	1	0	1	0	1
4	5	0	1	0	1	0	0
5	1	0	1	0	0	0	0
6	4	1	1	0	0	0	1
7	1	1	0	0	1	0	0
8	3	0	1	0	1	0	1

Figure 1 k-support values example with $k = 4$. The figure shows the co-lexicographical ordering up to column 5 used to compute $col(\mathsf{PBWT})_6$ and DA_6. Suppose that some run boundary b in $col(\mathsf{PBWT})_5$ is mapped to $col(\mathsf{PBWT})_6[4]$ (the green 0 in the $col(\mathsf{PBWT})_6$ column). We consider the 4-interval $\mathsf{DA}_6[\mathsf{FL}(b) - k + 2..\mathsf{FL}(b) + k - 1] = \mathsf{DA}_6[2..7]$, circled in green in the DA_6 column. We now consider all the possible subintervals of size $k - 1$, here 3, considering their minimum divergence array value. By definition $\mathsf{DA}_j[i]$ compares row i e row $i - 1$ in the co-lexicographical ordering so, with $k = 3$, we are considering four rows. For example, with $\mathsf{DA}_6[2]$ we are taking into account also row 1 to compute that divergence array value. These subintervals, including the additional rows, are identified by circles in the panel. We are interested in the optimal possible subinterval, so the one that involves the maximum common extension to the left d. In this example, the orchid one, i.e. $\mathsf{DA}_6[2..4]$, is the optimal one, with $d = 3$. In conclusion, we store the offset $\mathsf{off}_b = 2$ and $\mathsf{L}_b = 3$.

- $S_{k\text{-}MS[j].\mathsf{row}}[j - k\text{-}MS[j].\mathsf{len} + 1..j] = P[j - k\text{-}MS[j].\mathsf{len} + 1..j]$ *(there exists a match of length k-$MS[j]$.len shared between P, k-$MS[j]$.row and at least other $k - 1$ rows in S that ends in j)*
- *$P[j - k\text{-}MS[j].\mathsf{len}..j]$ does not occur as a suffix ending in the j-th column in any subset S' of sequences of S with $|S'|$ greater or equal of k (left maximality of the match)*
- *$k\text{-}MS[j].\mathsf{row} = -$ and $k\text{-}MS[j].\mathsf{len} = 0$ iff $P[j]$ does not occur at least k time in column j*

Given μ-PBWT, as described in Section 2.3, we can extend it with the k-support values and solve the k-SMEMs problem in a run-length encoding manner. Note that the k-support values can be pre-computed and queried in constant time, or they can be retrieved each time according to the complexity in Theorem 6. Observe that the computation of $k\text{-}MS.\mathsf{len}$ array involves updating column j, beginning with $j = 1$, and necessitates the modification of new additional arrays: len_k and len_t. We add a *support index* s_j, which is a position in the column j to verify whether at least k rows have the same left-maximal match up to the j-th column. Then:

- len_k stores in position j the length of the left-maximal matches shared by at least k rows (defined by the sub-interval that we get from s_j) in the panel
- len_t acts as the classical $MS.\mathsf{len}$ array. In column j, $\mathsf{len}_t[j]$ stores the length of the semi-left maximal match shared between $k\text{-}MS[j].\mathsf{row}$ and a row in the input panel S. We say *semi-left maximal* because, unlike classical the MS array, we cannot extend to the left a match if a column i, such that $1 \leq i < j$, does not contain at least k symbols $P[i]$,

We do not need to store these arrays in (entirely) memory – rather we store two variables with their values in column j. See Figure 2 for an example of len_k and len_t.

3.1.1 Computing k-MS and k-SMEMs

The computation of the k-MS array involves iterating starting from position $j = 1$ within both the array and the pattern P. Initially, if the first column contains at least k occurrences of the symbol $P[1]$, then we assign the values $k\text{-}MS[1].\mathsf{row} = s$ and $k\text{-}MS[1].\mathsf{len} = 1$, where

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
2	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
3	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
4	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
5	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
6	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
8	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
9	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
12	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
17	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
18	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
19	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
20	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
row	17	17	17	16	14	$\underline{14}$	–	19	12	$\underline{12}$	12	$\underline{12}$	18	18	$\underline{18}$
len$_t$	1	2	3	4	5	6	0	1	2	3	4	5	2	3	4
len$_3$	1	2	3	4	5	6	0	5	6	5	2	3	7	8	9
len	1	2	3	4	5	$\underline{6}$	0	1	2	$\underline{3}$	2	$\underline{3}$	2	3	$\underline{4}$

Figure 2 Example of 3-SMEMs results. We consider an input matrix M that represents a set of 20 sequences of length 15 and a pattern P of the same length. We show the 3-SMEMs using the same colour in M and P. In addition, we show the k-MS array (with underlined all the values that represent a SMEM), the len$_3$ array, and the len$_t$ array.

s belongs to the set S with the condition that $S_s[1] = P[1]$. If this condition is not met – that is, if there are fewer than k instances of $P[1]$ in the first column – we set k-$MS[1]$.row to a placeholder value (indicated by $-$) and k-$MS[1]$.len to 0.

We now describe how to update k-$MS[j]$ from k-$MS[j-1]$ for $j > 1$. To update the column of k-MS for j we need to follow k-$MS[j-1]$.row in the permutation induced by PA$_j$ to extend to column j the left-maximal match obtained in column $j-1$ if it is possible. Given that k-$MS[j-1]$.row is mapped in row i of the column j (via the FL function) in the PBWT, there are two cases that we have to consider: (1) we have a mismatch between the pattern P and row i in the j-th column, and (2) we have a match between the pattern P and row i in the j-th column. Within these two cases, there are some additional sub-cases that we have to consider. In what follows, we describe how to address these cases, and then, we show how to use len$_k$ and len$_t$ to compute k-MS and k-SMEMs.

Case 1: A mismatch occurs

We assume that we have a mismatch in the j-th column, i.e., col(PBWT)$_j[i] \neq P[j]$. If col(PBWT)$_j$ does not contain at least k occurrences of the symbol $P[j]$ then we have a *complete mismatch*. We represent this with k-$MS[j]$.row $= -$ and len$_t[j] = $ len$_k[j] = 0$, resetting the

k-MS computation in the next column – implying that we restart the computation from column $j+1$ as we were in column 1. Otherwise, if $k\text{-}MS[j].\mathsf{row}$ (where $k\text{-}MS[j].\mathsf{row}$ and $MS[j].\mathsf{row}$ are equal) and $\mathsf{len}_t[j]$ (which corresponds to the traditional $MS[j].\mathsf{len}$ value), then we apply the approach of Cozzi et al. [3]. This update involves the selection of a new run boundary b. We use the k-support values to update $\mathsf{len}_k[j]$ and s_{j+1} as follows: $\mathsf{len}_k[j] = L_b$ and $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. In this way, we consider the sub-interval of size k, consisting of rows that share the longest common suffix of length L_b up to column j.

Case 2: A match occurs

Next, we assume that we have a match in the j-th column, i.e., $\mathsf{col}(\mathsf{PBWT})_j[i] = P[j]$. If $\mathsf{col}(\mathsf{PBWT})_j$ does not contain at least k occurrences of the symbol $P[j]$ we have an *unfeasible match*. We handle this case as in the *complete mismatch* case described above. We recall that we use the support index s_j to identify the starting position of the sub-interval of size k in column j of the PBWT in which the rows share the longest common suffix up to column j. Observe that if we have $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j+k-1] = P[j]$ then we have a left-maximal match shared by at least k rows. It follows that we can update the Matching Statistics in the same manner as in the μ-PBWT: $k\text{-}MS[j].\mathsf{row} = k\text{-}MS[j-1].\mathsf{row}$, $s_{j+1} = \mathsf{FL}(s_j)$, $\mathsf{len}_t[j] = \mathsf{len}_t[j-1]+1$, and $\mathsf{len}_k[j] = \mathsf{len}_k[j-1]+1$. Informally, we follow the same sub-interval (by FL function) in column $j+1$, updating all the length values by 1 (due to the match) to consider the next column. We note that the condition $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j+k-1] = P[j]$ can be checked without scanning entirely $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j+k-1]$. In fact, the condition is satisfied iff $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j+k-1]$ lay in the same run and we can test this fact in logarithmic time (in the number of runs of the column j-th).

Finally, it is necessary to address the sub-cases that arise when there are symbols within $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j+k-1]$ that do not match $P[j]$. Again, this fact can be checked by looking at the runs of the first and the last symbols in this interval. If $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j+k-1]$ lay on different runs it means that we have at least a run of symbols that differ from $P[j]$ between these two runs. Thus, we need to use a different support index. To identify the new support index s_{j+1} it is necessary to trace a new row. Recall that all the information used to update the k-MS are stored at a run boundary, and therefore, we need to select a new run boundary b such that $\mathsf{col}_j(\mathsf{PBWT})[b] = P[j]$. This b is selected as in the mismatch case, assuming to consider i as the index of the first mismatch in $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j+k-1]$. At this point we can update $\mathsf{len}_t[j]$. For this purpose, we compute the length of the common suffix up to the j-th column between $k\text{-}MS[j-1].\mathsf{row}$ (that we are currently following due to the match) and $\mathsf{PA}_j[b]$ and compare it to $\mathsf{len}_t[j-1]+1$. We select the minimum of these two lengths to retain only the suffix that encompasses k rows. So, if we denote $\mathsf{lcs}_j(A,B)$ as the longest common suffix up to the j-th column shared by rows A and B, then we have $\mathsf{len}_t[j] = \min(\mathsf{lcs}_j(k\text{-}MS[j-1].\mathsf{row}, \mathsf{PA}_j[b]), \mathsf{len}_t[j-1]+1)$. Then we update $k\text{-}MS[j].\mathsf{row}$ using the prefix array samples as in μ-PBWT. Moreover, to update s_{j+1} and $\mathsf{len}_k[j]$ we use the k-support values as follows: $\mathsf{len}_k[j] = L_b$ and $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. In every case mentioned that requires jumping to a new row, we note that if there is no run available for the jump (for instance, when there are only two runs), then the algorithm followed is akin to what is done in the case of a complete mismatch.

Filling the k-MS array and computing k-SMEMs

To account for the lengths of all matches, we formulate the $k\text{-}MS.\mathsf{len}$ as follows: $k\text{-}MS[j].\mathsf{len} = \min(\mathsf{len}_t[j], \mathsf{len}_k[j])$, for all $j = 1..w$. This step can be performed after processing each column j. Finally, similar to the computation of SMEMs [1, 3], a k-SMEM of length $k\text{-}MS[j].\mathsf{len}$ occurs between P and row $k\text{-}MS[j].\mathsf{row}$, starting from position $j - k\text{-}MS[j].\mathsf{len} + 1$ in P, if $k\text{-}MS[j].\mathsf{len} \neq 0$ and either $j = w$ or $k\text{-}MS[j].\mathsf{len} \geq k\text{-}MS[j+1].\mathsf{len}$.

▶ **Theorem 9.** *Given a set S of h sequences of length w, an integer k, and query z of size w, we can compute the k-SMEMs for z using $\mathcal{O}(r)$ space.*

Providing a similar bound for the time complexity of computing k-SMEM in the μ-PBWT is an open problem that warrants consideration. We conjecture that it can be done in $\mathcal{O}(w \log r)$ time. Lastly, we illustrate the results of computing 3-SMEMs in Figure 2.

3.1.2 From k-MS to k-MPSC

We show now that we can build a solution for k-MPSC by using the k-MS array. We recall that each positional substring that belongs to a k-MPSC is contained in a k-SMEM. By the non-inclusion property of k-SMEM, each starting position of a k-SMEM is covered by one and only one k-SMEM. Combining the above two properties, it follows that there exists a k-MPSC in which each substring is a prefix of a k-SMEM. In addition, there exists only one k-SMEM that covers $P[w]$. Therefore, we can start building a solution to k-MPSC by adding this k-SMEM as a positional substring. We can now proceed iterating the process of including each time the single left-maximal match (covered by at least k rows in S) that ends in the column j prior to the starting column $j + 1$ of the last positional substring added to k-MPSC. This left-maximal match is represented by k-MS values in position j, and it is unique according to the definition of k-MS. The minimality condition is guaranteed by the fact that we cannot have a better solution, i.e., a single left-maximal match γ that covers two of the left-maximal matches, α and β, that were already added to k-MPSC. To see this, consider that if such a γ exists, it must begin prior to α owing to the left-maximal property and extend at least up to the identical ending position β at the j-th index. In this case, the definition of k-MS property implies that γ should be identified in position j of the k-MS array. This is a contradiction because in position j of k-MS array we already selected the left-maximal match β. Notice that this procedure extends Algorithm 1 to k-MPSC, as demonstrated in the following section where we prove its efficacy in computing the leftmost MPSC.

3.2 Solving the Leftmost, Rightmost, and Length-maximal MPSC Problems with MS and SMEMs

We now show how to use MS and SMEMs to solve the leftmost, rightmost, and length maximal MPSC problems. We first show that Algorithms 1 and 2 solve the leftmost (Lemma 10) and rightmost (Lemma 11) MPSC problems, respectively, in sublinear space. We recall that r denotes the number of runs in the PBWT of a set S of h sequences of length w, which is upper bounded by their size $n = h \cdot w$. The proofs of Lemma 10 and Lemma 11 are based on the non-inclusion property of SMEMs, which implies that the starting positions and ending positions of the set of SMEMs are unique. In other words, there cannot exist two SMEMs that start in the same position in the pattern P and two SMEMs that end in the same position in the pattern P.

Algorithm 1 Leftmost MPSC by MS.

```
1: function LeftMost($MS$)
2:     $j \leftarrow w$                                  ▷ $|MS| = w$
3:     $j' \leftarrow 0$
4:     while $j \neq 0$ do
5:         $j' \leftarrow j - MS[j]$.len
6:         report $(j'+1, j, MS[j]$.row$)$
7:         $j \leftarrow j'$
```

Algorithm 2 Rightmost MPSC by MS.

```
1: function RightMost($MS$)
2:     $i \leftarrow 1$                                  ▷ $|MS| = w$
3:     for $j = 1 \rightarrow w-1$ do
4:         if $MS[j]$.len $\geq MS[j+1]$.len then
5:             report $(i, j, MS[j]$.row$)$
6:             $i \leftarrow j+1$
7:     report $(i, w, MS[w]$.row$)$
```

▶ **Lemma 10** (Leftmost MPSC). *Given a panel S and the MS array of a pattern P with respect to S, Algorithm 1 computes the leftmost MPSC C of P by S in time $\mathcal{O}(|C|)$ and $\mathcal{O}(r)$ space.*

Proof. We seek a MPSC such that each i-th positional substrings starts not after any other i-th positional substring in a MPSC. It is easy to see that a leftmost MPSC is composed of positional substrings that are prefixes of SMEMs – or, in other words, that are left-maximal. Otherwise, we can extend the positional substring to the left, contradicting the assumption of having a leftmost MPSC. Scanning the MS array from right to left, we consider the SMEM that ends in the last position of the pattern. This SMEM is in the set of the leftmost MPSC by definition, and the fact that it is the only one that includes position w. We can compute its starting position j as $w - MS[w]$.len $+ 1$ and we can add $(j, w, MS[w]$.row$)$ to the set C. The set C is a leftmost MPSC of the columns j to w. A leftmost MPSC of columns 1 to $j-1$ must include the left-maximal match that ends at $j-1$. Given that $j' = (j-1) - MS[j-1]$.len $+ 1$, by definition of SMEM, it follows that we cannot have a SMEM that includes $MS[j-1]$.row in position $j-1$ and starts before j'. Thus j' is the starting position of the next positional substring (of length $MS[j-1]$.len) in the leftmost MPSC C of the columns j' to w. Iterating the previous procedure until reaching column 1 gives the leftmost MPSC. Since each iteration adds a positional substring to the cover with a single constant-time access of the MS array, the algorithm runs in time proportional to $|C|$. ◀

▶ **Lemma 11** (Rightmost MPSC). *Given a panel S of w-length strings and the MS array of a pattern P with respect to S, Algorithm 2 computes the rightmost MPSC of P by S in time $\mathcal{O}(w)$ and $\mathcal{O}(r)$ space.*

Proof. We are interested in a MPSC where each i-th positional substring ends no earlier than any other i-th positional substring within it. Symmetrically to the leftmost MPSC, the positional substrings of a rightmost MPSC are suffixes of SMEMs. Given the MS array, a position j is the ending position of a SMEM iff $MS[j]$.len $\geq MS[j+1]$.len or $j = w$. For simplicity, in this proof we assume that $MS[w+1].len = 0$. The construction of the rightmost MPSC is symmetric to that of the leftmost MPSC. As in the leftmost MPSC, we are not interested in overlaps between positional substrings. To compute the rightmost MPSC, we scan from left to right the MS array. By definition, the SMEM that starts in the first column is in the set of rightmost MPSC, being the only one that includes the first position. Denoting j as the ending position of this SMEM, we add the positional substring $(1, j, MS[j]$.row$)$ to the set C of the rightmost MPSC. At each position j where $MS[j]$.len $\geq MS[j+1]$.len, there are no other right-maximal matches, meaning there cannot exist an SMEM containing $MS[j]$.row at position j that extends beyond position j. Thus, js are the ending position

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
3	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0
4	1	1	1	0	1	1	1	1	0	0	1	0	0	0	0
5	0	1	0	1	0	0	0	0	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	1	0	0	1	1	0	0	0

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4

(a) Splitting of a SMEM into substrings from leftmost/rightmost MPSC. We show a panel M and a pattern P with the corresponding MS array. In the panel, we have dashed circles for the leftmost MPSC and dotted circles for the rightmost MPSC. With the same colour, we identify a SMEM.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
j/j'	←	2	←	←	←	6	←	←	←	←	11	←	←	←	15

(b) Example of computing leftmost MPSC (identified by circles in the P row) by the MS array. In the last line we show the jumps used to skip the overlaps as in Algorithm 1.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
i	1	–	–	–	–	6	–	–	–	–	11	–	13	–	–

(c) Example of computing rightmost MPSC (identified by circles in the P row) by the MS array. For the sake of simplicity, in the last line, we show the updating of value i as in Algorithm 2.

Figure 3 Example of the relationship between leftmost/rightmost MPSC and SMEMs on the same set of sequences of [11].

of any other positional substring $(i+1, j, MS[j].\mathsf{row})$ in the set C, where i is the ending column of the last substring added to C. Since each position of the MS array is scanned once, and as each iteration requires only a single constant-time access of the MS array, the algorithm runs in time proportional to w. ◀

Figure 3 shows how our algorithms compute the leftmost and the rightmost MPSC.

We now consider the problem of finding a length-maximal MPSC. Sanaullah et al. [11] showed the length-maximal MPSC problem can be solved in $\mathcal{O}(n + |Q|)$ space given that the leftmost MPSC, the rightmost MPSC, and the set Q of SMEMs have been computed in $\mathcal{O}(n)$ space, while the algorithm used to combine all of them to find a length-maximal MPSC requires $\mathcal{O}(|Q|)$ space. We showed that we can compute the leftmost and the rightmost MPSCs in $\mathcal{O}(r)$ space (Lemma 10 and Lemma 11) from the MS array. We also showed in Section 2.3 that the MS array and the set of SMEMs can be computed in $\mathcal{O}(r)$ space using the μ-PBWT. As a consequence, the following lemma holds.

▶ **Lemma 12.** *Given μ-PBWT for a set S of h strings of length w, a string P of length w, and the set Q of SMEMs shared by S and P, a length-maximal MPSC of P by S can be computed in $\mathcal{O}(r + |Q|)$ space.*

4 Results

We implemented our methods for computing k-MS and k-SMEMs with and without pre-computed k-support values, and compared these to the k-MPSC implementation by Sanaullah et al. [11]. Hereon, we refer to this method as `k-MPSC`.

4.1 Datasets

We evaluated the execution time and maximum memory usage for indexing and querying of all these methods using biallelic chromosome panels from the 1000 Genomes Project (1KGP) [13]. All the data is publicly available at `https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/`. We selected the panels for chromosomes 22, 18, and 2. These panels have 5,008 samples/rows and between 1M and 6M variations/columns. From these panels, we extracted 30 rows to use them as queries, hence we consider input panels with 4978 rows. The performance metrics were computed using `/usr/bin/time` on a machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256GB RAM and 8GB of swap, running Ubuntu 20.04.6 LTS.

4.2 Implementation Details and Experimental Setup

We augmented μ-PBWT implementation with the algorithms to compute k-SMEMs. We made all source code publicly avaliable at `https://github.com/dlcgold/muPBWT/tree/k-smem` (with pre-computed k-support values) and `https://github.com/dlcgold/muPBWT/tree/k-smem-live` (without pre-computed k-support values).

`k-MPSC` is written in `C++14` and necessitated a few modifications to the code. It was not possible to index panels as large as the ones from the 1KGP due to design choices on the dynamics allocation of the data structure. Moreover, `k-MPSC` implementation halted if an unfeasible column was encountered (a column with less than k symbols), which frequently occurred only after very few columns. To make a meaningful comparison, we edited the source code to solve the allocation issue and to restart the computation after an unfeasible column is encountered.

Empirical experimental performances for the computation of a leftmost MPSC, a rightmost MPSC, and a length-maximal MPSC (from the MS array and the set of SMEM) are directly proportional to the experimental results in Cozzi et al. [3]. Therefore, no additional dedicated experiments were conducted.

4.3 Results on 1000 Genomes Project Data

Table 1 reports the results of the indexing task and of the querying task with 30 queries. As anticipated, pre-computing the k-support values leads to an increase in computation times by up to approximately 100% compared to the variant without pre-computed values. We note that, due to the average number of runs in each column, we only need to store short bit-compressed integer vectors for the k-support values, implying that only a $\sim$5% increase in memory usage is needed for the implementation without these values stored. A fairly obvious note that warrants comment is that for $k = 1$, we do not require k-support values, thus the memory usage remains the same both with and without them. Regarding `k-MPSC`, the indexing phase consists of the PBWT computation of both panel and queries. Since `k-MPSC` needs a full PBWT, it requires almost two orders of magnitude more memory than both of our approaches. This is unsurprising in light of the work of Durbin [4], which stated that the whole set of data structures for PBWT require $13n$ bytes to be queried.

Table 1 Indexing and querying wall clock time and max memory usage comparison on chromosomes 2/18/22 panels from 1KGP, with 4978 rows and 30 queries.

			Wall Clock Time (seconds)			Max Memory usage (GB)		
			μ-PBWT			μ-PBWT		
Chr.	Task	k	(pre)	(no-pre)	`k-MPSC`	(pre)	(no-pre)	`k-MPSC`
2	*Index*	1	1381	1384	-	6.45	6.45	-
		50	1643	1387	-	6.62	6.45	-
		200	2798	1418	-	6.62	6.45	-
	Querying	1	254	247	-	5.87	5.87	-
		50	1687	2471	-	6.57	6.42	-
		200	3563	11834	-	8.45	8.31	-
18	*Index*	1	425	424	5750	3.26	3.26	168.58
		50	697	450	5750	3.36	3.26	168.58
		200	807	429	5750	3.36	3.26	168.58
	Querying	1	70	70	128	1.92	1.92	171.19
		50	739	817	229	2.13	2.08	171.30
		200	1740	4263	119	2.74	2.71	171.22
22	*Index*	1	191	189	2616	1.77	1.77	79.01
		50	304	193	2616	1.84	1.77	79.01
		200	400	194	2616	1.84	1.77	79.01
	Querying	1	31	32	130	0.98	0.98	82.31
		50	336	392	171	1.11	1.07	82.85
		200	726	2054	133	1.47	1.44	82.48

Regarding the querying performance results, the computation of k-support values during query time leads to an increase in execution time, which scales with the value of k due to the use of the Φ data structure. When k was equal to 200, the k-SMEMs computation requires up to a third of the time with the pre-computed values. Regarding `k-MPSC`, with random access in constant time to the complete set of data structures of the PBWT, we observe that it runs up to 20 times faster than μ-PBWT with pre-computed values and up to 60 times faster than the variant with query-time k-support values computation. Moreover, the experiments confirm that the k-MPSC algorithm does not linearly scale on k, as explained in Section 2.1. Regarding memory usage for querying, a similar analysis apply as for the indexing phase with the additional factor that we also have in memory the queries for the μ-PBWT. In addition, we suspect that the slight increase in memory usage with larger k values is due to the support variables used. Since in instances where the computation of MS requires pointer adjustments because there are not k equivalent values in a designated interval, we have to compute $\mathsf{lcs}(A, B)$.

Due to the memory usage of `k-MPSC`, we were unable to evaluate the performance on the larger panels, such as the one related to the chromosome 2 in the 1000 Genomes Project data. To ensure a comprehensive overview, Table 1 includes the outcomes achieved by μ-PBWT on the largest dataset (chromosome 2), which `k-MPSC` could not process within the allocated memory constraints. This demonstrates the advantages of using μ-PBWT which requires a sublinear amount of memory.

5 Conclusions

In this paper, we address the theoretical aspects of the haplotype threading problem, primarily aiming to offer practical solutions capable of scaling to the large biological datasets currently stored in biobanks. In particular, we presented two distinct results. First, we adapted the run-length encoding paradigm for the PBWT to compute k-Matching Statistics (k-MS) and the k-SMEMs in sublinear space. Next, we show that computing k-MS provides a theoretical framework to solving the k-Minimal Positional Substring Cover problem as well as all the variants of the MPSC problem in sublinear space. Our experimental results decisively show that our method achieves a significant reduction in memory usage for computing the k-MPSC problem, reducing it by no less than two orders of magnitude relative to the approach by Sanaullah et al. [11]. This enables our approach to scale to large datasets collected in contemporary biobanks. In turn, the development of μ-PBWT opens up unprecedented opportunities for comprehensive genetic studies – e.g., for genotyping and imputation workflows [14] – and exploration on a large scale.

References

1 Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. Data Structures for SMEM-Finding in the PBWT. In *International Symposium on String Processing and Information Retrieval*, pages 89–101. Springer, 2023.

2 Davide Cozzi. muPBWT k-SMEM. Software, European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 87253, swhId: `swh:1:dir:d3467768a54423c8294abfc44f87f18705b3ed02`, (visited on 04/06/2024). URL: `https://github.com/dlcgold/muPBWT/tree/k-smem`.

3 Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ-PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data. *Bioinformatics*, 39(9):btad552, 2023.

4 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.

5 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.

6 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):2:1–2:54, 2020. `doi:10.1145/3375890`.

7 Na Li and Matthew Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics*, 165(4):2213–2233, 2003.

8 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching: 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005. Proceedings 16*, pages 45–56. Springer, 2005.

9 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *Journal of Computational Biology*, 29(2):169–187, 2022.

10 Ahsan Sanaullah, Degui Zhi, and Shaoije Zhang. Haplotype threading using the positional Burrows-Wheeler transform. In *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

11 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Minimal positional substring cover is a haplotype threading alternative to Li and Stephens Model. *Genome Research*, 33(7):1007–1014, 2023. `doi:10.1101/gr.277673.123`.

12 Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. MONI Can Find k-MEMs. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

13 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

14 Naga Sai Kavya Vaddadi, Taher Mun, and Ben Langmead. Minimizing reference bias with an impute-first approach. *bioRxiv*, 2023. doi:10.1101/2023.11.30.568362.

Online Context-Free Recognition in OMv Time

Bartłomiej Dudek
Institute of Computer Science, University of Wrocław, Poland

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Abstract

One of the classical algorithmic problems in formal languages is the context-free recognition problem: for a given context-free grammar and a length-n string, check if the string belongs to the language described by the grammar. Already in 1975, Valiant showed that this can be solved in $\tilde{O}(n^{\omega})$ time, where ω is the matrix multiplication exponent. More recently, Abboud, Backurs, and Vassilevska Williams [FOCS 2015] showed that any improvement on this complexity would imply a breakthrough algorithm for the k-Clique problem. We study the natural online version of this problem, where the input string $w[1..n]$ is given left-to-right, and after having seen every prefix $w[1..t]$ we should output if it belongs to the language. The goal is to maintain the total running time to process the whole input. Even though this version has been extensively studied in the past, the best known upper bound was $O(n^3/\log^2 n)$. We connect the complexity of online context-free recognition to that of Online Matrix-Vector Multiplication, which allows us to improve the upper bound to $n^3/2^{\Omega(\sqrt{\log n})}$.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages

Keywords and phrases data structures, context-free grammar parsing, online matrix-vector multiplication

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.13

1 Introduction

Context-free languages, introduced by Chomsky already in 1959 [3], are one of the basic concepts considered in formal languages, with multiple applications in programming languages [2], NLP [11], computational biology [6], and databases [13]. A context-free language is a language generated by a context-free grammar, meaning that each production rule is of the form $A \to \alpha$, where A is a non-terminal symbol, and α is a string of terminal and non-terminal symbols (possibly empty). It was already established by Chomsky [3] that, without decreasing the expressive power, we can assume that the productions are of the form $A \to a$ and $A \to BC$, where A, B, C are non-terminal symbols, and a is a terminal symbol. From an algorithmic point of view, the natural (and very relevant with respect to the possible applications) question is whether, given such a grammar G and a string $w[1..n]$, we can efficiently check if $w \in \mathcal{L}(G)$. A simple application of the dynamic programming paradigm shows that this is indeed possible in $O(n^3)$ time (ignoring the dependency on the size of the grammar). This is usually called the Cocke–Younger–Kasami (CYK) approach [4, 12, 25]. In 1975, Valiant [23] designed a non-trivial algorithm that solves this problem in $O(BM(n))$ time, where $BM(n)$ denotes the complexity of multiplying two (Boolean) $n \times n$ matrices. Plugging in the currently best known bounds, $BM(n) = O(n^{\omega})$, where $\omega < 2.373$ [24]. See [9] for a somewhat more approachable description of Valiant's algorithm, and [19] for a very elegant simplification (achieving the same running time). This is of course a somewhat theoretical result, and given the practical nature of the problem it is not surprising that other approaches have been developed [5, 16, 17, 22], with high worst-case time complexities, but good behaviour on instances that are relevant in practice. However, the worst-case time complexity has not seen any improvement. In 2002, Lee [15] showed a conditional lower bound that provides some explanation for this lack of improvement: multiplying two (Boolean)

$n \times n$ matrices can be reduced to parsing a string of length $O(n^{1/3})$ for a grammar of size $O(n^2)$. This does exclude a combinatorial $O(gn^{3-\varepsilon})$ algorithm for parsing (more general problem than recognition), where g is the size of the grammar, but does not contradict the existence of e.g. $O(g^2n)$ time algorithm. However, in 2015 Abboud, Backurs, and Vassilevska Williams [1] showed a more general conditional lower bound: even for constant-size grammars, any improvement on the complexity of Valiant's recognition algorithm implies a breakthrough for the well-known k-Clique problem.

In some applications, the input string $w[1..n]$ is given character-by-character, and for each prefix $w[1..i]$ we should decide if it belongs to $\mathcal{L}(G)$ before reading the next character. This is known as the online CFG recognition. The goal is to minimise the total time to process all the characters. It is not hard to adapt the CYK approach to work in $O(n^3)$ total time for this variant, but this seems difficult (or perhaps impossible) for Valiant's algorithm. Graham, Harrison, and Ruzzo [8] designed a (slightly) subcubic algorithm, and Rytter [19] further improved the complexity to $O(n^3/\log^2 n)$. Surprisingly, no further improvements were achieved. On the lower bound, it is known that on a Turing machine, $\Omega(n^2/\log n)$ steps are required [7, 20]. This is however quite far from the upper bound, and assumes a somewhat restricted model of computation, and brings the natural question of understanding if a faster algorithm exists.

As the complexity of the offline CFG recognition is known to be close to that of (Boolean) matrix multiplication, it is natural to seek a connection between the complexity of its online variant with the so-called online matrix multiplication. As a tool for unifying the complexities of different dynamic problems, Henzinger, Krinninger, Nanongkai, and Saranurak [10] introduced the Online Matrix-Vector Multiplication problem:

▶ **Definition 1** (Online Matrix-Vector Multiplication (OMv)). *Given a matrix $M \in \{0,1\}^{n\times n}$, and a sequence of vectors $v_1, \ldots, v_n \in \{0,1\}^n$, the task is to output Mv_i before seeing v_{i+1}, for all $i = 1, \ldots, n-1$.*

and conjectured that no $O(n^{3-\epsilon})$ time algorithm exists (with the best known upper bound at the time being $O(n^3/\log^2 n)$):

▶ **Hypothesis 2** (OMv Hypothesis [10]). *Every (randomized) algorithm solving OMv must take total time $n^{3-o(1)}$.*

Surprisingly, Larsen and Williams [14] were soon able to construct a faster $n^3/2^{\Omega(\sqrt{\log n})}$ time algorithm. This does not refute the OMv hypothesis, but significantly improves the known upper bound, essentially by saying that we can shave any number of logarithms from the time complexity.

▶ **Theorem 3** ([14]). *There exists a randomized algorithm for OMv that runs in total $n^3/2^{\Omega(\sqrt{\log n})}$ time and succeeds with high probability*[1].

This suggests the possibility of leveraging the progress on the complexity of Online Matrix-Vector Multiplication to improve the complexity of online CFG recognition to improve on the $O(n^3/\log^2 n)$ time complexity from 1985.

[1] By succeeding with high probability we mean that there exists a constant $c > 0$ such that the algorithm succeeds with probability at least $1 - 1/n^c$.

Our contribution. We show that it is possible to use efficient OMv multiplication to speed up online context-free recognition:

▶ **Theorem 4.** *Let G be a context-free grammar, and w be a length-n string, revealed one character at a time. There exists a randomized algorithm that determines, after having seen $w[t]$, if $w[1..t] \in \mathcal{L}(G)$, in $n^3/2^{\Omega(\sqrt{\log n})}$ total time and succeeds with high probability.*

Our solution is based on the classical CYK dynamic-programming approach from 1960s [4,12,25] in which we calculate the set of non-terminals deriving each of the infixes of w. In order to avoid the $O(n^2)$ time for processing a new character $w[t]$, we maintain a division of the current prefix into segments of lengths that are powers of 2 present in the binary representation of t. For each of the segments, we build a structure responsible for processing suffixes $w[i..t]$ that start within the segment and end at t. We extensively use the approach from Theorem 3 for OMv, with a slight adaptation to matrices that grow in time. More precisely, we show that we can process a sequence of vectors $v_1, q_1, v_2, q_2, \ldots$ where $|q_i| = i$ in which we need to calculate $(v_1, \ldots, v_i) \times q_i$ online, before seeing v_{i+1}. This requires one more step of dividing the range of columns into segments of lengths that are powers of 2, and applying the structure from Theorem 3 for each of the segments separately. This results in the same running time as in the standard OMv problem, in which the matrix we multiply with does not change.

We note that our algorithm does not need to know the value of n in advance. In fact, our proof of Theorem 4 can be modified to show that the amortised time for processing the t-th character is $O(t^2/2^{\Omega(\sqrt{\log t})})$, so in particular after having seen $w[t]$ we know whether $w[1..t] \in \mathcal{L}(G)$, with the total time spent on $w[1], w[2], \ldots, w[t]$ being $O(t^3/2^{\Omega(\sqrt{\log t})})$, for every $t = 1, 2, \ldots$.

2 Preliminaries

Consider a context-free grammar $G = (V_N, V_T, P, S)$. Without loss of generality we assume that G is in Chomsky normal form [3,21], that is every production in P is either $A \to BC$ or $A \to c$ for $A, B, C \in V_N$ and $c \in V_T$. By $v \overset{\star}{\Rightarrow} s$ we denote that string s can be derived from non-terminal v in the grammar G.

We are given a string w of length n character-by-character and for each $t = 1..n$ need to decide if the string $w[1..t]$ belongs to $\mathcal{L}(G)$ or not. For every t, the answer should be provided before reading the $(t+1)$-th character and we call such a procedure *online*. Our algorithm will compute the set of all non-terminals that produce every infix of w: $U[i,j] = \{v \in V_N : v \overset{\star}{\Rightarrow} w[i..j]\}$. Then the t-th bit of the output is whether S belongs to $U[1,t]$ or not.

Our approach has polynomial dependence on the size of the grammar G, which we omit while stating the complexity of the parsing algorithm.

In the analysis of our algorithm we will consider sums of non-constant number of distinct expressions containing the Ω function. Unless stated otherwise, all the Ωs within one sum correspond to the same function, namely there exists one constant bounding all the expressions at the same time. An example of such sum appears in the following lemma that will be useful in the next section:

▶ **Lemma 5.** *For any constant $a > 0$, we have $\sum_{k=0}^{\log n} 2^{ak-\Omega(\sqrt{k})} = n^a/2^{\Omega(\sqrt{\log n})}$.*

Proof. Let $t = \log n$. As we discussed before, various Ω functions correspond to one particular Ω function, which means that we can read the expression $(*) = \sum_{k=0}^{t} 2^{ak-\Omega(\sqrt{k})}$ as: there exists a constant $c > 0$ such that $(*) \leqslant \sum_{k=0}^{t} 2^{ak-c\sqrt{k}}$. First we show for which $0 \leqslant k < t$ we can upper bound the k-th summand by the last element from the sum:

$$ak - c\sqrt{k} < at - c\sqrt{t}$$
$$\Updownarrow$$
$$c(\sqrt{t} - \sqrt{k}) < a(t-k) = a(\sqrt{t}-\sqrt{k})(\sqrt{t}+\sqrt{k})$$
$$\Updownarrow$$
$$\frac{c}{a} - \sqrt{t} < \sqrt{k}$$

So in particular, for $k \geqslant k_0 = (\frac{c}{a})^2$ we have that $ak - c\sqrt{k} \leqslant at - c\sqrt{t}$. For $k < k_0$ we have $2^{ak-c\sqrt{k}} < 2^{ak_0}$, so:

$$(*) \leqslant k_0 \cdot 2^{ak_0} + \sum_{k=k_0}^{t} 2^{at-c\sqrt{t}} \leqslant O(1) + t \cdot 2^{at-c\sqrt{t}} = O(2^{at-0.9c\sqrt{t}}) = 2^{at-\Omega(\sqrt{t})}. \qquad \blacktriangleleft$$

3 Parsing context-free grammars online

Our algorithm processes characters from the input one-by-one. While processing the t-th character it has already computed $U[i,j]$ for $1 \leqslant i \leqslant j < t$ and needs to compute $U[i,t]$ for $1 \leqslant i \leqslant t$. We maintain a division of the interval $[1..(t-1)]$ into $c = O(\log t)$ intervals: $[e_1, e_2), [e_2, e_3), \ldots, [e_c, e_{c+1})$ where $e_1 = 1, e_{c+1} = t$ and lengths of the intervals are exactly the powers of 2 in the binary representation of $t-1$, in the decreasing order. On a high level, for the j-th interval there is a data structure X_j responsible for computing $U[i,t]$ for $e_j \leqslant i < e_{j+1}$, based on the outputs from $X_{j'}$ for $j' > j$. We call such a data structure a *process*. We say that the *size* of process X_j is the length of the interval it corresponds to, that is $|[e_j, e_{j+1})| = e_{j+1} - e_j$. The processes are created and removed following the binary representation of t, and a process for interval $[a, a+2^k)$ exists only for $t = a+2^k, \ldots a+2^{k+1}-1$. In the following theorem we describe the calculations performed in each of the processes.

▶ **Theorem 6.** *Let $\mathcal{I} = [p, p+s)$ be an interval of positions from w. Consider the following sequence Q of at most s queries $Q_{p+s}, Q_{p+s+1}, \ldots$: in the t-th query we are given set $Q_t = \{(i,v) : v \overset{\star}{\Rightarrow} w[i,t], i \in [p+s, t]\}$ and need to compute $A_t = \{(i,v) : v \overset{\star}{\Rightarrow} w[i,t], i \in \mathcal{I}\}$.*

There exists a randomized algorithm answering online all queries from Q in total $s^3/2^{\Omega(\sqrt{\log s})}$ randomized time that succeeds with high probability.

Before we prove the above theorem, we show how to apply it to obtain the algorithm for parsing context-free grammars online.

▶ **Theorem 4.** *Let G be a context-free grammar, and w be a length-n string, revealed one character at a time. There exists a randomized algorithm that determines, after having seen $w[t]$, if $w[1..t] \in \mathcal{L}(G)$, in $n^3/2^{\Omega(\sqrt{\log n})}$ total time and succeeds with high probability.*

Proof. We show that Algorithm 1 correctly parses all prefixes of w online, in the desired time complexity. First, we show that the operations in Algorithm 1 satisfy the requirements on queries described in Theorem 6. Indeed, we always create a process β with $\mathcal{I} = [t+1-2^r, t+1)$ and the subsequent queries are $Q_{t+1}, Q_{t+2}, \ldots$, so in particular the first query concerns the position right after the end of $\mathcal{I}$, as required. Observe that due to line 7 the sequence of sizes

Algorithm 1 Parsing context-free grammar online.

```
    Input: Context-free grammar $G = (V_N, V_T, P, S)$

 1: $B := [\,]$                                   ▷ 1-based list of processes $B^1, B^2, \ldots$
 2: for $t = 1, 2, \ldots$ do
 3:     $Q_t := \{(t, v) : (v \to w[t]) \in P\}$
 4:     for $j = |B|$ to 1 do
 5:         $A := B^j.query(Q_t)$
 6:         $Q_t := Q_t \cup A$
 7:     let $r$ be maximal such that $\sum_{i=0}^{r-1} |B^{|B|-i}| = 2^r - 1$
 8:     remove the last $r$ processes from $B$
 9:     add $new\_process(\mathcal{I} = [t + 1 - 2^r, t + 1))$ to the end of $B$
10:     output whether $(1, S) \in Q_t$
```

of the processes follows the binary representation of t so we create a process of size 2^k for t such that $t \equiv 2^k \pmod{2^{k+1}}$ and the last query that we possibly process at β is Q_{t+2^k}, so β is queried at most $t + 2^k - t = 2^k = |\beta|$ times.

Now we calculate the complexity of the algorithm. We create a new process of size 2^k exactly $\lfloor \frac{n+2^k}{2^{k+1}} \rfloor < n/2^k$ times. Each process of size 2^k answers at most 2^k queries so we can directly apply Theorem 6 to bound the total running time of preprocessing and all queries processed by the process. Hence the total running time of the algorithm is upper bounded by:

$$\sum_{k=0}^{\log n} \frac{n}{2^k} \cdot \left(2^k\right)^3 / 2^{\Omega(\sqrt{k})} = n \cdot \sum_{k=0}^{\log n} 2^{2k - \Omega(\sqrt{k})} = n^3 / 2^{\Omega(\sqrt{\log n})}.$$

The last step follows by Lemma 5 and the claim holds. ◀

Proof of Theorem 6

In order to prove Theorem 6, we need to introduce some notation and insights following Rytter's variant of the Valiant's offline parser of context-free grammars [19]. We will operate on matrices of binary relations over the set of non-terminals V_N and we call such matrices *relational*. Formally, every element of a relational matrix is of the form $\{0,1\}^{V_N \times V_N}$. To simplify the notation, our relational matrices will be indexed by intervals $\mathcal{J}_1, \mathcal{J}_2 \subseteq [1, n]$ of consecutive numbers, corresponding to substrings of the input string w. We define $\otimes$-multiplication of matrices A, B with indices $\mathcal{J}_a \times \mathcal{J}_c$ and $\mathcal{J}_c \times \mathcal{J}_b$ respectively, as:

$$(A \otimes B)[i, j]^{X,Y} = \bigvee_{\substack{k \in \mathcal{J}_c \\ Z \in V_N}} A[i, k]^{X,Z} \cdot B[k, j]^{Z,Y} \quad \text{for } i \in \mathcal{J}_a, j \in \mathcal{J}_b, X, Y \in V_N$$

Similarly, we define *relational vectors* as vectors of subsets of V_N, that is their elements are of the form: $\{0,1\}^{V_N}$, and matrix-vector product $M \otimes F$ of matrix M (indexed with $\mathcal{J}_a \times \mathcal{J}_c$) and vector F (indexed with $\mathcal{J}_c$) as:

$$(M \otimes F)[i]^X = \bigvee_{\substack{k \in \mathcal{J}_c \\ Z \in V_N}} M[i, k]^{X,Z} \cdot F[k]^Z \quad \text{for } i \in \mathcal{J}_a, X \in V_N$$

▶ **Lemma 7** ([18]). *We can compute relational matrix-matrix $\otimes$-product in $|V_N|^3$ multiplications of two Boolean matrices and relational matrix-vector $\otimes$-product in $|V_N|^2$ multiplications of a Boolean matrix and a vector. The Boolean matrices and vectors that we multiply have the same size as the relational ones.*

Proof. By definition of $\otimes$-product, in order to multiply two relational matrices we iterate over all triples of X, Y, Z of non-terminals, create Boolean matrices $A'^{X,Z}, B^{Z,Y}$ where $A'^{X,Z}[i,j] = A[i,j]^{X,Z}$ and $B'^{Z,Y}[i,j] = B[i,j]^{Z,Y}$ and calculate $A' \cdot B'$ using the standard Boolean $(+,\cdot)$-product. Then $(A \otimes B)[i,j]^{X,Y} = \bigvee_{Z \in V_N} (A'^{X,Z} \cdot B'^{Z,Y})[i,j]$.

Matrix-vector $\otimes$-multiplication can be calculated analogously. ◀

Now we are able to show the main theorem of this section.

▶ **Theorem 6.** *Let $\mathcal{I} = [p, p+s)$ be an interval of positions from w. Consider the following sequence Q of at most s queries $Q_{p+s}, Q_{p+s+1}, \ldots$: in the t-th query we are given set $Q_t = \{(i,v) : v \overset{\star}{\Rightarrow} w[i,t], i \in [p+s,t]\}$ and need to compute $A_t = \{(i,v) : v \overset{\star}{\Rightarrow} w[i,t], i \in \mathcal{I}\}$.*

There exists a randomized algorithm answering online all queries from Q in total $s^3/2^{\Omega(\sqrt{\log s})}$ randomized time that succeeds with high probability.

Recall that $G = (V_N, V_T, P, S)$ is the considered grammar. Whenever we refer to $w[i, i-1]$ for any i, we mean an empty string.

Preprocessing. During the preprocessing phase we first run Rytter's algorithm [19] on $w[p..p+s-1]$ and compute $U[i,j]$ for all $p \leqslant i \leqslant j < p+s$ in $O(s^\omega)$ time[2]. Based on that we define a relational matrix V with rows and columns $p..(p+s)$ by setting

$$V[i,j]^{X,Y} = 1 \iff \exists_{Z \in V_N} \Big((X \to ZY) \in P \wedge Z \overset{\star}{\Rightarrow} w[i..j-1] \Big) \quad \text{for } p \leqslant i \leqslant j \leqslant p+s$$

Informally, this means that we can extend "to the left" every infix of w that starts at position j and can be derived from Y to an infix that starts at position i, ends at the same position and that can be derived from X. For empty infixes we set $V[i,i]^{X,Y} = 1 \iff X = Y$. When we do not specify the value of some entries of a matrix, it means that there are all zeros in that entry. For instance, for $i > j$ in V we have $V[i,j]^{X,Y} = 0$ for all $X, Y \in V_N$.

Now we calculate $V^\star = V^s$ with exponentiation by squaring, using $\otimes$-product at every step in total $O(s^\omega \log s) = \tilde{O}(s^\omega)$ time, by Lemma 7. Observe that $V^\star$ describes all possibilities of extending an infix "to the left" at most s times. As $j - i \leqslant s$, we never need more than s steps to extend an infix starting at position j to an infix starting at position i and then:

$$V^\star[i,j]^{X,Y} = 1 \iff \exists_{\substack{k_1 < \ldots < k_r \\ k_1 = i, k_r = j \\ Z_1, \ldots, Z_r \in V_N \\ Z_1 = X, Z_r = Y}} \Big(\forall_{1 \leqslant e < r} V[k_e, k_{e+1}]^{Z_e, Z_{e+1}} = 1 \Big) \quad \text{for } p \leqslant i \leqslant j \leqslant p+s$$

See Figure 1 for an illustration.

Invariant. During the process of answering queries, before receiving a subsequent query Q_t, we maintain a relational matrix H with similar properties as V, with rows $p..(p+s)$, but with columns $(p+s)..t$, that is:

$$H[i,j]^{X,Y} = 1 \iff \exists_{Z \in V_N} \Big((X \to ZY) \in P \wedge Z \overset{\star}{\Rightarrow} w[i..j-1] \Big) \quad \text{for } p \leqslant i \leqslant p+s \leqslant j \leqslant t$$

[2] In [19] is computed $VALID(k,\ell) = \bigcup_{k \leqslant i \leqslant j \leqslant \ell} \{(A,i,j) : A \in U[i,j]\}$.

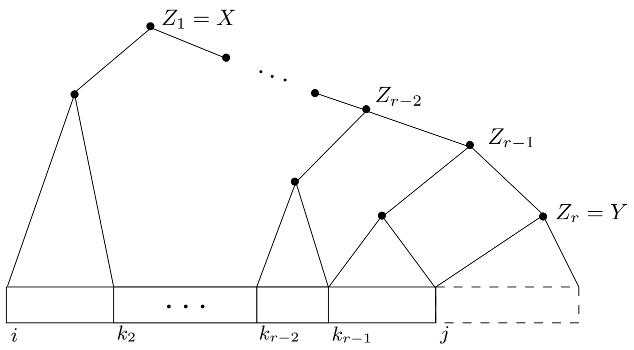


Figure 1 Illustration of the definition of $V^\star$. Note that we do not specify the endpoint of the last string, starting at position j and derived from Y, because we are only interested in the possible extensions "to the left" from such a string.

This matrix also describes extensions "to the left", but from an infix starting at position $j \geqslant p+s$ to an infix starting at position $i \leqslant p+s$. In order to satisfy the invariant, at the end of preprocessing we initialize $H[i, p+s] = V[i, p+s]$ for $p \leqslant i \leqslant p+s$.

Query. From the input set Q_t we create a relational vector $F[(p+s)..t]$ such that

$$F[j]^Y = 1 \iff Y \overset{\star}{\Rightarrow} w[j..t] \iff (j, Y) \in Q_t.$$

Let $A = H \otimes F$. Then $A[i]^X = 1 \implies X \overset{\star}{\Rightarrow} w[i..t]$ for $p \leqslant i \leqslant p+s$. However, this is not an equivalence yet, because we need to consider a larger number of extensions "to the left" using infixes fully contained in $\mathcal{I} = [p, p+s)$. For that we use matrix $V^\star$ and compute $A' = V^\star \otimes A$. Then we have $A'[i]^X = 1 \iff X \overset{\star}{\Rightarrow} w[i..t]$ for $p \leqslant i \leqslant p+s$ and we can construct the desired set A_t that can be returned from the procedure. As the last step of processing the query, we update matrix H by adding $(t+1)$-th column by definition: $H[i, t+1]^{X,Y} = 1 \iff \exists_{Z \in V_N} (X \to ZY) \in P \wedge A'[i]^Z = 1$.

Running time. The only operations that can take more than $O(s)$ time in the above procedure are the matrix-vector $\otimes$ multiplications $H \otimes F$ and $V^\star \otimes A$. Recall that by Lemma 7 it suffices to show how to perform these operations efficiently for matrices and vectors over the Boolean semiring, not the relational ones. In the case of $V^\star \otimes A$ we have one matrix $V^\star$ subsequently multiplied by different vectors A, so we can directly apply Theorem 3 and process online all the queries in total $s^3/2^{\Omega(\sqrt{\log s})}$ time.

For the multiplications $H \otimes F$ we need a slightly different approach, because the matrix H changes in time. Similarly as in Algorithm 1 we will divide the columns of H into intervals following the binary representation of the width of H and split H into a number of smaller square matrices. For each of the small matrices we will use the algorithm for OMv from Theorem 3.

▶ **Lemma 8.** *Consider the sequence of at most s operations, where in the j-th one we are given a binary vector v_j of length s and a binary vector q_j of length j and need to calculate $x_j = M_j \cdot q_j$ where M_j is the matrix with s rows and columns $v_1, \ldots, v_j$. There exists a randomized algorithm answering online all the queries in total $s^3/2^{\Omega(\sqrt{\log s})}$ time that succeeds with high probability.*

Proof. Similarly as in Algorithm 1, we maintain a partition of the interval $[1..j]$ into $c = O(\log j)$ intervals: $\mathcal{E}(j) = [e_1, e_2), [e_2, e_3), \ldots, [e_c, e_{c+1})$ where $e_1 = 1, e_{c+1} = j + 1$ and lengths of the intervals follow the binary representation of j, with $[e_1, e_2)$ being the largest one. Intervals correspond to subranges of columns of M_j and for an interval of length 2^k we divide its $s \times 2^k$ submatrix into $s/2^k$ square matrices of size $2^k \times 2^k$. For each such matrix we create a data structure for OMv multiplication, by Theorem 3.

In order to process a query, we first add the new column v_j, update the structure of intervals from $\mathcal{E}(j-1)$ to $\mathcal{E}(j)$ and run preprocessing for each of the newly-created matrices. To answer the query we divide q_j according to $\mathcal{E}(j)$ into vectors $q_j^1, \ldots, q_j^c$ and multiply each vector q_j^i by all the matrices of the same size as q_j and combine the results in one vector y_j of length s, see Figure 2. Then $x_j = \bigvee_{i=1}^{c} y_i$.

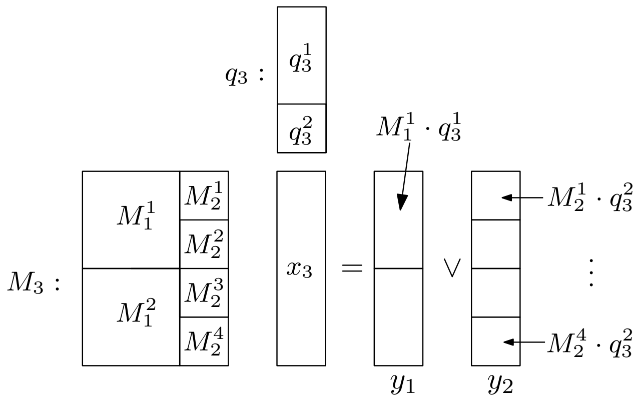


Figure 2 Example of calculating x_3 based on the results from multiplicating square matrices M_i^z by vector q_3^i for $z \in [1, s/|q_3^i|]$.

The correctness of the approach is immediate and now we need to calculate the total running time. While adding new columns to the considered matrix, we create a new interval of length 2^k for j such that $j \equiv 2^k \pmod{2^{k+1}}$, so in total less than $s/2^k$ times. We divide every interval into $s/2^k$ matrices of size $2^k \times 2^k$ and for each of them we create an OMv data structure that answers at most 2^k queries. By Theorem 3 we can process online all the queries for a single matrix in $2^{3k}/2^{\Omega(\sqrt{k})}$ total time. This gives us the following total running time of processing all the queries:

$$\sum_{k=0}^{\log s} s/2^k \cdot s/2^k \cdot (2^k)^{3-\Omega(\sqrt{k})} = s^2 \cdot \sum_{k=0}^{\log s} 2^{k-\Omega(\sqrt{k})} = s^3/2^{\Omega(\sqrt{\log s})}$$

where the last step follows from Lemma 5. ◀

Finally, the total running time of our algorithm is $\tilde{O}(s^\omega)$ for the preprocessing and $s^3/2^{\Omega(\sqrt{\log s})}$ for answering all the queries, which gives $s^3/2^{\Omega(\sqrt{\log s})}$ total time. This concludes the proof of Theorem 6.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. In *FOCS*, pages 98–117. IEEE Computer Society, 2015.

2 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

3 Noam Chomsky. On certain formal properties of grammars. *Inf. Control.*, 2(2):137–167, 1959.

4 John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes (technical report) (2nd revised ed. *Technical report, CIMS, NYU*, 1970.

5 Shay B. Cohen, Giorgio Satta, and Michael Collins. Approximate PCFG parsing using tensor decomposition. In *HLT-NAACL*, pages 487–496. The Association for Computational Linguistics, 2013.

6 Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

7 Hervé Gallaire. Recognition time of context-free languages by on-line turing machines. *Inf. Control.*, 15(3):288–295, 1969.

8 Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recognizer. *ACM Trans. Program. Lang. Syst.*, 2(3):415–462, 1980.

9 M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1978.

10 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015.

11 Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition.* Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

12 Tadao Kasami. An efficient recognition and syntax algorithm for context-free language. *Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bed-ford, MA.*, 1965.

13 Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. *Proc. VLDB Endow.*, 6(9):601–612, 2013.

14 Kasper Green Larsen and R. Ryan Williams. Faster online matrix-vector multiplication. In *SODA*, pages 2182–2189. SIAM, 2017.

15 Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

16 Adam Pauls and Dan Klein. K-best A* parsing. In *ACL/IJCNLP*, pages 958–966. The Association for Computer Linguistics, 2009.

17 Alexander M. Rush, David A. Sontag, Michael Collins, and Tommi S. Jaakkola. On dual decomposition and linear programming relaxations for natural language processing. In *EMNLP*, pages 1–11. ACL, 2010.

18 Wojciech Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3):12–22, 1985.

19 Wojciech Rytter. Context-free recognition via shortest paths computation: A version of Valiant's algorithm. *Theor. Comput. Sci.*, 143(2):343–352, 1995.

20 Joel I. Seiferas. A simplified lower bound for context-free-language recognition. *Inf. Control.*, 69(1-3):255–260, 1986.

21 Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

22 Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *ACL (1)*, pages 455–465. The Association for Computer Linguistics, 2013.

23 Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

24 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *SODA*, pages 3792–3835. SIAM, 2024.

25 Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. Control.*, 10(2):189–208, 1967.

When Is the Normalized Edit Distance over Non-Uniform Weights a Metric?

Dana Fisman
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Ilay Tzarfati
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Abstract

The well known Normalized Edit Distance (NED) [Marzal and Vidal 1993] is known to disobey the triangle inequality on contrived weight functions, while in practice it often exhibits a triangular behavior. Let d be a weight function on basic edit operations, and let NED_d be the resulting normalized edit distance. The question what criteria should d satisfy for NED_d to be a metric is long standing. It was recently shown that when d is the uniform weight function (all operations cost 1 except for no-op which costs 0) then NED_d is a metric. The question regarding non-uniform weights remained open. In this paper we answer this question by providing a necessary and sufficient condition on d under which NED_d is a metric.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases Normalized Edit Distance, Non-uniform Weights, Triangle Inequality, Metric

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.14

Funding *Ilay Tzarfati*: Supported by ISF grant 2507/21 and Frankel Center for Computer Science, BGU.

Acknowledgements We would like to thank Oded Margalit, Elina Sudit and Sandra Zilles for comments on an earlier draft of this paper.

1 Introduction

The question of quantifying the similarity between two strings is quite ancient [9, 11, 18, 17, 10, 19, 14]. A typical way to measure the distance between two strings, is the Levenshtein distance, aka, *edit distance* (ED) [11]. The edit distance between two strings $w_1, w_2 \in \Sigma^*$ is measured as the minimum *weight* of an *edit path* – a sequence of edit operations *delete*, *insert*, *replace*, or *no-op* – required to transform w_1 to w_2. In the case of uniform weights, all edit operations cost 1 except for *no-op* which costs 0. For example, $\text{ED}(\mathit{Jane}, \mathit{John}) = 3$ since we can transform the string *Jane* to *John* using the edit path $\alpha =$*no-op*(J),*replace*(a,o),*replace*(n,h),*replace*(e,n) which weighs 3 and there is no edit path transforming *Jane* to *John* that weighs less than 3. In many settings, a normalized version of the edit distance is required. To see why, note that for the same argument as above the distance between *JaneKennedy* and *JohnKennedy* is also 3 although clearly the latter pair of strings are much more similar to one another.

In [13] the well-known normalized version of the edit distance, henceforth NED, was suggested in which the distance between two non-empty strings w_1 and w_2 is the minimum *cost* of an edit path between w_1 and w_2. The *cost* of an edit path is the weight of edit operations along the path, divided by the length of the path. The cost of the edit path α above is thus $\frac{3}{4}$ but since $\alpha' =$*no-op*(J),*replace*(a,o),*insert*(h),*no-op*(n),*delete*(e) also transforms *Jane* to *John* we have that $\text{NED}(\mathit{Jane}, \mathit{John}) = \frac{3}{5}$. Similar arguments show that $\text{NED}(\mathit{JaneKennedy}, \mathit{JohnKennedy}) = \frac{3}{12}$, thus it is now apparent that *JaneKennedy* and *JohnKennedy* are more similar to one another (compared to *Jane* and *John*).

The above discussion concerned the case of uniform weights. However, in many applications using a normalized edit distance, such as text retrieval, signal processing, and computational biology, non-uniform weights are used. In the case of non-uniform weights, any basic edit operation has its own weight, e.g. $\mathit{delete}(a)$, $\mathit{insert}(a)$, $\mathit{delete}(b)$, $\mathit{replace}(a,b)$, etc. can cost differently. A function d assigning a weight to each edit operation is assumed, and each such function gives rise to a different version, NED_d, of the normalized edit distance of [13]. It was noted in [13] that NED_d may not satisfy the triangle inequality for certain weight functions d, though it is observed to behave well in practice often enough. The question under which criteria on d is NED_d a metric is standing since. This motivated the introduction of other definitions of normalized edit distance, e.g., the generalized edit distance (GED) [12], and the contextual edit distance (CED) [5]. A sufficient condition on d for GED_d to be a metric was given in [12][1] and in [5] it is shown that CED is a metric when d is the uniform weight. It was recently shown that under the uniform weights, NED is a metric, and that NED enjoys several nice properties that GED and CED do not [7]. The question under which criteria NED_d over a non-uniform weight function d is a metric was left unanswered.

In this paper we provide a necessary and sufficient condition on a weight function d on edit operations, in order for NED_d to be a metric. While it is reasonable to assume that d should be a metric (in the space of edit operations) we show that this is neither a necessary condition, nor a sufficient one. The exact criteria relaxes the requirement of the triangle inequality, makes an additional requirement on the cost of inserts and deletes, and in general concerns only edit operations we term *essentials*. We term d that satisfies these criteria *fine*. The proof that d being fine is also a sufficient condition for NED_d to be a metric generalizes and significantly simplifies the proof that NED_d is a metric in the uniform case [7].

The main result of the paper is the following theorem.

▶ **Theorem 1** (Necessary and Sufficient Condition). *Let* $d : (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \to [0, 1]$. *Let* $a, c \in \Sigma \cup \{\varepsilon\}$ *and* $b \in \Sigma$. *Let* $m = \sup\{\text{NED}_d(w_1, w_2) : w_1, w_2 \in \Sigma^*\}$. *A necessary and sufficient condition for* NED_d *to be a metric is that d satisfies the following properties after removing inessential edit operations.*

1. $d(a, c) = 0$ *iff* $a = c$
2. $d(a, c) = d(c, a)$
3. $d(a, b) + d(b, c) \geq \min\{d(a, c), d(a, \varepsilon) + d(\varepsilon, c)\}$
4. $d(\varepsilon, b) = d(b, \varepsilon) \geq \frac{m}{2}$

The rest of the paper is organized as follows. We provide some preliminaries in §2. In §3 we show that d being a metric is neither a necessary condition nor a sufficient one. In §4 we gradually develop the necessary condition on d, we term a weight function d satisfying these conditions *fine*. In §5 we show that d being fine is a sufficient condition for NED_d to be a metric. In §6 we provide some natural examples for weight functions that are fine, and discuss applications of NED_d in formal verification. Due to lack of space, the proofs regarding the examples in §6 are deferred to the full version of the paper.

2 Notations

Metric spaces

A metric space is an ordered pair $(\mathbb{M}, d)$ where $\mathbb{M}$ is a set and $d : \mathbb{M} \times \mathbb{M} \to \mathbb{R}$ is a *metric*, i.e., it satisfies the following properties for all $m_1, m_2, m_3 \in \mathbb{M}$:

[1] The condition is that d is a metric and all delete and insert operations cost the same.

1. $d(m_1, m_2) = 0$ iff $m_1 = m_2$;
2. $d(m_1, m_2) = d(m_2, m_1)$;
3. $d(m_1, m_3) \leq d(m_1, m_2) + d(m_2, m_3)$.

The first condition is referred to as *identity of indiscernibles*, the second as *symmetry*, and the third as the *triangle inequality*.

Words and Edit Operations

Let Σ be an alphabet and Σ^* (Σ^+) denote all the finite (non-empty) strings over Σ. The length of word $w = \sigma_1\sigma_2...\sigma_n$, denoted $|w|$, is n. We use $w[i]$ to denote the i-th letter of w, and $w[..i]$ for the prefix of w ending at the i-th letter. We denote the empty word by ε. Let $a, b \in \Sigma$. The usual edit operations are *delete a*, *insert a*, *replace a with b*, and *no-op a*. We use the following notations for them. Let $\hat{\Gamma} = (\Sigma \cup \{\varepsilon\})^2$. We use $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ to represent the pair (a, b). An *edit operation* is a letter in $\Gamma = \hat{\Gamma} \setminus \{\left[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right]\}$. The letter $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ denotes *replace a with b*, the letter $\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]$ denotes *no-op a*, the letter $\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]$ denote *delete a*, and the letter $\left[\begin{smallmatrix} \varepsilon \\ a \end{smallmatrix}\right]$ denotes *insert a*. This style of notation will come in handy in §5 when we prove the sufficient condition.

Edit Paths

An *edit path* between words w_1 and w_2 over Σ is a sequence $\left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix}\right]\left[\begin{smallmatrix} a_2 \\ b_2 \end{smallmatrix}\right]\dots\left[\begin{smallmatrix} a_m \\ b_m \end{smallmatrix}\right]$ of elements in Γ satisfying that $a_1a_2\dots a_m = w_1$ and $b_1b_2\dots b_m = w_2$. For instance, take $w_1 = aaa$ and $w_2 = bb$ then $\alpha = \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ is an edit path between w_1 and w_2, and $\alpha' = \left[\begin{smallmatrix} \varepsilon \\ b \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]$ is another edit path between w_1 and w_2. It is sometimes convenient to represent these edit paths as $aaa \mapsto b\_b$ and $\_aaa \mapsto bb\_\_$, respectively. In standard terminology the first edit path would correspond to *replace a* with b, *delete a*, *replace a* with b and the second to *insert b*, *replace a* with b, *delete a*, *delete a*. In §5 we use also strings over $\hat{\Gamma}$ which we refer to as *extended edit-paths.*

We use π_i for the projection of a tuple or a sequence of tuples on its i-th component. E.g. if $\alpha = \left[\begin{smallmatrix} \varepsilon \\ b \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}\right]$ then $\pi_1(\alpha) = \varepsilon aaa$ and $\pi_2(\alpha) = bb\varepsilon\varepsilon$. Let $\alpha = a_1a_2\dots a_k$ be a sequence of symbols over $\Sigma \cup \{\varepsilon\}$. We use $\mathsf{word}(\alpha)$ for the word obtained by concatenating these letters. For instance, $\mathsf{word}(\pi_1(\alpha)) = aaa$. Let $w_1, w_2 \in \Sigma^*$. Let α be an (extended) edit path between w_1, w_2. Then $\mathsf{word}(\pi_1(\alpha)) = w_1$ and $\mathsf{word}(\pi_2(\alpha)) = w_2$. We use $\mathsf{input}(\alpha)$ for $\mathsf{word}(\pi_1(\alpha))$ and $\mathsf{output}(\alpha)$ for $\mathsf{word}(\pi_2(\alpha))$.

Weight, Length and Costs of Edit Paths

Let $d : \Gamma \to [0, 1]$ be a function assigning cost for the basic edit operations. Although $\left[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right]$ is not an edit operation, it is sometimes convenient to assume d is defined on it as well, namely $d : \hat{\Gamma} \to [0, 1]$. When this is the case we simply assume $d(\varepsilon, \varepsilon) = 0$. Let $w_1 \in \Sigma^*$ and $w_2 \in \Sigma^+$. Let $\alpha = \left[\begin{smallmatrix} a_1 \\ b_1 \end{smallmatrix}\right]\left[\begin{smallmatrix} a_2 \\ b_2 \end{smallmatrix}\right]\dots\left[\begin{smallmatrix} a_n \\ b_n \end{smallmatrix}\right]$ be an edit path between w_1 and w_2. We say that the *length* of α is n, and that the *weight* of α is $\sum_{i=1}^{n} d(a_i, b_i)$. We denote them by $\mathsf{len}(\alpha)$ and $\mathsf{wgt}(\alpha)$ respectively. The cost of an edit path α, denoted $\mathsf{cost}(\alpha)$ is defined as $\frac{\mathsf{wgt}(\alpha)}{\mathsf{len}(\alpha)}$. [2]

[2] Note that it is well defined since $w_2 \in \Sigma^+$ guarantees that $\mathsf{len}(\alpha) \neq 0$. To obtain a definition that works also for $w_1 \in \Sigma^+$ and $w_2 \in \Sigma^*$ we can consider also edit paths from w_2 to w_1.

The Normalized Edit Distance (NED)

Let Σ be an alphabet and $d : \Gamma \to [0,1]$. Note that d may depend on the exact letters, and it could be that e.g. $d(a,b) \neq d(b,d)$ or that $d(\varepsilon,b) \neq d(\varepsilon,d)$. The Levenstein distance [11] (ED) and the Normalized Edit Distance [13] (NED) between w_1 and w_2 (with respect to d) can be defined as follows:

$$\text{ED}_d(w_1,w_2) = \min\{\mathsf{wgt}(\alpha) \ : \ \alpha \text{ is an edit path between } w_1 \text{ and } w_2 \ \}.$$

$$\text{NED}_d(w_1,w_2) = \min\{\mathsf{cost}(\alpha) \ : \ \alpha \text{ is an edit path between } w_1 \text{ and } w_2 \ \}.$$

The ED_d distance looks for an edit path with minimum weight, whereas NED_d looks for an edit path with minimum cost.

▶ **Example 2** (ED_d and NED_d)**.** For instance, consider the words $w_1 = abaad$, $w_2 = baaadc$ over $\Sigma = \{a,b,c,d\}$. Then both $\alpha_1 = \left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]\left[\begin{smallmatrix}b\\b\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}d\\d\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}\right]$ and $\alpha_2 = \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]\left[\begin{smallmatrix}b\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}d\\d\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}\right]$ are edit paths between w_1 and w_2. Consider first the setting of uniform weights, namely $d : \Gamma \to [0,1]$ is defined as $d(\sigma,\sigma) = 0$ and $d(\sigma,\sigma') = 1$ if $\sigma \neq \sigma'$. In this setting, we have that $\mathsf{wgt}(\alpha_1) = 1+0+0+0+1+0+1 = 3$ and $\mathsf{wgt}(\alpha_2) = 1+1+0+0+0+1 = 3$, so using ED α_1 and α_2 are equally good. However $\mathsf{len}(\alpha_1) = 7$ and $\mathsf{len}(\alpha_2) = 6$ so $\mathsf{cost}(\alpha_1) = \frac{3}{7}$ and $\mathsf{cost}(\alpha_2) = \frac{3}{6}$ thus using NED, α_1 is preferable.

Consider now the non-uniform weights $d(\sigma,\sigma') = 0.5$ for every $\sigma \neq \sigma'$, and $d(\sigma,\sigma) = 0$, $d(\sigma,\varepsilon) = d(\varepsilon,\sigma) = 1$ for every $\sigma,\sigma' \in \Sigma$. We get that $\mathsf{wgt}(\alpha_1) = 1+0+0+0+1+0+1 = 3$ and $\mathsf{wgt}(\alpha_2) = 0.5+0.5+0+0+0+1 = 2$ and so $\mathsf{cost}(\alpha_1) = \frac{3}{7}$ and $\mathsf{cost}(\alpha_2) = \frac{2}{6}$, thus α_2 is preferable.

▶ **Definition 3.** *An edit path α is termed* optimal *if $\mathsf{cost}(\alpha) = \text{NED}_d(\mathsf{input}(\alpha), \mathsf{output}(\alpha))$.*

3 A metric weight function is neither necessary nor sufficient

Let $d : \Gamma \to [0,1]$ we are interested in finding a necessary and sufficient condition on d for NED_d to be a metric. A reasonable conjecture is that d is a metric on the space Γ. We show that this is neither a sufficient nor a necessary condition.

We first show that d being a metric is not a sufficient condition for NED_d to be a metric.

▷ Claim 4. There exists d which is a metric while NED_d is not.

Proof. Let $\Sigma = \{a,b\}$ and let $d(\sigma,\sigma) = 0$ for every $\sigma \in \Sigma$. Let $d(a,b) = d(b,a) = 1$, $d(a,\varepsilon) = d(\varepsilon,a) = 0.1$ and $d(b,\varepsilon) = d(\varepsilon,b) = 1$. It is easy to verify that d is a metric.

We show now that NED_d breaks the triangle inequality. Take $w_1 = a$ and $w_3 = b$. Then $\text{NED}_d(a,b) = 0.55$ via the edit path that deletes a and inserts b namely $\left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}\right]$. Its weight is $0.1+1$ and its lengths is 2. Thus it costs $\frac{1.1}{2} = 0.55$.

Consider now going via $w_2 = baaaa$. Then $\alpha_{1,2} = \left[\begin{smallmatrix}a\\b\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}\right]\left[\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}\right]$ is an edit path between w_1 and w_2 and $\alpha_{2,3} = \left[\begin{smallmatrix}b\\b\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]\left[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}\right]$ is an edit path between w_2 and w_3. Notice that $\text{NED}_d(w_1,w_2) \leq \frac{1+4(0.1)}{5}$ and $\text{NED}_d(w_2,w_3) \leq \frac{4(0.1)}{5}$. Thus, $\text{NED}_d(w_1,w_2) + \text{NED}_d(w_2,w_3) \leq \frac{1.8}{5} = 0.36 < 0.55 = \text{NED}_d(w_1,w_3)$. Hence, the triangle inequality for NED_d breaks. ◁

▶ **Corollary 5.** *d being a metric is not a sufficient condition for NED_d to be a metric.*

Next we show that d being a metric is not a necessary condition for NED_d to be a metric: NED_d can be a metric although d breaks the triangle inequality or the symmetry condition.

▷ Claim 6. There exists NED_d which is a metric while d breaks the triangle inequality.

Proof. Let $\Sigma = \{a, b\}$ and let $d(\sigma, \sigma) = 0$ for every $\sigma \in \Sigma$. Let $d(a, b) = d(b, a) = 1$, $d(\varepsilon, a) = d(a, \varepsilon) = 0.4$ and $d(\varepsilon, b) = d(b, \varepsilon) = 0.5$. Then d is not a metric since going from a to b via ε is less costly than going directly (0.9 vs. 1). However, $\textsc{ned}_d$ is a metric. It is easy to see that the first two requirements of a metric hold for $\textsc{ned}_d$. Regarding the triangle inequality, while it seems at first that it breaks in going from a to b (as it does for d) this is not the case. The optimal edit path from a to b is $[{a \atop \varepsilon}][{\varepsilon \atop b}]$ whose cost is $\frac{0.5+0.4}{2} = 0.45$ which is smaller than going via ε which costs $\textsc{ned}_d(a, \epsilon) + \textsc{ned}_d(\epsilon, b) = 0.4 + 0.5$. The proof that $\textsc{ned}_d$ is a metric follows from the fact that it adheres to the sufficient and necessary conditions we provide. We come back to this in Remark 33. ◁

▷ Claim 7. There exists $\textsc{ned}_d$ which is a metric while d breaks symmetry.

Proof. Let $\Sigma = \{a, b\}$ and let $d(\sigma, \sigma) = 0$ for every $\sigma \in \Sigma$. Let $d(a, b) = 1$, $d(b, a) = 0.9$, $d(\varepsilon, a) = d(a, \varepsilon) = 0.4$ and $d(\varepsilon, b) = d(b, \varepsilon) = 0.45$. Then d is not a metric since $d(a, b) \neq d(b, a)$ breaks symmetry. However, $\textsc{ned}_d$ is a metric. Indeed, the symmetry of $\textsc{ned}_d$ does not break since it never uses in an optimal path the operation $[{a \atop b}]$ or $[{b \atop a}]$. For example, consider $w_1 = a$ and $w_2 = b$. Then $\textsc{ned}_d(w_1, w_2) = \mathsf{wgt}([{a \atop \varepsilon}][{\varepsilon \atop b}])/2 = 0.425$ and $\textsc{ned}_d(w_2, w_1) = \mathsf{wgt}([{b \atop \varepsilon}][{\varepsilon \atop a}])/2 = 0.425$ (and the fact that $d(b, a) = 0.9 \neq d(b, a) = 1$ doesn't come in the way). Here as well the proof that $\textsc{ned}_d$ is a metric is deferred to Remark 33. ◁

▶ **Corollary 8.** *d being a metric is not a necessary condition for $\textsc{ned}_d$ to be a metric.*

4 Necessary condition

We turn to extract necessary conditions on d for $\textsc{ned}_d$ to be a metric. We start by showing that, as expected, if $\textsc{ned}_d$ is a metric then d satisfies the first requirement of a metric. The proof relies on the following simple observation.

▷ Claim 9. Let $a, b \in \Sigma$. Then

1. $\textsc{ned}_d(a, \varepsilon) = d(a, \varepsilon)$ and $\textsc{ned}_d(\varepsilon, a) = d(\varepsilon, a)$
2. $\textsc{ned}_d(a, b) = \min\{d(a, b), \frac{1}{2}(d(a, \varepsilon) + d(\varepsilon, b))\}$

Proof. The first item holds since there is a single edit path from ε to $a \in \Sigma$: the edit path $[{\varepsilon \atop a}]$. Hence $\textsc{ned}_d(a, \varepsilon) = \frac{d(a,\varepsilon)}{1}$. The claim on $\textsc{ned}_d(\varepsilon, a)$ is symmetric.

The second item holds since there are exactly two edit paths from a to b: either $[{a \atop b}]$ or $[{a \atop \varepsilon}][{\varepsilon \atop b}]$. Thus $\textsc{ned}(a, b) = \min\{\frac{d(a,b)}{1}, \frac{d(a,\varepsilon)+d(\varepsilon,b)}{2}\}$. ◁

▷ Claim 10. If $\textsc{ned}_d$ is a metric then d must satisfy the identity of indiscernibles condition.

Proof. Let $a, b \in \Sigma$. By Claim 9, $\textsc{ned}_d(a, \varepsilon) = d(a, \varepsilon)$ and $\textsc{ned}_d(\varepsilon, a) = d(\varepsilon, a)$. Thus, $\textsc{ned}_d(a, \varepsilon) > 0$ implies $d(a, \varepsilon) > 0$. By symmetry we get $d(\varepsilon, a) > 0$. Consider now the case where $b = a$. We have $0 = \textsc{ned}_d(a, a) = \min\{d(a, a), \frac{1}{2}(d(a, \varepsilon) + d(\varepsilon, a))\}$. Since we have shown that the second argument is non-zero it follows that $d(a, a) = 0$. Last, consider the case where $b \neq \varepsilon$ and $b \neq a$. Assume towards contradiction the replace between some non-identical letters a and b is zero, then $\textsc{ned}_d(a, b) \leq 0$ via the direct path involving this replace contradicting that $\textsc{ned}_d$ satisfies the first requirement of a metric. ◁

The proof of Claim 6 shows that $\textsc{ned}_d$ can satisfy the condition of triangle inequality although d does not. The reason is that in $\textsc{ned}_d$ there are two options for a direct path between two letters a and b: either a replace or a delete followed by an insert. In the perspective of d a path from a to b that takes a short detour via ε is not direct. Hence the triangle inequality for d can be relaxed as stated below and as we show in the next section this relaxation suffices.

▷ Claim 11 (Relaxed Triangle Inequality). If $\textsc{ned}_d$ satisfies the triangle inequality then d should satisfy

$$d(a,b) + d(b,c) \geq \min\{d(a,c), d(a,\varepsilon) + d(\varepsilon,c)\}$$

for all $b \in \Sigma$ and $a, c \in \Sigma \cup \{\varepsilon\}$.

▶ Remark 12. Note that when $c = \varepsilon$ the requirement is $d(a,b)+d(b,\varepsilon) \geq \min\{d(a,\varepsilon), d(a,\varepsilon)+d(\varepsilon,\varepsilon)\}$ and given $d(\varepsilon,\varepsilon) = 0$ this amounts to

$$d(a,b) + d(b,\varepsilon) \geq d(a,\varepsilon)$$

which says that replacing and deleting cannot cost less than deleting. Similarly, when $a = \varepsilon$ this amounts to $d(\varepsilon,b) + d(b,c) \geq d(b,\varepsilon)$ which says that inserting and replacing cannot cost less than inserting.

Proof of Claim 11. We first consider the case that $c = \varepsilon$. Following Remark 12, assume towards contradiction that there exists $a, b \in \Sigma$ such that $d(a,b) + d(b,\varepsilon) < d(a,\varepsilon)$. Consider $w_1 = a$, $w_2 = b$, $w_3 = \varepsilon$. Let $\alpha_{1,3} = [\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}]$. Notice that it is the only possible edit path from w_1 to w_3 and thus the optimal. Let $\alpha_{1,2} = [\begin{smallmatrix} a \\ b \end{smallmatrix}]$ and $\alpha_{2,3} = [\begin{smallmatrix} b \\ \varepsilon \end{smallmatrix}]$. Hence $cost(\alpha_{1,2}) = d(a,b)$, $cost(\alpha_{2,3}) = d(b,\varepsilon)$ and $cost(\alpha_{1,3}) = d(a,\varepsilon)$. Since $\textsc{ned}_d$ satisfies the triangle inequality then we know that $cost(\alpha_{1,2}) + cost(\alpha_{2,3}) \geq cost(\alpha_{1,3})$ hence $d(a,b) + d(b,\varepsilon) \geq d(a,\varepsilon)$ in contradiction to the assumption. The case where $a = \varepsilon$ is similar.

Assume now neither a nor c is ε and assume towards contradiction that there exists $a, b, c \in \Sigma$ such that $d(a,b) + d(b,c) < \min\{d(a,c), d(a,\varepsilon) + d(\varepsilon,c)\}$. Let $i \in \mathbb{N}$ and consider $w_1 = a^{i+1}$, $w_2 = a^i b$ and $w_3 = a^i c$. Let $\alpha_{1,3} \in \Gamma^*$ be an optimal edit path between w_1 to w_3. Notice that either $\alpha_{1,3} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} a \\ c \end{smallmatrix}]$ or $\alpha_{1,3} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}] [\begin{smallmatrix} \varepsilon \\ c \end{smallmatrix}]$. Consider the two edit paths $\alpha_{1,2} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} a \\ b \end{smallmatrix}]$ and $\alpha_{2,3} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} b \\ c \end{smallmatrix}]$ between w_1 to w_2 and between w_2 to w_3, respectively.

- Case 1: $\alpha_{1,3} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} a \\ c \end{smallmatrix}]$.
  Then $\mathsf{wgt}(\alpha_{1,3}) = d(a,c)$, $\mathsf{len}(\alpha_{1,3}) = i+1$ and $\mathsf{cost}(\alpha_{1,3}) = \frac{d(a,c)}{i+1}$. In order for $\textsc{ned}_d$ to satisfy the triangle inequality $\mathsf{cost}(\alpha_{1,3}) \leq \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{2,3})$ must hold. Thus

$$\frac{d(a,c)}{i+1} \leq \frac{d(a,b)}{i+1} + \frac{d(b,c)}{i+1}$$

$$d(a,c) \leq d(a,b) + d(b,c)$$

  in contradiction to the assumption.
- Case 2: $\alpha_{1,3} = ([\begin{smallmatrix} a \\ a \end{smallmatrix}])^i [\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}] [\begin{smallmatrix} \varepsilon \\ c \end{smallmatrix}]$.
  Then $\mathsf{wgt}(\alpha_{1,3}) = d(a,\varepsilon) + d(\varepsilon,c)$, $\mathsf{len}(\alpha_{1,3}) = i+2$ and $\mathsf{cost}(\alpha_{1,3}) = \frac{d(a,\varepsilon)+d(\varepsilon,c)}{i+2}$. In order for $\textsc{ned}_d$ to satisfy the triangle inequality $\mathsf{cost}(\alpha_{1,3}) \leq \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{2,3})$ must hold. Thus,

$$\frac{d(a,\varepsilon) + d(\varepsilon,c)}{i+2} \leq \frac{d(a,b)}{i+1} + \frac{d(b,c)}{i+1}$$

$$\frac{(i+1)(d(a,\varepsilon) + d(\varepsilon,c))}{i+2} \leq d(a,b) + d(b,c)$$

By taking i to infinity we get that

$$\lim_{i\to\infty}\frac{(i+1)(d(a,\varepsilon)+d(\varepsilon,c))}{i+2}=d(a,\varepsilon)+d(\varepsilon,c)\le d(a,b)+d(b,c)$$

in contradiction to the assumption.
Hence either way we get that for NED_d to satisfy the triangle inequality then d should satisfy

$$d(a,b)+d(b,c)\ge\min\{d(a,c),d(a,\varepsilon)+d(\varepsilon,c)\}$$

for every $b\in\Sigma$ and $a,c\in\Sigma\cup\{\varepsilon\}$. $\triangleleft$

In Claim 4 we have shown that NED_d fails to be a metric although d is. Intuitively, the reason is that going through more and more insert and delete operations can decrease the overall cost. In the following, we will show that requiring insert and delete operations to be at least half of the costliest replace operation prevents this.

$\triangleright$ Claim 13 (At least half). If NED_d is a metric and $m=\sup\{\text{NED}_d(w_1,w_2) : w_1,w_2\in\Sigma^*\}$. Then d should satisfy the requirement $d(\varepsilon,b)=d(b,\varepsilon)\ge\frac{m}{2}$ for every $b\in\Sigma$.

Proof. First note that if $m = \sup\{\text{NED}_d(w_1,w_2) : w_1,w_2 \in \Sigma^*\}$ then $m = \sup\{\text{NED}_d(\sigma_1,\sigma_2) : \sigma_1,\sigma_2\in\Sigma\}$. Indeed the way to obtain the maximum cost of an edit path is using the edit operation with maximal cost, and using more than one such operation will not increase the total cost.

Suppose inserting/deleting some letter b costs c for some $c<\frac{m}{2}$. Consider the words $w_1=\sigma_1$, $w_3=\sigma_3$ and assume that w_1 and w_3 are such that $\text{NED}_d(w_1,w_3)=m$. Note that there are only two possible edit paths that transform w_1 to w_3. That is, either $\alpha_{1,3}=[\begin{smallmatrix}\sigma_1\\\sigma_3\end{smallmatrix}]$ or $\alpha_{1,3}=[\begin{smallmatrix}\sigma_1\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\\sigma_3\end{smallmatrix}]$. Hence $\text{NED}_d(w_1,w_3)=m$ implies $m=\min\{d(\sigma_1,\sigma_3),\frac{1}{2}(d(\sigma_1,\varepsilon)+d(\varepsilon,\sigma_3))\}$. This in turn implies that $d(\sigma_1,\sigma_3)\ge m$ and $d(\sigma_1,\varepsilon)+d(\varepsilon,\sigma_3)\ge 2m$.

Consider now the word $w_2=\sigma_3\cdot b^k$ for some $k\in\mathbb{N}$ where $k\ge 1$. Then we can transform w_1 to w_2 using the edit path $\alpha_{1,2}=[\begin{smallmatrix}\sigma_1\\\sigma_3\end{smallmatrix}]\cdot([\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}])^k$ or $\alpha_{1,2}=[\begin{smallmatrix}\sigma_1\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\\sigma_3\end{smallmatrix}]\cdot([\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}])^k$. To transform w_2 to w_3 we can use the edit path $\alpha_{2,3}=[\begin{smallmatrix}\sigma_3\\\sigma_3\end{smallmatrix}]([\begin{smallmatrix}b\\\varepsilon\end{smallmatrix}])^k$. Then the sum of the edit paths is one of the following:

1. In case of $\alpha_{1,2}=[\begin{smallmatrix}\sigma_1\\\sigma_3\end{smallmatrix}]\cdot([\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}])^k$:

$$\mathsf{cost}(\alpha_{1,2})+\mathsf{cost}(\alpha_{2,3})=\frac{d(\sigma_1,\sigma_3)+k\cdot d(\varepsilon,b)}{1+k}+\frac{k\cdot d(b,\varepsilon)}{1+k}\ge\frac{m+k\cdot c}{1+k}+\frac{k\cdot c}{1+k}=\frac{2k\cdot c+m}{1+k}$$

Since $\text{NED}_d(w_1,w_3)=m$ and since NED_d is a metric then by the triangle inequality we require $\mathsf{cost}(\alpha_{1,2})+\mathsf{cost}(\alpha_{2,3})\ge\frac{2k\cdot c+m}{1+k}\ge m=\text{NED}_d(w_1,w_3)$. Which entails that $2k\cdot c+m\ge m+mk$ and hence $c\ge\frac{m}{2}$.

2. In case of $\alpha_{1,2}=[\begin{smallmatrix}\sigma_1\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\\sigma_3\end{smallmatrix}]\cdot([\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}])^k$:

$$\mathsf{cost}(\alpha_{1,2})+\mathsf{cost}(\alpha_{2,3})=\frac{d(\sigma_1,\varepsilon)+d(\varepsilon,\sigma_3)+k\cdot d(\varepsilon,b)}{2+k}+\frac{k\cdot d(b,\varepsilon)}{1+k}\ge\frac{2m+k\cdot c}{2+k}+\frac{k\cdot c}{1+k}$$

Again, since $\text{NED}_d(w_1,w_3)=m$ and since NED_d is a metric by the triangle inequality we require $\mathsf{cost}(\alpha_{1,2})+\mathsf{cost}(\alpha_{2,3})\ge\frac{2m+k\cdot c}{2+k}+\frac{k\cdot c}{1+k}\ge m=\text{NED}_d(w_1,w_3)$. Which entails that

$$(2m+kc)(1+k)+kc(2+k)\ge m(2+k)(1+k)$$

$$2m+2mk+kc+k^2c+2kc+k^2c\ge 2m+3mk+mk^2$$

$$c(2k^2+3k)\ge mk+mk^2$$

$$c\ge\frac{mk^2+mk}{2k^2+3k}$$

Taking k to infinity we get that

$$\lim_{k\to\infty} \frac{mk^2+mk}{2k^2+3k} = \frac{m}{2}$$

Hence either way we get $c \geq \frac{m}{2}$. ◁

▶ **Definition 14.** *We say that an edit operation $\gamma \in \Gamma$ is* essential *if there exists $\alpha \in \Gamma^*$ such that α is an optimal edit path that uses γ. Otherwise γ is called* inessential.

For example in the proof of Claim 7 we can see that $[\begin{smallmatrix} a \\ b \end{smallmatrix}]$ and $[\begin{smallmatrix} b \\ a \end{smallmatrix}]$ are inessential since transforming a to b via ε is always preferable, while $[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}]$ is essential. We show that we can ignore inessential edit operations without changing the result.

▷ Claim 15 (Essentials suffice). Let Γ' be the restriction of Γ to only essential edit operations and let $d' : \Gamma' \to [0,1]$ be the restriction of d to Γ'. Then $\text{NED}_d(w_1, w_2) = \text{NED}_{d'}(w_1, w_2)$ for every $w_1, w_2 \in \Sigma^*$.

Proof. Let $w_1, w_2 \in \Sigma^*$ we will show that $\text{NED}_d(w_1, w_2) = \text{NED}_{d'}(w_1, w_2)$. From Definition 3 we know that an edit path α for which $\text{NED}_d(w_1, w_2) = \mathsf{cost}(\alpha)$ is an optimal edit path hence every edit operation in it is essential by Definition 14. Thus, all the edit operations in α exist in Γ'. It follows that $\text{NED}_{d'}(w_1, w_2) \leq \mathsf{cost}(\alpha)$ and since Γ' does not have additional edit operations compared to Γ (and they agree on the costs of the mutual ones) the cost of $\text{NED}_{d'}(w_1, w_2)$ cannot be bigger than $\mathsf{cost}(\alpha)$ or else α is not optimal in contradiction. Hence $\text{NED}_{d'}(w_1, w_2) = \text{NED}_d(w_1, w_2)$. ◁

It follows from Claim 15 that we can assume without loss of generality that there are no inessential operations in d.

▶ Remark 16. Note that $[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}]$ and $[\begin{smallmatrix} \varepsilon \\ a \end{smallmatrix}]$ are essential for every $a \in \Sigma$. This is since there is only one edit path from a to ε (and similarly from ε to a) and it involves these operations. Moreover, by Claim 9 if NED_d is a metric then $d(a, \varepsilon) = d(\varepsilon, a)$.

▷ Claim 17 (A bound on the cost of an essential replace). Let $a, b \in \Sigma$. If $[\begin{smallmatrix} a \\ b \end{smallmatrix}]$ is an essential edit operation then there exists $i \in \mathbb{N}$ such that $d(a,b) < (d(a,\varepsilon) + d(\varepsilon, b))(1 - 1/i)$.

Proof. Since $[\begin{smallmatrix} a \\ b \end{smallmatrix}]$ is essential there exists an optimal edit path α that uses it. Consider the shortest such optimal path. Note that $\mathsf{cost}(\alpha) = \frac{\mathsf{wgt}(\alpha) - d(a,b) + d(a,b)}{\mathsf{len}(\alpha)}$. Consider a new edit path α' that does the same edit operations as α apart from $[\begin{smallmatrix} a \\ b \end{smallmatrix}]$ which it will replace by $[\begin{smallmatrix} a \\ \varepsilon \end{smallmatrix}][\begin{smallmatrix} \varepsilon \\ b \end{smallmatrix}]$. Note that $\mathsf{input}(\alpha) = \mathsf{input}(\alpha')$ and $\mathsf{output}(\alpha) = \mathsf{output}(\alpha')$. Since α is optimal we know $\mathsf{cost}(\alpha) \leq \mathsf{cost}(\alpha')$. Moreover notice that $\mathsf{cost}(\alpha') = \frac{\mathsf{wgt}(\alpha) - d(a,b) + d(a,\varepsilon) + d(\varepsilon,b)}{\mathsf{len}(\alpha)+1}$.

Hence

$$\mathsf{cost}(\alpha) = \frac{\mathsf{wgt}(\alpha) - d(a,b) + d(a,b)}{\mathsf{len}(\alpha)} \leq \frac{\mathsf{wgt}(\alpha) - d(a,b) + d(a,\varepsilon) + d(\varepsilon,b)}{\mathsf{len}(\alpha) + 1} = \mathsf{cost}(\alpha')$$

Thus

$$\begin{gathered}(\mathsf{len}(\alpha)+1)(\mathsf{wgt}(\alpha) - d(a,b)) + (\mathsf{len}(\alpha)+1) \cdot d(a,b) \leq \\ \mathsf{len}(\alpha)(\mathsf{wgt}(\alpha) - d(a,b)) + \mathsf{len}(\alpha)(d(a,\varepsilon) + d(\varepsilon,b))\end{gathered}$$

implying

$$(\mathsf{wgt}(\alpha) - d(a,b)) + (\mathsf{len}(\alpha)+1) \cdot d(a,b) \leq \mathsf{len}(\alpha)(d(a,\varepsilon) + d(\varepsilon,b))$$

Therefore

$$\begin{aligned} d(a,b) &\leq \frac{\mathsf{len}(\alpha)(d(a,\varepsilon)+d(\varepsilon,b)) - (\mathsf{wgt}(\alpha) - d(a,b))}{\mathsf{len}(\alpha)+1} \\ &\leq \frac{\mathsf{len}(\alpha)(d(a,\varepsilon)+d(\varepsilon,b))}{\mathsf{len}(\alpha)+1} = \frac{(\mathsf{len}(\alpha)+1-1)(d(a,\varepsilon)+d(\varepsilon,b))}{\mathsf{len}(\alpha)+1} \\ &= d(a,\varepsilon)+d(\varepsilon,b) - \frac{d(a,\varepsilon)+d(\varepsilon,b)}{\mathsf{len}(\alpha)+1} = (d(a,\varepsilon)+d(\varepsilon,b))\left(1-\frac{1}{\mathsf{len}(\alpha)+1}\right) \end{aligned}$$

hence the claim holds for $i > \mathsf{len}(\alpha)+1$. ◁

In Claim 7 we have shown that symmetry of d is not a necessary condition for NED_d to be a metric. In Claim 9 we showed that insert and delete operations are essential and must be symmetric. The following claim clarifies that if we restrict d to the essential operations then symmetry must hold.

▷ Claim 18 (Symmetry of Essentials). Let $a,b \in \Sigma$, if $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ is an essential edit operation and NED_d is a metric then $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ is also essential and $d(a,b) = d(b,a)$.

Proof. Assume that $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ is an essential edit operation and NED_d is a metric. Assume towards contradiction that $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ is not essential. From Claim 17 we know that there exists $i \in \mathbb{N}$ such that $d(a,b) < (d(a,\varepsilon)+d(\varepsilon,b))(1-1/i)$. Consider $w_1 = a^{i+1}$, $w_2 = a^i b$, let $\alpha = ([\begin{smallmatrix}a\\a\end{smallmatrix}])^i [\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $\alpha' = ([\begin{smallmatrix}a\\a\end{smallmatrix}])^i [\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}]$. Notice that $\mathsf{cost}(\alpha) = \frac{d(a,b)}{i+1}$ and $\mathsf{cost}(\alpha') = \frac{d(a,\varepsilon)+d(\varepsilon,b)}{i+2}$. Since the only edit path that can cost less than α is α' we can check which of them is optimal. We argue that $\mathsf{cost}(\alpha) < \mathsf{cost}(\alpha')$. If this is the case then

$$\frac{d(a,b)}{i+1} < \frac{d(a,\varepsilon)+d(\varepsilon,b)}{i+2}$$

hence

$$d(a,b) < \frac{(i+1)\cdot(d(a,\varepsilon)+d(\varepsilon,b))}{i+2}$$

and so

$$d(a,b) < d(a,\varepsilon)+d(\varepsilon,b) - \frac{d(a,\varepsilon)+d(\varepsilon,b)}{i+2} = (d(a,\varepsilon)+d(\varepsilon,b))\cdot(1-\frac{1}{i+2})$$

And this holds since $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ is essential and so $d(a,b) < (d(a,\varepsilon)+d(\varepsilon,b))(1-\frac{1}{i})$, by Claim 17. Hence α is optimal which means that $\text{NED}_d(w_1,w_2) = \mathsf{cost}(\alpha)$ and since NED_d is a metric we know that $\mathsf{cost}(\alpha) = \text{NED}_d(w_2,w_1)$ as well. Consider now the optional optimal edit paths from w_2 to w_1. Let $\beta = ([\begin{smallmatrix}a\\a\end{smallmatrix}])^i [\begin{smallmatrix}b\\a\end{smallmatrix}]$ and $\beta' = ([\begin{smallmatrix}a\\a\end{smallmatrix}])^i [\begin{smallmatrix}b\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\a\end{smallmatrix}]$. We know that NED_d is a metric hence following Claim 9 we know that $\mathsf{cost}(\beta') = \mathsf{cost}(\alpha')$ hence we know that $\mathsf{cost}(\alpha) = \text{NED}_d(w_2,w_1) < \mathsf{cost}(\beta')$. Thus $\mathsf{cost}(\alpha) = \text{NED}_d(w_2,w_1) = \mathsf{cost}(\beta)$, implying $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ is essential too and moreover $d(a,b) = d(b,a)$. ◁

From Remark 16 we know that the only operations that can be inessential are $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ where $a,b \neq \varepsilon$. The following claim provides means to check if $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ is essential or not.

▷ Claim 19 (Essentialness Check). For every $a,b \in \Sigma$ we have $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ is inessential iff $d(a,b) \geq d(a,\varepsilon)+d(\varepsilon,b)$.

Proof. $\Longrightarrow$ From Claim 17 we know that if $[\substack{a\\b}]$ is essential then there exists $i \in \mathbb{N}$ such that $d(a,b) < (d(a,\varepsilon)+d(\varepsilon,b))(1-1/i)$. Thus if such i does not exist (which means that $[\substack{a\\b}]$ is inessential), then for every i we have

$$d(a,b) \geq (d(a,\varepsilon)+d(\varepsilon,b))(1-1/i)$$

hence $d(a,b) \geq d(a,\varepsilon)+d(\varepsilon,b)$.

$\Longleftarrow$ Assume towards contradiction that $d(a,b) \geq d(a,\varepsilon)+d(\varepsilon,b)$ and $[\substack{a\\b}]$ is essential. Let $n = d(a,b)$ and $m = d(a,\varepsilon)+d(\varepsilon,b)$. From the definition of essential, we know that there exists $w_1, w_2 \in \Sigma^*$ and $\alpha \in \Gamma^*$ such that α is an optimal path from w_1 to w_2 that uses $[\substack{a\\b}]$. Let k denote the number of occurrences of $[\substack{a\\b}]$ in α. Let $\mathsf{wgt}(\alpha) = p+n\cdot k$, and $\mathsf{len}(\alpha) = \ell+k$. Now notice that if we replace every occurrence of $[\substack{a\\b}]$ with $[\substack{a\\\varepsilon}][\substack{\varepsilon\\b}]$ we will get path α' from w_1 to w_2 where $\mathsf{cost}(\alpha') = \frac{p+m\cdot k}{\ell+2\cdot k} < \frac{p+n\cdot k}{\ell+k}$ in contradiction to α being an optimal path. $\lhd$

We are now ready to state the necessary condition on d for NED_d to be a metric.

▶ **Corollary 20** (Necessary Condition). *Let $a, c \in \Sigma \cup \{\varepsilon\}$ and $b \in \Sigma$. Let $m = \sup\{\text{NED}_d(w_1,w_2) \colon w_1, w_2 \in \Sigma^*\}$. A necessary condition for NED_d to be a metric is that d satisfies the following properties after removing inessential edit operations.*

1. $d(a,c) = 0$ *iff* $a = c$
2. $d(a,c) = d(c,a)$
3. $d(a,b)+d(b,c) \geq \min\{d(a,c), d(a,\varepsilon)+d(\varepsilon,c)\}$
4. $d(\varepsilon,b) = d(b,\varepsilon) \geq \frac{m}{2}$

Indeed, the first requirement is necessary by Claim 10, the second requirement by Claim 18, the third by Claim 11, and the forth by Claim 13.[3]

▶ Remark 21. Let $m = \sup\{\text{NED}_d(w_1,w_2) : w_1, w_2 \in \Sigma^*\}$. Note that we can assume without loss of generality that $m = 1$. If this is not the case then we define $d'(\sigma_1,\sigma_2) = \frac{1}{m}d(\sigma_1,\sigma_2)$. Then d' would satisfy that $\sup\{\text{NED}_{d'}(w_1,w_2) : w_1, w_2 \in \Sigma^*\} = 1$. In this case, it may be that there are σ_1, σ_2 for which $d'(\sigma_1,\sigma_2)$ are greater than 1 but such an edit operation is inessential.

▶ **Definition 22** (Fine Weight Function, Fine Metric). *We call a function $d : \Gamma \to [0,1]$ satisfying the conditions of Corollary 20* fine*. Note that if d is a metric it satisfies the first three requirements. If it also satisfied the fourth requirement, we call it a* fine metric*.*

In the next section, we show that if $d : \Gamma \to [0,1]$ is fine then NED_d is a metric. That is, d being fine is a sufficient and necessary condition for NED_d to be a metric.

5 Sufficient Condition

We turn to show that if d is fine then NED_d is a metric. That is, that the necessary condition provided in the previous section is also a sufficient.

▷ Claim 23. If d is fine then NED_d satisfies the identity of indiscernibles requirement.

[3] For GED, a sufficient condition was given in [12]. We conjecture that it is not a necessary condition, and GED may be a metric also when deletion of different letters costs differently as in d of Claim 7.

Proof. Assume d is fine, and let $w_1, w_2 \in \Sigma^*$.

1. Case $w_1 = w_2$. We show that $\text{NED}_d(w_1, w_2) = 0$. Since d is fine we know that $d(a, a) = 0$ for every $a \in \Sigma$. Thus, we can construct an edit path α that applies *no-op* to each letter which leads to that $\mathsf{wgt}(\alpha) = 0$. Hence $\text{NED}_d(w_1, w_2) = 0$.
2. Case $w_1 \neq w_2$. Let $\alpha \in \Gamma^*$ be an optimal edit path that transforms w_1 to w_2. Notice that α needs at least one edit operation, denote it γ, that is not *no-op*. Since d is fine we know that $\mathsf{wgt}(\gamma) > 0$. Hence $\text{NED}_d(w_1, w_2) = \mathsf{cost}(\alpha) = \frac{\mathsf{wgt}(\alpha)}{\mathsf{len}(\alpha)} \geq \frac{\mathsf{wgt}(\gamma)}{\mathsf{len}(\alpha)} > 0$. ◁

▷ Claim 24. If d is fine then NED_d satisfies the symmetry requirement.

Proof. Assume that d is fine and assume towards contradiction that there exists $w_1, w_2 \in \Sigma^*$ such that $\text{NED}_d(w_1, w_2) \neq \text{NED}_d(w_2, w_1)$. Assume w.l.o.g. that $\text{NED}_d(w_1, w_2) < \text{NED}_d(w_2, w_1)$. Let $\alpha_{1,2}$ be an optimal path that transforms w_1 to w_2. Let $\gamma \in \alpha_{1,2}$ such that $\gamma = [\begin{smallmatrix} a \\ b \end{smallmatrix}]$ where $a, b \in \Sigma \cup \{\varepsilon\}$ and either $a \neq \varepsilon$ or $b \neq \varepsilon$. From Definition 14 and Claim 18 we know that γ is essential and so is $[\begin{smallmatrix} b \\ a \end{smallmatrix}]$. Since d is fine we know that $d(a, b) = d(b, a)$. We refer to $[\begin{smallmatrix} b \\ a \end{smallmatrix}]$ as the opposite edit operation of $[\begin{smallmatrix} a \\ b \end{smallmatrix}]$. Note that if we replace every edit operation in $\alpha_{1,2}$ with its opposite edit operation, we will receive a new edit path $\alpha_{2,1}$ that transforms w_2 to w_1 and $\mathsf{cost}(\alpha_{2,1}) = \mathsf{cost}(\alpha_{1,2}) < \text{NED}_d(w_2, w_1)$, contradicting the definition of NED. ◁

We proceed to show that the triangle inequality also holds.

The idea of the proof of [7] that NED satisfies the triangle inequality for the uniform case is to take two edit paths $\alpha_{1,2}$ and $\alpha_{2,3}$ from words w_1 to w_2 and from w_2 to w_3 and extract from them an edit path $\alpha_{1,3}$ from w_1 to w_3 that costs at most their sum. We follow that idea but generalize and simplify the proof.

The heart of the simplification lies in finding a way to align the two edit paths so that their composition to a new edit path from w_1 to w_3 is seamless, and we can easily prove that it costs less than the sum.

We proceed by showing how to compose the two paths. The composition uses as an intermediate step a pair of extended edit paths $\alpha'_{1,2}, \alpha'_{2,3}$ that align the give edit paths $\alpha_{1,2}$ and $\alpha_{2,3}$, with respect to one another.[4]

▶ **Definition 25** (Alignment of edit paths). *Let $\alpha_{1,2}$ and $\alpha_{2,3}$ be such that $\mathsf{output}(\alpha_{1,2}) = \mathsf{input}(\alpha_{2,3})$. We say that $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ is the* alignment *of $\alpha_{1,2}$ and $\alpha_{2,3}$ if $\alpha'_{1,2}$ and $\alpha'_{2,3}$ are the shortest extended edit paths satisfying that*

- *$\alpha'_{1,2}$ is obtained from $\alpha_{1,2}$ by inserting some $[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}]$ letters,*
- *$\alpha'_{2,3}$ is obtained from $\alpha_{2,3}$ by inserting some $[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}]$ letters,*
- *and $\pi_2(\alpha'_{1,2}) = \pi_1(\alpha'_{2,3})$.*

The first requirement guarantees that the input and output of $\alpha'_{1,2}$ is the same as those of $\alpha_{1,2}$, and the second requirement gives the analogous guarantees regarding $\alpha'_{2,3}$ and $\alpha_{2,3}$. The third requirement strengthens the connection between $\mathsf{output}(\alpha_{1,2})$ and $\mathsf{input}(\alpha_{2,3})$ and demands that they agree not only on the letters of the interim word w_2, but also on the occurrences of ε. This in particular requires $\alpha'_{1,2}$ and $\alpha'_{2,3}$ to be of the same length. Using the $[\begin{smallmatrix} \sigma \\ \sigma' \end{smallmatrix}]$ notations, if we write $\alpha'_{1,2}$ and $\alpha'_{2,3}$ one above the other then the second and third lines are the same.

[4] Recall that an extended edit path is a string over $\hat{\Gamma}$, namely it may use $[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}]$ on top of the usual edit operations.

▶ **Example 26.** Let $w_1 = a_1a_2a_3$, $w_2 = b_1b_2$ and $w_3 = c_1c_2c_3c_4$. Then $\alpha_{1,2} = [\begin{smallmatrix}a_1\\b_1\end{smallmatrix}][\begin{smallmatrix}a_2\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}a_3\\b_2\end{smallmatrix}]$ is an edit path between w_1 and w_2 and $\alpha_{2,3} = [\begin{smallmatrix}\varepsilon\\c_1\end{smallmatrix}][\begin{smallmatrix}b_1\\c_2\end{smallmatrix}][\begin{smallmatrix}b_2\\c_3\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c_4\end{smallmatrix}]$ is an edit path between w_2 and w_3. Using the $\mapsto$ notation, we can write these as $a_1a_2a_3 \mapsto b_1\_b_2$ and $\_b_1b_2\_ \mapsto c_1c_2c_3c_4$.

Let $\alpha'_{1,2} = [\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}a_1\\b_1\end{smallmatrix}][\begin{smallmatrix}a_2\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}a_3\\b_2\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ and $\alpha'_{2,3} = [\begin{smallmatrix}\varepsilon\\c_1\end{smallmatrix}][\begin{smallmatrix}b_1\\c_2\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}b_2\\c_3\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c_4\end{smallmatrix}]$. Then $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ is their *alignment*. Using the $\mapsto$ notations these are $\_a_1a_2a_3\_ \mapsto \_b_1\_b_2\_$ and $\_b_1\_b_2\_ \mapsto c_1c_2\_c_3c_4$, on which it is perhaps easier to see that the output of $\alpha'_{1,2}$ and the input of $\alpha'_{2,3}$ agree also on ε positions.

Note that if $\pi_2(\alpha_{1,2})$ contains i occurrences of ε and $\pi_1(\alpha_{2,3})$ contains j occurrences of ε then there exist $\alpha'_{1,2}$ and $\alpha'_{2,3}$ of length at most $w_2 + i + j$ such that $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ is their alignment. Moreover, the alignment can be constructed iteratively by following $\pi_2(\alpha_{1,2})$ and $\pi_1(\alpha_{2,3})$ and if the current index (of the considered projections) is not the same, inserting $(\varepsilon, \varepsilon)$ to either $\alpha_{1,2}$ or $\alpha_{2,3}$ depending on which has advanced less (in terms of letters of w_2).

We are now ready to define the composition of $\alpha_{1,2}$ and $\alpha_{2,3}$.

▶ **Definition 27** (Compose). *Let $\alpha_{1,2}$ and $\alpha_{2,3}$ be such that $\mathsf{output}(\alpha_{1,2}) = \mathsf{input}(\alpha_{2,3})$ and let $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ be their alignment. Assume $\alpha'_{1,2} = [\begin{smallmatrix}a_1\\b_1\end{smallmatrix}][\begin{smallmatrix}a_2\\b_2\end{smallmatrix}]\dots[\begin{smallmatrix}a_k\\b_k\end{smallmatrix}]$ and $\alpha'_{2,3} = [\begin{smallmatrix}b_1\\c_1\end{smallmatrix}][\begin{smallmatrix}b_2\\c_2\end{smallmatrix}]\dots[\begin{smallmatrix}b_k\\c_k\end{smallmatrix}]$. Let $\alpha''_{1,3} = [\begin{smallmatrix}a_1\\c_1\end{smallmatrix}][\begin{smallmatrix}a_2\\c_2\end{smallmatrix}]\dots[\begin{smallmatrix}a_k\\c_k\end{smallmatrix}]$. Let $\alpha'_{1,3}$ be the edit path obtained from $\alpha''_{1,3}$ by replacing $[\begin{smallmatrix}a\\c\end{smallmatrix}]$ with $[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}]$ for every a, c for which $d(a, c) \geq d(a, \varepsilon) + d(\varepsilon, c)$. Finally, let $\alpha_{1,3}$ be the edit path obtained from $\alpha'_{1,3}$ by removing the $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ letters.*

▷ Claim 28. Let $\alpha_{1,2}$ and $\alpha_{2,3}$ be such that $\mathsf{output}(\alpha_{1,2}) = \mathsf{input}(\alpha_{2,3})$. If $\alpha_{1,3}$ is the result of composing $\alpha_{1,2}$ and $\alpha_{2,3}$ as per Definition 27 then $\alpha_{1,3}$ is an edit path between $\mathsf{input}(\alpha_{1,2})$ and $\mathsf{output}(\alpha_{2,3})$.

Proof. Let $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ be the alignment of $\alpha_{1,2}$ and $\alpha_{2,3}$. Then α and α' agree on their input and output for $\alpha \in \{\alpha_{1,2}, \alpha_{2,3}\}$ since they only differ in $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ letters. Let $\alpha''_{1,3}$ and $\alpha'_{1,3}$ be as described in Definition 27. It is easy to see that $\mathsf{input}(\alpha''_{1,3}) = \mathsf{input}(\alpha'_{1,2})$ and $\mathsf{output}(\alpha''_{1,3}) = \mathsf{output}(\alpha'_{2,3})$ since $\mathsf{input}([\begin{smallmatrix}a\\c\end{smallmatrix}]) = a = \mathsf{input}([\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}])$ and similarly $\mathsf{output}([\begin{smallmatrix}a\\c\end{smallmatrix}]) = c = \mathsf{output}([\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}])$. The claim follows by transitivity of equality. ◁

The following lemma is the heart of the proof that the triangle inequality holds for NED_d given d is fine.

▶ **Lemma 29.** *Assume $d : \Gamma \to [0, 1]$ is fine. Let $\alpha_{1,2}$ and $\alpha_{2,3}$ be such that $\mathsf{output}(\alpha_{1,2}) = \mathsf{input}(\alpha_{2,3})$. Let $\alpha''_{1,3}$, $\alpha'_{1,3}$ and $\alpha_{1,3}$ be as described in Definition 27. Let n be the number of occurrences of $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ in $\alpha'_{1,3}$. Then*
1. $\mathsf{len}(\alpha_{1,3}) \geq \max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\} - n$
2. $\mathsf{wgt}(\alpha_{1,3}) \leq \mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - n$
3. $\mathsf{cost}(\alpha_{1,3}) \leq \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{1,3})$

The proof relies on the following fact.

▶ **Fact 30.** *Let $d, e, n \in \mathbb{N}$ such that $e \leq d$. Then $\frac{e-n}{d-n} \leq \frac{e}{d}$*

We can now prove Lemma 29.

Proof of Lemma 29. Let $\langle\alpha'_{1,2}, \alpha'_{2,3}\rangle$ be the alignment of $\alpha_{1,2}$ and $\alpha_{2,3}$.
1. By the construction of the aligned edit paths there is no index i such that both $\alpha'_{1,2}[i] = [\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ and $\alpha'_{2,3}[i] = [\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$. Thus for every index i of $\alpha''_{1,3}$ either $\alpha'_{1,2}[i]$ is an element of $\alpha_{1,2}$ or $\alpha'_{2,3}[i]$ is an element of $\alpha_{2,3}$ (or both are). It follows that

$$\mathsf{len}(\alpha''_{1,3}) \geq \max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\}$$

Since $\alpha_{1,3}$ is obtained from $\alpha'_{1,3}$ by removing the occurrences of $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ and there are n such we get that

$$\mathsf{len}(\alpha_{1,3}) = \mathsf{len}(\alpha'_{1,3}) - n$$

Because $\mathsf{len}(\alpha'_{1,3}) \geq \mathsf{len}(\alpha''_{1,3})$ we get overall that

$$\mathsf{len}(\alpha_{1,3}) = \mathsf{len}(\alpha'_{1,3}) - n \geq \mathsf{len}(\alpha''_{1,3}) - n \geq \max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\} - n$$

as required.

2. First note that $\mathsf{wgt}(\alpha) = \mathsf{wgt}(\alpha')$ for all $\alpha \in \{\alpha_{1,2}, \alpha_{2,3}, \alpha_{1,3}\}$ since α and α' differ only by elements of the form $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ and since d is fine, by the first requirement, $d(\varepsilon,\varepsilon) = 0$.
Second, we claim that $\mathsf{wgt}(\alpha'_{1,3}) \leq \mathsf{wgt}(\alpha'_{1,2}) + \mathsf{wgt}(\alpha'_{2,3})$. This holds since for every element $[\begin{smallmatrix}a\\c\end{smallmatrix}]$ of $\alpha''_{1,3}$ there exists elements $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\c\end{smallmatrix}]$ in $\alpha'_{1,2}$ and $\alpha'_{2,3}$ respectively, where $a, b, c \in \Sigma \cup \{\varepsilon\}$. Hence for every respective element $[\begin{smallmatrix}a\\c\end{smallmatrix}]$ or respective two elements of $[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}]$ of $\alpha'_{1,3}$ there exists elements $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\c\end{smallmatrix}]$ in $\alpha'_{1,2}$ and $\alpha'_{2,3}$ respectively. To see how the corresponding weights relate we split into cases.

a. If both $a \neq \varepsilon$ and $c \neq \varepsilon$ then by the third requirement of being fine $\min\{d(a,c), d(a,\varepsilon)+d(\varepsilon,c)\} \leq d(a,b) + d(b,c)$ and according to the minimum $[\begin{smallmatrix}a\\c\end{smallmatrix}]$ or $[\begin{smallmatrix}a\\\varepsilon\end{smallmatrix}][\begin{smallmatrix}\varepsilon\\c\end{smallmatrix}]$ occurs in $\alpha'_{1,3}$.

b. If $a \neq \varepsilon$ and $c = \varepsilon$ then by the third requirement of being fine and Remark 12 we have $d(a,b) + d(b,\varepsilon) \geq d(a,\varepsilon)$.

c. If $a = \varepsilon$ and $c \neq \varepsilon$ then by the third requirement of being fine and Remark 12 we have $d(\varepsilon,b) + d(b,c) \geq d(\varepsilon,c)$.

Thus we get

$$\mathsf{wgt}(\alpha'_{1,3}) \leq \mathsf{wgt}(\alpha'_{1,2}) + \mathsf{wgt}(\alpha'_{2,3}) = \mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3})$$

Last, we note that each occurrence of $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ in $\alpha'_{1,3}$ corresponds to an occurrence of $[\begin{smallmatrix}\varepsilon\\b\end{smallmatrix}]$ in $\alpha_{1,2}$ and $[\begin{smallmatrix}b\\\varepsilon\end{smallmatrix}]$ in $\alpha_{2,3}$ for some $b \in \Sigma$. Let $b_1, b_2, \ldots, b_n$ be the respective letters in $\alpha_{1,2}$ or $\alpha_{2,3}$. The weight of $[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]$ in $\alpha'_{1,3}$ is 0 whereas the original components had some non-zero weight. Hence

$$\mathsf{wgt}(\alpha'_{1,3}) \leq \mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - \sum_{i=1}^{n}(\mathsf{wgt}([\begin{smallmatrix}\varepsilon\\b_i\end{smallmatrix}]) + \mathsf{wgt}([\begin{smallmatrix}b_i\\\varepsilon\end{smallmatrix}]))$$

From the forth requirement of being fine we know $d(\varepsilon,b) = d(b,\varepsilon) \geq \frac{1}{2}$. Thus $\sum_{i=1}^{n}(\mathsf{wgt}([\begin{smallmatrix}\varepsilon\\b_i\end{smallmatrix}]) + \mathsf{wgt}([\begin{smallmatrix}b_i\\\varepsilon\end{smallmatrix}])) \geq n$ and hence

$$\mathsf{wgt}(\alpha_{1,3}) = \mathsf{wgt}(\alpha'_{1,3}) \leq \mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - n$$

as required.

3. From items (2) and (1) we get

$$\frac{\mathsf{wgt}(\alpha_{1,3})}{\mathsf{len}(\alpha_{1,3})} \leq \frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - n}{\mathsf{len}(\alpha_{1,3})} \leq \frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - n}{\max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\} - n}$$

Applying Fact 30 we get

$$\frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3}) - n}{\max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\} - n} \leq \frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3})}{\max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\}}$$

Assume without loss of generality that $\mathsf{len}(\alpha_{1,2}) \geq \mathsf{len}(\alpha_{2,3})$ then

$$\begin{aligned}\frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3})}{\max\{\mathsf{len}(\alpha_{1,2}), \mathsf{len}(\alpha_{2,3})\}} &= \frac{\mathsf{wgt}(\alpha_{1,2}) + \mathsf{wgt}(\alpha_{2,3})}{\mathsf{len}(\alpha_{1,2})} \\ &= \frac{\mathsf{wgt}(\alpha_{1,2})}{\mathsf{len}(\alpha_{1,2})} + \frac{\mathsf{wgt}(\alpha_{2,3})}{\mathsf{len}(\alpha_{1,2})} \leq \frac{\mathsf{wgt}(\alpha_{1,2})}{\mathsf{len}(\alpha_{1,2})} + \frac{\mathsf{wgt}(\alpha_{2,3})}{\mathsf{len}(\alpha_{2,3})}\end{aligned}$$

Overall we get

$$\mathsf{cost}(\alpha_{1,3}) = \frac{\mathsf{wgt}(\alpha_{1,3})}{\mathsf{len}(\alpha_{1,3})} \leq \frac{\mathsf{wgt}(\alpha_{1,2})}{\mathsf{len}(\alpha_{1,2})} + \frac{\mathsf{wgt}(\alpha_{2,3})}{\mathsf{len}(\alpha_{2,3})} = \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{1,3})$$

as required. ◀

With this proof in place we can conclude that d being fine is a sufficient condition for NED_d to be a metric.

▶ **Theorem 31.** *Let $d : \hat{\Gamma} \to [0, 1]$ be fine. Then NED_d is a metric.*

Proof. Given $d : \hat{\Gamma} \to [0, 1]$ is fine it is easy to see that NED_d satisfies the first two requirements of a metric. To see that it also satisfies the triangle inequality, let $w_1, w_2, w_3 \in \Sigma^*$. Let $\alpha_{1,2}$ be an optimal edit path between w_1, w_2 and $\alpha_{2,3}$ an optimal edit path between w_2, w_3. Let $\alpha_{1,3}$ be the result of composing $\alpha_{1,2}, \alpha_{2,3}$ via Definition 27. By Claim 28, $\alpha_{1,3}$ is an edit path between w_1 and w_3 and by Lemma 29, $\mathsf{cost}(\alpha_{1,3}) \leq \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{1,3})$. Thus

$$\text{NED}_d(w_1, w_3) \leq \mathsf{cost}(\alpha_{1,3}) \leq \mathsf{cost}(\alpha_{1,2}) + \mathsf{cost}(\alpha_{1,3}) = \text{NED}_d(w_1, w_2) + \text{NED}_d(w_2, w_3)$$

as required. ◀

▶ **Corollary 32.** *NED_d is a metric if and only if d is fine.*

This corollary proves Theorem 1.

▶ Remark 33. Consider d of Claim 6. The first two requirements of being fine are obviously met. By Claim 19 the operations $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ are inessential. Hence the third requirement clearly hold. Note that $m = \sup\{\text{NED}_d(w_1, w_2) : w_1, w_2 \in \Sigma^*\} = 0.5$ and $\frac{m}{2} = 0.25$ hence the fourth requirement holds.

Consider d of Claim 7. The first requirement of being fine clearly holds. While it breaks symmetry, if we remove the inessential operations, namely $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ then symmetry is maintained. Since $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ are inessentials the third requirement holds as well. Finally the fourth requirement holds since $m = 0.45$ (as $[\begin{smallmatrix}a\\b\end{smallmatrix}]$ and $[\begin{smallmatrix}b\\a\end{smallmatrix}]$ are inessentials) and $\frac{m}{2} = 0.225$.

6 Discussion

Now that we have a sufficient and necessary condition for $d : \Gamma \to [0, 1]$ for NED_d to be a metric, it is easy to verify or come up with such d's for certain applications. We give some examples in §6.1. In §6.2 we discuss extensions to infinite words and applications in formal verification.

6.1 Examples for fine weight functions

Recall that given an alphabet Σ, we use Γ for $\hat{\Gamma} \setminus \{[\begin{smallmatrix}\varepsilon\\\varepsilon\end{smallmatrix}]\}$ where $\hat{\Gamma} = (\Sigma \cup \{\varepsilon\})^2$. Given a function $d : \Sigma \times \Sigma \to [0, 1]$ and given $c \in [\frac{1}{2}, 1]$ we augment it to a function $d^c : \Gamma \to [0, 1]$ as follows:

$$d^c(\sigma_1, \sigma_2) = \begin{cases} d(\sigma_1, \sigma_2) & \text{if } \sigma_1 \neq \varepsilon \text{ and } \sigma_2 \neq \varepsilon \\ c & \text{if } \sigma_1 = \varepsilon \text{ or } \sigma_2 = \varepsilon \end{cases}$$

Consider the case where $\Sigma = [0, n]$ for some $n \in \mathbb{N}$, that is Σ is a finite interval of the natural numbers, starting with 0. Then the following distance over Σ is fine.

▶ **Example 34** (Distances in $[0, n]$). Let $d_n : [0, n] \times [0, n] \to [0, 1]$ be defined as follows:

$$d_n(n_1, n_2) = \frac{|n_1 - n_2|}{n + 1}$$

▷ Claim 35. The weight function d_n^c is fine.

Consider now the case that $\Sigma = \mathbb{N}$, i.e., Σ is the set of naturals number. We can show that the following distance [16] is fine.

▶ **Example 36** (Distances in $\mathbb{N}$). Let $d_\mathbb{N} : \mathbb{N} \times \mathbb{N} \to [0, 1]$ be defined as follows:

$$d_\mathbb{N}(n_1, n_2) = 1 - \frac{1}{|n_1 - n_2| + 1}$$

▷ Claim 37. The weight function $d_\mathbb{N}^c$ is fine.

▶ **Example 38** (Distances between sets). Let $\Sigma = 2^A$ for some finite set of elements A. Let $d_{\mathsf{set}} : 2^A \times 2^A \to [0, 1]$ be defined as follows, where $\oplus$ denotes the symmetrical difference:

$$d_{\mathsf{set}}(S_1, S_2) = \frac{|S_1 \oplus S_2|}{|A|}$$

▷ Claim 39. The weight function d_{set}^c is fine.

In *model checking* [1, 2, 3], automata are defined with respect to a set $AP = \{p_1, p_2, \ldots, p_k\}$ of atomic propositions and the alphabet is $\Sigma = 2^{AP}$. We can use d_{set} to measure the distance between letters, but in a setting where a noise may alter the value of one of the atomic propositions it makes sense to define the distance between two letters as the Hamming distance between the two letters, divided by k for normalization.[5]

▶ **Example 40** (Distances in $\Sigma = 2^k$). Let $\Sigma = 2^k$. Let $d_{\mathsf{prop}} : 2^k \times 2^k \to [0, 1]$ be defined as follows:

$$d_{\mathsf{prop}}(v_1, v_2) = \frac{\textsc{hd}(v_1, v_2)}{k}$$

▷ Claim 41. The weight function d_{prop}^c is fine.

The transitions in automata used in model checking, are usually expressed using Boolean expressions over the set of atomic propositions, e.g. the Boolean expression $p_1 \wedge (\neg p_5 \vee p_7)$ abbreviates the set of letters $\sigma \in 2^k$ where the first bit is 1 and either the fifth bit is zero or the seventh bit is 1, and the rest of the bits can be anything. In general, a Boolean expression b is a compact way to represent the set of letters $\{\sigma \in 2^k \mid \sigma \models b\}$. This type of automata is a special case of *symbolic finite automata* (SFA) that are defined with respect to a concrete alphabet Σ and a symbolic alphabet Ψ of predicates over Σ (see [4] for an introduction to SFAs). The predicates are associated with a semantic function $[\![\cdot]\!]$ that maps a predicate ψ to a subset of Σ that consists of the concrete letters satisfying it. The distance d_{pred} between predicates ψ_1 and ψ_2 can thus be defined using d_{set} on $[\![\psi_1]\!]$ and $[\![\psi_2]\!]$.

▶ **Corollary 42.** *We have that* $\mathrm{NED}_{d_n^c}$, $\mathrm{NED}_{d_\mathbb{N}^c}$, $\mathrm{NED}_{d_{\mathsf{set}}^c}$, $\mathrm{NED}_{d_{\mathsf{prop}}^c}$ *and* $\mathrm{NED}_{d_{\mathsf{pred}}^c}$ *are metrics.*

[5] The Hamming distance, $\textsc{hd} : \bigcup_{k \in \mathbb{N}} (\Sigma^k \times \Sigma^k) \to \mathbb{N}$, is defined between two strings of the same length, as the number of positions in which they differ [9].

6.2 Applications in Formal Verification

The *robustness* question in verification, roughly speaking, asks how much a system S can be altered so that it still satisfies its specification T. Suppose the distance between words is given by dist, and that $[\![S]\!]$ is the set of computations induced by the system and $[\![T]\!]$ is the set of allowed computations according to the specification T. It is noted in [6] that the robustness question can be reduced to question of computing the distance between the languages $[\![S]\!]$ and $[\![T]\!]$ defined as: $\inf_{w_1 \in [\![S]\!]} \inf_{w_2 \in [\![\varphi]\!]} \mathsf{dist}(w_1, w_2)$. It is shown in [8, Theorem 18] that when S and T are given by non-deterministic finite automata and dist is NED over the uniform weights this can be computed in polynomial time. The proof is by building a so called *edit distance graph* of two NFAs, and using the fact that the infimum of the mean weights of paths from a set of origin nodes to a set of target nodes can be computed in polynomial time [6]. Since the same graph can be constructed for NED_d, with the only difference that the weights of edges follow the given d rather than follow the uniform weights, and since the proof in [6] works on any weighted graph in which the weights are rationals, we can conclude that NED_d between languages can be computed in polynomial time, if d gives rational weights. Note that this is the case in all examples considered in §6.1.

In formal verification, systems and specifications are usually defined over infinite words. It is thus desired to have a function $\mathsf{dist} : \Sigma^\omega \times \Sigma^\omega \to [0, 1]$ that measure the distance between two infinite words. In [8, Thm. 6] it was shown that $\overline{\omega}\text{-NED}(w_1, w_2)$ which is defined as $\limsup_{i \to \infty} \text{NED}(w_1[..i], w_2[..i])$ is a metric on infinite words. We can similarly define $\overline{\omega}\text{-NED}_d(w_1, w_2)$ as $\limsup_{i \to \infty} \text{NED}_d(w_1[..i], w_2[..i])$ and the same proof goes through. To compute the distance between two ultimately periodic words,[6] it is shown [8, Thm. 8] that it suffices to consider the best rotations of the periodic parts. Thus reducing computation of $\overline{\omega}$-NED to computation of NED, which can be done in polynomial time [13]. This proof works also for $\overline{\omega}\text{-NED}_d$ if d is non-uniform and gives rational weights. To compute the distance between S and T given by non-deterministic Büchi automata (NBA),[7] [8] requires a more sophisticated version of the edit graph, which tracks along a cycle the number of insert and deletes to coordinate that they are balanced, namely that the same number of letters is read in both automata. The same technique would work in the case of non-uniform weights. We conclude that the robustness question when S and T are NBAs and the considered distance is $\overline{\omega}\text{-NED}_d$ for some fine non-uniform weight function d that gives rational weights can also be computed in polynomial time.

References

1 C. Baier and J-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

2 Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition.* MIT Press, 2018. URL: `https://mitpress.mit.edu/books/model-checking-second-edition`.

3 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018. `doi:10.1007/978-3-319-10575-8`.

4 Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 47–67, 2017.

[6] Restricting infinite words to ultimately periodic words is common in formal verification since two regular ω-languages are equivalent iff they agree on the set of ultimately periodic words [15].

[7] NBA is the most common computational model used in formal verification.

5 Colin de la Higuera and Luisa Micó. A contextual normalised edit distance. In *Proceedings of the 24th International Conference on Data Engineering Workshops, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 354–361. IEEE Computer Society, 2008.

6 Emmanuel Filiot, Nicolas Mazzocchi, Jean-François Raskin, Sriram Sankaranarayanan, and Ashutosh Trivedi. Weighted transducers for robustness verification. In *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, pages 17:1–17:21, 2020.

7 Dana Fisman, Joshua Grogin, Oded Margalit, and Gera Weiss. The normalized edit distance with uniform operation costs is a metric. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 17:1–17:17, 2022.

8 Dana Fisman, Joshua Grogin, and Gera Weiss. A normalized edit distance on infinite words. In *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, pages 20:1–20:20, 2023.

9 R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950. `doi:10.1002/j.1538-7305.1950.tb00463.x`.

10 Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, December 1992. `doi:10.1145/146370.146380`.

11 Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

12 Yujian Li and Bi Liu. A normalized levenshtein distance metric. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1091–1095, 2007.

13 Andrés Marzal and Enrique Vidal. Computation of normalized edit distance and applications. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(9):926–932, 1993.

14 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001. `doi:10.1145/375360.375365`.

15 Büchi J. R. On a decision method in restricted second order arithmetic. In *Int. Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

16 Sanda Zilles. A distance on ℕ. Private communication, 2023.

17 David Sankoff and Joseph B. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

18 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974. `doi:10.1145/321796.321811`.

19 Achim Weigel and Frank Fein. Normalizing the weighted edit distance. In *12th IAPR International Conference on Pattern Recognition, Conference B: Patern Recognition and Neural Networks, ICPR 1994, Jerusalem, Israel, 9-13 October, 1994, Volume 2*, pages 399–402, 1994.

Efficient Construction of Long Orientable Sequences

Daniel Gabrić
University of Guelph, Canada

Joe Sawada
University of Guelph, Canada

Abstract

An orientable sequence of order n is a cyclic binary sequence such that each length-n substring appears at most once *in either direction.* Maximal length orientable sequences are known only for $n \leq 7$, and a trivial upper bound on their length is $2^{n-1} - 2^{\lfloor (n-1)/2 \rfloor}$. This paper presents the first efficient algorithm to construct orientable sequences with asymptotically optimal length; more specifically, our algorithm constructs orientable sequences via cycle-joining and a successor-rule approach requiring $O(n)$ time per bit and $O(n)$ space. This answers a longstanding open question from Dai, Martin, Robshaw, Wild [Cryptography and Coding III (1993)]. Our sequences are applied to find new longest-known orientable sequences for $n \leq 20$.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases orientable sequence, de Bruijn sequence, concatenation tree, cycle-joining, universal cycle

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.15

1 Introduction

Orientable sequences were introduced by Dai, Martin, Robshaw, and Wild [7] with applications related to robotic position sensing. In particular, consider an autonomous robot with limited sensors. To determine its location on a cyclic track labeled with black and white squares, the robot scans a window of n squares directly beneath it. For the position *and* orientation to be uniquely determined, the track must designed with the property that each length n window can appear at most once in *either direction.* A cyclic binary sequence (track) with such a property is called an *orientable sequence* of order n (an $\mathcal{OS}(n)$). By this definition, an orientable sequence does not contain a length-n substring that is a palindrome.

Example 1 Consider $\mathcal{S} = 001011$. In the forward direction, including the wraparound, $\mathcal{S}$ contains the six 5-tuples 00101, 01011, 10110, 01100, 11001, and 10010; in the reverse direction $\mathcal{S}$ contains 11010, 10100, 01001, 10011, 00110, and 01101. Since each substring is unique, $\mathcal{S}$ is an $\mathcal{OS}(5)$ with length (period) six.

Orientable sequences do not exist for $1 < n < 5$, and somewhat surprisingly, the maximum length M_n of an $\mathcal{OS}(n)$ is known only for $n = 1, 5, 6, 7$. Since the number of palindromes of length n is $2^{\lfloor (n+1)/2 \rfloor}$, a trivial upper bound on M_n is $(2^n - 2^{\lfloor (n+1)/2 \rfloor})/2 = 2^{n-1} - 2^{\lfloor (n-1)/2 \rfloor}$.

In addition to providing a tighter upper bound, Dai, Martin, Robshaw, and Wild [7] provide a lower bound on M_n by demonstrating the *existence* of $\mathcal{OS}(n)$s via cycle-joining with length L_n asymptotic to their upper bound; see Section 1.1 for the explicit upper and lower bounds.. They conclude by stating the following open problem relating to orientable sequences whose lengths (periods) attain the lower bound.

We note that the lower bound on the maximum period was obtained using an existence construction ... It is an open problem whether a more practical procedure exists for the construction of orientable sequences that have this asymptotically optimal period.

Recently, some progress was made in this direction by Mitchell and Wild [20]. They apply Lempel's lift [18] to obtain an $\mathcal{OS}(n)$ recursively from an $\mathcal{OS}(n-1)$. This construction can generate orientable sequences in $O(1)$-amortized time per bit; however, it requires exponential space, and there is an exponential time delay before the first bit can be output. Furthermore, they state that they "only *partially* answer the question, since the periods/lengths of the sequences produced are not asymptotically optimal."

Main result: By developing a parent rule to define a cycle-joining tree, we construct an $\mathcal{OS}(n)$ of length L_n in $O(n)$ time per bit using $O(n)$ space.

Outline. In Section 1.1, we review the lower bound L_n and upper bound U_n from [7]. In Section 2, we present necessary background definitions and notation, including a review of the cycle-joining technique. In Section 3, we provide a parent rule for constructing a cycle-joining tree composed of "reverse-disjoint" cycles. This leads to our $O(n)$ time per bit construction of orientable sequences of length L_n. In Section 4 we discuss the algorithmic techniques used to extend our constructed orientable sequences to find longer ones for $n \le 20$. We conclude in Section 5 with a summary of our results and directions for future research. An implementation of our construction is available for download at `http://debruijnsequence.org/db/orientable`.

1.1 Bounds on M_n

Dai, Martin, Robshaw, and Wild [7] gave a lower bound L_n and an upper bound U_n on the maximum length M_n of an $\mathcal{OS}(n)$.[1] Their lower bound L_n is the following, where μ is the Möbius function:

$$L_n = \left(2^{n-1} - \frac{1}{2}\sum_{d|n} \mu(n/d)\frac{n}{d}H(d)\right), \qquad \text{where} \qquad H(d) = \frac{1}{2}\sum_{i|d} i\left(2^{\lfloor \frac{i+1}{2} \rfloor} + 2^{\lfloor \frac{i}{2} \rfloor + 1}\right).$$

Their upper bound U_n is the following:[1]

$$U_n = \begin{cases} 2^{n-1} - \frac{41}{9}2^{\frac{n}{2}-1} + \frac{n}{3} + \frac{16}{9} & \text{if } n \bmod 4 = 0, \\ 2^{n-1} - \frac{31}{9}2^{\frac{n-1}{2}} + \frac{n}{3} + \frac{19}{9} & \text{if } n \bmod 4 = 1, \\ 2^{n-1} - \frac{41}{9}2^{\frac{n}{2}-1} + \frac{n}{6} + \frac{20}{9} & \text{if } n \bmod 4 = 2, \\ 2^{n-1} - \frac{31}{9}2^{\frac{n-1}{2}} + \frac{n}{6} + \frac{43}{18} & \text{if } n \bmod 4 = 3. \end{cases}$$

These bounds are calculated in Table 1 for n up to 20. This table also illustrates the length R_n of the $\mathcal{OS}(n)$ produced by the recursive construction by Mitchell and Wild [20], starting from an initial orientable sequence of length 80 for $n = 8$. The column labeled L_n^* indicates the longest known orientable sequences we discovered by applying a combination of techniques (discussed in Section 4) to our orientable sequences of length L_n.

[1] These bounds correspond to $\tilde{L}_n$ and $\tilde{U}_n$, respectively, as they appear in [7].

Table 1 Lower bounds R_n, L_n, L_n^* and upper bound U_n for M_n.

n	R_n	L_n	L_n^*	U_n
5	-	0	**6**	6
6	-	6	**16**	17
7	-	14	**36**	40
8	80	48	92	96
9	161	126	174	206
10	322	300	416	443
11	645	682	844	918
12	1290	1530	1844	1908
13	2581	3276	3700	3882
14	5162	6916	7694	7905
15	10325	14520	15394	15948
16	20650	29808	31483	32192
17	41301	61200	63135	64662
18	82602	124368	128639	129911
19	165205	252434	257272	260386
20	330410	509220	519160	521964

1.2 Related work

Recall the problem of determining a robot's position and orientation on a track. Suppose now that we allow the track to be non-cyclic. That is, the beginning of the track and the end of the track are not connected. Then the corresponding sequence that allows one to determine orientation and position is called an *acyclic orientable sequence*. One does not consider the substrings in the wraparound for this variation of an orientable sequence. Note that one can always construct an acyclic $\mathcal{OS}(n)$ from a cyclic $\mathcal{OS}(n)$ by taking the cyclic $\mathcal{OS}(n)$ and appending its prefix of length $n-1$ to the end. See the paper by Burns and Mitchell [5] for more on acyclic orientable sequences, which they call *aperiodic* 2*-orientable window sequences*. Alhakim et al. [2] generalize the recursive results of Mitchell and Wild [20] to construct orientable sequences over an alphabet of arbitrary size $k \geq 2$; they also generalize the upper bound, by Dai et al. [7], on the length of an orientable sequence. Rampersad and Shallit [21] showed that for every alphabet size $k \geq 2$ there is an infinite sequence such that for every sufficiently long substring, the reversal of the substring does not appear in the sequence. Fleischer and Shallit [11] later reproved the results of the previous paper using theorem-proving software. See [6, 19] for more work on sequences avoiding reversals of substrings.

2 Preliminaries

Let $\mathbf{B}(n)$ denote the set of all length-n binary strings. Let $\alpha = \mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_n \in \mathbf{B}(n)$ and $\beta = \mathtt{b}_1\mathtt{b}_2\cdots\mathtt{b}_m \in \mathbf{B}(m)$ for some $m, n \geq 0$. Throughout this paper, we assume $0 < 1$ and use lexicographic order when comparing two binary strings. More specifically, we say that $\alpha < \beta$ either if α is a prefix of β or if $\mathtt{a}_i < \mathtt{b}_i$ for the smallest i such that $\mathtt{a}_i \neq \mathtt{b}_i$. We say that α is a *rotation* of β if $m = n$ and there exist strings x and y such that $\alpha = xy$ and $\beta = yx$. The *weight* (density) of a binary string is the number of 1s in the string. Let $\bar{\mathtt{a}}_i$ denote the complement of bit $\mathtt{a}_i$. Let α^R denote the reversal $\mathtt{a}_n\cdots\mathtt{a}_2\mathtt{a}_1$ of α; α is a *palindrome* if $\alpha = \alpha^R$. For $j \geq 1$, let α^j denote j copies of α concatenated together. If $\alpha = \gamma^j$ for some non-empty string γ and some $j > 1$, then α is said to be *periodic*[2]; otherwise, α is said to be *aperiodic* (or *primitive*). For example, the English word `hotshots` $=$ (`hots`)^2 is periodic, but `hots` is aperiodic.

A *necklace class* is an equivalence class of strings under rotation; let $[\alpha]$ denote the set of strings in α's necklace class. We say α is a *necklace* if it is the lexicographically smallest string in $[\alpha]$. Let $\mathbf{N}(n)$ denote the set of length-n necklaces. A *bracelet class* is an equivalence class of strings under rotation and reversal; let $\langle\alpha\rangle$ denote the set of strings in α's bracelet class. Thus, $\langle\alpha\rangle = [\alpha] \cup [\alpha^R]$. We say α is a *bracelet* if it is the lexicographically smallest string in $\langle\alpha\rangle$. Note that in general, a bracelet is always a necklace, but a necklace need not be a bracelet. For example, the string 001011 is both a bracelet and a necklace, but the string 001101 is a necklace and is not a bracelet.

A necklace α is *symmetric* if it belongs to the same necklace class as α^R, i.e., both α and α^R belong to $[\alpha]$. By this definition, a symmetric necklace is necessarily a bracelet. If a necklace or bracelet is not symmetric, it is said to be *asymmetric*. Let $\mathbf{A}(n)$ denote the set of all asymmetric bracelets of order n. Table 2 lists all 60 necklaces of length $n = 9$ partitioned into asymmetric necklace pairs and symmetric necklaces. The asymmetric necklace pairs belong to the same bracelet class, and the first string in each pair is an asymmetric bracelet. Thus, $|\mathbf{A}(9)| = 14$. In general, $|\mathbf{A}(n)|$ is equal to the number of necklaces of length n minus the number of bracelets of length n; for $n = 6, 7, \ldots 15$, this sequence of values $|\mathbf{A}(n)|$ is given by 1, 2, 6, 14, 30, 62, 128, 252, 495, 968 and it corresponds to sequence A059076 in The On-Line Encyclopedia of Integer Sequences [25]. Asymmetric bracelets have been studied previously in the context of efficiently ranking/unranking bracelets [1]. One can test whether a string is an asymmetric bracelet in linear time using linear space; see Theorem 1.

▶ **Theorem 1.** *One can determine whether a string α is in $\mathbf{A}(n)$ in $O(n)$ time using $O(n)$ space.*

Proof. A string α will belong to $\mathbf{A}(n)$ if α is a necklace and the necklace of $[\alpha^R]$ is lexicographically larger than α. These tests can be computed in $O(n)$ time using $O(n)$ space [3]. ◀

Lemma 2 is considered a folklore result in combinatorics on words; see Theorem 4 in [4] for a variant of the lemma. We provide a short proof for the interested reader.

▶ **Lemma 2.** *A necklace α is symmetric if and only if there exists palindromes β_1 and β_2 such that $\alpha = \beta_1\beta_2$.*

Proof. Suppose α is a symmetric necklace. By definition, it is equal to the necklace of $[\alpha^R]$. Thus, there exist strings β_1 and β_2 such that $\alpha = \beta_1\beta_2 = (\beta_2\beta_1)^R = \beta_1^R\beta_2^R$. Therefore, $\beta_1 = \beta_1^R$ and $\beta_2 = \beta_2^R$, which means β_1 and β_2 are palindromes. Suppose there exists two palindromes β_1 and β_2 such that $\alpha = \beta_1\beta_2$. Since β_1 and β_2 are symmetric, we have that $\alpha^R = (\beta_1\beta_2)^R = \beta_2^R\beta_1^R = \beta_2\beta_1$. So α belongs to the same necklace class as α^R and hence is symmetric. ◀

▶ **Corollary 3.** *If $\alpha = 0^s\beta$ is a symmetric bracelet such that the string β begins and ends with 1 and does not contain 0^s as a substring, then β is a palindrome.*

[2] Periodic strings are are also known as *powers* in the literature. The term *periodic* is sometimes used to denote a string of the form $(\alpha\beta)^i\alpha$ where α is non-empty, β is possibly empty, $i \geq 1$, and $\frac{|(\alpha\beta)^i\alpha|}{|\alpha\beta|} \geq 2$. Under this definition, the word `alfalfa` is periodic, but `bonobo` is not.

Table 2 A listing of all 60 necklaces in $\mathbf{N}(9)$ partitioned into asymmetric necklace pairs and symmetric necklaces. The first column of the asymmetric necklaces corresponds to the 14 asymmetric bracelets $\mathbf{A}(9)$.

Asymmetric necklace pairs	**Symmetric necklaces**		
000001011 , 000001101	000000000	$000100\underline{011}$	$00\underline{1110111}$
000010011 , 000011001	$00000000\underline{1}$	$000\underline{101101}$	$00\underline{1111111}$
000010111 , 000011101	$0000000\underline{11}$	$000\underline{110011}$	$0101010\underline{11}$
000100101 , 000101001	$000000\underline{101}$	$000\underline{111111}$	$01010\underline{1111}$
000100111 , 000111001	$000000\underline{111}$	$00100100\underline{1}$	$010\underline{111111}$
000101011 , 000110101	$00000\underline{1001}$	$00100\underline{1111}$	$0110110\underline{11}$
000101111 , 000111101	$00000\underline{1111}$	$0010100\underline{11}$	$011\underline{011111}$
000110111 , 000111011	$0000\underline{10001}$	$00\underline{1010101}$	$01110\underline{1111}$
001001011 , 001001101	$0000\underline{10101}$	$00\underline{1011101}$	$0\underline{11111111}$
001010111 , 001110101	$0000\underline{11011}$	$001100\underline{111}$	111111111
001011011 , 001101101	$0000\underline{11111}$	$00\underline{1101011}$	
001011111 , 001111101			
001101111 , 001111011			
010110111 , 010111011			

2.1 Cycle joining

Given $\mathbf{S} \subseteq \mathbf{B}(n)$, a *universal cycle* U for $\mathbf{S}$ is a cyclic sequence of length $|\mathbf{S}|$ that contains each string in $\mathbf{S}$ as a substring (exactly once). Thus, an orientable sequence is a universal cycle. If $\mathbf{S} = \mathbf{B}(n)$ then U is known as a *de Bruijn sequence*. Given a universal cycle U for $\mathbf{S}$, a *successor rule* for U is a function $f : \mathbf{S} \to \{0, 1\}$ such that $f(\alpha)$ is the bit following α in U.

Cycle-joining is perhaps the most fundamental technique applied to construct universal cycles; for some applications, see [8, 9, 10, 12, 14, 16, 17, 23, 24]. If $\mathbf{S}$ is closed under rotation, then it can be partitioned into necklace classes (cycles); each cycle is disjoint. Let $\alpha = \mathtt{a}_1\mathtt{a}_2 \cdots \mathtt{a}_n$ and $\hat{\alpha} = \overline{\mathtt{a}}_1\mathtt{a}_2 \cdots \mathtt{a}_n$; we say $(\alpha, \hat{\alpha})$ is a *conjugate pair*. Two disjoint cycles can be joined if they each contain one string of a *conjugate pair* as a substring. This approach resembles Hierholzer's algorithm to construct an Euler cycle in an Eulerian graph [15].

Example 2 Consider disjoint subsets $\mathbf{S}_1 = [011111] \cup [001111]$ and $\mathbf{S}_2 = [010111] \cup [010101]$, where $n = 6$. Then $U_1 = 110011\underline{110111}$ is a universal cycle for $\mathbf{S}_1$ and $U_2 = 01\underline{010111}$ is a universal cycle for $\mathbf{S}_2$. Since $(110111, 010111)$ is a conjugate pair, $U = 110011\underline{110111} \cdot 01\underline{010111}$ is a universal cycle for $\mathbf{S}_1 \cup \mathbf{S}_2$.

If all necklace cycles can be joined via conjugate pairs to form a cycle-joining tree, then the tree defines a universal U for $\mathbf{S}$ with a corresponding successor rule (see Section 3 for an example).

For most universal cycle constructions, a corresponding cycle-joining tree can be defined by a rather simple *parent rule*. For example, when $\mathbf{S} = \mathbf{B}(n)$, the following are perhaps the *simplest* parent rules that define how to construct cycle-joining trees with nodes corresponding to $\mathbf{N}(n)$ [13, 22].

- **Last-0**: rooted at 1^n and the parent of every other node $\alpha \in \mathbf{N}(n)$ is obtained by flipping the last 0.
- **First-1**: rooted at 0^n and the parent of every other node $\alpha \in \mathbf{N}(n)$ is obtained by flipping the first 1.

- **Last-1**: rooted at 0^n and the parent of every other node $\alpha \in \mathbf{N}(n)$ is obtained by flipping the last 1.
- **First-0**: rooted at 1^n and the parent of every other node $\alpha \in \mathbf{N}(n)$ is obtained by flipping the first 0.

These rules induce the cycle-joining trees T_1, T_2, T_3, T_4 illustrated in Figure 1 for $n = 6$. Note that for T_3 and T_4, the parent of a node α is obtained by first flipping the highlighted bit and then rotating the string to its lexicographically least rotation to obtain a necklace. Each node α and its parent β are joined by a conjugate pair, where the highlighted bit in α is the first bit in one of the conjugates. For example, the nodes $\alpha = 011011$ and $\beta = 001011$ in T_2 from Figure 1 are joined by the conjugate pair $(110110, 010110)$.

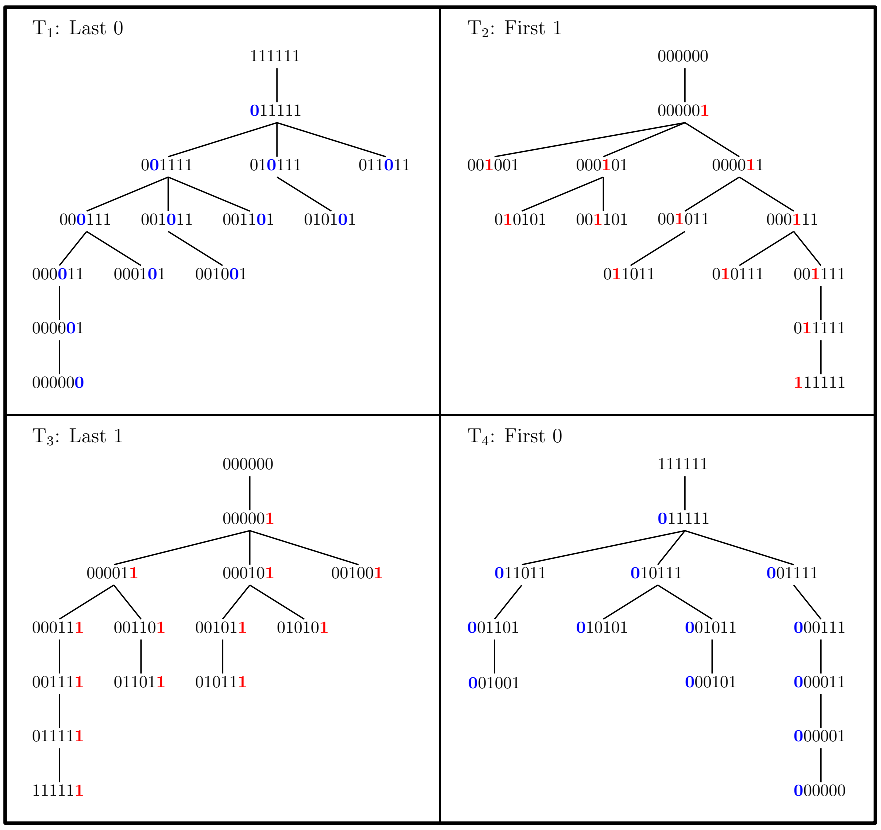


Figure 1 Cycle-joining trees for $\mathbf{B}(6)$ from simple parent rules.

3 An efficient cycle-joining construction of orientable sequences

Consider the set of asymmetric bracelets $\mathbf{A}(n) = \{\alpha_1, \alpha_2, \ldots, \alpha_t\}$. Recall, that each symmetric bracelet is a necklace. Let $\mathbf{S}(n) = [\alpha_1] \cup [\alpha_2] \cup \cdots \cup [\alpha_t]$. From [7], we have $|\mathbf{S}(n)| = L_n$. By its definition, there is no string $\alpha \in \mathbf{S}(n)$ such that $\alpha^R \in \mathbf{S}(n)$. Thus, a universal cycle for $\mathbf{S}(n)$ is an $\mathcal{OS}(n)$. For the rest of this section, we assume $n \geq 8$.

To construct a cycle-joining tree with nodes $\mathbf{A}(n)$, we apply a combination of three of the four simple parent rules described in the previous section. First, we demonstrate that there is no such parent rule, using at most two rules in combination. Observe, there are no necklaces in $\mathbf{A}(n)$ with weight 0, 1, 2, $n-2$, $n-1$, or, n. Thus, $0^{n-4}1011$ and $0^{n-5}10011$ are both necklaces in $\mathbf{A}(n)$ with minimal weight three. Similarly, 00101^{n-4} and 001101^{n-5} are necklaces in $\mathbf{A}(n)$ with maximal weight $n-3$. Therefore, when considering a parent rule for a cycle-joining tree with nodes $\mathbf{A}(n)$, the rule must be able to flip a 0 to a 1, or a 1 to a 0, i.e., if the rule applies a combination of the four rules from Section 2.1, it must include one of First-0 or Last-0, and one of First-1 and Last-1.

Let $\alpha = \mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_n$ denote a necklace in $\mathbf{A}(n)$; it must begin with 0 and end with 1. Then let

- first1(α) be the necklace $\mathtt{a}_1\cdots\mathtt{a}_{i-1}\mathbf{0}\mathtt{a}_{i+1}\cdots\mathtt{a}_n$, where i is the index of the first 1 in α;
- last1(α) be the necklace of $[\mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_{n-1}\mathbf{0}]$;
- first0(α) be the necklace of $[\mathbf{1}\mathtt{a}_2\cdots\mathtt{a}_n]$;
- last0(α) be the necklace $\mathtt{a}_1\cdots\mathtt{a}_{j-1}\mathbf{1}\mathtt{a}_{j+1}\cdots\mathtt{a}_n$, where j is the index of the last 0 in α.

Note that first1(α) and last0(α) are necklaces (easily observed by definition) obtained by flipping the i-th and j-th bit in α, respectively; last1(α) and first0(α) are the result of flipping a bit and rotating the resulting string to obtain a necklace. The next example illustrates that no two of the previous four parent rules can be applied in combination to obtain a spanning tree with nodes in $\mathbf{A}(n)$.

Example 3 Suppose p(α) is a parent rule that applies a combination of the four parent rules, first1, last1, first0, last0, to construct a cycle-joining tree with nodes $\mathbf{A}(n)$. The following examples are for $n = 10$ but generalize to larger n. In both cases, we see that at least three of the parent rules must be applied in p.

Suppose p does not use first0; it must apply last0. Consider three asymmetric bracelets in $\mathbf{A}(10)$: $\alpha_1 = 0000001011$, $\alpha_2 = 0000010111$, and $\alpha_3 = 0011001011$. Clearly, first1(α_1), last1(α_1), and last0(α_1) are symmetric. Thus, α_1 must be the root. Both first1(α_2) and last0(α_2) are symmetric; thus, p must apply last1. Note last0(α_3) is symmetric and last1(α_3) = 0001100101 is not a bracelet; thus, p must apply first1.

Suppose p does not use last0; it must apply first0. Consider three asymmetric bracelets in $\mathbf{A}(10)$: $\beta_1 = 0000100011$, $\beta_2 = 0001001111$, and $\beta_3 = 0001100111$. Clearly, first1(β_1), last1(β_1), and first0(β) are symmetric. Thus, β_1 must be the root. Both first1(β_2) and first0(β_2) are symmetric; thus, p must apply last1. Both last1(β_3) and first0(β_3) are symmetric; thus, p must apply first1.

Let r_n denote the asymmetric bracelet $0^{n-4}1011$. We choose to use r_n to be the root of our cycle-joining tree since it is the lexicographically smallest asymmetric bracelet of length n.

Parent rule for cycle-joining $\mathbf{A}(n)$: Let r_n be the root. Let α denote a non-root node in $\mathbf{A}(n)$. Then

$$\text{par}(\alpha) = \begin{cases} \text{first1}(\alpha) & \text{if first1}(\alpha) \in \mathbf{A}(n); \\ \text{last1}(\alpha) & \text{if first1}(\alpha) \notin \mathbf{A}(n) \text{ and last1}(\alpha) \in \mathbf{A}(n); \\ \text{last0}(\alpha) & \text{otherwise.} \end{cases} \quad (1)$$

▶ **Theorem 4.** *The parent rule* par(α) *in* (1) *induces a cycle-joining tree with nodes* $\mathbf{A}(n)$ *rooted at* r_n.

Let $\mathbb{T}_n$ denote the cycle-joining tree with nodes $\mathbf{A}(n)$ induced by the parent rule in (1); Figure 2 illustrates $\mathbb{T}_9$. The proof of Theorem 4 relies on the following lemma.

▶ **Lemma 5.** *Let* $\alpha \neq r_n$ *be an asymmetric bracelet in* $\mathbf{A}(n)$*. If neither* first1(α) *nor* last1(α) *are in* $\mathbf{A}(n)$*, then the last* 0 *in* α *is at index* $n-2$ *or* $n-1$*, and both* last0(α) *and* last1(last0(α)) *are in* $\mathbf{A}(n)$.

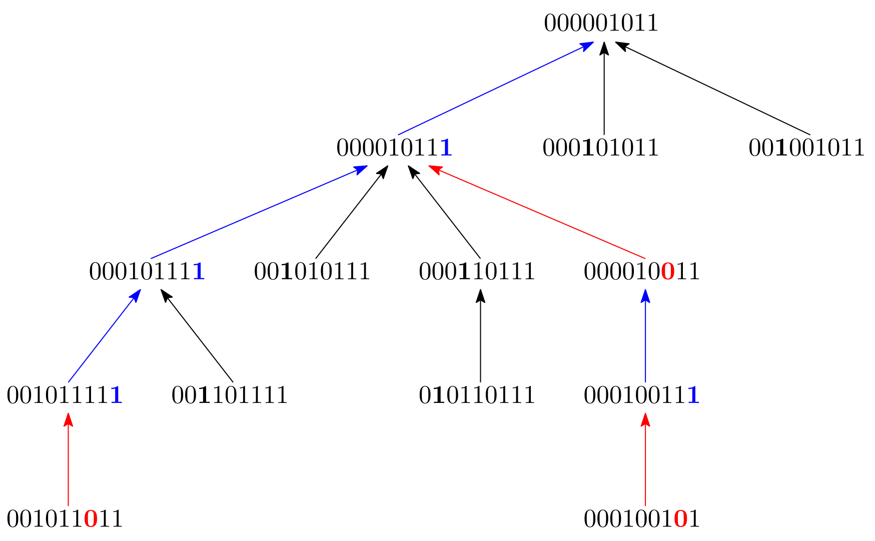


Figure 2 The cycle-joining tree $\mathbb{T}_9$. The black edges indicate that $\text{par}(\alpha) = \text{first1}(\alpha)$; the blue edges indicate that $\text{par}(\alpha) = \text{last1}(\alpha)$; the red edges indicate that $\text{par}(\alpha) = \text{last0}(\alpha)$.

Proof. Since α is an asymmetric bracelet, it must have the form $\alpha = 0^i1\beta01^j$ where $i, j \geq 1$ and $\beta0$ does not contain 0^{i+1} as a substring. Furthermore, $1\beta01^j < (1\beta01^j)^R$, which implies $\beta01^{j-1} < (\beta01^{j-1})^R$.

Suppose $j > 2$. Since $\text{last1}(\alpha) = 0^{i+1}1\beta01^{j-1}$ is not an asymmetric bracelet, we have $1\beta01^{j-1} \geq (1\beta01^{j-1})^R$. Thus, β begins with 1. Since $\text{first1}(\alpha) = 0^{i+1}\beta01^j$ is not an asymmetric bracelet, Lemma 2 implies $\beta01^j \geq (\beta01^j)^R$, contradicting the earlier observation that $\beta01^{j-1} < (\beta01^{j-1})^R$. Thus, the last 0 in α is at index $n-2$ or $n-1$.

Suppose $j = 1$ or $j = 2$. Then the last 0 in α must be at position $n-2$ or $n-1$. Write $\alpha = x0y$ where $y = 1$ or $y = 11$. Since α is a bracelet, it is straightforward to see that $\text{last0}(\alpha) = x1y$ is also a bracelet. If it is symmetric, Lemma 2 implies there exist palindromes β_1 and β_2 such that $\text{last0}(\alpha) = x1y = \beta_1\beta_2$. However, flipping the 1 in $x1y$ that allows us to obtain α implies that α is greater than or equal to the necklace in $[\alpha^R]$, contradicting the assumption that α is an asymmetric bracelet. Thus, $\text{last0}(\alpha)$ is an asymmetric bracelet.

Consider $\text{last1}(\text{last0}(\alpha)) = 0^{i+1}1\beta1^j$. Let $\beta = \mathtt{b}_1\mathtt{b}_2\cdots\mathtt{b}_m$. Suppose that $m = 0$. Then $\text{last1}(\text{last0}(\alpha)) = 0^{i+1}1^{j+1} \Rightarrow \text{last0}(\alpha) = 0^i1^{j+2}$. Since $j = 1$ or $j = 2$, we have that $\text{last0}(\alpha) = 0^i111$ or $\text{last0}(\alpha) = 0^i1111$. Now α is the result of flipping one of the 1s in $\text{last0}(\alpha)$ to a 0 and performing the appropriate rotation. But in every case, we end up with α being a symmetric necklace, a contradiction. Thus, assume $m \geq 1$. Suppose $\beta = 1^m$. Then, α is not an asymmetric bracelet, a contradiction. Suppose $\beta = 0^m$. If $j = 1$, then α is symmetric, a contradiction; if $j = 2$, then $\text{last1}(\text{last0}(\alpha)) = 0^{i+1}10^m11$ which is in $\mathbf{A}(n)$. For all other cases, β contains at least one 1 and at least one 0; $m \geq 2$. Since β does not contain 0^{i+1} as a substring, by Lemma 2, we must show that (i) $\beta1^{j-1} < 1^{j-1}\beta^R$, which implies $1\beta1^j < 1^j\beta^R1$, recalling that (ii) $\beta01^{j-1} < 1^{j-1}0\beta^R$. Let ℓ be the largest index of β such that $\mathtt{b}_\ell = 1$. Then $\mathtt{b}_{\ell+1}\cdots\mathtt{b}_m = 0^{m-\ell}$; note that $\mathtt{b}_{\ell+1}\cdots\mathtt{b}_m$ is the empty string when $\ell = m$. Suppose $j = 1$. From (ii), we have $\mathtt{b}_1 = 0$ and $\mathtt{b}_2\cdots\mathtt{b}_{\ell-1}10^{m-\ell} < 0^{m-\ell}1\mathtt{b}_{\ell-1}\cdots\mathtt{b}_2$. But this implies that $\mathtt{b}_2\cdots\mathtt{b}_{m-\ell+1} = 0^{m-\ell}$. Therefore, we have $\beta = 0^{m-\ell+1}\mathtt{b}_{m-\ell+2}\cdots\mathtt{b}_m < 0^{m-\ell}1\mathtt{b}_{\ell-1}\cdots\mathtt{b}_1 = \beta^R$, hence (i) is satisfied. Suppose

$j = 2$. If $\mathtt{b}_1 = 0$, then (i) is satisfied. Otherwise $\mathtt{b}_1 = 1$ and from (ii) $\mathtt{b}_2 = 0$. From (ii), we get that $\mathtt{b}_3 \cdots \mathtt{b}_{\ell-1} 1 0^{m-\ell} < 0^{m-\ell} \mathtt{b}_{\ell-1} \cdots \mathtt{b}_3$. This inequality implies that $\mathtt{b}_3 \cdots \mathtt{b}_{m-\ell+2} = 0^{m-\ell}$. Therefore, we have $\beta 1 = 10^{m-\ell+1} \mathtt{b}_{m-\ell+3} \cdots \mathtt{b}_m 1 < 10^{m-\ell} 1 \mathtt{b}_{\ell-1} \cdots \mathtt{b}_1 = 1\beta^R$, hence (i) is satisfied. Thus, last1(last0(α)) is an asymmetric bracelet. ◀

Proof of Theorem 4. Let α be an asymmetric bracelet in $\mathbf{A}(n) \setminus \{r_n\}$. We demonstrate that the parent rule par from (1) induces a path from α to r_n, i.e., there exists an integer j such that $\text{par}^j(\alpha) = r_n$. Note that r_n is the lexicographically smallest asymmetric bracelet of order n. By Lemma 5, $\text{par}(\alpha) \in \mathbf{A}(n)$. In the first two cases of the parent rule, $\text{par}(\alpha)$ is lexicographically smaller than α. If the third case applies, let $\alpha = 0^s 1 \beta$. From Lemma 5, last1(last0(α)) is an asymmetric bracelet. Thus, par(par(α)) is either first1(last0(α)) or last1(last0(α)); in each case the resulting asymmetric bracelet has 0^{s+1} as a prefix and is therefore lexicographically smaller than α. Therefore, the parent rule induces a path from α to r_n. ◀

A successor rule

Each application of the parent rule $\text{par}(\alpha)$ in (1) corresponds to a conjugate pair. For instance, consider the asymmetric bracelet $\alpha = 00010111\mathbf{1}$. The parent of α is obtained by flipping the last 1 to obtain $00010111\mathbf{0}$ (see Figure 2). The corresponding conjugate pair is $(\mathbf{1}00010111, \mathbf{0}00010111)$. Let $\mathbf{C}(n)$ denote the set of all strings belonging to a conjugate pair in the cycle-joining tree $\mathbb{T}_n$. Then the following is a successor rule for an $\mathcal{OS}(n)$:

$$f(\alpha) = \begin{cases} \overline{\mathtt{a}}_1 & \text{if } \alpha \in \mathbf{C}(n); \\ \mathtt{a}_1 & \text{otherwise.} \end{cases}$$

For example, if $\mathbf{C}(9)$ corresponds to the conjugate pairs to create the cycle-joining tree $\mathbb{T}_9$ shown in Figure 2, then the corresponding universal cycle is:

$$0\underline{\mathbf{0}00010111}11001011011001011110011011 1\underline{\mathbf{1}00010111}0010101110001101 1$$
$$1010110111000010011100010010100010011000001011001001011000101011,$$

where the two underlined strings belong to the conjugate pair $(\mathbf{1}00010111, \mathbf{0}00010111)$. In general, this rule requires exponential space to store the set $\mathbf{C}(n)$. However, in some cases, it is possible to test whether a string is in $\mathbf{C}(n)$ without pre-computing and storing $\mathbf{C}(n)$. In our successor rule for an $\mathcal{OS}(n)$, we use Theorem 1 to avoid pre-computing and storing $\mathbf{C}(n)$, thereby reducing the space requirement from exponential in n to linear in n.

Successor-rule g to construct an $\mathcal{OS}(n)$ of length L_n

Let $\alpha = \mathtt{a}_1 \mathtt{a}_2 \cdots \mathtt{a}_n \in \mathbf{S}(n)$ and let

- $\beta_1 = 0^{n-i} \mathbf{1} \mathtt{a}_2 \cdots \mathtt{a}_i$ where i is the largest index of α such that $\mathtt{a}_i = 1$ (first 1);
- $\beta_2 = \mathtt{a}_2 \mathtt{a}_3 \cdots \mathtt{a}_n \mathbf{1}$ (last 1);
- $\beta_3 = \mathtt{a}_j \mathtt{a}_{j+1} \cdots \mathtt{a}_n \mathbf{01} 1^{j-2}$ where j is the smallest index of α such that $\mathtt{a}_j = 0$ and $j > 1$ (last 0).

Let

$$g(\alpha) = \begin{cases} \overline{\mathtt{a}}_1 & \text{if } \beta_1 \text{ and first1}(\beta_1) \text{ are in } \mathbf{A}(n); \\ \overline{\mathtt{a}}_1 & \text{if } \beta_2 \text{ and last1}(\beta_2) \text{ are in } \mathbf{A}(n), \text{ and first1}(\beta_2) \text{ is not in } \mathbf{A}(n); \\ \overline{\mathtt{a}}_1 & \text{if } \beta_3 \text{ and last0}(\beta_3) \text{ are in } \mathbf{A}(n), \text{ and neither first1}(\beta_3) \text{ nor last1}(\beta_3) \text{ are in } \mathbf{A}(n); \\ \mathtt{a}_1 & \text{otherwise.} \end{cases}$$

Starting with any string in $\alpha \in \mathbf{S}(n)$, we can repeatedly apply $g(\alpha)$ to obtain the next bit in a universal cycle for $\mathbf{S}(n)$.

▶ **Theorem 6.** *The function g is a successor rule that generates an $\mathcal{OS}(n)$ with length L_n for the set $\mathbf{S}(n)$ in $O(n)$-time per bit using $O(n)$ space.*

Proof. Consider $\alpha = \mathtt{a}_1\mathtt{a}_2 \cdots \mathtt{a}_n \in \mathbf{S}(n)$. If α belongs to some conjugate pair in $\mathbb{T}_n$, then it must satisfy one of three possibilities stepping through the parent rule in 1:

- Both β_1 and first1(β_1) must be in $\mathbf{A}(n)$. Note, β_1 is a rotation of α when $\mathtt{a}_1 = 1$, where $\mathtt{a}_1$ corresponds to the first one in β_1.
- Both β_2 and last1(β_2) must both be in $\mathbf{A}(n)$, but additionally, first1(β_2) can not be in $\mathbf{A}(n)$. Note, β_2 is a rotation of α when $\mathtt{a}_1 = 1$, where $\mathtt{a}_1$ corresponds to the last one in β_2.
- Both β_3 and last0(β_3) must both be in $\mathbf{A}(n)$, but additionally, both first1(β_3) and last1(β_3) can not be in $\mathbf{A}(n)$. Note, β_3 is a rotation of α when $\mathtt{a}_1 = 0$, where $\mathtt{a}_1$ corresponds to the last zero in β_3.

Thus, g is a successor rule on $\mathbf{S}(n)$ that generates a cycle of length $|\mathbf{S}(n)| = L_n$. By Theorem 1, one can determine whether a string is in $\mathbf{A}(n)$ in $O(n)$ time using $O(n)$ space. Since there are a constant number of tests required by each case of g, the corresponding $\mathcal{OS}(n)$ can be computed in $O(n)$-time per bit using $O(n)$ space. ◀

4 Extending orientable sequences

The values from the column labeled L_n^* in Table 1 were found by extending an $\mathcal{OS}(n)$ of length L_n constructed in the previous section. Given an $\mathcal{OS}(n)$, $\mathtt{o}_1 \cdots \mathtt{o}_m$, the following approaches were applied to find longer $\mathcal{OS}(n)$s for $n \leq 20$:

1. For each index i, apply a standard backtracking search to see whether $\mathtt{o}_i \cdots \mathtt{o}_m\mathtt{o}_1 \cdots \mathtt{o}_{i-1}$ can be extended to a longer $\mathcal{OS}(n)$. We followed several heuristics: (a) find a maximal length extension for a given i, and then attempt to extend starting from index $i+1$; (b) find a maximal length extension over all i, then repeat; (c) find the "first" possible extension for a given i, and then repeat for the next index $i+1$. In each case, we repeat until no extension can be found for any starting index. This approach was fairly successful for even n, but found shorter extensions for n odd. Steps (a) and (b) were only applied to n up to 14 before the depth of search became infeasible.
2. Refine the search in the previous step so the resulting $\mathcal{OS}(n)$ of length m' has an odd number of 1s and at most one substring 0^{n-4}. Then we can apply the recursive construction by Mitchell and Wild [20] to generate an $\mathcal{OS}(n+1)$ with length $2m'$ or $2m'+1$. Then, starting from the sequences generated by recursion, we again apply the exhaustive search to find minor extensions (the depth of recursion is significantly reduced). This approach found significantly longer extensions to obtain $\mathcal{OS}(n+1)$s when $n+1$ is odd.

5 Future research directions

We present the first efficient algorithm to construct orientable sequences with asymptotically optimal length; it is a successor-rule-based approach that requires $O(n)$ time per bit and uses $O(n)$ space. This answers a long-standing open question by Dai, Martin, Robshaw, and Wild [7]. The full version of this paper includes an application of the recent concatenation-tree framework [22] that leads to constructions of our $\mathcal{OS}(n)$s in $O(1)$-amortized time per bit.

It also includes the results of applying our $\mathcal{OS}(n)$s to find some longer acyclic orientable sequences than reported in [5]. The binary results have recently been extended to arbitrary sized alphabets like $\{C, G, A, T\}$.

References

1 D. Adamson, V. V. Gusev, I. Potapov, and A. Deligkas. Ranking bracelets in polynomial time. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2021.4.

2 A. Alhakim, C. J. Mitchell, J. Szmidt, and P. R. Wild. Orientable sequences over non-binary alphabets. *manuscript*, 2023.

3 K. S. Booth. Lexicographically least circular substrings. *Inform. Process. Lett.*, 10(4/5):240–242, 1980. doi:10.1016/0020-0190(80)90149-0.

4 S. Brlek, S. Hamel, M. Nivat, and C. Reutenauer. On the palindromic complexity of infinite words. *Internat. J. Found. Comp. Sci.*, 15(02):293–306, 2004. doi:10.1142/S012905410400242X.

5 J. Burns and C. J. Mitchell. Position sensing coding schemes. In *Cryptography and Coding III (M.J.Ganley, ed.)*, pages 31–66. Oxford University Press, 1993.

6 J. Currie and P. Lafrance. Avoidability index for binary patterns with reversal. *Electronic J. Combinatorics*, 23((1) P1.36):1–14, 2016. doi:10.37236/5483.

7 Z. D. Dai, K. M. Martin, M. J. B. Robshaw, and P. R. Wild. Orientable sequences. In *Cryptography and Coding III (M.J.Ganley, ed.)*, pages 97–115. Oxford University Press, 1993.

8 T. Etzion. An algorithm for generating shift-register cycles. *Theoret. Comput. Sci.*, 44(2):209–224, 1986. doi:10.1016/0304-3975(86)90118-0.

9 T. Etzion. Self-dual sequences. *J. Combin. Theory Ser. A*, 44(2):288–298, 1987. doi:10.1016/0097-3165(87)90035-5.

10 T. Etzion and A. Lempel. Algorithms for the generation of full-length shift-register sequences. *IEEE Trans. Inform. Theory*, 30(3):480–484, 1984. doi:10.1109/TIT.1984.1056919.

11 L. Fleischer and J. O. Shallit. Words that avoid reversed factors, revisited. Arxiv preprint arXiv:1911.11704 [cs.FL], available at http://arxiv.org/abs/1911.11704, 2019.

12 H. Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *SIAM Review*, 24(2):195–221, 1982. doi:10.1137/1024041.

13 D. Gabrić, J. Sawada, A. Williams, and D. Wong. A framework for constructing de Bruijn sequences via simple successor rules. *Discrete Math.*, 241(11):2977–2987, 2018. doi:10.1016/j.disc.2018.07.010.

14 D. Gabrić, J. Sawada, A. Williams, and D. Wong. A successor rule framework for constructing k-ary de Bruijn sequences and universal cycles. *IEEE Trans. Inform. Theory*, 66(1):679–687, 2020. doi:10.1109/TIT.2019.2928292.

15 C. Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Annalen*, 6:30–32, 1873. doi:10.1007/BF01442866.

16 Y. Huang. A new algorithm for the generation of binary de Bruijn sequences. *J. Algorithms*, 11(1):44–51, 1990. doi:10.1016/0196-6774(90)90028-D.

17 C. J. A. Jansen, W. G. Franx, and D. E. Boekee. An efficient algorithm for the generation of DeBruijn cycles. *IEEE Trans. Inform. Theory*, 37(5):1475–1478, 1991. doi:10.1109/18.133272.

18 A. Lempel. On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers. *IEEE Trans. Comput.*, C-19(12):1204–1209, 1970. doi:10.1109/T-C.1970.222859.

19 R. Mercaş. On the aperiodic avoidability of binary patterns with variables and reversals. *Theoret. Comput. Sci.*, 682:180–189, 2017. doi:10.1016/j.tcs.2016.12.022.

20 C. J. Mitchell and P. R. Wild. Constructing orientable sequences. *IEEE Trans. Inform. Theory*, 68(7):4782–4789, 2022. doi:10.1109/TIT.2022.3158645.

21 N. Rampersad and J. O. Shallit. Words that avoid reversed subwords. *J. Combin. Math. Combin. Comput.*, 54:157–164, 2005.

22 J. Sawada, J. Sears, A. Trautrim, and A. Williams. Concatenation trees: A framework for efficient universal cycle and de Bruijn sequence constructions. Arxiv preprint arXiv:2308.12405 [math.CO], available at https://arxiv.org/abs/2308.12405, 2023.

23 J. Sawada and A. Williams. Constructing the first (and coolest) fixed-content universal cycle. *Algorithmica*, 85(6):1754–1785, 2023. doi:10.1007/s00453-022-01047-2.

24 J. Sawada and D. Wong. Efficient universal cycle constructions for weak orders. *Discrete Math.*, 343(10):112022, 2020. doi:10.1016/j.disc.2020.112022.

25 N. J. A. Sloane et al. OEIS Foundation Inc. (2024), The On-Line Encyclopedia of Integer Sequences, https://oeis.org.

Exploiting New Properties of String Net Frequency for Efficient Computation

Peaker Guo
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Patrick Eades
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Anthony Wirth
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Justin Zobel
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia

Abstract

Knowing which strings in a massive text are significant – that is, which strings are common and distinct from other strings – is valuable for several applications, including text compression and tokenization. Frequency in itself is not helpful for significance, because the commonest strings are the shortest strings. A compelling alternative is *net frequency*, which has the property that strings with positive net frequency are of maximal length. However, net frequency remains relatively unexplored, and there is no prior art showing how to compute it efficiently. We first introduce a characteristic of net frequency that simplifies the original definition. With this, we study strings with positive net frequency in Fibonacci words. We then use our characteristic and solve two key problems related to net frequency. First, SINGLE-NF, how to compute the net frequency of a given string of length m, in an input text of length n over an alphabet size σ. Second, ALL-NF, given length-n input text, how to report every string of positive net frequency (and its net frequency). Our methods leverage suffix arrays, components of the Burrows-Wheeler transform, and solution to the coloured range listing problem. We show that, for both problems, our data structure has $\mathcal{O}(n)$ construction cost: with this structure, we solve SINGLE-NF in $\mathcal{O}(m + \sigma)$ time and ALL-NF in $\mathcal{O}(n)$ time. Experimentally, we find our method to be around 100 times faster than reasonable baselines for SINGLE-NF. For ALL-NF, our results show that, even with prior knowledge of the set of strings with positive net frequency, simply confirming that their net frequency is positive takes longer than with our purpose-designed method. All in all, we show that net frequency is a cogent method for identifying significant strings. We show how to calculate net frequency efficiently, and how to report efficiently the set of plausibly significant strings.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words; Theory of computation → Design and analysis of algorithms

Keywords and phrases Fibonacci words, suffix arrays, Burrows-Wheeler transform, LCP arrays, irreducible LCP values, coloured range listing

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.16

Related Version *Full Version*: `http://arxiv.org/abs/2404.12701`

Supplementary Material *Software (Source Code)*: `https://github.com/peakergzf/string-net-frequency`, archived at `swh:1:dir:c037715e983062ea4fdf0529860b61c4be940240`

Funding This work was supported by the Australian Research Council, grant number DP190102078, and an Australian Government Research Training Program Scholarship.

Acknowledgements The authors thank William Umboh for insightful discussions. The authors also thank the anonymous reviewers for their suggestions. This research was supported by The University of Melbourne's Research Computing Services and the Petascale Campus Initiative.

1 Introduction

When analysing, storing, manipulating, or working with text, identification of notable (or significant) strings is typically a key component. These notable strings could form the basis of a dictionary for compression, be exploited by a tokenizer, or form the basis of trend detection. Here, a *text* is a sequence of characters drawn from a fixed alphabet, such as a book, collection of articles, or a Web crawl. A *string* is a contiguous sub-sequence of the text; in this paper, we seek to efficiently identify notable strings.

Given a text, T, and a string, S, the *frequency* of S is the number of occurrences of S in T. The frequency of a string is inherently a basis for its significance. However, frequency is in some sense uninformative. Sometimes a shorter string is frequent only because it is part of many different longer strings, or of a frequent longer string, or of many frequent longer strings. That is, the frequency of a string may be inflated by the occurrences of the longer strings that contain it. Moreover, every substring of a string of frequency f has frequency at least f. Indeed, the most frequent string in the text has length 1.

A compelling means of identifying notable strings is via *net frequency* (NF), introduced by Lin and Yu [23]. Let $T =$ `rstkstcastarstast$` be an input text. The highlighted string `st` has frequency five. But to arrive at a more helpful notion of the frequency of the string `st`, the occurrences of `st` in *repeated longer strings* – that is, `rst` and `ast` – should be excluded, leaving one occurrence left (underlined). Defined precisely in Section 3, net frequency (NF) captures this idea; indeed, the NF of `st` in T is 1. For now, strings with positive NF are those that are repeated in the text and are *maximal* (see Theorem 4, below): if extended to either left or right the frequency of the extended string would be 1. For the underlined `st` above, the frequency of both `kst` (left) and `stc` (right) is 1.

It is worth noting the difference between a string with positive NF and a *maximal repeat* [21, 33, 35]: when extending a string with positive NF, the frequency of the extended string becomes 1, whereas when extending a maximal repeat, the frequency of the extended string decreases, but does not necessarily become 1.

Motivation. NF has been demonstrated to be useful in tasks such as Chinese phoneme-to-character (and character-to-phoneme) conversion, the determination of prosodic segments in a Chinese sentence for text-to-speech output, and Chinese toneless phoneme-to-character conversion for Chinese spelling error correction [23, 24]. NF could also be complementary to tasks such as parsing in NLP and structure discovery in genomic strings. However, even though the original paper on NF [23] suggested that "suitable indexing can be used to improve efficiency", efficient structures and algorithms for NF were not explicitly described. In this work, we bridge this gap by delving into the properties of NF. Through these properties, we introduce efficient algorithms for computing NF.

Problem definition. Throughout, T is our length-n input text and S a length-m string in T. We consider two problems relating to computing NF in T: the Single-string Net Frequency problem SINGLE-NF and the All-strings Net Frequency problem ALL-NF:

- SINGLE-NF: given a text, T, and a query string, S, report the NF of S in T.
- ALL-NF: given a text, T, identify each string that has positive NF in T. Concretely, the identification could be one of the following two forms. ALL-NF-REPORT: for each string of positive NF, *report* one occurrence and its NF; or ALL-NF-EXTRACT: *extract* a multiset, where each element is a string with positive NF and its multiplicity is its NF.

Our contribution. In this work, we first reconceptualise NF through our new characteristic that simplifies the original definition. We then apply it and identify strings with positive NF in Fibonacci words. For SINGLE-NF, we introduce an $\mathcal{O}(m+\sigma)$ time algorithm, where m is the length of the query and σ is the size of the alphabet. This is achieved via several augmentation to suffix array from LF mapping to LCP array, as well as solution to the coloured range listing problem. For ALL-NF, we establish a connection to branching strings and LCP intervals, then solve ALL-NF-REPORT in $\mathcal{O}(n)$ time, and ALL-NF-EXTRACT in $\mathcal{O}(n \log \delta)$ time, where δ is a repetitiveness measure defined as $\delta := \max\{S(k)/k : k \in [n]\}$ and $S(k)$ denotes the number of distinct strings of length k in T. The cost is bounded by making a connection to irreducible LCP values. We also conducted extensive experiments and demonstrated the efficiency of our algorithms empirically. The code for our experiments is available at `https://github.com/peakergzf/string-net-frequency`.

2 Preliminaries

Strings. Let Σ be a finite alphabet of size σ. Given a character, x, and two strings, S and T, some of their possible concatenations are written as xS, Sx, ST, and TS. If S is a substring of T, we write $S \prec T$ or $T \succ S$. Let $[n]$ denote the set $\{1, 2, \ldots, n\}$. A substring of T with starting position $i \in [n]$ and end position $j \in [n]$ is written as $T[i \ldots j]$. A substring $T[1 \ldots j]$ is called a prefix of T, and $T[i \ldots n]$ is called a suffix of T. Let T_i denote the i^{th} suffix of T, $T[i \ldots n]$. An *occurrence* in the text T is a pair of starting and ending positions $(s, e) \in [n] \times [n]$. We say (i, j) is an occurrence *of string* S if $S = T[i \ldots j]$, and i is an occurrence of S if $S = T[i \ldots i + |S| - 1]$. The *frequency* of S, denoted by $f(S)$, is the number of occurrences of S in T. A string S is *unique* if $f(S) = 1$ and is *repeated* if $f(S) \geq 2$.

Suffix arrays and Burrows-Wheeler transform. The *suffix array* (SA) [27] of T is an array of size n where $SA[i]$ stores the text position of the i^{th} lexicographically smallest suffix. For a string S, let l and r be the smallest and largest positions in SA, respectively, where S is a prefix of the corresponding suffixes $T_{SA[l]}$ and $T_{SA[r]}$. Then, the closed interval $\langle l, r\rangle$ is referred to as the *SA interval* of S. The *inverse suffix array* (ISA) of a suffix array SA is an array of length n where $ISA[i] = j$ if and only if $SA[j] = i$. The *Burrows-Wheeler transform* (BWT) [29] of T is a string of length n where $BWT[i] = T[SA[i] - 1]$ for $SA[i] > 1$ and $BWT[i] = \$$ if $SA[i] = 1$. The *LF mapping* is an array of length n where $LF[i] = ISA[SA[i]-1]$ for $SA[i] > 1$, and $LF[i] = 1$ if $SA[i] = 1$.

LCP arrays and irreducible LCP values. The *longest common prefix array* (LCP) [17, 28] is an array of length n where the i^{th} entry in the LCP array stores the length of the longest common prefix between $T_{SA[i-1]}$ and $T_{SA[i]}$, which is denoted $lcp\left(T_{SA[i-1]}, T_{SA[i]}\right)$. An entry $LCP[i]$ is called *reducible* if $BWT[i-1] = BWT[i]$ and *irreducible* otherwise. The sum of irreducible LCP values was first bounded as $\mathcal{O}(n \log n)$ [16]. Later the bound has been refined with the development of *repetitiveness measures* [31]. Let r be the number of equal-letter runs in the BWT of T. The bound on the sum of irreducible LCP values was improved [15] to $\mathcal{O}(n \log r)$. Let $S(k)$ be the number of distinct strings of length k in T, and define $\delta := \max\{S(k)/k : k \in [n]\}$ [6, 20, 36]. The bound was further improved in the following result.

▶ **Lemma 1** ([18]). *The sum of irreducible LCP values is at most $\mathcal{O}(n \log \delta)$.*

Coloured range listing. The *coloured range listing (CRL)* problem is defined as follows. Preprocess a text T of length n such that, later, given a range $i, \ldots, j$, list the position of each distinct character ("colour") in $T[i \ldots j]$. The data structure introduced in [30] lists each such position in $\mathcal{O}(1)$ time, occupying $\mathcal{O}(n \log n)$ bits of space. Compressed structures for the CRL problem have also been introduced [10].

3 A Fresh Examination of Net Frequency

In this section, we lay the foundation for efficient net frequency (NF) computation by re-examining NF and proving several properties. Before we formally define NF, we first introduce the notion of *extensions.* The proofs of the results in this section are postponed to the full version.

▶ **Definition 2** (Extensions)**.** *Given a string S and two symbols $x, y \in \Sigma$, strings xS, Sy, and xSy are called the* left, right, *and* bidirectional extension *of S, respectively. A left or right extension is also called a* unidirectional extension*. We then define the following sets of extensions: $L(S) := \{x \in \Sigma : f(xS) \geq 2\}, R(S) := \{y \in \Sigma : f(Sy) \geq 2\}$, and $B(S) := \{(x, y) \in L(S) \times R(S) : f(xSy) \geq 1\}$.*

Note that the definition of $B(S)$ does not require that string xSy needs to repeat; only the unidirectional extensions, xS and Sy, must do so.

▶ **Definition 3** (Net frequency [23])**.** *Given a string S in T, the NF of S is zero if it is unique in T; otherwise S repeats and the NF of S is defined as*

$$\phi(S) := f(S) - \sum_{x \in L(S)} f(xS) - \sum_{y \in R(S)} f(Sy) + \sum_{(x,y) \in B(S)} f(xSy).$$

The two subtraction terms discount the occurrences that are part of longer repeated strings while the addition term compensates for double counting (occurrences of xS and Sy could correspond to the same occurrence of S), an inclusion-exclusion approach. We now introduce a fresh examination of NF that significantly simplifies the original definition and will be the backbone of our algorithms for NF computation later.

▶ **Theorem 4** (Net frequency characteristic)**.** *Given a repeated string S,*

$$\phi(S) = |\{ (x, y) \in \Sigma \times \Sigma : f(xS) = 1 \text{ and } f(Sy) = 1 \text{ and } f(xSy) = 1 \}| .$$

In the original definition of NF and in our characteristics, extensions are limited to adding only one character to one side of the string. It is intriguing to explore the impact of longer extensions. Surprisingly, the analogous quantity of NF with longer extensions is equal to NF.

▶ **Lemma 5.** *Given a repeated string S, for each $k \geq 1$, we have $\phi(S) = \phi_k(S)$ where*

$$\phi_k(S) := \left|\left\{ (X, Y) \in \Sigma^k \times \Sigma^k : f(XS) = 1 \text{ and } f(SY) = 1 \text{ and } f(XSY) = 1 \right\}\right| .$$

So far the definition and properties of NF have been formulated in terms of symbols from the alphabet. To facilitate our discussion on the properties and algorithms of NF later, we switch our focus away from symbols and reformulate NF in terms of occurrences. Recall that the frequency of a string S is the number of occurrences of S. Analogously, the NF of S is the number of *net occurrences* of S.

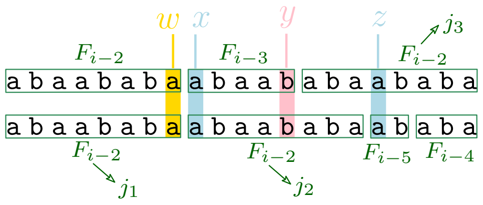


Figure 1 Illustration of proof of Theorem 8. Two factorisations of F_8 are depicted with rectangles.

▶ **Definition 6** (Net occurrence). *An occurrence* (i, j) *is a* net occurrence *if* $f(T[i \dots j]) \geq 2$, $f(T[i-1 \dots j]) = 1$, *and* $f(T[i \dots j+1]) = 1$. *When* $i = 1$, $f(T[i-1 \dots j]) = 1$ *is assumed to be true; when* $j = n$, $f(T[i \dots j+1]) = 1$ *is assumed to be true.*

When $f(xS) = 1$ and $f(Sy) = 1$, $f(xSy)$ is either 0 or 1. But when $f(T[i-1 \dots j]) = 1$ and $f(T[i \dots j+1]) = 1$, $f(T[i-1 \dots j+1])$ must be 1 and cannot be 0. Thus, the conditions in Definition 6 do not mention the bidirectional extension, $f(T[i-1 \dots j+1]) = 1$.

Net Frequency of Fibonacci Words: A Case Study

Let F_i be the i^{th} (finite) Fibonacci word over binary alphabet $\{\mathtt{a}, \mathtt{b}\}$, where $F_1 := \mathtt{b}, F_2 := \mathtt{a}$, and for each $i \geq 3$, $F_i := F_{i-1}F_{i-2}$. Note that $|F_i| = f_i$ where f_i is the i^{th} Fibonacci number. There has been an extensive line of research on Fibonacci words, from their combinatorial properties [19] to lower bounds and worst-case examples for strings algorithms [14].

In this section, we examine the NF of Fibonacci words, which later will help us obtain a lower bound on the sum of lengths of strings with positive NF in a text. Specifically, we assume $i \geq 7$, we regard F_i as our input text, and we study the net frequency of F_{i-2} and $S_i := F_{i-1}[1 \dots f_{i-1} - 2]$ in F_i.

Net Frequency of F_{i-2} in F_i

We begin by introducing some basic concepts in combinatorics on words [25]. A nonempty word u is a *repetition* of a word w if there exist words x, y such that $w = xu^k y$ for some integer $k \geq 2$. When $k = 2$, the repetition is called a *square*. A word v that is both a prefix and a suffix of w, with $v \neq w$, is called a *border* of w. Stronger results on the borders and squares of F_i have been introduced before [7, 13], but for our purposes, the following suffices.

▶ **Observation 7.** *F_{i-2} is a border and a square of F_i.*

Proof. We apply the recurrence and factorise F_i as follows. Occurrences of F_{i-2} as a border or a square of F_i are underlined. $F_i = F_{i-1}\ F_{i-2} = \underline{F_{i-2}}\ F_{i-3}\ \underline{F_{i-2}} = F_{i-2}\ F_{i-3}\ F_{i-3}\ F_{i-4} = F_{i-2}\ F_{i-3}\ F_{i-4}\ F_{i-5}\ F_{i-4} = \underline{F_{i-2}}\ \underline{F_{i-2}}\ F_{i-5}\ F_{i-4}$. ◀

▶ **Theorem 8.** *$\phi(F_{i-2}) \geq 1$.*

Proof. The proof is illustrated in Figure 1. In the following two factorisations of F_i, $F_i = F_{i-2}\ F_{i-3}\ F_{i-2}$ and $F_i = F_{i-2}\ F_{i-2}\ F_{i-5}\ F_{i-4}$, consider j_1, j_2, and j_3, three occurrences of F_{i-2}. Let w and y be the left extension characters of j_2 and j_3, respectively, and let x and z be the right extension characters of j_1 and j_2, respectively. Using the factorisation $F_i = F_{i-2}\ F_{i-3}\ F_{i-2}$, observe that $w = F_{i-2}[f_{i-2}]$, $x = F_{i-2}[1]$, and $y = F_{i-3}[f_{i-3}]$. Using the factorisation $F_i = F_{i-2}\ F_{i-2}\ F_{i-5}\ F_{i-4}$, we have $z = F_{i-5}[1]$. Thus, $x = z = \mathtt{a}$, and $w \neq y$ because the last character of consecutive Fibonacci words alternates. Therefore, j_1 and j_2 are not net occurrences of F_{i-2} in F_i and only j_3 is. ◀

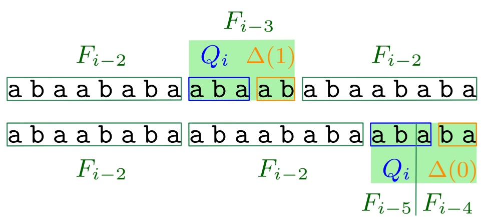


Figure 2 Illustration of Lemma 10 with F_8. Note that $F_{i-5} =$ `ab` and $F_{i-4} =$ `aba`.

Net Frequency of S_i in F_i

In the recurrence of Fibonacci word, F_{i-2} is appended to F_{i-1}, $F_i = F_{i-1}\ F_{i-2}$. When we reverse the order of the concatenation and prepend F_{i-2} to F_{i-1}, for example, notice that $F_6\ F_5 =$ `abaababa|abaab` and $F_5\ F_6 =$ `abaab|abaababa` only differ in the last two characters. Such property is referred to as *near-commutative* in [34]. In our case, we characterise the string that is *invariant* under such reversion with Q_i in the following definition.

▶ **Definition 9** (Q_i and $\Delta(j)$). *Let $Q_i := F_{i-5}\ F_{i-6} \cdots F_3\ F_2$ be the concatenation of $i-6$ consecutive Fibonacci words in decreasing length. For $j \in \{0,1\}$, we define $\Delta(j) :=$ `ba` if $j = 0$, and $\Delta(j) :=$ `ab` otherwise.*

In Figure 2, $F_{i-4}\ F_{i-5}$ and $F_{i-5}\ F_{i-4}$ only differ in the last two characters, and their common prefix is Q_i. The alternation between `ab` and `ba` was also observed in [8], but their focus was on capturing the length-2 suffix appended to the palindrome $F_i[1 \dots f_i - 2]$.

Observe that $F_{i-3} = F_{i-4}\ F_{i-5}$, the invariant discussed earlier is captured as follows.

▶ **Lemma 10.** $F_{i-3} = Q_i\ \Delta\,(1 - (i \bmod 2))$ *and* $F_{i-5}\ F_{i-4} = Q_i\ \Delta(i \bmod 2)$.

Proof. Let $P(i)$ be the statement $F_{i-3} = Q_i\ \Delta(1 - (i \bmod 2))$. We prove $P(i)$ by strong induction. Base case: observe that $F_{7-3} =$ `aba`, $Q_7 = F_2 =$ `a`, and $\Delta(1-(7 \bmod 2)) = \Delta(0) =$ `ba`. Inductive step: consider $k > 7$, assume that $P(j)$ holds for every $j \le k$. We now prove $P(k+1)$ holds. First, $F_{k-2} = F_{k-3}\ F_{k-4} = F_{k-4}\ F_{k-5}\ F_{k-4}$. Then, based on our inductive hypothesis, $F_{k-4} = Q_{k-1}\ \Delta(1-(k-1) \bmod 2) = F_{k-6}\ F_{k-7} \cdots F_3\ F_2\ \Delta(1-(k-1) \bmod 2)$. Substituting the second F_{k-4} in F_{k-2}, we have $F_{k-2} = F_{k-4}\ F_{k-5}\ F_{k-6}\ F_{k-7} \cdots F_3\ F_2\ \Delta(1-(k-1) \bmod 2) = Q_{k+1}\ \Delta(1-(k+1) \bmod 2)$. By induction, $P(i)$ holds for all i. $F_{i-5}\ F_{i-4} = Q_i\ \Delta(i \bmod 2)$ is proved similarly and the proof is postponed to the full version. ◀

Previously we defined S_i as the length $(f_{i-1} - 2)$ prefix of F_{i-1}, now we can see that this is to remove Δ ($|\Delta| = 2$). With Lemma 10, we now present the main result on the NF of S_i.

▶ **Theorem 11.** $\phi(S_i) \ge 2$.

Proof. It follows from Lemma 10 that $F_{i-1} = F_{i-2}\ F_{i-3} = F_{i-2}\ Q_i\ \Delta(1-(i \bmod 2))$. Then, $S_i = F_{i-1}[1 \dots f_{i-1} - 2] = F_{i-2}\ Q_i$. Consider the two occurrences of S_i, observe that the right extension characters of these occurrences are different: $\Delta(1-(i \bmod 2))[1] \ne \Delta(i \bmod 2)[1]$. (In Figure 2, $\Delta(1)[1] \ne \Delta(0)[1]$.) Therefore, both occurrences are net occurrences. ◀

▶ Remark 12. Theorem 8 and Theorem 11 show that there are at least three net occurrences in F_i (one of F_{i-2} and two of S_i). Empirically, we have verified that these are the only three net occurrences in F_i for each i until a reasonably large i. Future work can be done to prove this tightness.

Algorithm 1 for SINGLE-NF.

```
   Input  : $S \leftarrow$ a string;
1  $\phi \leftarrow 0$; // the NF of $S$
2  $\langle l, r \rangle \leftarrow$ the SA interval of $S$;
3  for $i \leftarrow CRL_{BWT}(l, r)$ do
4      $j \leftarrow LF[i]$;
5      if $|S| = \ell(i)$ and $|S| \geq \ell(j)$ then
           // see Theorem 15
6          $\phi \leftarrow \phi + 1$;
7  return $\phi$;
```

Algorithm 2 for ALL-NF-EXTRACT.

```
1  $\mathcal{N} \leftarrow \emptyset$;
   // $\mathcal{N}$ is a multiset of strings with positive NF.
   We write $\mathcal{N}|_S$ for the NF of $S$ in $\mathcal{N}$.
2  for $i \leftarrow 1, \ldots, n$ do
3      $j \leftarrow LF[i]$;
4      if $\ell(i) \geq \ell(j)$ then
5          $S \leftarrow T[C_i]$; // see Definition 16
6          $\mathcal{N}|_S \leftarrow \mathcal{N}|_S + 1$;
7  return $\mathcal{N}$;
```

4 New Algorithms for Net Frequency Computation

Our reconceptualisation of NF provides a basis for computation of NF in practice. In this section, we introduce our efficient approach for NF computation. The proofs of the results in this section are postponed to the full version.

4.1 SINGLE-NF Algorithm

To compute the NF of a query string S, it is sufficient to enumerate the SA interval of S and count the number of net occurrences of S. To determine which occurrence is a net occurrence, we need to check if the relevant extensions are unique. Locating the occurrences of the left extensions is achieved via LF mapping and checking for uniqueness is assisted by the LCP array. For convenience, we define the following. We then observe how to determine the uniqueness of a string as a direct consequence of a property of the LCP array.

▶ **Definition 13.** *For each* $1 \leq i \leq n-1$, $\ell(i) := \max(LCP[i], LCP[i+1])$.

▶ **Observation 14** (Uniqueness characteristic). *Let* $\langle l, r \rangle$ *be the SA interval of* S*, and let* $l \leq i \leq r$*, then* S *is unique if and only if* $|S| > \ell(i)$*, and* S *repeats if and only if* $|S| \leq \ell(i)$.

Now, we present the main result that underpins our SINGLE-NF algorithm.

▶ **Theorem 15** (Net occurrence characteristic). *Given an occurrence* (s, e) *in* T*, let* $S := T[s \ldots e]$*,* $i := ISA[s]$*, and* $j := LF[i]$*. Then,* (s, e) *is a net occurrence if and only if* $|S| = \ell(i)$ *and* $|S| \geq \ell(j)$.

Let $\langle l, r \rangle$ be the SA interval of S and let f be the frequency of S. With Theorem 15, we have an $\mathcal{O}(m + f)$ time SINGLE-NF algorithm by exhaustively enumerating $\langle l, r \rangle$. Note that it takes $\mathcal{O}(m)$ time to locate $\langle l, r \rangle$ [1] and $\mathcal{O}(f)$ to enumerate the interval. However, with the data structure for CRL, we can improve this time usage. Specifically, observe that if we preprocess the BWT of T for CRL, then, instead of enumerating each position within $\langle l, r \rangle$, we only need to examine each position that corresponds to a distinct character of $BWT[l \ldots r]$. Observe that each such character is precisely a distinct left extension character. We write $CRL_{BWT}(l, r)$ for such set of positions. Our algorithm for SINGLE-NF is presented in Algorithm 1, which takes $\mathcal{O}(m + \sigma)$ time where σ is a loose upper bound on the number of distinct characters in $BWT[l \ldots r]$.

4.2 ALL-NF Algorithms

From Theorem 15, observe that for each position in the suffix array, only one string occurrence could be a net occurrence, namely, the occurrence that corresponds to a repeated string with a unique right extension. This occurrence will be a net occurrence if the repeated string also has a unique left extension. For convenience, we define the following.

▶ **Definition 16** (Net occurrence candidate)**.** *For each $i \in [n]$, let $C_i := (SA[i], SA[i] + \ell(i) - 1)$ be the* net occurrence candidate *at position i. We write $T[C_i]$ for the string $T[SA[i] \dots SA[i] + \ell(i) - 1]$, the* candidate string *at position i.*

In our approach for solving SINGLE-NF, Theorem 15 is applied within a SA interval. To solve ALL-NF, there is an appealing direct generalisation that would apply Theorem 15 to each candidate string in the entire suffix array. However, there is a confound: consecutive net occurrence candidates in the suffix array do not necessarily correspond to the same string. To mitigate this confound, we introduce a hash table representing a multiset that maps each string with positive NF to a counter that keeps track of its NF. With this, Algorithm 2 iterates over each row of the suffix array, identifies the only net occurrence candidate C_i, then increment the NF of $T[C_i]$, if C_i is indeed a net occurrence.

▶ Remark 17. Algorithm 2 is natural for ALL-NF-EXTRACT, but cannot support ALL-NF-REPORT without the extraction first. In contrast, our second ALL-NF method, Algorithm 3, which we will discuss next, supports both ALL-NF-EXTRACT and ALL-NF-REPORT without having to complete the other first. ⌟

We next consider the only strings that could have positive NF. A string S is *branching* [26] in T if S is the longest common prefix of two distinct suffixes of T.

▶ **Lemma 18.** *For every non-branching string S, $\phi(S) = 0$.*

Now, we make the following observation, which aligns with the previous result.

▶ **Observation 19.** *Each net occurrence candidate is an occurrence of a branching string.*

The SA intervals of branching strings are better known as the *LCP intervals* in the literature.

▶ **Definition 20** (LCP interval [1])**.** *An* LCP interval *of LCP value ℓ, written as ℓ-$\langle l, r\rangle$, is an interval $\langle l, r\rangle$ that satisfies the following: $LCP[l] < \ell$, $LCP[r+1] < \ell$, for each $l + 1 \le i \le r$, $LCP[i] \ge \ell$, and there exists $l + 1 \le k \le r$ such that $LCP[k] = \ell$,*

Traversing the LCP intervals is a standard task and can be accomplished by a stack-based algorithm: examples include Figure 7 in [17] and Algorithm 4.1 in [1]. These algorithms were originally conceived for emulating a bottom-up traversal of the internal nodes in a suffix tree using a suffix array and an LCP array. In a suffix tree, an internal node has multiple child nodes and thus its corresponding string is branching.

Thus, Algorithm 3 is an adaptation of the LCP interval traversal algorithms in [1, 17] with an integration of our NF computation. Notice that in the i^{th} iteration of the algorithm, we set Boolean variable `for_next` to true if $\ell(i) = LCP[i+1]$. That is, `for_next` is true if the current net occurrence candidate that we are examining corresponds to an LCP interval that will be pushed onto the stack in the next iteration, $i + 1$.

Note that the correctness of Algorithms 1–3 follows from the correctness of Theorem 15.

Algorithm 3 for ALL-NF-REPORT or ALL-NF-EXTRACT.

```
$s \leftarrow \emptyset$;
// an empty stack; the standard stack operations used in the algorithm are: s.push( ), s.top( ), and s.pop( )
$s.push(\langle 0,0,0\rangle)$;
// an LCP interval len-$\langle lb, rb\rangle$ with NF $\phi$ is written as $\langle len, lb, \phi\rangle$; note that rb is not used in this algorithm
// $\langle 0,0,0\rangle$ is the LCP interval for the empty string
for_next $\leftarrow$ false;
// for_next = true indicates that the current net occurrence is for the interval that will be pushed onto the stack in the next iteration
function process_interval($I$):
    // I: an LCP interval
    if $I.\phi > 0$ then
        $j \leftarrow SA[I.lb]$;
        $S \leftarrow T[j \ldots j + I.len]$;
        // to be reported or extracted
        $\phi(S) = I.\phi$;
for $i \leftarrow 2 \ldots n$ do
    $lb \leftarrow i - 1$;
    while $LCP[i] < s.top().len$ do
        $I \leftarrow s.pop()$;
        process_interval($I$);
        $lb \leftarrow I.lb$;
    if $LCP[i] > s.top().len$ then
        $s.push(\langle LCP[i], lb, 0\rangle)$;
        if for_next then
            $s.top().\phi \leftarrow s.top().\phi + 1$;
            for_next $\leftarrow$ false;
    $j \leftarrow LF[i]$;
    if $\ell(i) \geq \ell(j)$ then
        if $LCP[i] = \ell(i)$ then
            $s.top().\phi \leftarrow s.top().\phi + 1$;
        else for_next = true;
while s is not empty do
    process_interval($s.pop()$);
```

Analysis of the ALL-NF algorithms. When Algorithm 3 is used for ALL-NF-REPORT, it runs in $\mathcal{O}(n)$ time in the worst case. We can also use Algorithm 3 for ALL-NF-EXTRACT. To analyse the asymptotic cost for ALL-NF-EXTRACT (using either Algorithm 2 or Algorithm 3), we first define the following.

▶ **Definition 21.** *Given an input text T, let $\mathcal{S} := \{S \prec T : \phi(S) > 0\}$ be the set of strings with positive NF in T. Then, we define $N := \sum_{S \in \mathcal{S}} |S|$ and $L := \sum_{S \in \mathcal{S}} \phi(S) \cdot |S|$.*

With these definitions, we first present the following bounds.

▶ **Lemma 22.** $\sum_{S \in \mathcal{S}} \phi(S) \leq n$ *and* $|\mathcal{S}| \leq n$.

For ALL-NF-EXTRACT, when a hash table is used, Algorithm 2 takes $\mathcal{O}(L)$ time while Algorithm 3 only takes $\mathcal{O}(N)$, both in expectation. Note that for each $S \in \mathcal{S}$, in Algorithm 2, S is hashed $\phi(S)$ times, but in Algorithm 3, S is only hashed once.

Since $N \leq L$, a lower bound on N is also a lower bound on L, and an upper bound on L is also an upper bound on N. The next two results present a lower bound on N and an upper bound on L.

▶ **Lemma 23.** $N \in \Omega(n)$.

Proof. We use our results on Fibonacci words. From Theorem 8 and Theorem 11, $N(F_i) \geq |F_{i-2}| + |F_{i-2}\, Q_i| = f_{i-2} + \left(f_{i-2} + \sum_{j=2}^{i-5} f_j\right)$. Using the equality $\sum_{j=1}^{i} f_j = f_{i+2} - 1$, we have $L(F_i) \geq f_{i-2} + (f_{i-2} + f_{i-3} - 1 - f_1)$. With further simplification, $N(F_i) \geq f_i - 2$. ◀

We can similarly show that $L(F_i) \geq f_i + f_{i-2} - 2$. Next, we present an upper bound on L.

▶ **Theorem 24.** $L \in \mathcal{O}(n \log \delta)$.

Proof. First observe that $L = \sum_{i \in [n]\,:\, net\_occ(C_i)} \ell(i)$ where $net\_occ(C_i)$ denotes that C_i is a net occurrence. Thus, L can be expressed as the sum of certain LCP values. Next, when C_i is a net occurrence, its left extension is unique, which means $LCP[i]$ or $LCP[i+1]$ is irreducible. Notice that each irreducible $LCP[i]$ contributes to L at most twice due to C_i or C_{i-1}. It follows that L is at most twice the sum of irreducible LCP values. Using Lemma 1, we have the desired result. ◀

Table 1 Statistics for each dataset, T. The first three datasets are news collections. Definition 21 explains $\mathcal{S}$, N, and L. As described in Section 5.1, it is practical to bound the length of each query by 35: in parentheses, therefore, we also include the values of N and L with a length upper bound (u.b.) of 35 on the individual strings. That is, we replace $\mathcal{S}$ with $\{S \prec T : \phi(S) > 0 \text{ and } |S| \leq 35\}$. Also recall that L and N are used in the asymptotic costs of our ALL-NF algorithms.

T	n ($\times 10^6$)	$\lvert\Sigma\rvert$	$\lvert\mathcal{S}\rvert$	N (with u.b.)	L (with u.b.)
NYT	435.3	89	$0.1n$	$1.7n$ ($1.4n$)	$2.7n$ ($2.2n$)
APW	152.2	92	$0.1n$	$1.6n$ ($1.3n$)	$2.6n$ ($2.1n$)
XIE	98.9	91	$0.1n$	$1.7n$ ($1.4n$)	$2.8n$ ($2.2n$)
DNA	505.9	4	$0.001n$	$0.5n$ ($0.005n$)	$1.1n$ ($0.007n$)

5 Experiments

In this section, we evaluate the effectiveness of our SINGLE-NF and ALL-NF algorithms empirically. The datasets used in our experiments are news collections from TREC 2002 [37] and DNA sequences from Genbank [5]. Statistics are in Table 1. Relatively speaking, there are far fewer strings with positive NF in the DNA data because, with a smaller alphabet, the extensions of strings are less variant, and DNA is more nearly random in character sequence than is English text. All of our experiments are conducted on a server with a 3.0GHz Intel(R) Xeon(R) Gold 6154 CPU. All the algorithms are implemented in C++ and GCC 11.3.0 is used. Our implementation is available at `https://github.com/peakergzf/string-net-frequency`.

5.1 SINGLE-NF Experiments

For the news datasets, each query string is randomly selected as a concatenation of several consecutive space-delimited strings. We set a query-string length lower bound of 5 because we regard very short strings as not noteworthy. We set a practical upper bound of 35 because there are few strings longer than 35 with positive NF based on our preliminary experimental results. For DNA, each query string is selected by randomly choosing a start and end position from the text.

Algorithms. As discussed in Section 1, there are no prior efficient algorithms for SINGLE-NF. Thus, we came up with two reasonable baselines, CSA and HSA, and compare their performance against our new efficient algorithms, CRL and ASA.

- CRL: presented in Algorithm 1. We implement the algorithm for coloured range listing (CRL) following [30], which uses structures for *range minimum query* [9].
- ASA: removing the CRL augmentation from Algorithm 1, but keeping all other augmentations, hence the name *augmented suffix array* (ASA). Specifically, we replace "$CRL_{BWT}(l, r)$" with "$\langle l, r\rangle$" in Line 3 of Algorithm 1.
- CSA: algorithmically the same as ASA, but the data structure used is the *compressed suffix array* (CSA) [32]. We use the state-of-the-art implementation of CSA from the SDSL library (`https://github.com/simongog/sdsl-lite`). Specifically, their *Huffman-shaped wavelet tree* [11, 12] was chosen based on our preliminary experimental results.
- HSA: *Hash table-augmented suffix array* (HSA) is a naive baseline approach that does not use LF, LCP, or CRL, but only augments the suffix array with hash tables to maintain the frequencies of the extensions. These hash tables are later used to determine if an extension is unique or not.

Table 2 Average SINGLE-NF query time (in microseconds) over all the queries, repeated queries ($f \geq 2$), and queries with positive NF ($\phi > 0$). The query set from NYT has 2×10^6 queries in total, 38.3% repeated, while 1.4% have positive NF. The query set from DNA has 3×10^6 queries in total, 51.9% repeated, while 2.5% have positive NF.

Dataset	Algorithm	All	$f \geq 2$	$\phi > 0$
NYT	CRL	**3.9**	**7.3**	**12.6**
	ASA	9.4	21.4	39.7
	HSA	695.0	1813.9	3755.4
	CSA	1002.1	2595.7	4884.3
DNA	CRL	**6.8**	**10.1**	5.5
	ASA	64.9	122.4	**3.3**
	HSA	5655.5	10884.9	11.8
	CSA	6348.6	12209.0	10.9

The asymptotic running times of CRL, ASA, CSA, and HSA are $\mathcal{O}(m+\sigma), \mathcal{O}(m+f), \mathcal{O}(m+f\log\sigma)$, and $\mathcal{O}(m+f\cdot\sigma)$, respectively, where σ is the size of the alphabet and the query has length m and frequency f.

Comparing CRL against ASA, we expect CRL to be faster for more frequent queries as ASA needs to enumerate the entire SA interval of the query string while CRL does not. ASA is compared against CSA to illustrate the trade-off between query time and space usage: ASA is expected to be faster while CSA is expected to be more space-efficient, and indeed this trade-off is observed in our experiments. We also compare ASA against HSA to demonstrate the speedup provided by the augmentations of LF and LCP.

Results. The average query time of each algorithm is presented in Table 2. Since the results from the three news datasets exhibit similar behaviours, only the results from NYT are included: henceforth, NYT is the representative for the three news datasets. Overall, our approaches, CRL and ASA, outperform the baseline approaches, HSA and CSA, on both NYT and DNA, but all the algorithms are slower on the DNA data because the query strings are much more frequent. Notably, CRL outperforms the baseline by a factor of up to almost 1000, across all queries, validating the improvement in the asymptotic cost. Since non-existent and unique queries have zero NF by definition, we next specifically look at the results on repeated queries.

All approaches are slower when the query string is repeated, as further NF computation is required after locating the string in the data structure. For this reason we additionally report results on queries with positive NF. For NYT, similar relative behaviours are observed, but for DNA, all algorithms are significantly faster, likely because strings with positive NF on DNA data are shorter and have much lower frequency. For the same reason of queries being less frequent, ASA is faster than CRL on DNA queries with positive NF because the advantage of CRL over ASA is more apparent when the queries are more frequent.

Further results on CRL and ASA. Previously we have seen that, empirically, the augmentation of CRL accelerates our SINGLE-NF algorithm, but is that the case for queries of different frequency and length? We now investigate how query string frequency and length contribute to SINGLE-NF query time of CRL and ASA.

For NYT we do not consider strings with frequency greater than 2000, as we observe that these are rare outliers that obscure the overall trend. For each frequency $f \in [0, 2000]$ (or length $l \in [5, 35]$) and each algorithm A, a data point is plotted as the average time taken

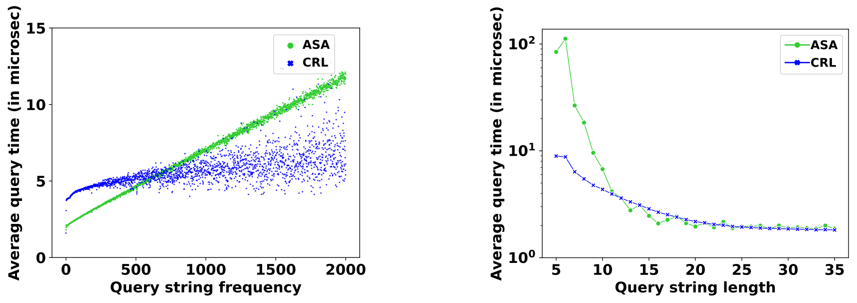


Figure 3 Average SINGLE-NF query time (in microseconds) of ASA and CRL against query string frequency (left) and length (right) on the NYT dataset. Note that the y-axis on the right is scaled logarithmically.

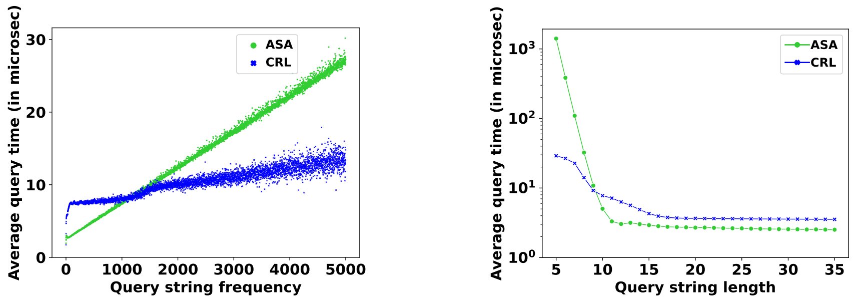


Figure 4 Average SINGLE-NF query time (in microseconds) of ASA and CRL against query string frequency (left) and length (right) on the DNA dataset. Note that the y-axis on the right is scaled logarithmically.

by A over all the query strings with frequency f (or length l). Since there are far more data points in the frequency plot than the length plot, we use a scatter plot for frequency (left of Figure 3) while a line plot for length (right of Figure 3). For frequency, as we anticipated, CRL is faster than ASA on more frequent queries. Empirically, on NYT, the turning point seems to be around 700. Note that the plot for CRL seems more scattered, likely because its time usage does not depend on frequency, but depends on query length. For length, on very short queries, CRL is faster because these queries are highly frequent. Then, generally, query time does not increase as the query strings become longer because they tend to correspondingly become less frequent. This suggests that length is not as significant as frequency in affecting the query time of these approaches.

We similarly examine the effect of frequency and length on ASA and CRL query time using the DNA data. As the query strings are more frequent in DNA than in NYC, we use a higher frequency upper bound of 5000 and the results are presented in Figure 4. The most notable difference between the DNA and NYT is that the former has a much smaller alphabet. Thus, the gap between ASA and CRL becomes more evident for more frequent queries. Additionally, it is notable that the frequency plot for CRL on the DNA dataset appears less scattered compared to the NYT plot, also because of a much smaller alphabet.

▶ Remark 25. Although asymptotically CRL is faster than ASA, we have seen that empirically ASA is faster on less frequent queries. This suggests a hybrid algorithm that switches between ASA and CRL depending on the frequency of the query string.

Table 3 Asymptotic cost and average time (in seconds) for ALL-NF. Build involves building an augmented suffix array including the off-the-shelf suffix array, the LCP array, and the LF mapping of text T. See Definition 21 for L and N. Recall that $N \leq L$ and $L \in \mathcal{O}(n \log \delta)$.

Task	Approach	Cost	Average time			
			NYT	APW	XIE	DNA
Build	prior alg.	$\mathcal{O}(n)$	186.7	61.3	39.0	219.6
Extract	Alg. 2	$\mathcal{O}(L)$	38.8	13.6	8.6	6.6
	Alg. 3	$\mathcal{O}(N)$	100.1	33.2	21.9	76.2
Report	Alg. 2	$\mathcal{O}(L+n)$	65.6	20.8	14.3	6.7
	Alg. 3	$\mathcal{O}(n)$	231.6	81.2	52.6	77.4

5.2 ALL-NF Experiments

In this section, we present the analyse and empirical results for the two tasks ALL-NF-REPORT and ALL-NF-EXTRACT. In this setting, each dataset from Table 1 is taken directly an as input text, without having to generate queries. Each reported time is an average of five runs. As seen in Table 3, for ALL-NF-EXTRACT, Algorithm 2 is consistently faster than Algorithm 3 in practice, even though $L \geq N$ (see Definition 21). We believe this is because Algorithm 2 is more cache-friendly and does not involve stack operations. Each algorithm is slower for ALL-NF-REPORT than ALL-NF-EXTRACT, likely due to random-access requirements. For Algorithm 2, although DNA is the largest dataset, the method is faster than on other datasets because there are far fewer strings with positive NF. However, this is not the case for Algorithm 3, because it has to spend much time on other operations, regardless of whether an occurrence is a net occurrence.

Comparing these results to those of the SINGLE-NF methods, observe that, for NYT, calculation of NF for each string with $\phi > 0$ takes on average 12.6 microseconds, or a total of around 548.5 seconds for the complete set of such strings – which is only possible if the set of these strings is known before the computation begins. Using ALL-NF, these NF values can be determined in about 39 seconds for extraction and a further 65 seconds to report.

6 Conclusion and Future Work

Net frequency is a principled method for identifying which strings in a text are likely to be significant or meaningful. However, to our knowledge there has been no prior investigation of how it can be efficiently calculated. We have approached this challenge with fresh theoretical observations of NF's properties, which greatly simplify the original definition. We then use these observations to underpin our efficient, practical algorithmic solutions, which involve several augmentations to the suffix array, including LF mapping, LCP array, and solutions to the colour range listing problem. Specifically, our approach solves SINGLE-NF in $\mathcal{O}(m + \sigma)$ time and ALL-NF in $\mathcal{O}(n)$ time, where n and m are the length of the input text and a string, respectively, and σ is the size of the alphabet. Our experiments on large texts showed that our methods are indeed practical.

We showed that there are at least three net occurrences in a Fibonacci word, F_i, and verified that these are the only three for each i until reasonably large i. Proving there are exactly three is an avenue of future work. We also proved that $\Omega(n) \leq N \leq L \leq \mathcal{O}(n \log \delta)$. Closing this gap remains an open problem. Another open question is determining a lower bound for SINGLE-NF. We have focused on static text with exact NF computation in this work. It would be interesting to address dynamic and streaming text and to consider how approximate NF calculations might trade accuracy for time and space usage improvements. Future research could also explore how *bidirectional indexes* [2, 3, 4, 22] can be adapted for NF computation.

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. `doi:10.1016/S1570-8667(03)00065-0`.

2 Yuma Arakawa, Gonzalo Navarro, and Kunihiko Sadakane. Bi-directional r-indexes. In *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 11:1–11:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.CPM.2022.11`.

3 Djamal Belazzougui and Fabio Cunial. Smaller fully-functional bidirectional BWT indexes. In *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2020. `doi:10.1007/978-3-030-59212-7_4`.

4 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler Transform. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2013. `doi:10.1007/978-3-642-40450-4_12`.

5 Dennis A. Benson, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, James Ostell, Kim D. Pruitt, and Eric W. Sayers. Genbank. *Nucleic Acids Research*, 46(Database-Issue):D41–D47, 2018. `doi:10.1093/nar/gkx1094`.

6 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):8:1–8:39, 2021. `doi:10.1145/3426473`.

7 Larry J. Cummings, D. Moore, and J. Karhumäki. Borders of Fibonacci strings. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 20:81–88, 1996.

8 Aldo de Luca. A combinatorial property of the Fibonacci words. *Information Processing Letters*, 12(4):193–195, 1981. `doi:10.1016/0020-0190(81)90099-5`.

9 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. `doi:10.1137/090779759`.

10 Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013. `doi:10.1016/j.tcs.2012.08.004`.

11 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. `doi:10.1007/978-3-319-07959-2_28`.

12 Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *ACM Journal of Experimental Algorithmics*, 18, 2013. `doi:10.1145/2444016.2461327`.

13 Costas S. Iliopoulos, Dennis W. G. Moore, and William F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1-2):281–291, 1997. `doi:10.1016/S0304-3975(96)00141-7`.

14 Hiroe Inoue, Yoshiaki Matsuoka, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Factorizing strings into repetitions. *Theory of Computing Systems*, 66(2):484–501, 2022. `doi:10.1007/S00224-022-10070-3`.

15 Juha Kärkkäinen, Dominik Kempa, and Marcin Piatkowski. Tighter bounds for the sum of irreducible LCP values. *Theoretical Computer Science*, 656:265–278, 2016. `doi:10.1016/j.tcs.2015.12.009`.

16 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. `doi:10.1007/978-3-642-02441-2_17`.

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 *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. `doi:10.1007/3-540-48194-X_17`.

18 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler Transform conjecture. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1002–1013. IEEE, 2020. `doi:10.1109/FOCS46700.2020.00097`.

19 Kaisei Kishi, Yuto Nakashima, and Shunsuke Inenaga. Largest repetition factorization of Fibonacci words. In *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 284–296. Springer, 2023. `doi:10.1007/978-3-031-43980-3_23`.

20 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023. `doi:10.1109/TIT.2022.3224382`.

21 M. Oguzhan Külekci, Jeffrey Scott Vitter, and Bojian Xu. Efficient maximal repeat finding using the Burrows-Wheeler Transform and wavelet tree. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 9(2):421–429, 2012. `doi:10.1109/TCBB.2011.127`.

22 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36. IEEE Computer Society, 2009. `doi:10.1109/BIBM.2009.42`.

23 Yih-Jeng Lin and Ming-Shing Yu. Extracting Chinese frequent strings without dictionary from a Chinese corpus and its applications. *Journal of Information Science and Engineering*, 17(5):805–824, 2001. URL: `https://jise.iis.sinica.edu.tw/JISESearch/pages/View/PaperView.jsf?keyId=86_1308`.

24 Yih-Jeng Lin and Ming-Shing Yu. The properties and further applications of Chinese frequent strings. *International Journal of Computational Linguistics and Chinese Language Processing*, 9(1), 2004. URL: `http://www.aclclp.org.tw/clclp/v9n1/v9n1a7.pdf`.

25 M. Lothaire. *Combinatorics on words, Second Edition.* Cambridge mathematical library. Cambridge University Press, 1997.

26 Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. In *Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000, Montreal, Canada, June 21-23, 2000, Proceedings*, volume 1848 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2000. `doi:10.1007/3-540-45123-4_27`.

27 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`.

28 Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2004. `doi:10.1007/978-3-540-27810-8_32`.

29 Burrows Michael and Wheeler David. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*, 1994.

30 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 657–666. ACM/SIAM, 2002. URL: `http://dl.acm.org/citation.cfm?id=545381.545469`.

31 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2022. `doi:10.1145/3434399`.

32 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Computing Surveys*, 54(2):26:1–26:32, 2022. `doi:10.1145/3432999`.

33 Julian Pape-Lange. On extensions of maximal repeats in compressed strings. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 27:1–27:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.CPM.2020.27`.

34 Giuseppe Pirillo. Fibonacci numbers and words. *Discrete Mathematics*, 173(1-3):197–207, 1997. `doi:10.1016/S0012-365X(94)00236-C`.

35 Mathieu Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165–169, 2001. `doi:10.1016/S0020-0190(01)00152-1`.

36 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. `doi:10.1007/s00453-012-9618-6`.

37 Ellen M. Voorhees. Overview of TREC 2003. In *Proceedings of The Twelfth Text REtrieval Conference, TREC 2003, Gaithersburg, Maryland, USA, November 18-21, 2003*, volume 500-255 of *NIST Special Publication*, pages 1–13. National Institute of Standards and Technology (NIST), 2003. URL: `http://trec.nist.gov/pubs/trec12/papers/OVERVIEW.12.pdf`.

Closing the Gap: Minimum Space Optimal Time Distance Labeling Scheme for Interval Graphs

Meng He
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Kaiyu Wu
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

We present a distance labeling scheme for an interval graph on n vertices that uses at most $3 \lg n + \lg \lg n + O(1)$ bits per vertex to answer distance queries, which ask for the distance between two given vertices, in constant time. Our labeling scheme improves the distance labeling scheme of Gavoille and Paul for connected interval graphs which uses at most $5 \lg n + O(1)$ bits per vertex to achieve constant query time. Our improved space cost matches a lower bound proven by Gavoille and Paul within additive lower order terms and is thus optimal. Based on this scheme, we further design a $6 \lg n + 2 \lg \lg n + O(1)$ bit distance labeling scheme for circular-arc graphs, with constant distance query time, which improves the $10 \lg n + O(1)$ bit distance labeling scheme of Gavoille and Paul.

We give a $n/2 + O(\lg^2 n)$ bit labeling scheme for chordal graphs which answers distance queries in $O(1)$ time. The best known lower bound is $n/4 - o(n)$ bits.

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

Keywords and phrases Distance Labeling, Interval Graph, Circular-Arc Graph, Chordal Graph

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.17

Funding This work was supported by NSERC.

1 Introduction

A notion closely related to that of a centralized data structure for computing a query is the notion of a *labeling scheme*. Formally introduced by Peleg [29], a labeling scheme assigns a relatively short label to each vertex (using an encoder function), and to answer a query, uses only the labels of the vertices involved in the query (using a decoder function). Thus a labeling scheme can be viewed as a distributed form of a data structure, where we split the data structure among the vertices of the graph. The distributed nature of the data structure is highly applicable in distributed settings, where the computation only has access to the data stored at the node and not the overall topology of the network (nor the data at other nodes). By distributing the data structure, we avoid large centralized data structures, which are costly and often will not fit in faster levels of memory. Furthermore, the length of the labels are important as the labels will need to be transmitted between the nodes of a network. Thus the quality of a labeling scheme is measured as the worst case label length (i.e. we wish to split the data structure as evenly as possible) and the worst case time to decode the labels to answer the query. One important property to note is that a labeling scheme can be trivially converted to a more traditional data structure, by simply storing all the labels. Thus the total space of a labeling scheme will be at least as much as the space for an optimal data structure. However, it is often the case that labeling schemes will use more overall space than the optimal data structure due to the distributed nature of the model.

Many operations on graphs and trees have been considered in the labeling model. For instance, labeling schemes computing adjacency in general undirected graphs [8] or in trees [4] have been considered. In subclasses of graphs, hereditary graphs classes with at least $2^{\Omega(n^2)}$ members have adjacency labelings that use optimal space [10]. For trees, there are many operations that are considered, aside from adjacency. For example, labelings checking ancestry [15], computing the lowest/least common ancestor [21], and the ancestor at any given depth [16]. The distance operation, which returns the distance between two vertices of a graph, has been considered for many classes, such as general graphs [19, 5], planar graphs [22], interval graphs [17], and trees [16]. Labeling schemes for the distance operation is highly applicable to network routing [13, 14].

The distance operation, is a fundamental operation in graphs, and has been extensively studied outside of the labeling model. As distance queries are able to compute adjacency queries on unweighted graphs, the space needed is at least as much as for any data structure or labeling for the adjacency query. One solution to the distance query is to precompute and store the distances between all pairs of vertices, which incurs a quadratic space cost, but for general graphs, such a space cost is unavoidable. To achieve better space costs, work has focused on designing approximate solutions [28, 1]. For instance, Patrascu and Roditty [28] constructs a data structure occupying $O(n^{5/3})$ words of space which computes the approximate distance within a factor of 2. Other work has focused on subclasses of graphs which admits smaller space solutions, such as planar graphs [26, 25], interval graphs [17, 23] and chordal graphs [31, 27].

Here we consider labeling schemes for the distance query. The graph classes we consider are unweighted interval graphs, circular arc graphs and chordal graphs. These graphs are *intersection graphs*, where the edges can be encoded in the intersection structure of sets. That is for every vertex v, we can associate it with a set s_v so that two vertices u, v are adjacent exactly when $s_u \cap s_v \neq \emptyset$. We say that the collection of sets $\{s_v; v \in V\}$ is an *intersection model* of the graph. An *interval graph* is thus a graph where we can find an intersection model where the sets are intervals on the real line, or simply, the intersection graph of intervals on the real line. A *circular arc* graph is the intersection graph of arcs on a circle, and a *chordal graph* is the intersection graph of subtrees (a set of connected nodes, rather than an entire subtree rooted at some node) in a tree. These graphs have nice combinatorial structures where many otherwise NP-Hard problems (such as maximum independent set, clique etc...), can be solved on them in polynomial time. They also have applications in compiler design [30], operations research [9] and bioinformatics [34] among others where the specific objects they study can be modeled by these classes of graphs.

Particularly for interval graphs, there is a gap between the lower and upper bounds of a distance labeling scheme, where the lower bound is $3 \lg n - O(\lg \lg n)$ bits while the upper bound is $5 \lg n + O(1)$ bits [17]. And one of our aims is to close this gap and give tight results for this class of graphs.

1.1 Related Work

For the distance labeling model, many classes of graphs have been considered. For general graphs, a distance labeling scheme occupying $\frac{\lg 3}{2} n + o(n)$ bits (about $0.795n$) exists [5] with $O(1)$ decode (i.e time to compute the query) time along with a matching $\Omega(n)$ bit lower bound [19]. For planar graphs, A lower bound of $\Omega(n^{1/3})$ is shown [19]. The best labeling scheme uses $O(\sqrt{n})$ bits [22], but incurs a matching $O(\sqrt{n})$ decode time. With a bit more space, a labeling scheme using $O(\sqrt{n \lg n})$ bits can be decoded in $O(\lg^3 n)$ time [22]. For interval graphs, Gavoille and Paul [17] gave a $5 \lg n + O(1)$ bit labeling scheme with $O(1)$

decode time along with a $3 \lg n - o(\lg n)$ bit lower bound. For circular arc graphs, they gave a $10 \lg n + O(1)$ bit labeling scheme with $O(1)$ decode time. For permutation graphs, Katz et al. [24] gave a $O(\lg^2 n)$ bit labeling scheme with $O(\lg n)$ decode time.

On trees, we also have a variety of queries based on the ancestor-descendant relationships, among others:

- Adjacency: determine if one vertex is the parent of another. A $\lg n + O(1)$ bit scheme with $O(1)$ decode time is given by Alstrup et al. [4], along with a matching lower bound.
- Ancestry: determine if one vertex is an ancestor of another. A $\lg n + O(\lg \lg n)$ bit scheme is given by Fraigniaud and Korman [15], and a matching lower bound is given by Alstrup et al. [3].
- Lowest common ancestor (LCA): determine the label of the lowest common ancestor of two vertices. A $2.318 \lg n + o(\lg n)$ bit labeling scheme with $O(1)$ decode time is given by Gawrychowski [21], while the lower bound is $1.008 \lg n$ bits [7]. If we also need to return a predetermined k bit label of the lower common ancestor, then Alstrup et al. [7] gives a labeling scheme of length $(3 + k) \lg n$ bits with $O(1)$ decode time. If $k = \Theta(\lg n)$, then a matching $\Theta(\lg^2 n)$ lower bound is shown by Peleg [29].
- Level ancestor: return the label of the ancestor of a vertex v at depth d. A $\frac{1}{2} \lg^2 n + O(\lg n)$ bit scheme is given by Alstrup et al. [6], which matches a lower bound of $\frac{1}{2} \lg^2 n - \lg n \lg \lg n$ of Freedman et al. [16].
- Distance: return the distance between between two vertices. A $\frac{1}{4} \lg^2 n + o(\lg^2 n)$ bit labeling scheme is given by Freedman et al. [16], with a matching lower bound given by Alstrup et al. [6].

We refer to the survey of Gavoille and Peleg [18], and references therein, for a survey of labeling schemes and their applications in distributed computing.

1.2 Our Results

Our main contribution is a $3 \lg n + \lg \lg n + O(1)$ bit labeling scheme for interval graphs with $O(1)$ decode time, which improves the $5 \lg n + O(1)$ labeling scheme given by Gavoille and Paul [17]. This matches the $3 \lg n - o(\lg n)$ lower bound they proved up to lower order terms. We further note that Gavoille and Paul assumed that the interval graph is connected in their paper, and did not discuss what do if it were disconnected, while our solution works for general interval graphs. We also first consider connected interval graphs, and give a $3 \lg n + O(1)$ bit distance labeling scheme with $O(1)$ decode time, before generalizing it.

To do this, we adapt the distance algorithm on interval graphs of He et al. [23] which uses level ancestor queries. The main advancement compared to similar structures such that of Chen et al. [12] is the fact that a tree encoding the distances can be constructed such that a level-order traversal of the tree gives exactly the vertices in a left to right scan of the (left endpoints of the) intervals, and thus comparisons of these endpoints can be done by comparing properties of the corresponding nodes of the tree. We note that Gavoille and Paul [17] stated that such an approach using level ancestor queries was impossible, as any labeling scheme for level ancestors queries would need $\Omega(\lg^2 n)$ bits [16], and this may be the reason for the gap between upper and lower bounds ($5 \lg n$ vs $3 \lg n$) to exist. The key insight for our approach is that, although we use level ancestor queries as our basis, we do not need to compute the exact ancestor node (and thus its label), but rather some properties of that ancestor. These properties are also not exact, but approximate (for example, rather than the exact index of a node visited in a post-order traversal, we only need to know whether this index is less than some integer i), which allows us to bypass the level ancestor query lower bound. This optimal scheme for interval graphs also immediately improves the distance labeling scheme of circular arc graphs from $10 \lg n + O(1)$ bits [17] to $6 \lg n + 2 \lg \lg n + O(1)$ bits, with $O(1)$ decode time.

We then apply the labeling scheme for interval graphs to chordal graphs to obtain the first distance labeling schemes for chordal graphs. We obtain a distance labeling scheme of length $n/2 + O(\lg^2 n)$ bits. We note that as the lower bound on chordal graphs data structures is $n^2/4 - o(n)$ bits via an enumeration argument [27, 33], any distance labeling scheme will require $n/4 - o(n)$ bit label lengths.

2 Preliminaries

2.1 Definitions and Notation

We will use the standard graph theoretic notations. Let $G = (V, E)$ be a graph. We set $n = |V|$ the number of vertices and $m = |E|$ the number of edges. As is standard, we will use the word-RAM model with $\omega = \Theta(\lg n)$ bit words.

We use the standard definitions of graph and tree operations. For graphs, the operations we use are

- `adjacent`(u, v) which tests if two vertices u and v are adjacent.
- `distance`(u, v) which returns the (unweighted) distance between two vertices u and v.

For trees the standard operations we use are

- `depth`(v) which returns the depth of node v.
- `lev_anc`(v, d) which returns the ancestor of a node v at a given depth d.
- `parent`(v) which returns the parent node of the given node v.
- `LCA`(u, v) which returns the lowest (i.e. largest depth) common ancestor of two given nodes u and v.
- `node_rank`$_X(v)$ which returns the index of v in the X traversal of the tree, where X is `PRE`, `POST` or `LEVEL` indicating a preorder, post-order or level-order (i.e. a breadth-first traversal where we visit the children from left to right) traversal of the tree.

A labeling scheme is a distributed data structure, where each vertex of the graph contains a piece of the data structure and queries must be computed using only those pieces available at the relevant vertices. Formally, a *distance labeling scheme* for a graph G with n vertices is a pair of functions (L, f) where $L(G, v)$, typically referred to as a encoder or marker algorithm, computes a label from a vertex v of G, and f, typically referred to as a decoder algorithm, computes the distance between two vertices given their labels. That is $f(L(G, u), L(G, v)) = \texttt{distance}_G(u, v)$. The size or length of the labeling scheme is the maximum length over all possible labels: $\max_{G,v} |L(G, v)|$. We note that there is a dichotomy between L and f, where L can be computed using information from the entire graph, f cannot and can use only the labels, without further information about the original graph that the labels come from.

2.2 Interval Graph

An interval graph is a graph where the intersection model is a set of closed interval on the real line, where we write the interval as $I_v = [l_v, r_v]$. By sorting the endpoints (and breaking ties such that left endpoints come before right endpoints to preserve the intersection, and arbitrarily otherwise) we may assume that the endpoints are distinct integers in the range $[1, 2n]$. We will name the vertices $1, \ldots, n$, in the order of their left endpoints, so that for two vertices $u < v$ we have $l_u < l_v$. We will now review some of the lemmas used in the computation of distances in interval graphs.

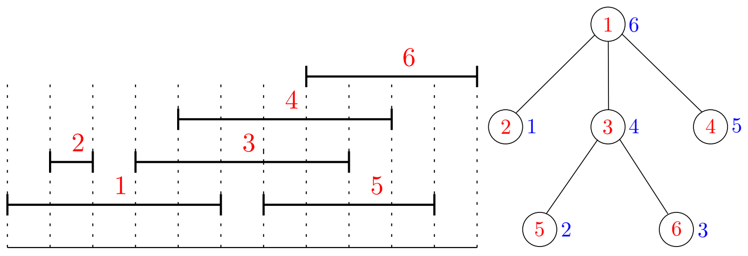


Figure 1 An interval graph on 6 vertices. The intersection model is shown on the left, while the distance tree (blue labels next to the nodes represent post-order numbers) is shown on the right.

For each vertex v, we define its parent, $\texttt{parent}(v)$, to be the minimum vertex u (i.e. the one with minimal left endpoint) adjacent to v. This can be expressed by the following formula: $\arg\min\{l_u \mid r_u \geq l_v\}$.

Using this parent-child relationship (where the root of the tree is the vertex v with $1 = v = \texttt{parent}(v)$), we may build a tree T which we will call the *distance tree*, where for each internal node, its children are in sorted order (by left endpoint).

Crucial to the data structure is the index of the nodes of the tree in some traversal of the tree denoted by $\texttt{node\_rank}_X(T, v)$, where X is $\texttt{PRE}, \texttt{POST}$ or $\texttt{LEVEL}$. We will omit the tree T when the tree being referred to is clear. The main property of this tree is that the vertex order sorted by left endpoint is exactly the vertex order obtained in a level-order traversal of the tree:

▶ **Lemma 1** (Lemma 7 of [23]). *Let G be an interval graph with distance tree $T(G)$ and vertices u, v. Then $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$ if and only if $l_u < l_v$.*

This is quite intuitive since we ordered the children of every node of the tree by their left endpoints, so that the property can propagate up the tree. Furthermore, by the property of these traversals, in the case that $\texttt{depth}(u) = \texttt{depth}(v)$, $\texttt{node\_rank}_X(u) < \texttt{node\_rank}_X(v)$ if and only if $l_u < l_v$ for $X = \texttt{PRE}, \texttt{POST}, \texttt{LEVEL}$, as on each level of the tree, the traversals visit the nodes from left to right; see Figure 1.

The shortest path algorithm used in previous works [12, 2, 27, 23] is the recursive algorithm given in Algorithm 1, for two vertices in the same connected component of the interval graph. The correctness can be summarized as the following lemma (though not explicitly stated as a lemma in some previous papers):

▶ **Lemma 2** (Lemma 8 [27], Lemma 4,6 [12]). *Let G be an interval graph with distance tree $T(G)$, and u, v be two vertices in the same connected component of G with $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$ (i.e. $u < v$). Let the node to root path of v be $v = v_{\texttt{depth}(v)}, \ldots, v_0 = r$, and i be the first (i.e. largest) index where $l_{v_i} \leq r_u$. Then a shortest path from u to v is $u = v_{\texttt{depth}(v)}, \ldots, v_i, u$, and furthermore, i is $\texttt{depth}(u) - 1$, $\texttt{depth}(u)$ or $\texttt{depth}(u) + 1$.*

Algorithm 1 Shortest Path computation between vertices u and v with $u < v$.

```
1: path = empty
2: while true do
3:    if adjacent $(u, v)$ then
4:       path = path, $v, u$
5:       return path
6:    path = path, $v$
7:    $v \leftarrow \texttt{parent}(v)$
```

The above lemma also allows us to compute the distance, as there are at most 3 candidates for v_i, which we may check individually using `lev_anc`.

3 Distance Labeling in Interval Graphs

In this section, we will consider distance labeling schemes for interval graphs. We will first assume that our graph is connected, and then generalize it to arbitrary interval graphs. As Gavoille and Paul [17] showed, any distance labeling scheme requires at least $3\lg n - O(\lg\lg n)$ bits. Thus, our goal is to give a labeling scheme that uses $3\lg n + \lg\lg n + O(1)$ bits matching their lower bound up to lower order terms. We will use the distance computation method outlined in Lemma 2 and level ancestor queries as the basis of our scheme.

3.1 Labeling Scheme for Connected Interval Graphs

Now we consider the distance computation method outlined in Lemma 2. Given two vertices u and v such that $u < v$ (we assume $u \neq v$ as that is trivial), we computed the path to the root in the distance tree from v as $v = v_{\mathtt{depth}(v)}, \dots, v_0$ and computed the first index i such that v_i is adjacent to u. In Lemma 2, we had 3 candidates for v_i. We will now narrow it down to 2 by examining the relative positions of u and v in the tree.

▶ **Lemma 3.** *Let G be an interval graph with distance tree T. Let u and v be two vertices of G such that $u < v$. Depending on the positioning of u and v we define the ancestor w (of v) as follows:*

1. *Suppose that* $\mathtt{node\_rank}_{\mathtt{POST}}(u) < \mathtt{node\_rank}_{\mathtt{POST}}(v)$. *Let* $w = \mathtt{lev\_anc}(v, \mathtt{depth}(u))$.
2. *Suppose that* $\mathtt{node\_rank}_{\mathtt{POST}}(u) > \mathtt{node\_rank}_{\mathtt{POST}}(v)$. *Let* $w = \mathtt{lev\_anc}(v, \mathtt{depth}(u) + 1)$.

Then $w = v_i$ if u and w are adjacent. Otherwise, $w = v_{i-1}$.

Proof. See Figure 2 for an illustration of the two cases in the lemma statement. Define $d = \mathtt{depth}(u)$ if $\mathtt{node\_rank}_{\mathtt{POST}}(u) < \mathtt{node\_rank}_{\mathtt{POST}}(v)$ and $d = \mathtt{depth}(u) + 1$ if $\mathtt{node\_rank}_{\mathtt{POST}}(u) > \mathtt{node\_rank}_{\mathtt{POST}}(v)$. Then $w = v_d$ as defined in this lemma. By the definition of `parent`, $l_{v_j} > l_{v_{j-1}}$ for every j. Then $l_{v_j} > l_{v_d} > l_u$ for all $j > d$. Thus for any v_j with $j > d$, v_j cannot be adjacent to u, as otherwise $l_{v_j} \le r_u$ and thus u would be considered as a possible vertex in the definition of $\mathtt{parent}(v_j)$. But as $v_{j-1} = \mathtt{parent}(v_j)$, we must have $l_{v_d} \le l_{v_{j-1}} < l_u$, a contradiction.

In the case where u and w are adjacent, $w = v_i$ by definition as it is the first vertex adjacent to u on the path towards the root (Lemma 2). Otherwise, suppose that u and w are not adjacent. By the definition of a post-order traversal, we have $\mathtt{node\_rank}_{\mathtt{POST}}(w) \ge \mathtt{node\_rank}_{\mathtt{POST}}(v)$ since ancestors are visited later in the traversal. In the first case, since $\mathtt{depth}(w) = \mathtt{depth}(u)$ and $\mathtt{node\_rank}_{\mathtt{POST}}(u) < \mathtt{node\_rank}_{\mathtt{POST}}(v) \le \mathtt{node\_rank}_{\mathtt{POST}}(w)$, we have $l_u < l_w$. Furthermore, since $\mathtt{depth}(\mathtt{parent}(w)) = \mathtt{depth}(u) - 1$, $l_{\mathtt{parent}(w)} < l_u$ because it comes before u in a level-order traversal (it has a smaller depth). In the second case where $\mathtt{node\_rank}_{\mathtt{POST}}(u) > \mathtt{node\_rank}_{\mathtt{POST}}(v)$, because $\mathtt{depth}(w) = \mathtt{depth}(u) + 1$, we have $l_w > l_u$. This also implies that $\mathtt{depth}(\mathtt{parent}(w)) = \mathtt{depth}(u)$. By assumption $\mathtt{node\_rank}_{\mathtt{POST}}(u) > \mathtt{node\_rank}_{\mathtt{POST}}(v)$ which implies that $\mathtt{node\_rank}_{\mathtt{POST}}(u) > \mathtt{node\_rank}_{\mathtt{POST}}(\mathtt{parent}(w))$, so we have $l_{\mathtt{parent}(w)} < l_u$ as it comes before u in a level-order traversal.

Therefore, in either situation, we have the inequalities $l_{\mathtt{parent}(w)} < l_u < l_w$. By definition of `parent` we have $r_{\mathtt{parent}(w)} > l_w$. Therefore, $l_{\mathtt{parent}(w)} < l_u < r_{\mathtt{parent}(w)}$ and thus u and $\mathtt{parent}(w)$ are adjacent. Hence $\mathtt{parent}(w)$ is the first vertex adjacent to u on the path towards the root from v so $w = v_{i-1}$ (Lemma 2). ◀

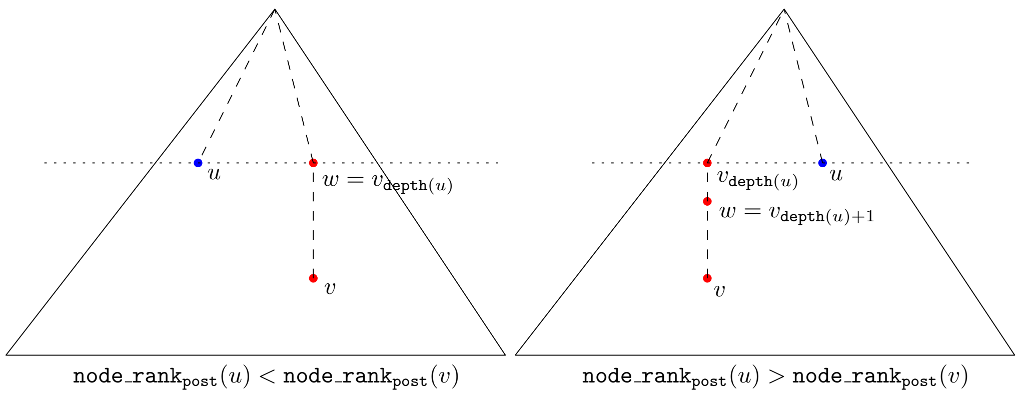

u
$w = v_{\mathtt{depth}(u)}$
v
$\mathtt{node\_rank}_{\mathtt{post}}(u) < \mathtt{node\_rank}_{\mathtt{post}}(v)$
$v_{\mathtt{depth}(u)}$
u
$w = v_{\mathtt{depth}(u)+1}$
v
$\mathtt{node\_rank}_{\mathtt{post}}(u) > \mathtt{node\_rank}_{\mathtt{post}}(v)$

Figure 2 The two cases based on the relative positioning of u and $v_{\mathtt{depth}(u)}$ on the level $\mathtt{depth}(u)$. In the first case, u is to the left of the ancestor of v, while in the second case, u to the right. The node w, which will later be called the *representative* of v with respect to u, is the smallest depth ancestor that is after u in a level-order traversal. In the first case, w would be on the same level as u, while in the second w is on the next level.

The node w, which is the smallest depth ancestor with $l_w > l_u$ (i.e. to the right of u in the intersection model and in a level-order traversal of the tree) will be denoted as the *representative* of v with respect to u. This is because to compute the distance between u and v, it suffices to determine whether w is adjacent to u or not, and to compute the distance between v and w. Since w is an ancestor of v, this distance is simply the difference in the depths of w and v. We will rewrite Algorithm 1 as Algorithm 2 to take advantage of this observation.

Algorithm 2 Distance computation between vertices u and v with $u < v$.

```
1: if $\mathtt{node\_rank}_{\mathtt{POST}}(u) < \mathtt{node\_rank}_{\mathtt{POST}}(v)$ then
2:     $w \leftarrow \mathtt{lev\_anc}(v, \mathtt{depth}(u))$
3: else
4:     $w \leftarrow \mathtt{lev\_anc}(v, \mathtt{depth}(u) + 1)$
5: distance = $\mathtt{depth}(v) - \mathtt{depth}(w)$
6: if $\mathtt{adjacent}(u, w)$ then
7:     distance = distance+1
8: else
9:     distance = distance+2
10: return distance
```

To convert Algorithm 2 into an algorithm that uses only labels, we need to do the following steps using only labels:

1. Test whether $u < v$
2. Compute $\mathtt{depth}(v)$ and $\mathtt{depth}(u)$
3. Compute $\mathtt{node\_rank}_{\mathtt{POST}}(v)$ and $\mathtt{node\_rank}_{\mathtt{POST}}(u)$
4. Compute (an approximation of) `lev_anc`
5. Compute `adjacent` using the approximation of `lev_anc`.

Our labeling scheme will consist of the following 3 integers for each vertex v, each using $\lceil \lg n \rceil$ bits:

1. $\mathtt{depth}(v)$
2. $\mathtt{node\_rank}_{\mathtt{POST}}(v)$
3. $\mathtt{node\_rank}_{\mathtt{POST}}(\mathtt{last}(v))$

Here, $\texttt{last}(v)$ is the rightmost neighbour of v (i.e. the neighbour with the largest left endpoint). In the case where v is the rightmost vertex, the rightmost neighbour of v is to its left, and we consider this to be an invalid case and set $\texttt{last}(v) =$ nothing (and we will not need it in our computation).

An important property of $\texttt{last}(v)$ is

▶ **Lemma 4.** *Let G be an interval graph with distance tree T. Let v be a vertex that is not the rightmost vertex. Then every vertex w such that $v < w \leq \texttt{last}(v)$ is adjacent to v.*

Proof. Let w be a vertex such that $v < w \leq \texttt{last}(v)$ (by our naming convention, this is also an inequality on the left endpoints of these vertices). Since $\texttt{last}(v)$ is adjacent to v, we have $l_v < l_{\texttt{last}(v)} < r_v$. Thus we have $l_v < l_w \leq l_{\texttt{last}(v)} < r_v$, so w is adjacent to v. ◀

Now we show how to compute the steps using our labels.

Step 1: Decide if $u < v$. By Lemma 1, we have $u < v$ exactly when $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$. If $\texttt{depth}(u) < \texttt{depth}(v)$, then the inequality of $\texttt{node\_rank}_{\texttt{LEVEL}}$ is implied. Otherwise, if $\texttt{depth}(u) = \texttt{depth}(v)$, then $\texttt{node\_rank}_{\texttt{LEVEL}}(u) < \texttt{node\_rank}_{\texttt{LEVEL}}(v)$ if and only if $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. If neither is the case, then we have $v < u$, so we switch the two vertices in the algorithm.

Step 2,3: Compute $\texttt{depth}(u), \texttt{depth}(v)$, $\texttt{node\_rank}_{\texttt{POST}}(u), \texttt{node\_rank}_{\texttt{POST}}(v)$. This is immediate as we store them as part of the label.

Step 4,5: Compute adjacent using the approximation of lev_anc. We will use $\texttt{node\_rank}_{\texttt{POST}}(v)$ as our approximation of $\texttt{node\_rank}_{\texttt{POST}}(w)$ in our calculations. To compute $\texttt{adjacent}(u, w)$ using $\texttt{node\_rank}_{\texttt{POST}}(v)$, we will use the following lemma (see Figure 3):

▶ **Lemma 5.** *Let G be an interval graph and T be its distance tree. Let u and v be two vertices such that $u < v$. Let w be the representative of v with respect to u, as defined in Lemma 3. The following two cases mirror the two cases used to define w:*
Suppose that $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. Then u and w are adjacent if and only if $\texttt{node\_rank}_{\texttt{POST}}(v) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$ or $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$.
Suppose that $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v)$. Then u and w are adjacent if and only if $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$.

Proof. First we examine the relationship between the post-order ranks of u and $\texttt{last}(u)$ (i.e. $\texttt{node\_rank}_{\texttt{POST}}(u)$ and $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$). Since $\texttt{last}(u)$ is the rightmost neighbour of u, it comes after u in level-order, so $\texttt{depth}(\texttt{last}(u)) \geq \texttt{depth}(u)$. Since $\texttt{last}(u)$ is adjacent to u, $l_{\texttt{parent}(\texttt{last}(u))} \leq l_u$ by the definition of parent. Thus $\texttt{depth}(\texttt{last}(u)) = \texttt{depth}(\texttt{parent}(\texttt{last}(u))) + 1 \leq \texttt{depth}(u) + 1$. Thus $\texttt{last}(u)$ is either on the same level as u or the level below. If it is on the same level as u, then as it is to the right of u, we have $\texttt{node\_rank}_{\texttt{POST}}(u) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$. If it is on the level below u, then since $\texttt{depth}(\texttt{parent}(\texttt{last}(u))) = \texttt{depth}(u)$, we have

$$\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{parent}(\texttt{last}(u))) \leq \texttt{node\_rank}_{\texttt{POST}}(u)$$

Thus by checking the relationship between their post-order ranks we may determine the depth of $\texttt{last}(u)$ (and vice versa).

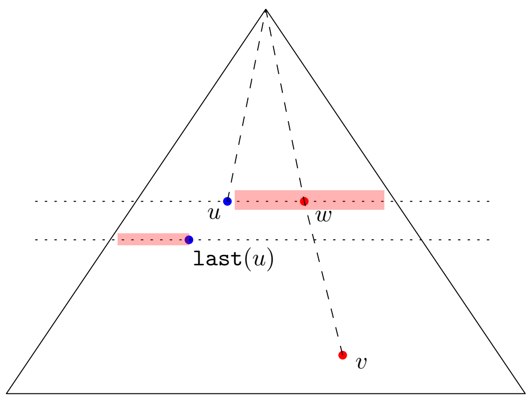


Figure 3 To determine if w is adjacent to u, we need to check if w lies in the shaded region between u and $\texttt{last}(u)$ in level-order. To do so, we will need to compare the relative positioning of w and $\texttt{last}(u)$ when they are on the same level. If w is on the level above $\texttt{last}(u)$ (so it must come before it in level-order), then w is in the shaded region. If w is on the level below $\texttt{last}(u)$ (so it must come after it in level-order), then w cannot be in the shaded region.

In the first case, suppose that $\texttt{node\_rank}_{\texttt{POST}}(u) < \texttt{node\_rank}_{\texttt{POST}}(v)$. By the definition of the representative of v with respect to u, $\texttt{depth}(w) = \texttt{depth}(u)$, and w is to the right of u on that level (so we have $\texttt{node\_rank}_{\texttt{POST}}(w) > \texttt{node\_rank}_{\texttt{POST}}(v) > \texttt{node\_rank}_{\texttt{POST}}(u)$). By Lemma 4, u and w are adjacent if and only if

$$\texttt{node\_rank}_{\texttt{LEVEL}}(w) \in (\texttt{node\_rank}_{\texttt{LEVEL}}(u), \texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u))]$$

If $\texttt{depth}(\texttt{last}(u)) = \texttt{depth}(u)$ (equivalently, $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) > \texttt{node\_rank}_{\texttt{POST}}(u)$), then all three nodes are on the same level, so we may translate it them to post-order numbers as $\texttt{node\_rank}_{\texttt{POST}}(w) \in (\texttt{node\_rank}_{\texttt{POST}}(u), \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))]$. The first half is satisfied by assumption, so we are left with just $\texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$. If $\texttt{depth}(\texttt{last}(u)) = \texttt{depth}(u)+1$ (equivalently, $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$), then the range $(\texttt{node\_rank}_{\texttt{LEVEL}}(u), \texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u))]$, restricted to the level $\texttt{depth}(u)$ contains all the nodes on the level $\texttt{depth}(u)$ to the right of u, which w satisfies by definition (as it is a node on level $\texttt{depth}(u)$ to the right of u). Hence, in this case, u and w are adjacent if and only if $\texttt{node\_rank}_{\texttt{POST}}(v) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$ or $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$.

In the second case, we assume that $\texttt{node\_rank}_{\texttt{POST}}(u) > \texttt{node\_rank}_{\texttt{POST}}(v)$. Therefore, we have $\texttt{depth}(w) = \texttt{depth}(u)+1$, and $\texttt{node\_rank}_{\texttt{POST}}(w) < \texttt{node\_rank}_{\texttt{POST}}(u)$ (by the properties of a post-order traversal). Again u and w are adjacent if and only if $\texttt{node\_rank}_{\texttt{LEVEL}}(w) \in (\texttt{node\_rank}_{\texttt{LEVEL}}(u), \texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u))]$. If $\texttt{depth}(\texttt{last}(u)) = \texttt{depth}(u)$ (equivalently, $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) > \texttt{node\_rank}_{\texttt{POST}}(u)$), then as $\texttt{depth}(w) = \texttt{depth}(\texttt{last}(u)) + 1$, w comes after $\texttt{last}(u)$ in level-order, so $\texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{LEVEL}}(w)$. Therefore, w cannot be adjacent to u. If $\texttt{depth}(\texttt{last}(u)) = \texttt{depth}(u) + 1 = \texttt{depth}(w)$ (equivalently, $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$), then the restriction of the range $(\texttt{node\_rank}_{\texttt{LEVEL}}(u), \texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u))]$ to level $\texttt{depth}(w) = \texttt{depth}(\texttt{last}(u))$ are exactly the nodes on level $\texttt{depth}(w)$ in the range $(-\infty, \texttt{node\_rank}_{\texttt{LEVEL}}(\texttt{last}(u))]$. Converted to a post-order traversal, this is the range $(-\infty, \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))]$ for those nodes on level $\texttt{depth}(w)$. w satisfies this exactly when $\texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$ (so that $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(w) \leq \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u))$). Hence, in this case, u and w are adjacent if and only if $\texttt{node\_rank}_{\texttt{POST}}(v) < \texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(u)) < \texttt{node\_rank}_{\texttt{POST}}(u)$. ◀

Algorithm 3 Distance computation between vertices u, v by their labels $L(u), L(v)$.

```
1: if depth(v) < depth(u) or (depth(v) = depth(u) and node_rank_POST(v) < node_rank_POST(u)) then
2:     switch u and v {Now we have u < v}
3: if node_rank_POST(u) < node_rank_POST(v) then
4:     distance ← depth(v) − depth(u)
5:     if node_rank_POST(v) ≤ node_rank_POST(last(u)) or node_rank_POST(last(u)) < node_rank_POST(u) then
6:         distance = distance+1
7:     else
8:         distance = distance+2
9: else
10:    distance ← depth(v) − depth(u) − 1
11:    if node_rank_POST(v) < node_rank_POST(last(u)) < node_rank_POST(u) then
12:        distance = distance+1
13:    else
14:        distance = distance+2
15: return distance
```

To summarize the above, the decoding function f is given by algorithm 3. Note that line 6 and line 12 corresponds to the case in Lemma 5 where u and w are adjacent. Therefore, we add 1 to the distance from v to w to obtain the distance from v to u. In line 8 and 14, u and w are not adjacent, but u and $\texttt{parent}(w)$ are by Lemma 3, and the distance between u and w is 2. It is clear that algorithm 3 runs in $O(1)$ time, using only the labels of u and v.

Regarding preprocessing, we claim that if the intervals' endpoints are given in sorted order, then all the labels can be constructed in $O(n)$ time. Otherwise, we can spend $O(n \log n)$ additional time sorting the endpoints. If the interval graph is given but not an intersection model, we may use a linear time ($O(n + m)$) interval graph recognition algorithm [11] to construct an intersection model of G in sorted order. Details for preprocessing are omitted due to space constraints. Thus we have the main theorem for this section:

▶ **Theorem 6.** *Let G be a connected interval graph. Then there exists a distance labeling scheme occupying at most $3\lceil \lg n \rceil = 3 \lg n + O(1)$ bits per vertex and can compute* `distance` *in $O(1)$ time. Furthermore, if the intervals are given in sorted order, then the labels can be constructed in $O(n)$ time.*

3.2 General Interval Graphs

Previously, we had assumed that our interval graphs were connected. For general interval graphs, it is possible that for two vertices u and v, their labels would be identical in two graphs, one where u and v belong to the same connected component G_j and one where they belong to different connected components G_j and $G_{j'}$. Thus the labels computed previously is insufficient to compute the distance as it cannot decide if the two vertices belong to the same connected component. As noted, Gavoille and Paul [17] assumed the graph were connected and did not discuss how to generalize it to the disconnected case. To generalize to general interval graphs, it suffices to be able to determine if two vertices are in the same component or not. One way to solve this is to add $\lg n$ bits to the label to state which component the vertex is in, but that would make the labels too costly. Instead, we will use the fact that the label size depends on the size of the component, and the number of components of a given size scales inversely with that size.

Define the ranges $[2^i, 2^{i+1})$ for $i = 0, \dots, \lfloor \lg n \rfloor$. For each component G_j, the size of the component n_j falls into one of these ranges. For range $[2^i, 2^{i+1})$, the number of components G_j falling into this range is at most $n/2^i$. Thus, to identify the components, we need to store i, the range that its size falls into, and c_i, an identifier for which component in that range. For a component of size n_j falling in the range $[2^i, 2^{i+1})$, these identifiers use at most $\lg(\lfloor \lg n \rfloor + 1)$ and $\lg(n/2^i)$ bits respectively. Also as $n_j < 2^{i+1}$, each of the three integers comprising of the labels of the vertices of G_j has size at most $\lceil \lg n_j \rceil \leq i + 1$ bits. Thus in total, for a component G_j of size $n_j \in [2^i, 2^{i+1})$, a label has size at most

$$\lg \lg n + 1 + \lg n - (i + 1) + 3(i + 1) = \lg n + 2i + \lg \lg n + O(1)$$

Since $i \leq \lfloor \lg n \rfloor$, this is at most $3 \lg n + \lg \lg n + O(1)$ bits. Thus our extension to general interval graphs is the following theorem.

▶ **Theorem 7.** *Let G be an interval graph. Then there exists a distance labeling scheme occupying at most $3 \lg n + \lg \lg n + O(1)$ bits per vertex and can compute* `distance` *in $O(1)$ time. Furthermore, if the intervals are given in sorted order, then the labels can be constructed in $O(n)$ time.*

Using this, we have an immediate application to circular arc graphs. Following the framework of Gavoille and Paul [17], we unroll the circular arc graph twice. That is, we start from the angle $\theta = 0$ and sweep the circle twice, writing down the endpoints of the arcs. In this fashion, each arc is recorded twice, once as its original $[\theta_1, \theta_2]$, and once on the second unroll, $[\theta_1 + 2\pi, \theta_2 + 2\pi]$ [1]. After unrolling, we obtain an interval graph, where each vertex of the original circular arc graph corresponds to two vertices in the unrolled interval graph. The distance can then be calculated using the following lemma:

▶ **Lemma 8** (Lemma 3.5 [17]). *For an circular arc graph G, unrolled twice into an interval graph $\tilde{G}$. For every $i < j$, $\texttt{distance}_G(x_i, x_j) = \min\{\texttt{distance}_{\tilde{G}}(x_i^1, x_j^1), \texttt{distance}_{\tilde{G}}(x_j^1, x_i^2)\}$, where x_i are sorted by their left endpoints in increasing i and x^1, x^2 are the two copies of the arc x in the interval graph.*

Thus it suffices to store two (interval) vertex labels for each vertex of the circular-arc graph, and so the length of the label is $6 \lg n + 2 \lg \lg n + O(1)$ bits.

▶ **Corollary 9.** *Let G be a circular arc graph. Then there exists a distance labeling scheme occupying at most $6 \lg n + 2 \lg \lg n + O(1)$ bits per vertex and can compute* `distance` *in $O(1)$ time. Furthermore, if G is connected, then $6 \lg n + O(1)$ bit labels suffices. If the arcs are given in sorted order, then the labels can be constructed in $O(n)$ time.*

4 Chordal Graph

4.1 Background

4.1.1 Chordal Graph Structure

One of the many equivalent definitions of chordal graph is that a chordal graph is the intersection graph of subtrees (i.e. connected sets of nodes) in a tree [20]. Thus, for a chordal graph G, there exists a tree T and a set of subtrees $\{T_v; v \in V\}$ of T such that two vertices

[1] Some more work need to be done for arcs which contain our origin angle.

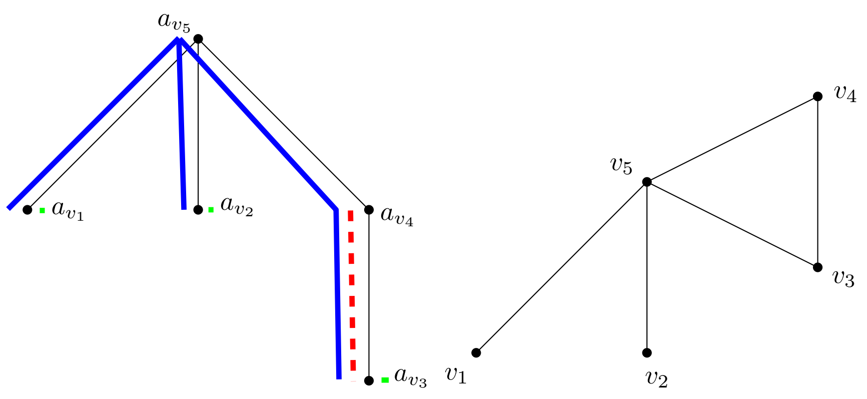


Figure 4 Left: The underlying tree for a chordal graph is depicted in black. Each of the leaves has a subtree containing only that leaf (the green dots), and these correspond to vertices v_1, v_2, v_3. T_{v_4} is depicted with a red dashed segment and T_{v_5} in blue bold segments. The apex of each path is labeled. Note that every subtree has a distinct apex. Right: The chordal graph generated by this set of subtrees.

u and v are adjacent if and only if the subtrees T_u and T_v intersect (at some node). If we further root the tree, then each subtree T_v corresponding to a vertex v has a unique smallest depth node a_v, which we will denote as the *apex* of the subtree (and of the vertex). To create a data structure, we would like to make some modifications to the tree T with additional exploitable properties.

Munro and Wu [27] showed that we may choose T such that the number of nodes is exactly n. Furthermore, this rooted tree T and the subtrees T_v corresponding to vertices has the property that for two vertices u and v, their apexes are distinct: $a_u \neq a_v$. See Figure 4 for an example. In this way, we have a bijection between vertices of the graph G and nodes of tree T. Thus it make sense to talk about the vertex v which corresponds to a node of T, and we will name the nodes of T as a_v for the vertex v whose apex is that node.

To characterize adjacency, we look at the relationship between the apexes of the two vertices u and v. If the subtrees rooted at a_u and at a_v are disjoint, then the subtrees T_u and T_v do not intersect and u and v are not adjacent. Otherwise, one of a_u and a_v is an ancestor of the other, say a_u is an ancestor of a_v. Then u and v are adjacent exactly when the subtree T_u corresponding to u contains the node a_v. Thus we may define a set of vertices S_v at each node a_v of T which is the set of ancestors of a_v, whose subtrees contain a_v (i.e. the set of ancestors which are adjacent to v).

Finally, we observe that in the degenerate case that T is a path, then all subtrees T_v become paths. By viewing T as a subset of the real number line, we see that all subtrees T_v are intervals, and thus the graph generated is an interval graph.

4.1.2 Chordal Graph Distance Algorithm

Munro and Wu [27] gave an algorithm computing the distance between two vertices u and v. In the case that a_u is an ancestor of a_v (or vice versa), the problem reduces to that of an interval graph by restricting the tree T to the path from a_v to the root. Otherwise, if a_u and a_v belong to different subtrees, we first compute the lowest common ancestor a_h of a_u and a_v. We then compute the distance from h to u and from h to v (with a caveat). Since a_h is an ancestor of a_u and a_v, we reduce to the interval graph case. Analogous to the distance tree in interval graphs, Munro and Wu construct a distance tree for chordal graphs.

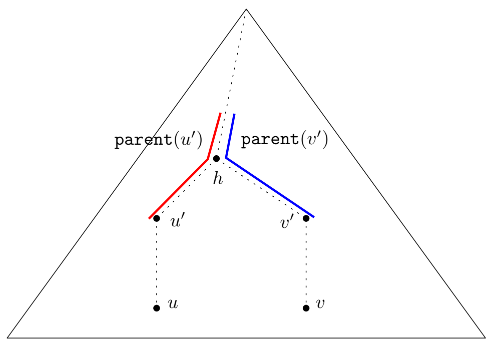


Figure 5 Left bolded (red) segment represents part of the subtree corresponding to $\texttt{parent}(u')$, which must contain the node $a_{u'}$, and the right (blue) segment for $\texttt{parent}(v')$. Here the apex $a_{u'}$ of the subtree corresponding to a vertex u' is labelled as u'. It can be seen that both $\texttt{parent}(u')$ and $\texttt{parent}(v')$ pass through a_h so they are adjacent.

We note that for a vertex v, when we restrict the tree T to the path from a_v to the root, the set of vertices S_v corresponds exactly to those vertices p such that $l_p < l_v$ and $r_p \geq l_v$ (when viewing this path as a subset of the real number line, with the coordinates being the depth of the node). Therefore, the formula $(\arg\min\{l_u \mid r_u \geq l_v\})$ used to define the distance tree in interval graphs can also be applied to chordal graphs. The parent $\texttt{parent}(v)$ of a vertex v in the distance tree T_D of a chordal graph is the vertex $\texttt{parent}(v) \in S_v$ with the smallest depth apex $a_{\texttt{parent}(v)}$. This is well-defined since the apexes are distinct.

Since this notion of parent matches the parent of an interval graph generated from T by restricting to any path towards the root, the distance tree T_D can be seen as the union of all distance trees of interval graphs generated by taking paths from leaves of T to the root. However, due to the multiple interval graph distance trees being unioned, we do not have the nice ordering property in Lemma 1.

When applying the interval graph distance algorithm between u and h (and between v and h), we reach their representatives u' and v', and compute $\texttt{distance}(u, h) = \texttt{distance}(u, u') + \texttt{distance}(u', h)$, where $\texttt{distance}(u', h) = 1$ if u' and h are adjacent, and $\texttt{distance}(u', h) = 2$ if not (a shortest path being u', $\texttt{parent}(u')$, h, similarly for v'). Finally these two paths must be joined. Naively, the path could be the shortest path from u to h followed by the shortest from h to v. So depending on whether u' and h, and v' and h are adjacent, the path from u' to v' has length either 2, 3, or 4. However, it is shown that $\texttt{parent}(u')$ and $\texttt{parent}(v')$ are adjacent (see Figure 5), so that in the case that both u' and v' are not adjacent to h, a shortest path is $u', \texttt{parent}(u'), \texttt{parent}(v'), v'$ which has length 3, rather than $u', \texttt{parent}(u'), h, \texttt{parent}(v'), v'$ with length 4. Thus the distance between u' and v' is either 2 or 3. Determining this distance between u' and v' is the bottleneck for the distance computation as it is shown that $\texttt{distance}(u, v) = \texttt{distance}(u, u') + \texttt{distance}(u', v') + \texttt{distance}(v', v)$.

Munro and Wu show that the distance between u' and v' is 2 exactly when there exists some vertex h' such that h' is adjacent to both u' and v' (i.e. the subtree $T_{h'}$ corresponding to h' contains both nodes $a_{u'}$ and $a_{v'}$). See Figure 6. If u' and v' are both adjacent to h, then the vertex h takes this role. However, even if u' and v' are not adjacent to h, such an h' can exist (which is independent of the adjacencies between u' and v' with h). Such a vertex h' would exist in both the sets $S_{u'}$ and $S_{v'}$, and so determining whether h' exists is equivalent to determining whether the intersection $S_{u'} \cap S_{v'} = \emptyset$.

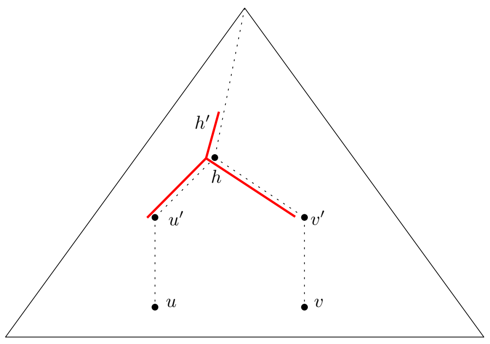


Figure 6 Part of the subtree corresponding to the vertex h' is depicted. The vertex h' is adjacent to both u' and v' so that the distance between them is 2.

This problem of preprocessing sets to answer queries of the form: determine whether the intersection of two of the given sets is empty, is the *set intersection oracle* problem. Thus the chordal graph distance problem can be reduced to it. Munro and Wu further argued that the set intersection oracle problem can also be reduced to the chordal graph distance problem so that the two are equivalent (up to a constant factor of the input sizes). The set intersection oracle problem is conjectured to be difficult (Conjecture 3 and some follow up discussions of Pătraşcu and Roditty[28] state that, to solve it using $O(1)$ time we must use $\Omega(n^2)$ space, even if the sets are small), so computing distances in chordal graphs quickly would also be difficult. Munro and Wu's [27] algorithm described above can be summarized in the following lemma.

▶ **Lemma 10.** *Let G be a chordal graph, T be the intersection model as described by Munro and Wu [27] with exactly n nodes, and T_D be the distance tree. Let u, v be two vertices, and let a_u, a_v be their apexes. We consider the general case where a_u and a_v do not have any ancestry relationship. Let h be the vertex such that $a_h = \texttt{LCA}(a_u, a_v)$. Let u' be the* representative *of u with respect to h, similarly for v'. Then the distance between u and v is $\texttt{distance}(u, v) = \texttt{distance}(u, u') + \texttt{distance}(v, v') + 2 + \mathbf{1}(C)$, where C is this condition: there does not exist any vertex h' such that $T_{h'}$ contains both the nodes $a_{u'}$ and $a_{v'}$.*

4.2 Labeling Scheme

Using the above `distance` algorithm, our labeling scheme must be able to check/compute the following steps.

- Given two vertices u and v, determine whether one is an ancestor of the other, and if not, locate $h = \texttt{LCA}(T, u, v)$. In the positive case, it reduces to an interval graph distance query.
- Locate u' and v', the representatives of u and v with respect to h in T_D.
- Compute $\texttt{distance}(u, u')$ and $\texttt{distance}(v, v')$.
- Decide the value of C for the exact distance.

Our labeling scheme will thus consist of the following:

- $\texttt{depth}(T_D, v), \texttt{node\_rank}_{\texttt{POST}}(T_D, v)$.
- A labeling scheme for `LCA` in T which can return $\texttt{depth}(T_D, h)$ and $\texttt{node\_rank}_{\texttt{POST}}(T_D, h)$ of the lowest common ancestor h of two vertices u and v.
- A bitvector of length $n/2$. Its content will be defined later.

The lowest common ancestor scheme we will use is the result of Alstrup et al. [7] which computes arbitrary labels of the lowest common ancestor of two nodes in $O(1)$ time.

▶ **Lemma 11** (Corollary 4.1 of [7]). *There exists an* `LCA`*-labeling scheme for Trees with predefined labels of fixed length k whose worst-case label size is at most $(3+k)\lfloor \log n \rfloor + 1$ bits, with $O(1)$ decode time.*

As our predefined labels that we must return are of size $2\lg n + O(1)$ bits, this `LCA`-labeling scheme uses uses $2\lg^2 n + O(\lg n)$ bits per label.

To perform step 1, we use the `LCA`-labeling scheme, and check if the returned node is u or v, by checking the label $\texttt{node\_rank}_{\texttt{POST}}(T_D, h)$ of the returned node against $\texttt{node\_rank}_{\texttt{POST}}(T_D, u)$ and $\texttt{node\_rank}_{\texttt{POST}}(T_D, v)$. If one is an ancestor of the other, we revert to the interval graph labeling scheme. However, since we do not have $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(v))$, we cannot decide whether to add 1 or 2 in Algorithm 3 (lines 5-8, 11-14). To decide this we will store which term (+1 or +2) to add in the bitvector for deciding C as described below. Otherwise, we obtain the distance tree T_D labels of u, v and h. We again note that v' is the representative of v with respect to h is the node w in algorithm 2. Thus $\texttt{distance}(v, v')$ is simply $\texttt{depth}(v) - \texttt{depth}(v')$, and we do not need the value $\texttt{node\_rank}_{\texttt{POST}}(\texttt{last}(v))$ in the interval graph labeling scheme.

Finally, we need to decide the condition C. As discussed earlier, the condition is equivalent to the set intersection oracle problem. It is conjectured that to compute it in $O(1)$ time, it is necessary to store the result of the queries explicitly, rather than trying to compute it from the sets. Therefore, we will pre-compute the value of C for every pair of vertices. For a pair of vertices u and v where one is an ancestor of the other, we do not need to decide the value of C for this pair, as the computation of the distance between this pair of vertices reduces to the interval graph distance case. In this case, we must determine whether to add 1 or to add 2 in Algorithm 3. We use the bitvector C to store whether to add 1 (the bit 0) or to add 2 (the bit 1). Thus for every vertex v, for each other vertex u, either u is an ancestor or descendant of v, so we store in C a bit for the interval graph distance computation, or u is not an ancestor or descendant of v in which case we store a bit for the chordal graph distance computation of u and v. For a vertex u, the index into the bitvector is $\texttt{node\_rank}_{\texttt{POST}}(T_D, u)$. We may distribute this bitvector more evenly by storing the value for only those vertices u such that $(\texttt{node\_rank}_{\texttt{POST}}(T_D, u) - \texttt{node\_rank}_{\texttt{POST}}(T_D, v)) \mod n \leq n/2$, so that we store $n/2$ bits per vertex in the worst case. For any pair of vertices u and v, this bit for the distance computation is stored in the bitvector of one of the vertices.

The preprocessing time is dominated by the precomputation of the condition C for all pairs of vertices. We may compute all the results of the set intersection oracle problem via matrix multiplication. Details are omitted due to space constraints. Thus, we obtain the following theorem:

▶ **Theorem 12.** *Let G be a chordal graph. Then there exists a distance labeling scheme with maximum label size $n/2 + O(\lg^2 n)$ bits which can compute* `distance` *in $O(1)$ time. The labels can be constructed in $O(n^\omega)$ time where $\omega < 2.371552$ is the matrix multiplication exponent [32].*

References

1 Ittai Abraham and Cyril Gavoille. On approximate distance labels and routing schemes with affine stretch. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 404–415. Springer, 2011. `doi:10.1007/978-3-642-24100-0_39`.

2 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct data structures for families of interval graphs. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, pages 1–13, 2019. doi:10.1007/978-3-030-24766-9_1.

3 Stephen Alstrup, Philip Bille, and Theis Rauhe. Labeling schemes for small distances in trees. *SIAM Journal on Discrete Mathematics*, 19(2):448–462, 2005. doi:10.1137/S0895480103433409.

4 Stephen Alstrup, Søren Dahlgaard, and Mathias Bæk Tejs Knudsen. Optimal induced universal graphs and adjacency labeling for trees. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 1311–1326, 2015. doi:10.1109/FOCS.2015.84.

5 Stephen Alstrup, Cyril Gavoille, Esben Bistrup Halvorsen, and Holger Petersen. Simpler, faster and shorter labels for distances in graphs. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 338–350, USA, 2016. Society for Industrial and Applied Mathematics.

6 Stephen Alstrup, Inge Li Gørtz, Esben Bistrup Halvorsen, and Ely Porat. Distance Labeling Schemes for Trees. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 132:1–132:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2016.132.

7 Stephen Alstrup, Esben Bistrup Halvorsen, and Kasper Green Larsen. Near-optimal labeling schemes for nearest common ancestors. In *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 972–982, 2014. doi:10.1137/1.9781611973402.72.

8 Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. *SIAM Journal on Discrete Mathematics*, 33(1):116–137, 2019. doi:10.1137/16M1105967.

9 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 735–744. ACM, 2000. doi:10.1145/335305.335410.

10 Marthe Bonamy, Louis Esperet, Carla Groenland, and Alex Scott. Optimal labelling schemes for adjacency, comparability, and reachability. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 1109–1117, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3406325.3451102.

11 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.

12 Danny Z. Chen, D. T. Lee, R. Sridhar, and Chandra N. Sekharan. Solving the all-pair shortest path query problem on interval and circular-arc graphs. *Networks*, 31(4):249–258, 1998. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0037%28199807%2931%3A4%3C249%3A%3AAID-NET5%3E3.0.CO%3B2-D.

13 Lenore J Cowen. Compact routing with minimum stretch. *Journal of Algorithms*, 38(1):170–183, 2001. doi:10.1006/jagm.2000.1134.

14 Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *Journal of Algorithms*, 46(2):97–114, 2003. doi:10.1016/S0196-6774(03)00002-6.

15 Pierre Fraigniaud and Amos Korman. An optimal ancestry labeling scheme with applications to XML trees and universal posets. *J. ACM*, 63(1), February 2016. doi:10.1145/2794076.

16 Ofer Freedman, Paweł Gawrychowski, Patrick K. Nicholson, and Oren Weimann. Optimal distance labeling schemes for trees. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 185–194, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087801.3087804.

17 Cyril Gavoille and Christophe Paul. Optimal distance labeling for interval graphs and related graph families. *SIAM J. Discret. Math.*, 22(3):1239–1258, July 2008. doi:10.1137/050635006.

18 Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Distrib. Comput.*, 16(2–3):111–120, September 2003. doi:10.1007/s00446-002-0073-5.

19 Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004. doi:10.1016/j.jalgor.2004.05.002.

20 Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974. doi:10.1016/0095-8956(74)90094-X.

21 Paweł Gawrychowski, Fabian Kuhn, Jakub Łopuszański, Konstantinos Panagiotou, and Pascal Su. Labeling schemes for nearest common ancestors through minor-universal trees. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, pages 2604–2619, USA, 2018. Society for Industrial and Applied Mathematics.

22 Pawel Gawrychowski and Przemyslaw Uznanski. Better distance labeling for unweighted planar graphs. *Algorithmica*, 85(6):1805–1823, 2023. doi:10.1007/S00453-023-01133-Z.

23 Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ISAAC.2020.25.

24 Michal Katz, Nir A. Katz, and David Peleg. Distance labeling schemes for well-separated graph classes. *Discrete Applied Mathematics*, 145(3):384–402, 2005. doi:10.1016/j.dam.2004.03.005.

25 Hung Le and Christian Wulff-Nilsen. Optimal approximate distance oracle for planar graphs. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 363–374. IEEE, 2021. doi:10.1109/FOCS52979.2021.00044.

26 Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2517–2537. SIAM, 2021. doi:10.1137/1.9781611976465.149.

27 J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, pages 67:1–67:12, 2018. doi:10.4230/LIPIcs.ISAAC.2018.67.

28 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.

29 David Peleg. Informative labeling schemes for graphs. *Theoretical Computer Science*, 340(3):577–593, 2005. Mathematical Foundations of Computer Science 2000. doi:10.1016/j.tcs.2005.03.015.

30 Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 315–329, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

31 Gaurav Singh, N. S. Narayanaswamy, and G. Ramakrishna. Approximate distance oracle in o(n 2) time and o(n) space for chordal graphs. In M. Sohel Rahman and Etsuji Tomita, editors, *WALCOM: Algorithms and Computation - 9th International Workshop, WALCOM 2015, Dhaka, Bangladesh, February 26-28, 2015. Proceedings*, volume 8973 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2015. doi:10.1007/978-3-319-15612-5_9.

32 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 3792–3835. SIAM, 2024. doi:10.1137/1.9781611977912.134.

33 Nicholas C. Wormald. Counting labelled chordal graphs. *Graphs and Combinatorics*, 1(1):193–200, 1985. `doi:10.1007/BF02582944`.

34 Peisen Zhang, Eric A. Schon, Stuart G. Fischer, Eftihia Cayanis, Janie Weiss, Susan Kistler, and Philip E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA. *Computer Applications in the Biosciences*, 10(3):309–317, 1994. `doi:10.1093/bioinformatics/10.3.309`.

Algorithms for Galois Words: Detection, Factorization, and Rotation

Diptarama Hendrian
Tokyo Medical and Dental University, Japan

Dominik Köppl
University of Yamanashi, Japan

Ryo Yoshinaka
Tohoku University, Japan

Ayumi Shinohara
Tohoku University, Japan

Abstract

Lyndon words are extensively studied in combinatorics on words – they play a crucial role on upper bounding the number of runs a word can have [Bannai+, SIAM J. Comput.'17]. We can determine Lyndon words, factorize a word into Lyndon words in lexicographically non-increasing order, and find the Lyndon rotation of a word, all in linear time within constant additional working space. A recent research interest emerged from the question of what happens when we change the lexicographic order, which is at the heart of the definition of Lyndon words. In particular, the alternating order, where the order of all odd positions becomes reversed, has been recently proposed. While a Lyndon word is, among all its cyclic rotations, the smallest one with respect to the lexicographic order, a Galois word exhibits the same property by exchanging the lexicographic order with the alternating order. Unfortunately, this exchange has a large impact on the properties Galois words exhibit, which makes it a nontrivial task to translate results from Lyndon words to Galois words. Up until now, it has only been conjectured that linear-time algorithms with constant additional working space in the spirit of Duval's algorithm are possible for computing the Galois factorization or the Galois rotation.

Here, we affirm this conjecture as follows. Given a word T of length n, we can determine whether T is a Galois word, in $O(n)$ time with constant additional working space. Within the same complexities, we can also determine the Galois rotation of T, and compute the Galois factorization of T online. The last result settles Open Problem 1 in [Dolce et al., TCS 2019] for Galois words.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Galois Factorization, Alternating Order, Word Factorization Algorithm, Regularity Detection

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.18

Supplementary Material *Software (Source code)*: `https://github.com/koeppl/galoisword`

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number 23H04378
Ryo Yoshinaka: JSPS KAKENHI Grant Numbers 18K11150 and 20H05703
Ayumi Shinohara: JSPS KAKENHI Grant Number 21K11745

1 Introduction

A *Galois word* is a word that is strictly smaller than all its cyclic rotations with respect to the so-called alternating order, where symbols at odd positions are compared in the usual lexicographic order, but symbols at the remaining positions in the opposite order. While `aab` is clearly the smallest word among all its cyclic rotations `aba` and `baa` under the lexicographic order, `aab` is larger than `aba` under the alternating order because the `b` in the second position is smaller than `a`. In fact, `aba` is a Galois word. Readers familiar with Lyndon words may identify aab to be, nevertheless, a Lyndon word because it is strictly smaller than all its cyclic rotations with respect to the *lexicographic* order. While the definition of Lyndon and Galois words only differ by the used order, the combinatorial differences are astonishing. For instance, on the one hand, Lyndon words cannot have proper borders, i.e., factors appearing both as a prefix *and* as a suffix (but shorter than the word itself). On the other hand, Galois words such as aba can have proper borders of odd lengths [16, Proposition 3.1].

The name *Galois word* has been coined by Reutenauer [16], who introduced these words and derived the naming by a bijection of Galois words and homographic classes of Galois numbers. In the same paper [16], Reutenauer defined a unique factorization of a generalization of Lyndon words, a class of words covering Galois words. Here, we call this factorization *Galois factorization* since we only cover Galois words within the scope of this paper. The Galois factorization is a factorization of a word into a sequence of non-increasing Galois words. Later, Dolce et al. [5] could characterize the first factor of the Galois factorization [5, Theorem 33], and also provide a characterization of Galois words by their prefixes [5, Theorem 32]. However, Dolce et al. left it as an open problem ([5, Open Problem 1]) to find a computation algorithm similar to Duval's algorithm [7] computing the Lyndon factorization. In this paper, we solve this problem by introducing a factorization algorithm (Algorithm 2 and Theorem 32) in the spirit of Duval's algorithm, computing the Galois factorization in linear time with constant additional working space online.

Asides from the above results, we are only aware of the following two articles dealing with Galois words. First, Dolce et al. [6] studied generalized Lyndon-words, among them also Galois words, with respect to infinite orders. Finally, Burcroff and Winsor [3] gave a characterization of infinite generalized Lyndon words, and properties of how finite generalized Lyndon words can be infinitely extended.

2 Related Work

While we covered, to the best of our knowledge, all published results explicitly dealing with Galois words above, Galois words have a strong relation with Lyndon words and the alternating order.

Lyndon. Regarding the former, an exhaustive list of results would go beyond the scope of this paper. We nevertheless highlight that the Lyndon factorization (the aforementioned factorization when outputting factors that are Lyndon words) can be computed in linear time with constant additional space with Duval's algorithm [7]. The same algorithm allows us to judge whether a word is Lyndon. Shiloach [17] gave a linear-time algorithm computing the Lyndon rotation of a primitive word T, i.e., its cyclic rotation that is Lyndon, in constant additional working space.

Alternating Order. Regarding the latter, much work focused on implementing a Burrows–Wheeler transform (BWT) [4] based on the alternating order. While the classic BWT sorts all cyclic rotations of a given input word in lexicographic order, the alternating BWT [11] sorts the cyclic rotations with respect to the alternating order. Gessel et al. [11] not only introduced this BWT variant, but also gave an inversion to restore the original word. Subsequently, Giancarlo et al. [12] gave a linear-time algorithm for computing the alternating BWT. To this end, they compute the Galois rotation of the input word T, i.e., the cyclic rotation of T that is Galois. However, their algorithm computing the Galois rotation needs an auxiliary integer array of length n. Compared to space-efficient algorithms computing the classic

BWT (e.g. [15] with linear time and space linear in the *bit size* of the input text), this is a major bottleneck, but a necessary precomputation step of their algorithm constructing the alternating BWT. Giancarlo et al. [12] also showed how to invert the alternating BWT in linear time. In a follow-up [13], Giancarlo et al. put their discovered properties of the alternating BWT into larger context by covering BWT variants based on a generalized ordering. In that article, they also showed that the alternating BWT can be turned into a compressed self-index that supports pattern counting queries in time linear in the pattern length. The space of the index can be related with the number of symbol runs even when augmented for queries to locate all pattern occurrences in the text, by adapting r-indexing [10] techniques to the alternating BWT.

Our Contribution. This paper makes three contributions to the research on Galois words. First, we give an algorithm (Theorem 25 and Algorithm 1) in Section 5 that checks, for a given word of length n, whether this word is Galois, in $O(n)$ time with constant additional working space. Second, we give an algorithm (Theorem 32 and Algorithm 2) in Section 6 that computes the Galois factorization in $O(n)$ time with constant additional working space online. Finally, we show how to find the Galois rotation (Theorem 36 and Algorithm 3) in Section 7 that in $O(n)$ time with constant additional working space online, paving the way for constructing the alternating BWT in $o(n)$ working space.

We stress that, having an efficient Galois factorization algorithm allows us to merge indexing techniques for the alternate BWT with the bijective BWT [14, 1] to give rise to a BWT-variant that indexes Galois words, whose indexing capabilities are left as future work.

3 Preliminaries

Let Σ be a set of symbols called an *alphabet*. The set of words over Σ is denoted by Σ^*. The empty word is denoted by ε. The *length* of a word $W \in \Sigma^*$ is denoted by $|W|$. The i-th symbol of a word W is denoted by $W[i]$ for $1 \le i \le |W|$ and the factor of W that begins at position i and ends at position j is $W[i..j]$ for $1 \le i \le j \le |W|$. We define $W[i..j] = \varepsilon$ if $i > j$. A word B is a *border* of W if B is a prefix and a suffix of W. We say a border B of W is *proper* if $B \neq W$. For a word T we call an integer $p \in [1..|T|]$ a *period* of T if $T[i+p] = T[i]$ for all $i \in [1..|T|-p]$. In particular, $|T|$ is always a period of T. Let $\mathsf{Per}_o(W)$ and $\mathsf{Per}_e(W)$ be the shortest odd and even period of W if any, respectively. We set $\mathsf{Per}_o(W) = |W|+1$ or $\mathsf{Per}_e(W) = |W|+1$ if W does not have an odd or an even period, respectively. Since the length of a word itself is a period, a word of odd length always has an odd period and a word of even length always has an even period. For a rational number α, let W^α be the word obtained by concatenating W α times. Let W^ω be the infinite repetition of W. We call a word $V \in \Sigma^*$ *primitive* if the fact $V = U^k$ for some word $U \in \Sigma^*$ and an integer $k \ge 1$ implies $V = U$ and $k = 1$. We say that two words X and Y have the same *length-parity* if $|X| \mod 2 = |Y| \mod 2$, i.e., their lengths are both either odd or even.

We denote the standard lexicographic order over words with $\prec_{\text{lex}}$. We define the *alternating order* on words as follows: Given two distinct words S and T such that $S^\omega \neq T^\omega$, with the first mismatching symbol pair at a position j, i.e., $S^\omega[1..j-1] = T^\omega[1..j-1]$ and $S^\omega[j] \neq T^\omega[j]$, we write $S \prec_{\text{alt}} T$ if either (a) j is odd and $S^\omega[j] < T^\omega[j]$, or (b) j is even and $S^\omega[j] > T^\omega[j]$. In addition we denote by $S =_{\text{alt}} T$ if $S^\omega = T^\omega$. For instance, $\texttt{aba} \prec_{\text{alt}} \texttt{aab}$ but $\texttt{aab} \prec_{\text{lex}} \texttt{aba}$; $\texttt{b} \prec_{\text{lex}} \texttt{bba}$ but $\texttt{bba} \prec_{\text{alt}} \texttt{b}$; $\texttt{aba} =_{\text{alt}} \texttt{abaaba}$. We define $\varepsilon \succ_{\text{alt}} X$ for all $X \in \Sigma^+$. We denote by $S \preceq_{\text{alt}} T$ if either $S \prec_{\text{alt}} T$ or $S =_{\text{alt}} T$. We further write $S \sqsubset_{\text{alt}} T$ if $S \prec_{\text{alt}} T$ but neither S is a prefix of T nor vice versa.

We introduce $S \sqsubset_{\text{alt}} T$ for the following reason: For two words S and T with $S \prec_{\text{alt}} T$, it is generally not true that $SU \prec_{\text{alt}} TU$ for all words U (e.g., ab $\prec_{\text{alt}}$ aba but abac $\prec_{\text{alt}}$ abc). However, for $\sqsubset_{\text{alt}}$ we have:

▶ **Fact 1.** *For two words S and T with $S \sqsubset_{alt} T$, it holds that $SU \sqsubset_{alt} TU$ for all words U.*

We also make use of the following additional facts:

▶ **Fact 2.** *For two words S and T with $S^\omega = T^\omega$, there exists a primitive word U integers a and b such that $S = U^a$ and $T = U^b$.*

▶ **Fact 3.** *T is non-primitive if and only if $|T|/p$ is an integer of at least two for p being T's smallest period. If* $\mathsf{Per}_o(T) = |T|$*, then T is primitive.*

▶ **Example 4.** We cannot switch Per_o with Per_e in Fact 3. A counter-example is the non-primitive word $T_1 =$ abaaba, for which we have $\mathsf{Per}_o(T_1) = 3$ but $\mathsf{Per}_e(T_1) = 6$. Also, for $T_2 =$ aa we have $\mathsf{Per}_o(T_2) = 1$ but $\mathsf{Per}_e(T_2) = 2$.

The following property holds for any two periods of a word.

▶ **Lemma 5** ([9]). *Let p and q be periods of a word T. If $p + q - r \leq |T|$, then r is also a period of T, where r is the greatest common divisor of p and q.*

A word is called *Galois* if it is, among all its cyclic rotations, the smallest with respect to $\prec_{\text{alt}}$. By definition, a Galois word has to be primitive (otherwise, it has an identical cyclic rotation that is not strictly larger). The following properties hold of Galois words.

▶ **Lemma 6** ([5, Theorem 14]). *A primitive word T is Galois if and only if T is smaller than all its suffixes, with respect to $\prec_{alt}$.*

▶ **Lemma 7** ([5, Theorem 32]). *A word T is Galois if and only if for any factorization $T = UV$ with $U, V \in \Sigma^+$, one of the following condition holds: (1) $U \prec_{alt} T$ if $|U|$ is even; (2) $U \succ_{alt} T$ if $|U|$ is odd.*

▶ **Lemma 8** ([5, Lemma 35], [16, Proposition 3.1]). *If a Galois word T has a proper border B, then the length of B is odd.*

For example aba and abba are Galois words with a proper border. Unlike for Lyndon words (cf. [7, Proposition 1.3]), it does not hold that, if U and V are Galois words then UV is Galois if $U \prec_{\text{alt}} V$. For instance, aba $\prec_{\text{alt}}$ c but abac is not Galois because ac $\prec_{\text{alt}}$ abac.

4 Characteristics of Pre-Galois Words

In this section, we define pre-Galois words and study their properties. The observations we make here will lead us to helpful tools that we will leverage for proposing the three algorithms in the subsequent sections, namely 1. determining Galois words, 2. computing the Galois factorization, and 3. computing the Galois rotation of a word.

▶ **Definition 9** (Pre-Galois word). *A word T is a* pre-Galois *word if every proper suffix S of T satisfies one or both of the following conditions: (1) S is a prefix of T; (2) $S \succ_{alt} T$.*

In particular, a Galois word is pre-Galois. However, the converse is in general not true; for example $T =$ abaab is pre-Galois but not Galois because ab $\prec_{\text{alt}}$ abaab. In what follows, we introduce a basic property of pre-Galois words.

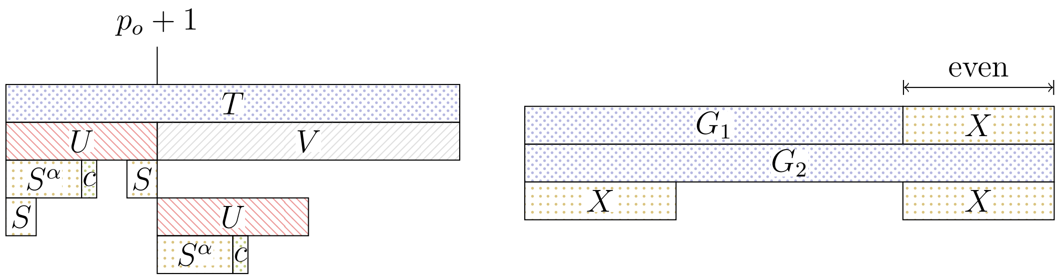


Figure 1 Left: Sketch of the proofs of Lemma 11 and Lemma 12. Right: Sketch of the proof of Lemma 14. As both Galois roots are prefixes of T, we obtain a border X of G_2 with even length which contradicts Lemma 8.

▶ **Lemma 10.** *Let U be a word that is not pre-Galois. Then, for any word V, UV is not pre-Galois. The contraposition is that any prefix of a pre-Galois word is pre-Galois.*

Proof. By definition there exists a proper suffix S of U such that $S \sqsubset_{\mathrm{alt}} U$. By Fact 1, $S \cdot V \sqsubset_{\mathrm{alt}} U \cdot V$ holds. ◀

Next, we study properties of periods of pre-Galois words.

▶ **Lemma 11.** *Let T be a pre-Galois word that has an odd period. Let $p_o = \mathsf{Per}_o(T)$ be the shortest odd period of T. Then $T[1..p_o]$ is Galois.*

Proof. Let $U = T[1..p_o]$ and $T = UV$. By Lemma 10, U is pre-Galois. Assume U is not Galois. Then there exists a proper suffix S of U such that S is a prefix of U and $S \preceq_{\mathrm{alt}} U$. Since p_o is odd and the shortest odd period of T, U is primitive according to Fact 3, and we obtain two observations: First, by Fact 2, if $S^\omega = U^\omega$ then $S = U$, a contradiction to S being proper. Hence, $S \prec_{\mathrm{alt}} U$ must hold.

Second, there exists a rational number $\alpha \geq 1$ and a symbol $c \in \Sigma$ such that $S^\alpha c$ is a prefix of U, $S^\omega[1..|S^\alpha c|] \prec_{\mathrm{alt}} S^\alpha c$, and $|S^\alpha c| < |U|$. See the left of Figure 1 for a sketch. By definition, we have $S^{\alpha-1} = U[1..|S^{\alpha-1}|]$. If $|S|$ is odd, we have $T[|S|+1..|T|] \sqsubset_{\mathrm{alt}} T$, which implies that T is not pre-Galois. Otherwise, if $|S|$ is even, $|U| + |S^{\alpha-1}c| \leq |T|$ since p_o is the shortest odd period of T and therefore $|T| \geq 2|U|$ with $|U| \geq |S^{\alpha-1}c|$. Here, we have $T[|U|-|S|+1..|U|+|S^{\alpha-1}c|] = S^\omega[1..|S^\alpha c|]$. Therefore, $T[|U|-|S|+1..|U|+|S^{\alpha-1}c|] \sqsubset_{\mathrm{alt}} T[1..|S^\alpha c|]$, which implies T is not pre-Galois. ◀

A similar property also holds for even periods.

▶ **Lemma 12.** *Let T be a pre-Galois word that has an even period. Let $p_e = \mathsf{Per}_e(T)$ be the shortest even period of T. Then $T[1..p_e]$ is Galois if primitive.*

Proof. We follow the proof of Lemma 11 by replacing U there with $T[1..p_e]$. We also give there p_e the role of p_o. We can do that because we assume that U is primitive, so we obtain a proper border S of U like in the previous proof. ◀

We are in particular interested in prefixes of pre-Galois words that are Galois. To formalize this idea, we define *Galois roots* of a pre-Galois word as follows.

▶ **Definition 13** (Galois root). *Let P be a prefix of a pre-Galois word T. We call P a* Galois root *of T if $|P|$ is a period of T and P is Galois.*

In addition to our aforementioned example $T = \texttt{abaab}$, $\texttt{aba}$ is a Galois root of T. Also, the words $T[1..p_o]$ in Lemma 11 and $T[1..p_e]$ Lemma 12, are Galois roots of T if they are, respectively, primitive. Note that a pre-Galois word T has at least one Galois root, namely T's prefix of length equal to T's shortest period. While a pre-Lyndon word has exactly one Lyndon root, a pre-Galois word can have two different Galois roots:

▶ **Lemma 14.** *A pre-Galois word T can have at most two Galois roots, and their lengths have different parities.*

Proof. Assume that there are two Galois roots G_1 and G_2 with the same length-parity. Then the length difference of G_1 and G_2 is even. Without loss of generality, suppose that $|G_1| < |G_2|$. Then the suffix $X = G_2[|G_1| + 1..]$ of G_2 is also a prefix of G_2 since $T = G_1^{\alpha} = G_2^{\beta}$ for rational numbers α and β. Hence, X is a border of G_2 with even length, which is impossible due to Lemma 8. See the right of Figure 1 for a sketch. ◀

In what follows, we name the odd-length and the even-length Galois root, if they exist, by G_{o} and G_{e}, respectively. By Lemma 14, they are well-defined. For example, consider $T = \texttt{aba}$. The two prefixes $\texttt{ab}$ and $\texttt{aba}$ are both Galois, for which $T = (\texttt{ab})^{3/2} = (\texttt{aba})^1$.

From Lemma 12, $T[1..p_e]$ is Galois only when it is primitive. Next, we consider the case where $T[1..p_e]$ is not primitive.

▶ **Lemma 15.** *Let T be an even-length pre-Galois word with no even-length Galois root, i.e., $T[1..p_e]$ is not primitive, where $p_e = \mathsf{Per}_e(T)$. Then there exists $G_o = T[1..p_o]$ such that $T = G_o^k G'_o$ with $k \geq 2$, where $p_o = \mathsf{Per}_o(T)$ and G'_o is a prefix of G_o.*

Proof. Since $|T|$ is even, T has a period of even length. Let p_e be the shortest even period of T. By Lemma 12, $U = T[1..p_e]$ is Galois if U is primitive. Since U is not primitive and p_e is the smallest even period of T, we have $\mathsf{Per}_o(T) = p_o = p_e/2$. Thus, there is an odd-length prefix $G_o = T[1..p_o]$ of U such that $U = G_o^2$. ◀

▶ **Lemma 16.** *Let G_o be an odd-length Galois root of a pre-Galois word $T = G_o^k G'_o$ with $k \geq 2$ and G'_o is a prefix of G_o. Then T has no even-length Galois root.*

Proof. Since $T = G_{\mathrm{o}}^k G'_{\mathrm{o}}$, $2|G_{\mathrm{o}}|$ is a period of T. Assume that T has a shorter even period $p_e < 2|G_{\mathrm{o}}|$. By Lemma 8, G_{o} does not have a proper border of even length. Because the two conditions (a) $G_{\mathrm{o}}G_{\mathrm{o}}$ has even length and (b) $p_e \in [|G_{\mathrm{o}}| + 1..2|G_{\mathrm{o}}| - 1]$ would imply that $G_{\mathrm{o}}G_{\mathrm{o}}$ (and thus G_{o} due to the length of p_e) has a border of even length, p_e must be less than $|G_{\mathrm{o}}|$. However, by the periodicity lemma (Lemma 5), there exists an odd period shorter than G_{o}, which contradicts that $|G_{\mathrm{o}}|$ is the shortest odd-length Galois period of T. ◀

▶ **Example 17.** Let $T = \texttt{abaa}$ be an even-length Galois word. T has the odd-length Galois root $\texttt{aba}$. By appending $\texttt{b}$ to T, we obtain $\texttt{abaab}$, which is pre-Galois with no even-length Galois root. $T \cdot \texttt{b}$ can be written as $(\texttt{aba})^{5/3}$, a fractional repetition of $\texttt{aba}$.

By Lemmas 15 and 16, if T has an even period, $T[1..p_e]$ is either G_e or G_{o}^2.

5 Determining Galois Words

The algorithm we propose checks if a word T is Galois by reading T from left to right. For that, we want to maintain the Galois roots of the prefix of T read so far. To this end, we study the Galois roots of $T' = T \cdot z$, i.e., when appending a symbol z to T. Our main observation can be stated as follows:

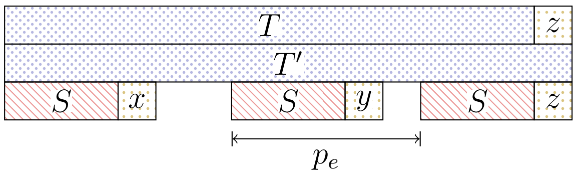


Figure 2 Sketch of the proof of Lemma 20. The caption p_e can be also considered as p_o for the latter case.

▶ **Theorem 18.** *Let T be a pre-Galois word, $p_o = \mathsf{Per}_o(T)$, and $p_e = \mathsf{Per}_e(T)$. Given a symbol z, the extension $T' = T \cdot z$ is a pre-Galois word if and only if both conditions $T'[1..|T| - p_o + 1] \preceq_{alt} T'[p_o + 1..|T| + 1]$ and $T'[1..|T| - p_e + 1] \preceq_{alt} T'[p_e + 1..|T| + 1]$ hold.*

In what follows, we break down the statement of this theorem into two lemmas for each direction. First, we consider the case where T' cannot be a pre-Galois word.

▶ **Lemma 19.** *Let T be a pre-Galois word, $p_o = \mathsf{Per}_o(T)$, and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T' = T \cdot z$, such that either $T'[1..|T| - p_o + 1] \succ_{alt} T'[p_o + 1..|T| + 1]$ or $T'[1..|T| - p_e + 1] \succ_{alt} T'[p_e + 1..|T| + 1]$. Then the extension T' is not a pre-Galois word.*

Proof. We treat here only the case involving p_o because the other case involving p_e can be proved similarly. If $p_o = |T|+1$, $T'[1..|T|-p_o+1] = T'[p_o+1..|T|+1] = \varepsilon$. Thus, this case does not meet the requirements of the lemma statement. It remains to consider $p_o \leq |T|$. For that, suppose $T'[1..|T|-p_o+1] \succ_{\text{alt}} T'[p_o+1..|T|+1]$. Since $T'[1..|T|-p_o+1] \neq T'[p_o+1..|T|+1]$, $T'[p_o + 1..|T| + 1]$ not a prefix of T'. Thus, T' is not pre-Galois. ◀

For all other cases, we show that T' is a pre-Galois word.

▶ **Lemma 20.** *Let T be a pre-Galois word, $p_o = \mathsf{Per}_o(T)$, and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T' = T \cdot z$, such that $T'[1..|T| - p_o + 1] \preceq_{alt} T'[p_o + 1..|T| + 1]$ and $T'[1..|T| - p_e + 1] \preceq_{alt} T'[p_e + 1..|T| + 1]$. Then the extension T' is a pre-Galois word.*

Proof. We prove the contraposition, i.e., if T' is not a pre-Galois word, either $T'[1..|T| - p_o + 1] \succ_{\text{alt}} T'[p_o + 1..|T| + 1]$ or $T'[1..|T| - p_e + 1] \succ_{\text{alt}} T'[p_e + 1..|T| + 1]$ holds.

Suppose T' is not a pre-Galois word. Since T is pre-Galois and T' is not pre-Galois, there exists a suffix S of T such that S is a prefix of T but $S \cdot z$ is not a prefix of T' and $S \cdot z \prec_{\text{alt}} T'$, i.e., $S = T'[1..|S|]$ and $S \cdot z \sqsubset_{\text{alt}} T'[1..|S| + 1]$. In what follows we consider three cases. The first case is that S is the empty word. But then $z \prec_{\text{alt}} T'[1]$, and therefore T' is not pre-Galois. The other cases concern the length-parity of T and S.

Consider T and S have the *same* length-parity. Since p_e is even, $T'[1..|T| - p_e]$ and S also have the same length-parity. Since S is a proper border of T, $p_e \leq |T| - |S|$. Here we have $S = T'[1..|S|] = T'[|T| - p_e - |S| + 1..|T| - p_e]$. Let $x = T'[|S| + 1]$ and $y = T'[|T| - p_e + 1]$. See Figure 2 for a sketch. Since $|T|$ is pre-Galois, we have $S \cdot y \succeq_{\text{alt}} S \cdot x$. Moreover, $S \cdot x \succ_{\text{alt}} S \cdot z$ holds by $T'[1..|S| + 1] \succ_{\text{alt}} S \cdot z$. Thus we have $S \cdot y \succ_{\text{alt}} S \cdot z$. Since $T'[1..|T| - p_e]$ and $|S|$ have the same parity, we have $T'[1..|T| - p_e + 1] \succ_{\text{alt}} T'[p_e + 1..|T| + 1]$.

Consider T and S have *different* length-parity. Since p_o is odd, $T'[1..|T| - p_o]$ and $|S|$ have the same parity. Note that S is a proper border of T thus $p_o \leq |T| - |S|$. Here we have $S = T'[1..|S|] = T'[|T| - p_o - |S| + 1..|T| - p_o]$. Let $x = T'[|S| + 1]$ and $y = T'[|T| - p_o + 1]$. Since $|T|$ is pre-Galois, we have $S \cdot y \succeq_{\text{alt}} S \cdot x$. Moreover $S \cdot x \succ_{\text{alt}} S \cdot z$ holds by $T'[1..|S| + 1] \succ_{\text{alt}} S \cdot z$. Thus we have $S \cdot y \succ_{\text{alt}} S \cdot z$. Since $T'[1..|T| - p_o]$ and $|S|$ have the same parity, we have $T'[1..|T| - p_o + 1] \succ_{\text{alt}} T'[p_o + 1..|T| + 1]$. ◀

Next we show how periods change when we append a symbol to a pre-Galois word T. Here, we focus on p_o first. The cases for p_e can be proven in a similar way. The claim of the first lemma follows by definition.

▶ **Lemma 21.** *Let T be a pre-Galois word and $p_o = \mathsf{Per}_o(T)$. Consider a symbol z and the extension $T' = T \cdot z$, such that $z = T'[|T| - p_o + 1]$. Then the extension T' has $\mathsf{Per}_o(T') = p_o$.*

▶ **Lemma 22.** *Let T be a pre-Galois word and $p_o = \mathsf{Per}_o(T)$. Consider a symbol z and the extension $T' = T \cdot z$ with $T'[1..|T| - p_o + 1] \prec_{alt} T'[p_o + 1..|T| + 1]$. Then,*

$$\mathsf{Per}_o(T') = \begin{cases} |T'| & \text{if } |T'| \text{ is odd,} \\ |T'| + 1 & \text{otherwise.} \end{cases}$$

Proof. If $|T|$ has no odd period, i.e., $p_o = |T|+1 = |T'|$, then $|T|$ is even. Thus, $|T'|$ is odd and $\mathsf{Per}_o(T') = |T'|$. Otherwise, suppose that $|T|$ has an odd period. Assume T' has odd period $p'_o < |T'|$. Thus, we have $T'[1..|T| - p'_o + 1] = T'[p'_o + 1..|T| + 1]$. Let $S = T'[1..|T| - p'_o]$ and $y = T'[|T| - p_o + 1]$. Since T is pre-Galois, we have $S \cdot z \preceq_{\text{alt}} S \cdot y$. However, by $T'[1..|T| - p_o + 1] \prec_{\text{alt}} T'[p_o + 1..|T| + 1]$, we have $S \cdot y \prec_{\text{alt}} S \cdot z$, which is a contradiction. Therefore, T' has no odd period p'_o with $p'_o < |T'|$, which implies $\mathsf{Per}_o(T') = |T'|$ if $|T'|$ is odd or $\mathsf{Per}_o(T') = |T'| + 1$ if $|T'|$ is even. ◀

In a similar way, we can show the following lemmas.

▶ **Lemma 23.** *Let T be a pre-Galois word and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T' = T \cdot z$, such that $z = T'[|T| - p_e + 1]$. Then the extension T' has $\mathsf{Per}_e(T') = p_e$.*

▶ **Lemma 24.** *Let T be a pre-Galois word and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T'[1..|T| - p_e + 1] \prec_{alt} T'[p_e + 1..|T| + 1]$. Then,*

$$\mathsf{Per}_e(T') = \begin{cases} |T'| & \text{if } |T'| \text{ is even,} \\ |T'| + 1 & \text{otherwise.} \end{cases}$$

With Algorithm 1 we give algorithmic instructions in how to verify whether an input word T is Galois. For each position in T, the algorithm performs a constant number of symbol comparisons on T. Storing only the two periods p_e and p_o of the processed prefix up so far, it thus runs in linear time with a constant number of words extra to the input word T. We obtain the following theorem:

▶ **Theorem 25.** *Given a word T, we can verify whether T is Galois in $O(|T|)$ time with $O(1)$ working space.*

6 Computing the Galois Factorization Online

In this section we present an online algorithm for computing the Galois factorization of a given word. We first start with a formal definition of the Galois factorization, introduce a key property called $\mathsf{SPref}(T)$, and then show how to compute $\mathsf{SPref}(T)$. The Galois factorization of a word T is defined as follows.

▶ **Definition 26** (Galois factorization). *A factorization $T = G_1 \cdot G_2 \cdots G_k$ is the* Galois factorization *of T if G_i is Galois for $1 \leq i \leq k$ and $G_1 \succeq_{alt} G_2 \succeq_{alt} \cdots \succeq_{alt} G_k$ holds.*

It is known that every word admits just one Galois factorization (see [16, Théorème 2.1] or [5]). We denote the Galois factorization $T = G_1 \cdot G_2 \cdots G_k$ of T by $\mathsf{GF}(T) = (G_1, G_2, \ldots, G_k)$. The Galois factorization has the following property.

Algorithm 1 Determining whether a word is Galois, see Theorem 25.

```
Function IsGalois($T$)                              // Assume $|T| \geq 2$, otherwise always true
    $p_o = 1$; $p_e = 2$;
    for $i$ from 2 to $|T|$ do                      // Loop-Invariant: $T[1..i-1]$ is pre-Galois
        if $i$ is odd then
            if $p_e < i$ then
                if $T[i] < T[i - p_e]$ then return False;             // Lemma 19
                else if $T[i] > T[i - p_e]$ then $p_e = i + 1$;       // Lemma 24
            if $p_o < i$ then
                if $T[i] < T[i - p_o]$ then $p_o = i$;                // Lemma 22
                else if $T[i] > T[i - p_o]$ then return False;        // Lemma 19
        else
            if $p_e < i$ then
                if $T[i] < T[i - p_e]$ then $p_e = i$;                // Lemma 24
                else if $T[i] > T[i - p_e]$ then return False;        // Lemma 19
            if $p_o < i$ then
                if $T[i] < T[i - p_o]$ then return False;             // Lemma 19
                else if $T[i] > T[i - p_o]$ then $p_o = i + 1$;       // Lemma 22
    if $p_o = |T|$ then                                               // Is $T$ primitive?
        return True                                   // $T$ is Galois by Lemma 11
    else if $p_e = |T|$ and $p_e \neq 2p_o$ then
        return True                      // $T$ is Galois by Lemma 12 and Lemma 15.
    return False               // $T$ is pre-Galois but not primitive (hence not Galois)
```

▶ **Lemma 27** ([5, Theorem 33]). *Let* $\mathsf{GF}(T) = (G_1, G_2, \ldots, G_k)$ *be the Galois factorization of a word* T *of length* n*. Let* P *be the shortest non-empty prefix of* T *such that*

$$P \succeq_{alt} T \text{ if } |P| \text{ is even and } P \preceq_{alt} T \text{ if } |P| \text{ is odd.} \tag{1}$$

Then we have

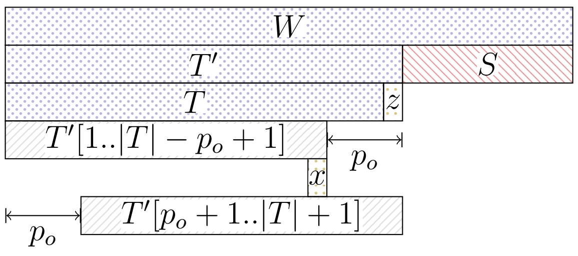
$$P = \begin{cases} G_1^2 & \text{if } |G_1| \text{ is odd, } m \text{ is even, and } m < k, \\ G_1 & \text{otherwise,} \end{cases}$$

where the integer m *is the multiplicity of* G_1*, i.e.,* $G_i = G_1$ *for* $i \leq m$*, but* $G_{m+1} \neq G_1$*.*

We denote such P in Lemma 27 by $\mathsf{SPref}(T)$. If we can compute $\mathsf{SPref}(T)$ for any word T, we can compute the Galois factorization of T by recursively computing $\mathsf{SPref}(T)$ from the suffix remaining when removing the prefix $\mathsf{SPref}(T)$. To this end, we present a way to compute $\mathsf{SPref}(T)$ by using the periods of T.

▶ **Lemma 28.** *Let* T *be a pre-Galois word,* $p_o = \mathsf{Per}_o(T)$*, and* $p_e = \mathsf{Per}_e(T)$*. Consider a symbol* z *and the extension* $T' = T \cdot z$*, such that (a)* $T'[1..|T| - p_o + 1] \succ_{alt} T'[p_o + 1..|T| + 1]$ *or (b)* $T'[1..|T| - p_e + 1] \succ_{alt} T'[p_e + 1..|T| + 1]$ *hold (both (a) and (b) can hold at the same time). Then, for any word* S*, we have* $\mathsf{SPref}(T'S) = T'[1..p]$ *such that*

$$p = \begin{cases} \min\{p_o, p_e\} & \text{if } T'[1..|T| - p_o + 1] \succ_{alt} T'[p_o + 1..|T| + 1] \\ & \quad \text{and } T'[1..|T| - p_e + 1] \succ_{alt} T'[p_e + 1..|T| + 1], \text{ ((a) and (b))} \\ p_o & \text{if } T'[1..|T| - p_e + 1] \preceq_{alt} T'[p_e + 1..|T| + 1], \text{ ((b) but not (a))} \\ p_e & \text{otherwise. ((a) but not (b))} \end{cases} \tag{2}$$



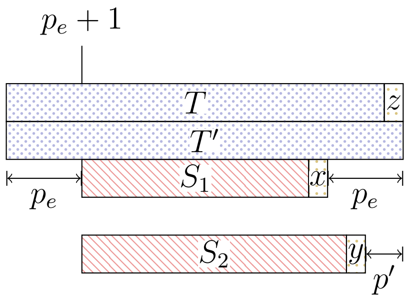


Figure 3 Sketch of the proof of Lemma 28 (left) and of Lemma 29 (right).

Proof. Let $W = T'S$ for some arbitrary word $W \in \Sigma^*$. Suppose $T'[1..|T| - p_o + 1] \succ_{\text{alt}} T'[p_o{+}1..|T|{+}1]$ holds. Let $x = T'[|T|{-}p_o{+}1]$. Since T and $T'[1..|T|{-}p_o]$ have different length-parities, we have $T \cdot x \prec_{\text{alt}} T \cdot z$, which implies $T'[1..p_o] \prec_{\text{alt}} W$. See the left of Figure 3 for a sketch. Similarly, we can show that $T'[1..p_e] \prec_{\text{alt}} W$ if $T'[1..|T|{-}p_e{+}1] \succ_{\text{alt}} T'[p_e{+}1..|T|{+}1]$.

Next, we show the minimality of p. Consider a prefix X such that $|X| < p$. Let $T[1..q]$ be the longest prefix of T such that $|X|$ is its period. Then, we have $T[1..q-|X|] = T'[|X|+1..q]$ and $T[1..q-|X|+1] \neq T'[|X|+1..q+1]$. Since T is pre-Galois, we have $T[1..q-|X|+1] \prec_{\text{alt}} T'[|X|+1..q+1]$. If $|X|$ is odd, $X \succ_{\text{alt}} T[1..q+1]$, otherwise if $|X|$ is even, $X \prec_{\text{alt}} T[1..q+1]$, which does not satisfy the condition of Equation (1) in Lemma 27. ◀

By using the property shown in Lemma 28, we can compute $\mathsf{GF}(W)$ by computing $\mathsf{SPref}(W)$ recursively. For example, given $W = UV$ with $U = \mathsf{SPref}(W)$, after computing $\mathsf{SPref}(W)$ to get U, we recurse on the remaining suffix V and compute $\mathsf{SPref}(V)$ to get the next factor. We can modify Algorithm 1 to output $\mathsf{SPref}(W)$ in $O(\ell)$ time, where ℓ is the length of the longest pre-Galois prefix of W. However, it takes time if we compute $\mathsf{GF}(W)$ by the recursive procedure, especially when ℓ is much larger than $|\mathsf{SPref}(W)|$. To tackle this problem, we use the following property.

▶ **Lemma 29.** *Let T be a pre-Galois word and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T' = T{\cdot}z$, such that $T'[1..|T|{-}p_e{+}1] \succ_{alt} T'[p_e{+}1..|T|{+}1]$. Let $\mathsf{SPref}(T') = T'[1..p]$. If $p = p_e$ and $|T| \geq 2p$, we have $\mathsf{SPref}(T'[p+1..]) = T'[p+1..2p] = T'[1..p]$.*

Proof. Since $T'[1..|T| - p_e]$ and $T'[p_e + 1..|T| - p_e]$ have the same length-parity, we have $T'[p_e{+}1..|T|{-}p_e{+}1] \succ_{\text{alt}} T'[2p_e{+}1..|T|{+}1]$. Assume that $T[p_e{+}1..]$ has a period $p' < p_e$ and $T'[p_e{+}1..|T|{-}p'{+}1] \succ_{\text{alt}} T'[p_e{+}p'{+}1..|T|{+}1]$. Since T is pre-Galois, $T'[1..p_e] = T'[p_e{+}1..2p_e]$ has no even border. Thus p' is odd. Let $S_1 = T'[p_e + 1..|T| - p_e]$, $S_2 = T'[p_e + 1..|T| - p']$, $x = T'[|T|{-}p_e{+}1]$, and $y = T'[|T|{-}p'{+}1]$. By $T'[p_e{+}1..|T|{-}p'{+}1] \succ_{\text{alt}} T'[p_e{+}p'{+}1..|T|{+}1]$, we have $S_2 \cdot y \succ_{\text{alt}} S_2 \cdot z$. See the right of Figure 3 for a sketch. Since S_1 and S_2 have different parities, we have $S_1 \cdot y \prec_{\text{alt}} S_1 \cdot z$. Moreover, by $T'[p_e{+}1..|T|{-}p_e{+}1] \succ_{\text{alt}} T'[2p_e{+}1..|T|{+}1]$, we have $S_1 \cdot x \succ_{\text{alt}} S_1 \cdot z$, which implies $S_1 \cdot x \succ_{\text{alt}} S_1 \cdot y$. However, since $T[p_e + 1..]$ is pre-Galois, $S_1 \cdot x \preceq_{\text{alt}} S_1 \cdot y$, which contradicts $S_1 \cdot x \succ_{\text{alt}} S_1 \cdot y$. Therefore, $T[p_e + 1..]$ has no period p' with $p' < p_e$. ◀

Let $U = \mathsf{SPref}(W)$, $|U|$ is even, and $W = U^k V$ for some $k \geq 2$. By Lemma 29, we know that $\mathsf{SPref}(U^j V) = U$ for $1 \leq j < k$ without computing it explicitly. Next, we consider the case when $|\mathsf{SPref}(W)|$ is odd.

Algorithm 2 Computing the Galois factorization, as claimed in Theorem 32.

```
Function GaloisFactorization($T$)
    Append \$ to $T$;
    Fact = ( ) empty list; $i = 0$;
    while $i \leq |T|$ do
        $p_o = 1$; $p_e = 2$;
        for $j$ from 2 to $|T| - i$ do
            $p = |T| + 2$; $p'_e = p_e$; $p'_o = p_o$;
            if $j$ is odd then
                if $p_e < j$ then
                    if $T[i + j] < T[i + j - p_e]$ then $p = \min\{p, p_e\}$;          // Lemma 28
                    else if $T[i + j] > T[i + j - p_e]$ then $p'_e = j + 1$;          // Lemma 24
                if $p_o < j$ then
                    if $T[i + j] < T[i + j - p_o]$ then $p'_o = j$;                   // Lemma 22
                    else if $T[i + j] > T[i + j - p_o]$ then $p = \min\{p, p_o\}$;   // Lemma 28
            else
                if $p_e < j$ then
                    if $T[i + j] < T[i + j - p_e]$ then $p'_e = j$;                   // Lemma 24
                    else if $T[i + j] > T[i + j - p_e]$ then $p = \min\{p, p_e\}$;   // Lemma 28
                if $p_o < j$ then
                    if $T[i + j] < T[i + j - p_o]$ then $p = \min\{p, p_o\}$;        // Lemma 28
                    else if $T[i + j] > T[i + j - p_o]$ then $p'_o = j + 1$;          // Lemma 22
            if $p \neq |T| + 2$ then
                while $j > p$ do
                    if $p = p_e$ and $p_e = 2p_o$ then                                 // Lemma 29
                        Append $T[i..i + p_o - 1]$ and $T[i + p_o..i + 2p_o - 1]$ to Fact;
                    else
                        Append $T[i..i + p - 1]$ to Fact;
                    $i = i + p$; $j = j - p$;
                    $p = p_e$;
                break;
            $p_e = p'_e$; $p_o = p'_o$;
    return Fact;
```

▶ **Lemma 30.** *Let T be a pre-Galois word, $p_o = \mathsf{Per}_o(T)$, and $p_e = \mathsf{Per}_e(T)$. Consider a symbol z and the extension $T' = T \cdot z$, such that $T'[1..|T| - p_o + 1] \succ_{alt} T'[p_o + 1..|T| + 1]$. Let $P = T'[1..p] = \mathsf{SPref}(T')$. If $p = p_o$ and $|T| \geq 3p$, we have $\mathsf{SPref}(T'[p + 1..]) = T'[p + 1..3p] = T'[1..2p]$.*

Proof. Since T is pre-Galois, $T'[1..p_o] = T'[p_o + 1..2p_o]$ does not have an even border. Thus $\mathsf{Per}_o(T[p_o + 1..]) = p_o$ and $\mathsf{Per}_e(T[p_o + 1..]) = 2p_o$. Moreover, since $T'[1..|T| - p_o]$ and $T'[p_o + 1..|T| - p_o]$ have different parities, we have $T'[p_o + 1..|T| - p_o + 1] \prec_{\text{alt}} T'[2p_o + 1..|T| + 1]$. Next, $T'[p_o + 1..|T| - 2p_o + 1] \succ_{\text{alt}} T'[3p_o + 1..|T| + 1]$ holds, since $T'[1..|T| - p_o]$ and $T'[p_o + 1..|T| - 2p_o]$ have the same parity. Therefore, $\mathsf{SPref}(T'[p_o + 1..]) = T'[p_o + 1..3p] = T'[1..2p_o]$. ◀

Let $U = \mathsf{SPref}(W)$, $|U|$ is odd, and $W = U^kV$ for some $k \geq 3$. By Lemmas 29 and 30, we know that $\mathsf{SPref}(U^{k-2j-1}V) = U^2$ for $0 \leq j < \lceil k/2 \rceil$ without computing it explicitly.

Lemmas 28, 29, and 30 are used to factorize a pre-Galois word when we extended it. However, we can not use the Lemmas to factorize T when T is pre-Galois but not Galois and the input is terminated. Although we can factorize T by finding $P = \mathsf{SPref}(T)$ in Lemma 27, we need an additional procedure to find such P. To simplify our algorithm, we append a terminal symbol \$ that is smaller than all symbols of Σ. In particular, all other symbols in W are different from \$.[1] Here we show that the appended \$ determines a Galois factor of length one, thus it does not affect the factorization result.

▶ **Lemma 31.** *Consider a symbol \$ that does not appear in a word T and $\$ \prec c$ for any $c \in \Sigma$. Then, $\mathsf{GF}(T) = (G_1, G_2, \dots, G_k)$ iff $\mathsf{GF}(T \cdot \$) = (G_1, G_2, \dots, G_k, \$)$.*

Proof. Let $\mathsf{GF}(T) = (G_1, G_2, \dots, G_k)$. Here, $G_1 \succeq_{\text{alt}} G_2 \succeq_{\text{alt}} \dots \succeq_{\text{alt}} G_k$. Since $\$ \prec c$ for any $c \in \Sigma$, we have $G_1 \succeq_{\text{alt}} G_2 \succeq_{\text{alt}} \dots \succeq_{\text{alt}} G_k \succeq_{\text{alt}} \$$. Therefore, $\mathsf{GF}(T \cdot \$) = (G_1, G_2, \dots, G_k, \$)$. Similarly, let $\mathsf{GF}(T \cdot \$) = (G_1, G_2, \dots, G_k, \$)$. Here, $G_1 \succeq_{\text{alt}} G_2 \succeq_{\text{alt}} \dots \succeq_{\text{alt}} G_k \succeq_{\text{alt}} \$$. Therefore, $\mathsf{GF}(T) = (G_1, G_2, \dots, G_k)$. ◀

This gives us the final ingredient for introducing the algorithmic steps for computing the Galois factorization, which we present as pseudo code in Algorithm 2.

▶ **Theorem 32.** *The Galois factorization of a word T can be computed in $O(|T|)$ time and $O(1)$ additional working space, excluding output space.*

Proof. The correctness of Algorithm 2 is proven by Lemmas 28, 29, 30, and 31. Next, we prove the time complexity of Algorithm 2. The time complexity of the algorithm is bounded by the number of iterations of the inner loop (Line 6). The algorithm increments j until it finds a prefix to be factorized (Line 22). Here we show that $j \leq 3\ell + 1$, where ℓ is the length of the factorized prefix. Let $p = p_e$. If $j < 2p + 1$, it is clear that $j \leq 3\ell + 1$, where $\ell = p$. Otherwise, if $j \geq 2p + 1$, the algorithm factorizes the prefix recursively k times, such that $kp \geq j$ and $(k + 1)p > j$. Thus, we have $\ell = kp$ which implies $j \leq 3\ell + 1$. On the other hand, let $p = p_o$. If $j < 3p + 1$, its clear that $j \leq 3\ell + 1$, where $\ell = p$. Otherwise, if $j \geq 3p + 1$, the algorithm factorizes the prefix recursively k times, such that $kp \geq j$ and $(k + 2)p > j$. Thus, we have $\ell = kp$ which implies $j \leq 3\ell + 1$. Therefore, the number of iterations of the inner loop is $O(|T|)$, since the total length of the factors is $|T|$. ◀

7 Computing Galois rotation

While we can infer the Lyndon rotation of a word T from the Lyndon factorization of $T \cdot T$, the same kind of inference surprisingly does not work for Galois words [5, Example 41]. Here, we present an algorithm computing the Galois rotation, using constant additional working space. The algorithm is a modification of Algorithm 2. We start with formally defining the Galois rotation of a word.

▶ **Definition 33.** *Let W be a primitive word. A rotation $T = VU$ is a* Galois rotation *of $W = UV$ if T is Galois.*

[1] Without \$, the last factor we report might be just pre-Galois, not Galois. So we have to break the last factor into Galois factors. If W ends with the unique symbol \$, then \$ cannot be included in another Galois factor of W; it has to stay alone as a Galois factor of length one, and thus we cannot end with the last factor being just pre-Galois.

Algorithm 3 Computing the Galois rotation of T, as claimed in Theorem 36.

```
Function GaloisRotation($T$)
    $i = 0$;
    while $i \leq 3|T|$ do
        $p_o = 1$; $p_e = 2$;
        for $j$ from 2 to $3|T| - i$ do
            $p = 3|T| + 2$; $p'_e = p_e$; $p'_o = p_o$;
            if $j$ is odd then
                if $p_e < j$ then
                    if $T[i+j] < T[i+j-p_e]$ then $p = \min\{p, p_e\}$;          // Lemma 28
                    else if $T[i+j] > T[i+j-p_e]$ then $p'_e = j + 1$;          // Lemma 24
                if $p_o < j$ then
                    if $T[i+j] < T[i+j-p_o]$ then $p'_o = j$;                   // Lemma 22
                    else if $T[i+j] > T[i+j-p_o]$ then $p = \min\{p, p_o\}$;    // Lemma 28
            else
                if $p_e < j$ then
                    if $T[i+j] < T[i+j-p_e]$ then $p'_e = j$;                   // Lemma 24
                    else if $T[i+j] > T[i+j-p_e]$ then $p = \min\{p, p_e\}$;    // Lemma 28
                if $p_o < j$ then
                    if $T[i+j] < T[i+j-p_o]$ then $p = \min\{p, p_o\}$;          // Lemma 28
                    else if $T[i+j] > T[i+j-p_o]$ then $p'_o = j + 1$;          // Lemma 22
            if $p \neq 3|T| + 2$ then
                while $j > p$ do
                    $i = i + p$; $j = j - p$;
                    $p = p_e$;
                break;
            $p_e = p'_e$; $p_o = p'_o$;
            if $p_o \geq |T|$ and $p_e \geq |T|$ then
                return $(i \mod |T|) + 1$;
```

To describe our algorithm computing Galois rotations, we study a property of the Galois factorization for repetitions of a Galois word.

▶ **Lemma 34.** *Let T be a Galois word and $P = \mathsf{SPref}(T^k)$ for some rational number $k \geq 2$. Then $|P| \geq |T|$.*

Proof. Suppose that $|P| < |T|$ and $|P|$ is even. Since T is primitive, $|P|$ is not a period of T^2 by Lemma 5. Thus, there exist a position $i < 2|T|$ such that $P^\omega[i] \neq T^2[i]$. Moreover, we have $P \succeq_{\text{alt}} T^k$ by Lemma 27, which implies $P \succ_{\text{alt}} T^2$. However, we have $P \prec_{\text{alt}} T =_{\text{alt}} T^2$ by Lemma 7, which is a contradiction. The case that $|P|$ is odd leads to a similar contradiction, and thus $|P| \geq |T|$ must hold. ◀

We then use the above property to show the following lemma, which is the core of our algorithm.

▶ **Lemma 35.** *Let W be the Galois rotation of a primitive word T. Given $TTT = UWWV$ with $|U| < |T|$, let $\mathsf{GF}(U) = (G_1, G_2, \ldots, G_k)$ and $\mathsf{GF}(WWV) = (H_1, H_2, \ldots, H_l)$. Then we have $\mathsf{GF}(UWWV) = (G_1, G_2, \ldots, G_k, H_1, H_2, \ldots, H_l)$.*

Proof. For $U = \varepsilon$, the claim trivially holds with $V = W$ and $\mathsf{GF}(UWWV) = \mathsf{GF}(WWV) = (H_1, H_2, \ldots, H_l)$. In the rest of the proof we assume $U \neq \varepsilon$. Let $\mathsf{GF}(U) = (G_1, G_2, \ldots, G_k)$ and $\mathsf{GF}(WWV) = (H_1, H_2, \ldots, H_l)$. Because $W = V \cdot U$, U (and in particular G_k) is a suffix of W. By the definition of the Galois factorization, we have $G_1 \succeq_{\text{alt}} G_2 \succeq_{\text{alt}} \ldots \succeq_{\text{alt}} G_k$ and $H_1 \succeq_{\text{alt}} H_2 \succeq_{\text{alt}} \ldots \succeq_{\text{alt}} H_l$. By showing $G_k \succeq_{\text{alt}} H_1$, we obtain that the factorization $(G_1, G_2, \ldots, G_k, H_1, H_2, \ldots, H_l)$ of $UWWV$ admits the properties of the Galois factorization, which proves the claim.

To that end, we first observe that G_k is a proper suffix of W and W is Galois, thus $G_k \succ_{\text{alt}} W$. Next, if G_k is not a prefix of W, there exists a position $i \leq |G_k| < |W|$ such that $G_k^{\omega}[i] \neq W[i]$. Otherwise if G_k is a prefix of W, G_k is a border of W and $|W| - |G_k|$ is a period of W. Assuming that $|G_k|$ is a period of W, the greatest common divisor gcd of $|G_k|$ and $|W| - |G_k|$ is a period of G_k by the periodicity lemma (Lemma 5), and gcd is a factor of $|W|$. However this is impossible since W is primitive; thus $|G_k|$ cannot be a period of W. Hence, there exists a position $i \leq |W|$ such that $G_k^{\omega}[i] \neq W[i]$. We therefore know that the first k Galois factors in $\mathsf{GF}(U)$ and $\mathsf{GF}(UWWV)$ are the same since we cannot extend G_k further without losing the property to be Galois. Moreover, W is a prefix of H_1 by Lemma 34. Thus, there exists a position $j \leq |W| \leq |H_1|$ such that $G_k^{\omega}[j] \neq H_1[j]$, which implies $G_k \succ_{\text{alt}} H_1$. Therefore, we have $G_1 \succeq_{\text{alt}} G_2 \succeq_{\text{alt}} \ldots \succeq_{\text{alt}} G_k \succeq_{\text{alt}} H_1 \succeq_{\text{alt}} H_2 \succeq_{\text{alt}} \ldots \succeq_{\text{alt}} H_l$, which implies $\mathsf{GF}(UWWV) = (G_1, G_2, \ldots, G_k, H_1, H_2, \ldots, H_l)$. ◀

With Lemma 35, we now have a tool to find the Galois rotation W of T in TTT by determining H_1 and knowing that W is a prefix of H_1. Since we can write $TTT = UWWV$ with $W = VU$ and $|U| < |T|$, the goal is to determine U. For U, we know that all its Galois factors have even or odd periods shorter than $|T|$, so it suffices to find the first Galois factor in TTT for which both periods are at least $|T|$ long (cf. Lemma 34).

In detail, let G be the first factor of $\mathsf{GF}(TTT)$ with $|G| \geq T$. From Lemma 35 we know that the Galois rotation W of a word T is the prefix of G whose length is $|T|$, i.e. $W = G[1..|T|]$. Algorithm 3 describes an algorithm to compute the Galois rotation of an input word T. The algorithm scans TTT sequentially from the beginning, mimicking our Galois factorization algorithm, except that it does not output the factors. At the time where we set $p_o \geq |T|$ and $p_e \geq |T|$, we know that the length of next factor we compute is $|T|$ or longer. At that time, we can determine G. To this end, we output the starting position i of G when reaching the condition that $p_o \geq |T|$ and $p_e \geq |T|$. In a post-processing, we determine $W = (TTT)[i..i + |T| - 1]$, where W is the Galois rotation of T. To keep the additional working space constant, we do not load three copies of T into memory, but use that fact that $(TTT)[k] = T[((k-1) \mod |T|) + 1]$ for $k > |T|$ when processing the input TTT.

▶ **Theorem 36.** *The Galois rotation of a word T can be computed in $O(|T|)$ time and $O(1)$ additional working space.*

8 Experiments

We have implemented our algorithms in C++. The software is freely available by the link on the title page. For a short demonstration, we computed the Galois factors of files from the Canterbury, the Calgary [2] and the Pizza&Chili corpus [8], and depict the results in Table 1. We have omitted those files that contain a zero byte, which is prohibited in our implementation. The experiments run on WSL with Intel Core i7-10700K CPU. To compare the time with a standard Lyndon factorization algorithm, we used the implementation of Duval's algorithm from `https://github.com/cp-algorithms/cp-algorithms`.

Table 1 Counting the number of Galois factors for various datasets. The alphabet size is denoted by σ. Counts are listed in the # columns, together with a time evaluation with Duval's algorithm computing the Lyndon factorization. *Upper part:* The Canterbury and Calgary corpus datasets. *Lower part:* The Pizza&Chili corpus datasets. Note that we used different time units for the upper table (microseconds) and lower table (seconds).

			Galois		Lyndon	
file	σ	size [KB]	#	time [µs]	#	time [µs]
ALICE29.TXT	74	152	14	3070	3	192
ASYOULIK.TXT	68	125	7	2435	2	134
BIB	81	111	25	2372	6	110
BOOK2	96	610	20	12 182	27	555
CP.HTML	86	24	7	544	8	21
FIELDS.C	90	11	18	237	13	12
GRAMMAR.LSP	76	3	10	78	8	6
LCET10.TXT	84	426	12	8599	6	438
NEWS	98	377	24	7440	24	375
PAPER1	95	53	19	1016	9	40
PAPER2	91	82	14	1593	16	66
PAPER3	84	46	11	908	14	39
PAPER4	80	13	8	267	6	12
PAPER5	91	11	9	237	6	10
PAPER6	93	38	12	740	15	32
PLRABN12.TXT	81	481	4	9801	6	559
PROGC	92	39	15	788	12	36
PROGL	87	71	84	1423	77	63
PROGP	89	49	14	944	12	38
XARGS.1	74	4	6	88	9	5

			Galois		Lyndon	
file	σ	size [KB]	#	time [s]	#	time [s]
DBLP.XML	97	296 135	3	5.969	15	0.294
DNA	97	403 927	26	7.799	18	0.360
PROTEINS	27	1 184 051	29	24.384	30	1.091
SOURCES	230	210 866	23	4.307	35	0.179

References

1 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piątkowski. Indexing the bijective BWT. In *Proc. CPM*, volume 128 of *LIPIcs*, pages 17:1–17:14, 2019. `doi:10.4230/LIPIcs.CPM.2019.17`.

2 Timothy C. Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989. `doi:10.1145/76894.76896`.

3 Amanda Burcroff and Eric Winsor. Generalized Lyndon factorizations of infinite words. *Theor. Comput. Sci.*, 809:30–38, 2020. `doi:10.1016/J.TCS.2019.11.003`.

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 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019. doi:10.1016/j.tcs.2018.12.015.

6 Francesco Dolce, Antonio Restivo, and Christophe Reutenauer. Some variations on Lyndon words (invited talk). In *Proc. CPM*, volume 128 of *LIPIcs*, pages 2:1–2:14, 2019. doi:10.4230/LIPIcs.CPM.2019.2.

7 Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. doi:10.1016/0196-6774(83)90017-2.

8 Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1–1.12:31, 2008. doi:10.1145/1412228.1455268.

9 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

10 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018. doi:10.1137/1.9781611975031.96.

11 Ira M. Gessel, Antonio Restivo, and Christophe Reutenauer. A bijection between words and multisets of necklaces. *Eur. J. Comb.*, 33(7):1537–1546, 2012. doi:10.1016/j.ejc.2012.03.016.

12 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. The alternating BWT: An algorithmic perspective. *Theor. Comput. Sci.*, 812:230–243, 2020. doi:10.1016/j.tcs.2019.11.002.

13 Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. A new class of string transformations for compressed text indexing. *Inf. Comput.*, 294:105068, 2023. doi:10.1016/J.IC.2023.105068.

14 Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *ArXiv 1201.3077*, 2012. arXiv:1201.3077.

15 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. SODA*, pages 408–424, 2017. doi:10.1137/1.9781611974782.26.

16 Christophe Reutenauer. Mots de Lyndon généralisés. *Séminaire Lotharingien de Combinatoire*, 54(B54h):1–16, 2005.

17 Yossi Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981. doi:10.1016/0196-6774(81)90013-4.

Simplified Tight Bounds for Monotone Minimal Perfect Hashing

Dmitry Kosolobov
Institute of Natural Sciences and Mathematics, Ural Federal University, Ekaterinburg, Russia

Abstract

Given an increasing sequence of integers $x_1, \ldots, x_n$ from a universe $\{0, \ldots, u-1\}$, the monotone minimal perfect hash function (MMPHF) for this sequence is a data structure that answers the following rank queries: $rank(x) = i$ if $x = x_i$, for $i \in \{1, \ldots, n\}$, and $rank(x)$ is arbitrary otherwise. Assadi, Farach-Colton, and Kuszmaul recently presented at SODA'23 a proof of the lower bound $\Omega(n \min\{\log\log\log u, \log n\})$ for the bits of space required by MMPHF, provided $u \ge n2^{2^{\sqrt{\log\log n}}}$, which is tight since there is a data structure for MMPHF that attains this space bound (and answers the queries in $O(\log u)$ time). In this paper, we close the remaining gap by proving that, for $u \ge (1+\epsilon)n$, where $\epsilon > 0$ is any constant, the tight lower bound is $\Omega(n \min\{\log\log\log \frac{u}{n}, \log n\})$, which is also attainable; we observe that, for all reasonable cases when $n < u < (1+\epsilon)n$, known facts imply tight bounds, which virtually settles the problem. Along the way we substantially simplify the proof of Assadi et al. replacing a part of their heavy combinatorial machinery by trivial observations. However, an important part of the proof still remains complicated. This part of our paper repeats arguments of Assadi et al. and is not novel. Nevertheless, we include it, for completeness, offering a somewhat different perspective on these arguments.

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

Keywords and phrases monotone minimal perfect hashing, lower bound, MMPHF, hash

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.19

Funding This work was carried out with financial support from the Ministry of Science and Higher Education of the Russian Federation (project 075-02-2024-1428 for the development of the regional scientific and educational mathematical center "Ural Mathematical Center").

1 Introduction

The *monotone minimal perfect hash function (MMPHF)* is a data structure built on an increasing sequence $x_1 < \cdots < x_n$ of integers from a universe $\{0, \ldots, u-1\}$ that answers the following *rank* queries: $rank(x) = i$ if $x = x_i$ for some i, and $rank(x)$ is arbitrary otherwise.

The MMPHF is an important basic building block for succinct data structures (e.g., see [2, 6, 5, 14, 13, 9]). It turns out that the relaxation that permits to return arbitrary answers when x does not belong to the stored sequence leads to substantial memory savings: as was shown by Belazzougui, Boldi, Pagh, and Vigna [3], it is possible to construct an MMPHF that occupies $O(n \min\{\log\log\log u, \log n\})$ bits of space with $O(\log u)$-time queries, which is a remarkable improvement over the $\Omega(n \log \frac{u}{n})$ bits required to store the sequence $x_1, \ldots, x_n$ itself.

Until very recently, the best known lower bound for the space of the MMPHF was $\Omega(n)$ bits, which followed from the same bound for the minimal perfect hashing (see [11, 16, 17]). In 2023, this bound was improved by Assadi, Farach-Colton, and Kuszmaul [1]: they proved that, surprisingly, the strange space upper bound $O(n \min\{\log\log\log u, \log n\})$ is actually tight, provided $u \ge n2^{2^{\sqrt{\log\log n}}}$. Thus, the problem was fully settle for almost all possible u. Their proof utilized a whole spectre of sophisticated combinatorial techniques: a "conflict graph" of possible data structures, the fractional chromatic number for this graph, the duality of linear programming, non-standard graph products, and intricate probabilistic arguments.

In this paper we simplify their proof, removing all mentioned concepts except the intricate probabilistic arguments, and we slightly extend the result: our lower bound is $\Omega(n \min\{\log\log\log \frac{u}{n}, \log n\})$, where $u \geq (1+\epsilon)n$ for an arbitrary constant $\epsilon > 0$ (the Ω hides an $\epsilon \log \frac{1}{\epsilon}$ factor). Further, we show that this bound is tight by devising a simple extension of the MMPHF data structure of Belazzougui et al. [3]. We also observe that, for all reasonable cases $n < u < (1+\epsilon)n$, tight bounds can be obtained using known facts.

All ingredients required for our simplification were actually already present in the paper by Assadi et al. For instance, we eventually resort to essentially the same problem of coloring random sequences (see below) and the same probabilistic reasoning. Our extension of the MMPHF by Belazzougui et al. [3] is also not difficult. It seems that the simpler proof and the extension were overlooked.

The paper is organized as follows. In Section 2, we define our notation and discuss weaker lower bounds and tight upper bounds, including our extension of the MMPHF from [3]. Section 3 provides a short way to connect the lower bound to certain colorings of the universe. Section 4 shows how the problem can be further reduced to the coloring of certain random sequences on a very large universe $u = 2^{2^{n^3}}$; Assadi et al. make a similar reduction but their explanation includes unnecessary non-standard graph products and does not cover the extended range $(1+\epsilon)n \leq u$. The material in all these first sections is quite easy. Finally, Section 5 is a core of the proof, which, unfortunately, still involves complicated arguments. This part is exactly equivalent to Lemma 4.2 in [1], which was also the most challenging part in [1]. For the self-containment of the paper, instead of directly citing Lemma 4.2 from [1], we decided to offer a somewhat different view on the same arguments. Depending on their disposition, the reader might find our exposition more preferrable or vice versa; it shares the central idea with [1] but reaches the goal through a slightly different path.

2 Tight Upper Bounds

Denote $[p..q] = \{k \in \mathbb{Z}\colon p \leq k \leq q\}$, $[p..q) = [p..q{-}1]$, $(p..q] = [p{+}1..q]$, $(p..q) = [p{+}1..q{-}1]$. Throughout the text, u denotes the size of the universe $[0..u)$, from which the hashed sequence $x_1, \ldots, x_n$ is sampled, and n denotes the size of this sequence. All logarithms have base 2. To simplify the notation, we assume that $\log\log\log \frac{u}{n} = \Omega(1)$, for any $\frac{u}{n} > 0$.

Let us overview known upper bounds for the space required by the MMPHF. The MMPHF of Belazzougui et al. [3] offers $O(n \log\log\log u)$ bits of space. When $n2^{2^{\sqrt{\log\log n}}} \leq u \leq 2^{2^{\mathrm{poly}(n)}}$, it is tight due to the lower bound of [1]. For larger u, we can construct a perfect hash $h : [0..u) \to [1..n]$ that occupies $O(n \log n)$ bits and bijectively maps the hashed sequence $x_1, \ldots, x_n$ onto $[1..n]$ (e.g., it might be the classical two-level scheme [12] or a more advanced hash [4]) and we store an array $A[1..n]$ such that, for $i \in [1..n]$, $A[h(x_i)]$ is the rank of x_i. This scheme takes $O(n \log n)$ bits, which is tight when $u \geq 2^{2^{\mathrm{poly}(n)}}$, again due to the lower bound of [1] since $n \log\log\log 2^{2^{\mathrm{poly}(n)}} = \Theta(n \log n)$. For small u (like $u = \Theta(n)$), we can simply store a bit array $B[0..u{-}1]$ such that $B[x] = 1$ iff $x = x_i$. Such "data structure" takes u bits and can answer the rank queries for the MMPHF (very slowly). With additional $o(n)$ bits [8, 15], one can answer in $O(1)$ time the rank queries on this array and, also, the following *select queries*: given $i \in [1..n]$, return the position of the ith 1 in the array.

The described MMPHFs give the tight space upper bound $O(n \min\{\log\log\log u, \log n\})$, for $u \geq n2^{2^{\sqrt{\log\log n}}}$, and an upper bound $O(n)$, for $u = O(n)$. Let us construct an MMPHF that occupies $O(n \log\log\log \frac{u}{n})$ bits, for $2n \leq u < n2^{2^{\sqrt{\log\log n}}}$, which, as we show below, is tight. (We note that a construction similar to ours was alluded in [6, 7].) We split the range $[0..u)$ into n buckets of length $b = \frac{u}{n}$. For $i \in [1..n]$, denote by n_i the number of elements of the sequence $x_1, \dots, x_n$ that are contained in the ith bucket. We construct the MMPHF of Belazzougui et al. for each bucket, thus consuming $O(\sum_{i=1}^n n_i \log\log\log\frac{u}{n}) = O(n\log\log\log\frac{u}{n})$ bits of space. Then, we build a bit array $B[1..2n]$ such that $B[\sum_{j=1}^{i-1} n_j + i] = 1$, for each $i \in [1..n]$, and all other bits are zeros; it is convenient to view B as the concatenation of bit strings 10^{n_i}, for $i \in [1..n]$, where 0^{n_i} denotes a bit string with n_i zeros. The bit array B supports select queries and takes $O(n)$ bits. To answer the rank query for $x \in [0..u)$, our MMPHF first calculates $i = \lfloor x/b \rfloor + 1$ (the index of the bucket containing x), then it computes the position k of the ith 1 in the bit array B using the select query, and the answer is equal to $k - i$ plus the answer of the query $rank(x - (i-1)b)$ in the MMPHF associated with the ith bucket. We, however, cannot afford to store n pointers to the MMPHFs associated with the buckets. Instead, we concatenate the bit representations of these MMPHFs and construct another bit array N of length $O(n\log\log\log\frac{u}{n})$ where the beginning of each MMPHF in the concatenation is marked by 1 (and all other bits are zeros). The array N also supports the select queries and the navigation to the MMPHF associated with the ith bucket is performed by finding the ith 1 in the array N using the select query.

With the described data structures, we obtain tight upper bounds for all $u \ge (1+\epsilon)n$, where $\epsilon > 0$ is an arbitrary constant. In particular, when $(1+\epsilon)n \le u < 2n$, the upper bound $O(u)$ is equal to $O(n)$ and it is known to be tight: an analysis of the minimal perfect hashing [16, 17] implies the lower bound $\Omega(n)$ for the MMPHF, provided $u \ge (1+\epsilon)n$. More precisely, the bound is $(u-n)\log\frac{u}{u-n} - O(\log n)$, which holds for arbitrary $u > n$ (see [4, 16, 17]). For illustrative purposes, we rederive this bound in a proof sketch provided in Section 4.

It remains to analyse the range $n < u < (1+\epsilon)n$. To this end, we restrict our attention only to matching upper and lower bounds that are greater than $\Omega(\log n)$. This is a reasonable restriction because all these data structures are usually implemented on the word RAM model, where it is always assumed that $\Theta(\log n)$ bits are available for usage. Thus, we analyse only the first term of the difference $(u-n)\log\frac{u}{u-n} - O(\log n)$ assuming that it is at least twice greater than the $O(\log n)$ term. Suppose that $u = (1+\alpha)n$, where $0 < \alpha < \frac{1}{4}$ and α is not necessarily constant. Then, we obtain $(u-n)\log\frac{u}{u-n} = n\alpha\log\frac{1+\alpha}{\alpha} = \Theta(n\alpha\log\frac{1}{\alpha})$. It is known that the bit array $B[0..u{-}1]$ such that $B[x] = 1$, for $x = x_i$, can be encoded into $\log\binom{u}{n}$ bits, which can be treated as an MMPHF for the sequence $x_1, \dots, x_n$. By the well-known entropy inequality [10], we obtain $\log\binom{u}{n} \le n\log\frac{u}{n} + (u-n)\log\frac{u}{u-n} = n\log(1+\alpha) + n\alpha\log\frac{1+\alpha}{\alpha} \le O(n\alpha + n\alpha\log\frac{1}{\alpha}) = O(n\alpha\log\frac{1}{\alpha})$, which coincides with the lower bound $(u-n)\log\frac{u}{u-n} - O(\log n) = \Omega(n\alpha\log\frac{1}{\alpha})$ (rederived in Section 4) and, thus, is tight. In particular, for constant $\alpha = \epsilon$, we obtain the bound $\Omega(n)$ that hides $\epsilon\log\frac{1}{\epsilon}$ under the Ω.

Hereafter, we assume that all presented results hold for sufficiently large u. We will mostly consider the case $u \le 2^{2^{\mathrm{poly}(n)}}$, which also implies sufficiently large n.

3 From Data Structures to Colorings

Let us first consider deterministic MMPHFs. Randomized MMPHFs are briefly discussed in Remark 2 in the end of Section 4.

Given positive integers u and $n < u$, the MMPHF that uses S bits of space is a data structure that can encode any increasing sequence $x_1 < \cdots < x_n$ from $[0..u)$ into S bits to support the rank queries: for $x \in [0..u)$, $rank(x) = i$ if $x = x_i$ for some $i \in [1..n]$, and $rank(x)$ is arbitrary otherwise. We assume a very powerful model of computation: the query algorithm has unbounded computational capabilities and has unrestricted access to its S bits of memory. Formally, it can be modelled as a function $rank\colon [0..u) \times \{0,1\}^S \to [1..n]$ that takes as its arguments an integer $x \in [0..u)$ and the content of the S-bit memory and outputs the rank of x in a sequence encoded in these S bits (note that the same bits might correctly encode many different sequences); it is guaranteed that any increasing sequence $x_1, \dots, x_n$ has at least one encoding $c \in \{0,1\}^S$ that provides correct queries for it, i.e., $rank(x_i, c) = i$, for $i \in [1..n]$. Our goal is to prove that such function can exist only if $S \geq \Omega(n \log\log\log \frac{u}{n})$, provided $(1+\epsilon)n \leq u \leq 2^{2^{\mathrm{poly}(n)}}$, for constant $\epsilon > 0$.

The function $rank\colon [0..u) \times \{0,1\}^S \to [1..n]$ can be viewed as a family of 2^S colorings of the range $[0..u)$: each "memory content" $c \in \{0,1\}^S$ colors any $x \in [0..u)$ into the color $rank(x,c)$, one of n colors $[1..n]$. We say that such a coloring *encodes* a sequence $x_1 < \dots < x_n$ if the color of x_i is i, for $i \in [1..n]$. Note that one coloring may encode many distinct sequences and one sequence may be encoded by different colorings of $[0..u)$.

▶ **Example 1.** For $u = 17$, the following coloring of $[0..u)$ (the colors are both highlighted and denoted by indices 1–5 below) encodes the sequences $3, 6, 7, 10, 14$, and $1, 2, 4, 9, 12$, and $1, 6, 11, 15, 16$, to name a few:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	1	3	3	2	3	1	4	4	3	5	2	5	4	5

We thus have deduced that the MMPHF provides a family of 2^S colorings that encode all possible sequences of size n from $[0..u)$. Now, if we prove that any such all-encoding family must have at least C colorings, then we will have $2^S \geq C$, which implies the space lower bound $S \geq \log C$. In what follows, we show that $C \geq (\log\log \frac{u}{n})^{\Omega(n)}$ when $(1+\epsilon)n \leq u \leq 2^{2^{\mathrm{poly}(n)}}$, hence proving the lower bound $S \geq \Omega(n \log\log\log \frac{u}{n})$.

4 Coloring of Random Sequences

As in [1], we utilize the following probabilistic argument. Consider a random process that generates size-n sequences from $[0..u)$ in such a way that any fixed coloring of $[0..u)$ encodes the generated sequence with probability at most $1/C$. Now if a family of colorings of $[0..u)$ is such that any size-n sequence from $[0..u)$ can be encoded by some of its colorings (i.e., it is an "all-encoding" family as the one provided by the MMPHF), then it necessarily contains at least C colorings since any sequence generated by our random process is encoded with probability 1 by one of the colorings. Let us illustrate this reasoning by sketching a proof of a weaker lower bound for our problem (which can also serve as a proof of the space lower bound for the minimal perfect hash function on size-n sequences from $[0..u)$).

Consider a process that generates all size-n sequences from $[0..u)$ uniformly at random. Fix an arbitrary coloring of $[0..u)$ with colors $[1..n]$. Denote by c_i the number of elements $x \in [0..u)$ with color i. Since the coloring might encode at most $c_1 \cdots c_n$ distinct size-n sequences from $[0..u)$, the probability that a random sequence is encoded by it is at most $c_1 \cdots c_n / \binom{u}{n}$. Since $\sum_{i=1}^{n} c_i = u$, the maximum of $c_1 \cdots c_n$ is attained when all c_i are equal, so $c_1 \cdots c_n \leq (\frac{u}{n})^n$. Thus, we obtain $c_1 \cdots c_n / \binom{u}{n} \leq (\frac{u}{n})^n / \binom{u}{n}$ and, hence, any "all-encoding" family must contain at least $\binom{u}{n} / (\frac{u}{n})^n$ colorings, which, after applying the logarithm, implies the space lower bound $\log(\binom{u}{n} / (\frac{u}{n})^n)$ for the MMPHF. Finally, the entropy inequality [10] $\log \binom{u}{n} \geq n \log \frac{u}{n} + (u-n) \log \frac{u}{u-n} - O(\log n)$ gives the lower bound $(u-n) \log \frac{u}{u-n} - O(\log n)$, which is bounded by $\Omega(n\epsilon \log \frac{1}{\epsilon}) = \Omega(n)$ when $(1+\epsilon)n \leq u$ for constant $\epsilon > 0$.

It is evident from this sketch that our random process must be more elaborate than a simple random pick.

In what follows we essentially repeat the scheme from [1]. Namely, for the special case $u = 2^{2^{n^3}}$, we devise a random process generating size-n sequences from $[0..u)$ such that any fixed coloring encodes its generated sequence with probability at most $1/n^{\Omega(n)}$. Hence, the number of colorings in the family provided by the MMPHF is at least $n^{\Omega(n)}$, which, after applying the logarithm, implies the space lower bound $\Omega(n \log n) = \Omega(n \log\log\log \frac{u}{n})$. All other possible u are reduced to this special case. Let us start with this reduction.

Reduction of arbitrary u to very large u. Suppose that, for any n and $u = 2^{2^{n^3}}$, we are able to devise a random process that generates size-n sequences from $[0..u)$ in such a way that any coloring of $[0..u)$ encodes the generated sequence with probability at most $1/n^{\Omega(n)}$. Now let us fix arbitrary u and n such that $(1+\epsilon)n \le u \le 2^{2^{\mathrm{poly}(n)}}$, for constant $\epsilon > 0$.

Case (i) $u \ge 2^{2^{n^3}}$. For this case, the same random process that generates size-n sequences from $[0..2^{2^{n^3}}) \subseteq [0..u)$ gives the probability at most $1/n^{\Omega(n)}$, again implying the space lower bound $\Omega(n \log n)$ as above, which is equal to $\Omega(n \log\log\log \frac{u}{n})$ when $2^{2^{n^3}} \le u \le 2^{2^{\mathrm{poly}(n)}}$.

Case (ii) $(1+\epsilon)n \le u < 2^{2^8} n$. Since $n \log\log\log \frac{u}{n} = \Theta(n)$, the lower bound $\Omega(n)$ for this case was obtained above.

Case (iii) $2^{2^8} n \le u < 2^{2^{n^3}}$. The key observation is that while our hypothesised random process cannot be applied to generate sequences of size n (since u is too small), it can generate smaller sequences, for instance, of size $\bar{n} \le (\log\log u)^{1/3}$ (since $2^{2^{\bar{n}^3}} \le u$). Accordingly, we compose a random process that generates a size-n sequence as follows: it splits the range $[0..u)$ into $n/\bar{n}$ equal blocks of length $\bar{u} = u/(n/\bar{n})$ and independently generates a size-$\bar{n}$ sequence inside the first block, a size-$\bar{n}$ sequence inside the second block, etc. The generation inside each block is performed using our hypothesised random process, which is possible provided $\bar{u} \ge 2^{2^{\bar{n}^3}}$. This inqualily is satisfied by putting $\bar{n} = \lfloor (\log\log \frac{u}{n})^{1/3} \rfloor$ (note that $\bar{n} \ge 2$ since $\frac{u}{n} \ge 2^{2^8}$): $\bar{u} \ge u/n = 2^{2^{\log\log \frac{u}{n}}} \ge 2^{2^{\bar{n}^3}}$. Fix an arbitrary coloring of $[0..u)$. The probability that the generated size-n sequence is encoded by this coloring is equal to the product of $n/\bar{n}$ probabilities that its independently generated size-$\bar{n}$ subsequences are encoded by the coloring restricted to the corresponding blocks, which gives $(1/\bar{n}^{\Omega(\bar{n})})^{n/\bar{n}} = 1/\bar{n}^{\Omega(n)}$ (note that, technically, our assumption that gives each probability $1/\bar{n}^{\Omega(\bar{n})}$ requires the colors in the ith block to be from $(i\bar{n}..(i+1)\bar{n}]$ but, clearly, any other colors in the ith block make the probability that the corresponding size-$\bar{n}$ subsequence is encoded by this coloring even lower.) The latter, after applying the logarithm, yields the space lower bound $\Omega(n \log \bar{n})$, which is $\Omega(n \log\log\log \frac{u}{n})$ when $\bar{n} = \lfloor (\log\log \frac{u}{n})^{1/3} \rfloor$.

▶ Remark 2. Using an argument akin to Yao's principle, Assadi et al. showed that the lower bound for randomized MMPHFs is the same as for deterministic. We repeat their argument for completeness, albeit without their unnecessary graph products etc.

A randomized MMPHF has unrestrictedly access to a tape of random bits, which does not take any space. Denote by X the set of all size-n sequences in $[0..u)$. The MMPHF receives a sequence $x \in X$ and a random tape r and encodes x into a memory content $d^r(x) \in \{0,1\}^*$. Thus, unlike the deterministic case, the space size depends on $x \in X$ and on the randomness r. Naturally, the space occupied by such MMPHF is defined as $d = \max_{x \in X} \mathrm{E}_r[|d^r(x)|]$, where the expectation is for the random r. Note that, when the tape r is fixed, the algorithm becomes deterministic: it can be modelled as a function $\mathit{rank}^r \colon [0..u) \times \{0,1\}^* \to [1..n]$ that, for any memory content $c \in \{0,1\}^*$, defines a coloring of $[0..u)$ into colors $[1..n]$.

Denote by P a random distribution on X such that any fixed coloring of $[0..u)$ encodes $x \in P$ with probability at most $1/C$. Obviously, $d = \max_{x\in X} \mathrm{E}_r[|d^r(x)|] \ge \mathrm{E}_{x\in P}\mathrm{E}_r[|d^r(x)|] = \mathrm{E}_r\mathrm{E}_{x\in P}[|d^r(x)|]$. By the averaging argument, there is a tape r^* such that $\mathrm{E}_r\mathrm{E}_{x\in P}[|d^r(x)|] \ge \mathrm{E}_{x\in P}[|d^{r^*}(x)|]$. Therefore, $\mathrm{E}_{x\in P}[|d^{r^*}(x)|] \le d$. By Markov's inequality, $\mathrm{Pr}_{x\in P}(|d^{r^*}(x)| \le$

$2d) \geq \frac{1}{2}$. Denote by M all possible memory contents $d^{r^*}(x)$ for all $x \in P$ such that $|d^{r^*}(x)| \leq 2d$. Evidently, we have the lower bound $\frac{1}{2}\log|M| \leq d$ and $\Pr_{x\in P}(d^{r^*}(x) \in M) \geq \frac{1}{2}$. Each $c \in M$ determines a coloring of $[0..u)$. Since the probability that a random $x \in P$ is encoded by this coloring is $\frac{1}{C}$, we have $\Pr_{x\in P}(d^{r^*}(x) \in M) \leq \frac{|M|}{C}$. Thus, we obtain $|M| \geq \frac{C}{2}$, which implies the space bound $d \geq \Omega(\log C)$, the same as in the deterministic case.

5 Random Sequences on Large Universes

In this section, we always assume that $u = 2^{2^{n^3}}$ and $n \geq 2$. Our random process generating size-n sequences from $[0..u)$ is essentially a variation of the process by Assadi et al. [1]. Unlike the previous sections, it is not precisely a simplification of the arguments from [1], rather a different perspective on them. We first overview main ideas of Assadi et al. and, then, define our process by modifying them.

5.1 Definition of the random process

Let us fix an arbitrary coloring of the range $[0..u)$ into colors $[1..n]$. On a very high level, the random process devised by Assadi et al. is as follows: choose $n-1$ lengths $b_2 > \cdots > b_n$, then pick uniformly at random x_1 from $[0..u)$, then pick uniformly at random x_2 from $(x_1..x_1{+}b_2)$, then x_3 from $(x_2..x_2{+}b_3)$, etc. Intuitively, if it is highly likely that at least $\frac{n}{2}$ of the picks $x_i \in (x_{i-1}..x_{i-1}{+}b_i)$ were such that the fraction of elements with color i in the range $(x_{i-1}..x_{i-1}{+}b_i)$ is at most $O(\frac{1}{n})$, then the probability that the randomly generated sequence $x_1, \ldots, x_n$ is encoded by our fixed coloring is at most $O(\frac{1}{n})^{n/2} = \frac{1}{n^{\Omega(n)}}$. Unfortunately, for any $b_2, \ldots, b_n$, there are colorings where this is not true. However, as it was shown in [1], when picking $b_2, \ldots, b_n$ randomly from a certain distribution such that $b_2 \gg \cdots \gg b_n$, one might guarantee that, whenever we encounter a "dense" range $(x_{i-1}..x_{i-1}{+}b_i)$ where at least an $\Omega(\frac{1}{n})$ fraction of colors are i, the final range $(x_{n-1}..x_{n-1}{+}b_n)$ with very high probability will contain at least a $\frac{2}{n}$ fraction of colors i. Therefore, it is highly likely that such "dense" ranges appear less than $\frac{n}{2}$ times since otherwise the range $(x_{n-1}..x_{n-1}{+}b_n)$ has no room for the color n of element x_n as it already contains $\frac{n}{2}$ colors from $[1..n)$, each occupying a $\frac{2}{n}$ fraction of the range. Hence, with very high probability, the random process generating the size-n sequence will encounter such "dense" ranges $(x_{i-1}..x_{i-1}{+}b_i)$ at most $\frac{n}{2}$ times and at least $\frac{n}{2}$ ranges will contain an $O(\frac{1}{n})$ fraction of the picked color, which leads to the probability $\frac{1}{n^{\Omega(n)}}$ that the generated sequence will be encoded by our fixed coloring.

We alter the outlined scheme introducing a certain "rigid" structure into our random process. The range $[0..u)$ is decomposed into a hierarchy of blocks with $L = n^{n^2-n}$ levels (the choice of L is explained below): $[0..u)$ is split into n^n equal blocks, which are called the blocks of level 1, each of these blocks is again split into n^n equal blocks, which are the blocks of level 2, and so on: for $\ell < L$, each block of level ℓ is split into n^n equal blocks of level $\ell+1$. Thus, we have $(n^n)^L = n^{n^{n^2-n+1}}$ blocks on the last level L. Observe that $(n^n)^L \leq u$ since $\log\log((n^n)^L) = \Theta(n^2 \log n) \ll n^3 = \log\log u$. For simplicity, we assume that $(n^n)^L$ divides u (otherwise we could round u to the closest multiple of $(n^n)^L$, ignoring some rightmost elements of $[0..u)$). The length of the last level blocks is set to $u/(n^n)^L$ (their length will not play any role, it is set to this number just to make everything fit into u).

Our random process consists of two parts: first, we pick a sequence of levels $\ell_2 < \cdots < \ell_n$; then, we pick the elements $x_1, \ldots, x_n$. The chosen levels $\ell_2, \ldots, \ell_n$ will determine the sizes of n nested blocks from which the elements $x_1, \ldots, x_n$ are sampled. Formally, it is as follows (see Fig. 1):

1. The levels $\ell_2, \dots, \ell_n$ are chosen by consecutively constructing a sequence of nested intervals $[\ell_2..\ell'_2) \supset \dots \supset [\ell_n..\ell'_n)$: whenever $[\ell_i..\ell'_i)$ is already chosen, for $i \in [1..n)$ (assuming $[\ell_1..\ell'_1) = [0..L)$), we split $[\ell_i..\ell'_i)$ into n^n equal disjoint intervals and pick as $[\ell_{i+1}..\ell'_{i+1})$ one of them uniformly at random, except the first one containing ℓ_i.
2. The elements $x_1, \dots, x_n$ are chosen by consecutively constructing a sequence of nested blocks $[b_2..b'_2) \supset \dots \supset [b_n..b'_n)$ from levels $\ell_2, \dots, \ell_n$, respectively: whenever $[b_i..b'_i)$ is already chosen, for $i \in [1..n)$ (assuming $[b_1..b'_1) = [0..u)$), we pick x_i uniformly at random from the range $[b_i..b'_i{-}b)$, where b is the block length for level ℓ_{i+1} (recall that $b = u/(n^n)^{\ell_{i+1}}$), and choose as $[b_{i+1}..b'_{i+1})$ the block $[kb..(k{+}1)b)$ closest to the right of x_i, i.e., $(k{-}1)b \le x_i < kb$; the element x_n is chosen uniformly at random from $[b_n..b'_n)$.

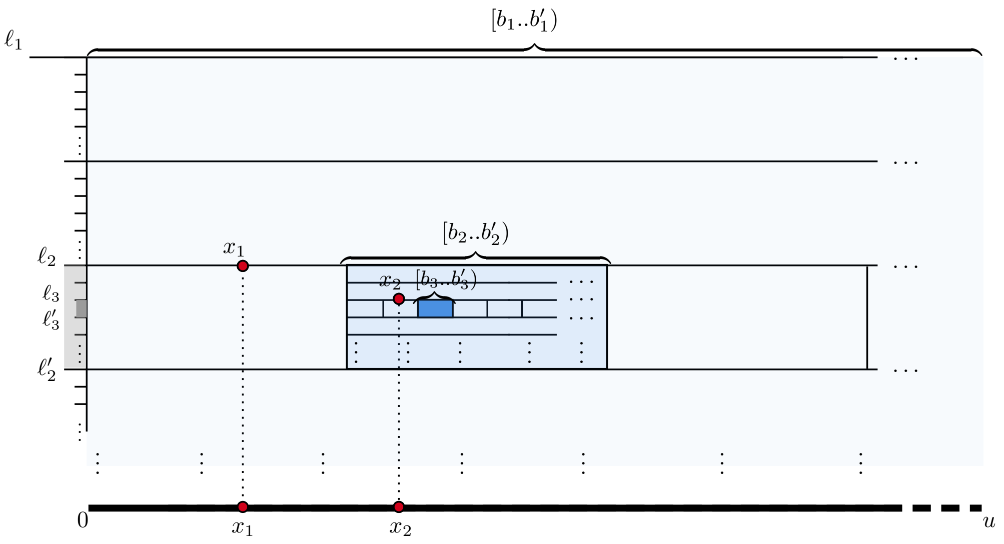


Figure 1 A schematic image of the first intervals $[\ell_1..\ell'_1)$, $[\ell_2..\ell'_2)$, $[\ell_3..\ell'_3)$, the first blocks $[b_1..b'_1)$, $[b_2..b'_2)$, $[b_3..b'_3)$, and the first elements x_1, x_2 generated by our process. The set $[0..u)$ is depicted as the line at the bottom. The left vertical "ruler" depicts some levels (not all): the larger divisions denote the levels that could be chosen as ℓ_2 and the smaller divisions could be chosen as ℓ_3. The intervals $[\ell_2..\ell'_2)$ and $[\ell_3..\ell'_3)$ are painted in two shades of gray. For $i \in [1..3]$, each block $[b_i..b'_i)$ is associated with a rectangle that includes all subblocks of $[b_i..b'_i)$ from levels $[\ell_i..\ell'_i)$; the rectangles are painted in shades of blue; we depict inside the rectangle of $[b_i..b'_i)$ lines corresponding to levels that could be chosen as ℓ_{i+1} and we outline contours of blocks from the level ℓ_{i+1}. The elements x_1, x_2 are chosen from $[0..u)$ but it is convenient to draw them also on the lines corresponding to the respective levels ℓ_2 and ℓ_3, so it is easier to see that the frist level-ℓ_2 block to the right of x_1 is $[b_2..b'_2)$ and the first level-ℓ_3 block to the right of x_2 is $[b_3..b'_3)$.

We say that the process *reaches* a block B (respectively, a level ℓ) on the ith stage of recursion if it assigns $[b_i..b'_i) = B$ (respectively, $\ell_i = \ell$) during its work. Note that the length of $[\ell_{i+1}..\ell'_{i+1})$ is $\frac{L}{(n^n)^i}$. Hence, the number $L = n^{n^2-n} = (n^n)^{n-1}$ is large enough to allow the described $n-1$ recursive splits of $[0..L)$. We have $\ell_1 < \ell_2 < \dots < \ell_n$ (assuming $\ell_1 = 0$) since, while choosing $[\ell_{i+1}..\ell'_{i+1})$ from $[\ell_i..\ell'_i)$, the process excludes from consideration the first interval $[\ell_i..\ell_i{+}\frac{L}{(n^n)^i})$. Note that, as a byproduct, many levels from $[0..L)$ are not reacheable.

The process can be viewed as a variation on the design of Assadi et al. restrained to the introduced block structure. Alternatively, it can be viewed as a recursion: for $i \in [1..n)$, it takes as its input an interval $[\ell_i..\ell'_i)$ and a block $[b_i..b'_i)$ from level ℓ_i (assuming $[\ell_1..\ell'_1) = [0..L)$ and $[b_1..b'_1) = [0..u)$), chooses randomly ℓ_{i+1} from a set of $n^n - 1$ evenly spaced levels in

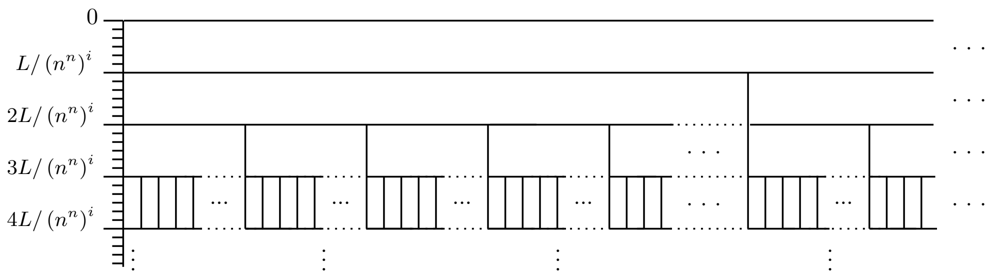


Figure 2 A schematic partition of all blocks into disjoint subsets for a fixed $i \in [1..n)$.

$(\ell_i..\ell'_i)$, then picks x_i, and invokes the recursion to generate the elements $x_{i+1}, \dots, x_n$ inside the closest level-ℓ_{i+1} block to the right of x_i, setting $[\ell_{i+1}..\ell'_{i+1})$ as the next interval of levels. In this view, the process is not split into two parts and it constructs the levels and blocks simultaneously, which is possible since the levels are chosen independently of the blocks.

The nestedness of the intervals $[\ell_i..\ell'_i)$ and the blocks $[b_i..b'_i)$ implies that, whenever the process reaches a block $[b_n..b'_n)$ on level ℓ_n, we can uniquely determine the sequence of intervals $[\ell_2..\ell'_2), \dots, [\ell_n..\ell'_n)$ and blocks $[b_2..b'_2), \dots, [b_n..b'_n)$ that were traversed. It is instructive to keep in mind the following view (Fig. 2). Fix $i \in [1..n)$. Split $[0..L)$ into $(n^n)^i$ equal disjoint intervals: $I = \{[k\frac{L}{(n^n)^i}..(k+1)\frac{L}{(n^n)^i})\}_{k\in[0..(n^n)^i)}$. The set of all blocks can be partitioned into disjoint subsets as follows: each subset is determined by an interval $[\ell..\ell')$ from I and a level-ℓ block $[b..b')$ and consists of all subblocks of $[b..b')$ from levels $[\ell..\ell')$ (including $[b..b')$ itself). Then, the $(i+1)$th recursive invocation of our process (which chooses x_{i+1}) necessarily takes as its input an interval $[\ell..\ell')$ from I and a level-ℓ block $[b..b')$, and subsequently it can reach only subblocks from the corresponding set in the partition. Note, however, that not all subblocks are reachable since, first, some levels are unreachable, as was noted above, and, second, the process ignores the leftmost subblock of its current block when it chooses the next block (since this subblock is not located to the right of any element x in the block). For the same reason, not all intervals $[\ell..\ell') \in I$ and level-ℓ blocks $[b..b')$ might appear as inputs of the recursion.

5.2 Analysis of the random process

Fix an arbitrary coloring of $[0..u)$ into colors $[1..n]$, which will be used untill the end of this section. We call a subset of $[0..u)$ *dense* for color i if at least a $\frac{2}{n}$ fraction of its elements have color i; we call it *sparse* otherwise. The color is not specified if it is clear from the context.

Analysis plan. For each $i \in [1..n)$ and each block $[b_i..b'_i)$ that might appear on the ith stage of our recursion, we show that whenever the process reaches $[b_i..b'_i)$, the level ℓ_{i+1} it randomly chooses (among $n^n - 1$ choices) admits, with high probability $1 - \frac{1}{n^{\Omega(n)}}$, a partition of all level-ℓ_{i+1} blocks inside $[b_i..b'_i)$ into two disjoint families (see Fig. 3): a "sparse" set $\bar{S}$, whose union is sparse for color i, and an "inherently dense" set $\bar{D}$ of blocks, each of which is dense for color i and almost all subblocks of $\bar{D}$ on each subsequent level $\ell \in [\ell_{i+1}..\ell'_{i+1})$ are dense for i too, where "almost all" means that only a $\frac{1}{n^{\Omega(n)}}$ fraction of level-ℓ subblocks of $\bar{D}$ might be sparse. Thus, whenever the process chooses x_i from the "inherently dense" set on an ith stage, it will with high probability $1 - \frac{1}{n^{\Omega(n)}}$ end up inside a block $[b_n..b'_n)$ dense for color i, where it picks x_n in the end. Therefore, to have a room for one element x_n with color n, such hits into "inherently dense" sets could happen on less than $\frac{n}{2}$ different stages i in the recursion, with high probability. We deduce from this, like in the scheme from [1] outlined above, that at least $\frac{n}{2}$ stages i of the recursion pick x_i from the sparse sets $\bar{S}$, with high probability, which allows us to estimate by $O(\frac{1}{n})^{\frac{n}{2}} = \frac{1}{n^{\Omega(n)}}$ the probability that the generated sequence is correctly colored in our fixed coloring. Now let us formalize this.

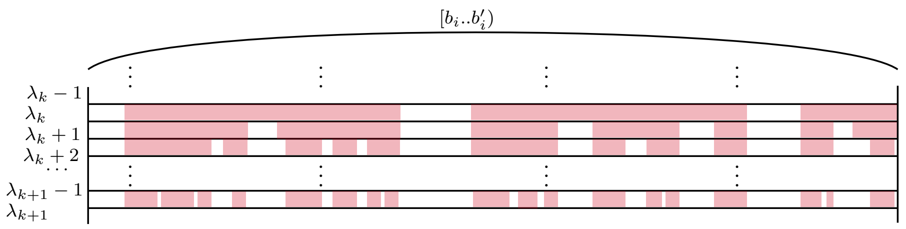


Figure 3 The lines depict consecutive levels $[\lambda_k - 1..\lambda_{k+1}]$ inside a block $[b_i..b'_i)$; we assume that $[\ell_{i+1}..\ell'_{i+1}) = [\lambda_k..\lambda_{k+1})$. The red regions denote the dense sets D_ℓ, for $\ell \in [\lambda_k - 1..\lambda_{k+1})$. The image is supposed to show the case when each such D_ℓ takes a large portion of $D_{\lambda_k - 1}$, so that $D_{\lambda_k - 1}$ might (approximately) serve as our "inherently dense" set $\bar{D}_k$ for level λ_k in the block.

Constructing $\bar{S}$ and $\bar{D}$. Fix $i \in [1..n)$. Let $H = [\ell_i..\ell'_i)$ and $B = [b_i..b'_i)$ be, respectively, an interval of levels and a level-ℓ_i block that could be reached by our process on the ith stage of recursion (assuming that $[\ell_1..\ell'_1) = [0..L)$ and $[b_1..b'_1) = [0..u)$). For $\ell \in H$, denote by S_ℓ the union of all sparse blocks from levels $[\ell_i..\ell]$ that are subsets of B. Due to the nestedness of blocks, S_ℓ is equal to the union of $S_{\ell-1}$ (assuming $S_{\ell-1} = \emptyset$ for $\ell = \ell_i$) and all sparse level-ℓ blocks that are subsets of B disjoint with $S_{\ell-1}$. Obviously, the set S_ℓ is sparse. Denote $D_\ell = B \setminus S_\ell$, the complement of S_ℓ, which is equal to the union of all dense level-ℓ blocks that are subsets of B disjoint with S_ℓ (note that some dense level-ℓ blocks could be subsets of S_ℓ if they were covered by larger sparse blocks). Consult Figure 3 in what follows.

Denote $\lambda_0 = \ell_i$ and $\lambda_k = \lambda_{k-1} + \frac{|H|}{n^n}$, for $k \in [1..n^n]$. The intervals $[\lambda_k..\lambda_{k+1})$, for all $k \in [1..n^n)$, are all possible choices for the random interval $[\ell_{i+1}..\ell'_{i+1})$. To choose the interval is to choose $k \in [1..n^n)$. We slightly relax the scheme outlined in the plan: for each of the choices $k \in [1..n^n)$, we define a set $\bar{S}_k$ that will contain a $\frac{2}{n} + \frac{1}{2^{n/8}}$ fraction of colors i, so it might be not precisely sparse as in the plan. We call sets with this fraction of a given color *almost sparse*. If B itself is almost sparse, we define $\bar{S}_k = B$ and $\bar{D}_k = \emptyset$, for all $k \in [1..n^n)$. Otherwise, i.e., when B is not almost sparse, we are to prove that, for a randomly chosen level ℓ_{i+1}, with high probability $1 - \frac{1}{n^{\Omega(n)}}$ not only the blocks composing $D_{\ell_{i+1}}$ are dense but also most of their subblocks on levels $\ell \in [\ell_{i+1}..\ell'_{i+1})$ are dense for color i. The sets $\bar{D}_k$ will be constructed using the sets $D_{\ell_{i+1}}$ with this property. So, assume that B is not almost sparse.

Let $D_{\ell_i - 1} = B$. Since the sets D_ℓ are nested (i.e., $D_{\ell-1} \supseteq D_\ell$), the "fraction of space" which any $D_{\ell+\delta}$ occupies inside any D_ℓ is $\frac{|D_{\ell+\delta}|}{|D_\ell|}$ (a number between 0 and 1). Therefore, for $k \in [0..n^n)$, the fraction of space which each of the sets $D_{\lambda_k}, D_{\lambda_k+1}, \ldots, D_{\lambda_{k+1}-1}$ occupies inside the set $D_{\lambda_k - 1}$ is at least $q_{\lambda_k} = \frac{|D_{\lambda_{k+1}-1}|}{|D_{\lambda_k - 1}|}$. Observe that $\prod_{k=0}^{n^n-1} q_{\lambda_k}$ is equal to the fraction of space that $D_{\ell'_i - 1}$ occupies in B. This product is greater than $\frac{1}{2^{n/8}}$ since otherwise the fraction of colors i in $B = S_{\ell'_i - 1} \cup D_{\ell'_i - 1}$ is at most $\frac{2}{n} + \frac{1}{2^{n/8}}$, contrary to our assumption that B is not almost sparse. Hence, $\prod_{k=0}^{n^n-1} q_{\lambda_k} > \frac{1}{2^{n/8}}$. The product with this many (namely n^n) factors cannot contain many even mildly small values q_{λ_k} provided the result is as large as $\frac{1}{2^{n/8}}$: indeed, if we have at least $n^{n/2}$ factors that are at most $1 - \frac{1}{n^{n/4}}$, then we already obtain $(1 - \frac{1}{n^{n/4}})^{n^{n/2}} = ((1 - \frac{1}{n^{n/4}})^{n^{n/4}})^{n^{n/4}} = O(\frac{1}{e^{n^{n/4}}})$, much smaller than $\frac{1}{2^{n/8}}$. Thus, less than $n^{n/2}$ numbers $q_{\lambda_1}, \ldots, q_{\lambda_{n^n - 1}}$ can be less than $1 - \frac{1}{n^{n/4}}$ and, for most $k \in [1..n^n)$, we have $q_{\lambda_k} \geq 1 - \frac{1}{n^{n/4}}$. Therefore, the probability that $q_{\ell_{i+1}} < 1 - \frac{1}{n^{n/4}}$, where ℓ_{i+1} is chosen uniformly at random among the levels $\lambda_1, \ldots, \lambda_{n^n-1}$, is at most $\frac{n^{n/2}}{n^n-1} = O(\frac{1}{n^{n/2}})$. For $k \in [1..n^n)$, we call the level λ_k *abnormal for B* if $q_{\lambda_k} < 1 - \frac{1}{n^{n/4}}$, and *normal* otherwise.

Let us define, for each normal level λ_k with $k \in [1..n^n)$, a partition of B into an almost sparse set $\bar{S}_k$ and an "inherently dense" set $\bar{D}_k$ that were announced above. One might suggest that $\bar{D}_k$ can be defined as D_{λ_k}. Indeed, it seems to have the alluded property. However, observe that even when the randomly chosen x_{i+1} "hits" D_{λ_k}, the block $[b_{i+1}..b'_{i+1})$, which is the first level-λ_k block to the right of x_{i+1}, might not be a subset of D_{λ_k}. So, there is no "inheritance" after hitting D_{λ_k}. Our trick is to define $\bar{D}_k$ as D_{λ_k-1} minus the rightmost blocks from level λ_k on each maximal interval in D_{λ_k-1} (see Fig. 4). This trick is the reason why we defined the number q_{λ_k} as the fraction of space that $D_{\lambda_{k+1}-1}$ occupies in D_{λ_k-1}, not in D_{λ_k}. To formalize this, let us decompose D_{λ_k-1} into maximal intervals: $D_{\lambda_k-1} = [d_1..d'_1) \cup \cdots \cup [d_t..d'_t)$, where $d'_j < d_{j+1}$ for $j \in [1..t)$. All the intervals are aligned on block boundaries for blocks from both levels $\lambda_k - 1$ and λ_k. Denote by b the block length on level λ_k. Since the block length on level $\lambda_k - 1$ is $n^n b$, the length of each interval is at least $n^n b$ and the distance between the intervals is at least $n^n b$. If $D_{\lambda_k-1} = B$, define $\bar{D}_k = B$ and $\bar{S}_k = \emptyset$; otherwise, define $\bar{D}_k = [d_1..d'_1{-}b) \cup \cdots \cup [d_t..d'_t{-}b)$ and $\bar{S}_k = B \setminus \bar{D}_k$. Hence, $\bar{S}_k$ is S_{λ_k-1} plus t blocks of size b. Since $\bar{S}_k$ contains at least $t-1$ disjoint intervals each with length at least $n^n b$ (those intervals are the distances between the intervals of D_{λ_k-1}), these added blocks constitute at most a $2\frac{b}{n^n b} = \frac{2}{n^n}$ fraction of the size of S_{λ_k-1}. Hence, the fraction of colors i in $\bar{S}_k$ is at most $\frac{2}{n} + \frac{2}{n^n}$, which is enough for $\bar{S}_k$ to be almost sparse.

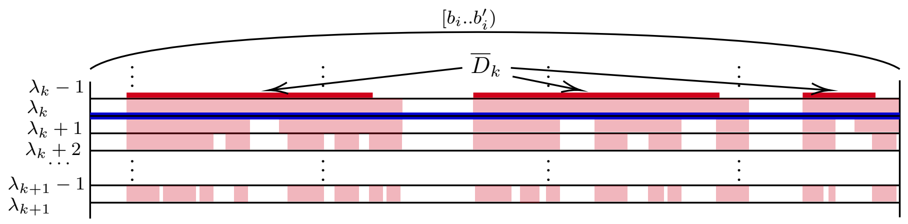


Figure 4 The lines depict consecutive levels $[\lambda_k - 1..\lambda_{k+1}]$ inside a block $[b_i..b'_i)$. The level λ_k is emphasized by the blue color. The red region under line representing level ℓ depicts D_ℓ. The set D_{λ_k-1} consists of three maximal intervals; accordingly, $\bar{D}_k$ is drawn as three thick red lines over D_{λ_k-1} (the gap to the right of each line represents the lacking rightmost block from level λ_k).

For each abnormal level λ_k in the block B, we call all subblocks of B from levels $[\lambda_k..\lambda_{k+1})$ *abnormal*; for each normal level λ_k with $k \in [1..n^n)$, we call all sparse subblocks of D_{λ_k-1} from levels $[\lambda_k..\lambda_{k+1})$ *abnormal.* Thus, for the fixed $i \in [1..n)$ and the fixed level-ℓ_i block B that could be reached by our process on the ith stage of recursion, we have defined abnormal levels, abnormal blocks, and the sets $\bar{D}_k$ and $\bar{S}_k$, for all normal levels λ_k that could be chosen as ℓ_{i+1} for the $(i+1)$th stage of recursion. Analogously, for all $i \in [1..n)$, we define abnormal levels, abnormal blocks, and sets $\bar{D}_k$ and $\bar{S}_k$ for all blocks B that could possibly be reached by our process on the ith stage of recursion. All non-abnormal blocks are called *normal.*

▶ Remark 3. The crucial observation about the normal blocks is as follows: if the last block $[b_n..b'_n)$ reached by our process on the nth stage is normal, then, for each $i \in [1..n)$, this last block can be sparse for color i only if the element x_i was chosen by the process from an almost sparse set $\bar{S}_k$ defined in the corresponding block $B = [b_i..b'_i)$ for level $\ell_{i+1} = \lambda_k$ (this level λ_k must be normal for B since the last block is normal). This behaviour corresponds to our expectations outlined in the beginning of this section: whenever x_i "hits" an "inherently dense" set $\bar{D}$ corresponding to the level ℓ_{i+1} in the block B, it is guaranteed that the last block $[b_n..b'_n)$ will be dense for color i, provided this last block is normal. The idea is that the abnormal blocks are unlikely to appear as last blocks in the process and we will be able to restrict our attention only to normal blocks.

Probability to end up in an abnormal block. For $i \in [1..n)$, denote by $\mathcal{B}_i$ all blocks that could possibly be reached by the process on the ith stage of recursion. Let us estimate the probability that the last block $[b_n..b'_n)$ produced by our process is abnormal as follows:

$$\sum_{i=1}^{n-1} \sum_{B \in \mathcal{B}_i} \Pr\begin{pmatrix}\text{the process}\\ \text{reaches block } B\end{pmatrix} \cdot \Pr\begin{pmatrix}[b_n..b'_n) \text{ ends up being abnormal}\\ \text{after the process reaches } B\end{pmatrix}.$$

For each fixed i, the second sum is through disjoint events "the process reaches block B", for $B \in \mathcal{B}_i$; thus, the sum of the probabilities for these events (with the fixed i) is 1. Therefore, if we prove that, for the fixed i and any fixed $B \in \mathcal{B}_i$, the probability of the event "$[b_n..b'_n)$ ends up being abnormal after the process reaches B" is at most $O(\frac{1}{n^{n/4}})$, then the total sum is bounded as follows:

$$\sum_{i=1}^{n-1} \sum_{B \in \mathcal{B}_i} \Pr\begin{pmatrix}\text{the process}\\ \text{reaches block } B\end{pmatrix} \cdot O\left(\frac{1}{n^{n/4}}\right) = \sum_{i=1}^{n-1} O\left(\frac{1}{n^{n/4}}\right) = O\left(\frac{n}{n^{n/4}}\right) \le O\left(\frac{1}{n^{n/8}}\right). \quad (1)$$

Suppose that the process reaches a block B on the ith stage of recursion. Case (i): it may end up in an abnormal block $[b_n..b'_n)$ if ℓ_{i+1} happens to be an abnormal level. The probability of this is $O(\frac{1}{n^{n/2}})$ since, as was shown, less than $n^{n/2}$ of $n^n - 1$ possible choices for the level ℓ_{i+1} are abnormal and ℓ_{i+1} is chosen uniformly at random. Case (ii): the probability that $[b_n..b'_n)$ ends up being abnormal while ℓ_{i+1} is normal can be estimated as follows (the sum is taken over all normal levels among $\lambda_1, \ldots, \lambda_{n^n-1}$ defined for our fixed block B):

$$\sum_{\substack{\ell \in [\lambda_k..\lambda_{k+1})\\ \text{for normal } \lambda_k}} \Pr(\ell_n = \ell) \cdot \Pr\begin{pmatrix}[b_n..b'_n) \text{ is abnormal}\\ \text{block of level } \ell\end{pmatrix}. \quad (2)$$

Since the events "$\ell_n = \ell$" are disjoint, the sum of $\Pr(\ell_n = \ell)$ is 1 (note that $\Pr(\ell_n = \ell) = 0$, for ℓ unreachable on the nth stage). Therefore, if, for any normal λ_k and $\ell \in [\lambda_k..\lambda_{k+1})$, we estimate by $O(\frac{1}{n^{n/4}})$ the probability that $[b_n..b'_n)$ ends up being an abnormal subblock of B on level ℓ, then the sum (2) is upperbounded by $O(\frac{1}{n^{n/4}})$.

Our random process is designed in such a way that, for any $\ell \in [\lambda_k..\lambda_{k+1})$, it reaches on the nth stage any reacheable level-ℓ subblock of B with equal probability. Since, for any normal level λ_k, we have $q_{\lambda_k} \ge 1 - \frac{1}{n^{n/4}}$, the fraction of abnormal subblocks of B on any level $\ell \in [\lambda_k..\lambda_{k+1})$ is at most $\frac{1}{n^{n/4}}$. However, not all level-ℓ subblocks are reachable since the process always ignores the leftmost block when it chooses the block for the next stage (for instance, when we uniformly at random pick one of level-λ_k subblocks of B for the recursion to stage $i+1$, we cannot choose the leftmost subblock, as it is not located to the right of any $x_i \in B$). Since the number of subblocks for the choice is always at least n^n, the dismissed leftmost subblock renders unreachable at most a $\frac{1}{n^n}$ fraction of level-ℓ subblocks of B. Such dismissals happen for each of the stages $i, i+1, \ldots, n-1$. Hence, the fraction of unreachable level-ℓ subblocks of B is at most $\frac{n}{n^n}$. Consequently, the probability that one of the (equally probable) reachable level-ℓ subblocks of B is abnormal is at most $\frac{1}{n^{n/4}}/(1 - \frac{n}{n^n}) = O(\frac{1}{n^{n/4}})$.

Adding cases (i) and (ii), we obtain the probability $O(\frac{1}{n^{n/2}} + \frac{1}{n^{n/4}}) = O(\frac{1}{n^{n/4}})$ to reach an abnormal block after reaching the block B, which, due to the sum (1), leads to the total probability $O(\frac{1}{n^{n/8}})$ that the last block $[b_n..b'_n)$ in the process ends up being abnormal.

Probability of correct coloring. Now we estimate the probability that the increasing size-n sequence $x_1, \ldots, x_n$ generated by our process is correctly encoded by our fixed coloring of $[0..u)$, i.e., the color of x_i is i, for each $i \in [1..n]$. Suppose that the process generates a sequence $x_1, \ldots, x_n$ and, during its work, reaches levels $\ell_1, \ldots, \ell_n$ and blocks $[b_1..b'_1), \ldots, [b_n..b'_n)$ such that the block $[b_n..b'_n)$ is normal. For each $i \in [1..n)$, let $[b_i..b'_i) = \bar{S}^i \cup \bar{D}^i$ be the described above partition of level-ℓ_{i+1} subblocks of $[b_i..b'_i)$ into an almost sparse set $\bar{S}^i$ and an "inherently dense" set $\bar{D}^i$ (we use the upper indices to avoid confusion with the notation $\bar{S}_k, \bar{D}_k$ used for the partitions on different levels ℓ_{i+1}, not on different stages as we do now). Then, the sequence $x_1, \ldots, x_n$ might be correctly encoded by our fixed coloring of $[0..u)$ only if we had $x_i \in \bar{S}^i$, for at least $\frac{n}{2}$ stages $i \in [1..n)$, since otherwise the whole block $[b_n..b'_n)$ will be dense for at least $\frac{n}{2}$ different colors from $[1..n)$, lacking a room for color n to paint $x_n \in [b_n..b'_n)$ (here we rely on Remark 3 about normal blocks above).

According to this observation, the probability that the generated sequence is correctly encoded can be estimated by the sum of the following numbers (a) and (b):

(a) the probability that $[b_n..b'_n)$ is abnormal,

(b) the probability that the sequence $x_1, \ldots, x_n$ is correctly encoded, subject to the condition that $x_i \in \bar{S}^i$, for at least $\frac{n}{2}$ stages $i \in [1..n)$ of the process that produced $x_1, \ldots, x_n$.

This estimation covers all possible generated sequences except those for which the process ended up in a normal block $[b_n..b'_n)$ but less than $\frac{n}{2}$ stages $i \in [1..n)$ had $x_i \in \bar{S}^i$; but this case can be dismissed since the probability for such sequences to be correctly encoded is zero, as was observed above (they have no room for color n in $[b_n..b'_n)$). We have already deduced that (a) is $O(\frac{1}{n^{n/8}})$. It remains to estimate (b) as $\frac{1}{n^{\Omega(n)}}$.

Fix $M \subseteq [1..n)$ such that $|M| \geq \frac{n}{2}$. Let us estimate the probability p_M that the generated sequence $x_1, \ldots, x_n$ is correctly encoded and satisfies the following condition: $x_i \in \bar{S}^i$, for $i \in M$, and $x_i \in \bar{D}^i$, for $i \notin M$, where $i \in [1..n)$ are the stages of the process that produced $x_1, \ldots, x_n$. We then can upperbound (b) by $\sum_M p_M$, where the sum is through all $M \subseteq [1..n)$ such that $|M| \geq \frac{n}{2}$. If we show that $p_M < \frac{1}{n^{\Omega(n)}}$, then this sum can be bounded by $\frac{2^n}{n^{\Omega(n)}}$, which is equal to $\frac{1}{n^{\Omega(n)}}$. Indeed, for a fixed M and $i \in M$, when the process reaches the ith stage of recursion, the probability that the randomly chosen x_i belongs the set $\bar{S}^i$ and has color i is at most $\frac{2}{n} + \frac{1}{2^{n/8}} \leq \frac{3}{n}$ since the set $\bar{S}^i$ is almost sparse (note that the probability is zero if $\bar{S}^i = \emptyset$). Then, the probability that x_i belongs to $\bar{S}^i$ and has color i, for all $i \in M$, is at most $(\frac{3}{n})^{|M|} \leq (\frac{3}{n})^{n/2} = \frac{1}{n^{\Omega(n)}}$.

References

1 S. Assadi, M. Farach-Colton, and W. Kuszmaul. Tight bounds for monotone minimal perfect hashing. In *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 456–476. SIAM, 2023. doi:10.1137/1.9781611977554.ch2.

2 D. 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, 2014. doi:10.1145/2591796.2591885.

3 D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. In *Proc. SODA*, pages 785–794. SIAM, 2009. doi:10.1137/1.9781611973068.86.

4 D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009. doi:10.1007/978-3-642-04128-0_61.

5 D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020. doi:10.1145/3381417.

6 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):1–19, 2014. `doi:10.1145/2635816`.

7 D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms (TALG)*, 11(4):1–21, 2015. `doi:10.1145/2629339`.

8 D. Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.

9 R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. 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`.

10 T. M. Cover and J. A. Thomas. Information theory and statistics. *Elements of Information Theory*, 1(1):279–335, 1991. `doi:10.1002/0471200611`.

11 M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984. `doi:10.1137/0605009`.

12 M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. `doi:10.1145/828.1884`.

13 T. Gagie, G. Navarro, and B. Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020. `doi:10.1145/3375890`.

14 R. Grossi, A. Orlandi, and R. Raman. Optimal trade-offs for succinct string indexes. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37*, pages 678–689. Springer, 2010. `doi:10.1007/978-3-642-14165-2_57`.

15 G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE, 1989. `doi:10.1109/SFCS.1989.63533`.

16 K. Mehlhorn. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pages 170–175. IEEE, 1982. `doi:10.1109/SFCS.1982.80`.

17 J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41(4):203–207, 1992. `doi:10.1016/0020-0190(92)90181-T`.

Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space

Dmitry Kosolobov
Ural Federal University, Ekaterinburg, Russia

Nikita Sivukhin
Ural Federal University, Ekaterinburg, Russia

Abstract

The notions of synchronizing and partitioning sets are recently introduced variants of locally consistent parsings with a great potential in problem-solving. In this paper we propose a deterministic algorithm that constructs for a given readonly string of length n over the alphabet $\{0, 1, \dots, n^{\mathcal{O}(1)}\}$ a variant of a τ-partitioning set with size $\mathcal{O}(b)$ and $\tau = \frac{n}{b}$ using $\mathcal{O}(b)$ space and $\mathcal{O}(\frac{1}{\epsilon}n)$ time provided $b \ge n^{\epsilon}$, for $\epsilon > 0$. As a corollary, for $b \ge n^{\epsilon}$ and constant $\epsilon > 0$, we obtain linear time construction algorithms with $\mathcal{O}(b)$ space on top of the string for two major small-space indexes: a sparse suffix tree, which is a compacted trie built on b chosen suffixes of the string, and a *longest common extension* (LCE) index, which occupies $\mathcal{O}(b)$ space and allows us to compute the longest common prefix for any pair of substrings in $\mathcal{O}(n/b)$ time. For both, the $\mathcal{O}(b)$ construction storage is asymptotically optimal since the tree itself takes $\mathcal{O}(b)$ space and any LCE index with $\mathcal{O}(n/b)$ query time must occupy at least $\mathcal{O}(b)$ space by a known trade-off (at least for $b \ge \Omega(n/\log n)$). In case of arbitrary $b \ge \Omega(\log^2 n)$, we present construction algorithms for the partitioning set, sparse suffix tree, and LCE index with $\mathcal{O}(n \log_b n)$ running time and $\mathcal{O}(b)$ space, thus also improving the state of the art.

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

Keywords and phrases (τ, δ)-partitioning set, longest common extension, sparse suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.20

Related Version *Full Version*: `https://arxiv.org/abs/2105.03782`

Funding Supported by the Russian Science Foundation (RSF), project 20-71-00050.

1 Introduction

Indexing data structures traditionally play a central role in algorithms on strings and in information retrieval. Due to constantly growing volumes of data in applications, the attention of researchers in the last decades was naturally attracted to small-space indexes. In this paper we study two closely related small-space indexing data structures: a sparse suffix tree and a longest common extension (LCE) index. We investigate them in the general framework of (deterministic) locally consistent parsings that was developed by Cole and Vishkin [7], Jeż [21, 20, 22, 19], and others [1, 11, 12, 13, 15, 28, 29, 32] (the list is not exhausting) and was recently revitalized in the works of Birenzwige et al. [5] and Kempa and Kociumaka [25] where two new potent concepts of partitioning and synchronizing sets were introduced.

The sparse suffix tree (*SST*) for a given set of b suffixes of a string is a compacted trie built on these suffixes. It can be viewed as the suffix tree from which all suffixes not from the set were removed (details follow). The tree takes $\mathcal{O}(b)$ space on top of the input string and can be easily constructed in $\mathcal{O}(n)$ time from the suffix tree, where n is the length of the string. One can build the suffix tree in $\mathcal{O}(n)$ time [10] provided the letters of the string are sortable in linear time. However, if at most $\mathcal{O}(b)$ space is available on top of the input, then in general there is not enough memory for the full suffix tree and the problem, thus, becomes much more difficult. The $\mathcal{O}(b)$ bound is optimal since the tree itself takes $\mathcal{O}(b)$

space. The construction problem with restricted $\mathcal{O}(b)$ space naturally arises in applications of the sparse suffix tree and the sparse suffix array (which is easy to retrieve from the tree) where we have to index data in the setting of scarce memory. As is common in algorithms on strings, it is assumed that the input string is readonly, its letters are polynomially bounded integers $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$, and the space is at least polylogarithmic, i.e., $b \geq \log^{\Omega(1)} n$. We note, however, that in supposed usages the memory restrictions can often be relaxed even more to $b \geq n^{\epsilon}$, for constant $\epsilon > 0$.

The $\mathcal{O}(b)$-space construction problem was posed by Kärkkäinen and Ukkonen [24] who showed how to solve it in linear time for the case of evenly spaced b suffixes. In a series of works [2, 5, 11, 14, 18, 23], the problem was settled for the case of randomized algorithms: an optimal linear $\mathcal{O}(b)$-space Monte Carlo construction algorithm for the sparse suffix tree was proposed by Gawrychowski and Kociumaka [14] and an optimal linear $\mathcal{O}(b)$-space Las-Vegas algorithm was described by Birenzwige et al. [5]. The latter authors also presented the best up-to-date deterministic solution that builds the sparse suffix tree within $\mathcal{O}(b)$ space in $\mathcal{O}(n \log \frac{n}{b})$ time [5] (log is in base 2 unless explicitly stated otherwise). All these solutions assume (as we do too) that the input string is readonly and its alphabet is $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$; the case of rewritable inputs is apparently very different, as was shown by Prezza [30].

The LCE index, crucial in string algorithm applications, preprocesses a readonly input string of length n so that one can answer queries $\mathsf{lce}(p, q)$, for any positions p and q, computing the length of the longest common prefix of the suffixes starting at p and q. The now classical result of Harel and Tarjan states that the LCE queries can be answered in $\mathcal{O}(1)$ time provided $\mathcal{O}(n)$ space is used [16]. In [3] Bille et al. presented an LCE index that, for any given user-defined parameter b, occupies $\mathcal{O}(b)$ space on top of the input string and answers queries in $\mathcal{O}(\frac{n}{b})$ time. In [26] it was proved that this time-space trade-off is optimal provided $b \geq \Omega(n/\log n)$ (it is conjectured that the same trade-off lower bound holds for a much broader range of values b; a weaker trade-off appears in [4, 6]). In view of these lower bounds, it is therefore natural to ask how fast one can construct, for any parameter b, an LCE index that can answer queries in $\mathcal{O}(\frac{n}{b})$ time using $\mathcal{O}(b)$ space on top of the input. The space $\mathcal{O}(b)$ is optimal for this query time and the construction algorithm should not exceed it. The issue with the data structure of [3] is that its construction time is unacceptably slow, which motivated a series of works trying to solve this problem. As in the case of sparse suffix trees, the problem was completely settled in the randomized setting: an optimal linear $\mathcal{O}(b)$-space Monte Carlo construction algorithm for an LCE index with $\mathcal{O}(\frac{n}{b})$-time queries was presented by Gawrychowski and Kociumaka [14] and a Las-Vegas construction with the same time and space was proposed by Birenzwige et al. [5] provided $b \geq \Omega(\log^2 n)$. The best deterministic solution is also presented in [5] and runs in $\mathcal{O}(n \log \frac{n}{b})$ time answering queries in slightly worse time $\mathcal{O}(\frac{n}{b}\sqrt{\log^* n})$ provided $b \geq \Omega(\log n)$ (the previous best solution was from [34] and it runs in $\mathcal{O}(n \cdot \frac{n}{b})$ time but, for some exotic parameters b, has slightly better query time). The input string is readonly in all these solutions and the alphabet is $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$.

For a broad range of values b, we settle both construction problems, for sparse suffix trees and LCE indexes, in $\mathcal{O}(b)$ space in the deterministic case. Specifically, given a readonly string of length n over the alphabet $\{0, 1, \ldots, n^{\mathcal{O}(1)}\}$, we present two algorithms: one that constructs the sparse suffix tree, for any user-defined set of b suffixes such that $b \geq \Omega(\log^2 n)$, in $\mathcal{O}(n \log_b n)$ time using $\mathcal{O}(b)$ space on top of the input; and another that constructs an LCE index with $\mathcal{O}(\frac{n}{b})$-time queries, for any parameter b such that $b \geq \Omega(\log^2 n)$, in $\mathcal{O}(n \log_b n)$ time using $\mathcal{O}(b)$ space on top of the input. This gives us optimal $\mathcal{O}(b)$-space solutions with $\mathcal{O}(\frac{1}{\epsilon} n) = \mathcal{O}(n)$ time when $b \geq n^{\epsilon}$, for constant $\epsilon > 0$, which arguably includes most interesting cases. As can be seen in Table 1, our result beats the previous best solution in virtually all settings since $n \log_b n = o(n \log \frac{n}{b})$, for $b = o(n)$.

Table 1 LCE indexes deterministically constructible in $\mathcal{O}(b)$ space on a readonly input, for $b \ge \Omega(\log^2 n)$.

Algorithm	Tanimura et al. [34]	Birenzwige et al. [5]	Theorem 4
Query time	$\mathcal{O}(\frac{n}{b}\log\min\{b, \frac{n}{b}\})$	$\mathcal{O}(\frac{n}{b}\sqrt{\log^* n})$	$\boldsymbol{\mathcal{O}(\frac{n}{b})}$
Construction in $\mathcal{O}(b)$ space	$\mathcal{O}(n \cdot \frac{n}{b})$	$\mathcal{O}(n \log \frac{n}{b})$	$\boldsymbol{\mathcal{O}(n \log_b n)}$

In order to achieve these results, we develop a new algorithm that, for any given parameter $b \ge \Omega(\log^2 n)$, constructs a so-called τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$. This result is of independent interest.

We note that there is another natural model where the input string is packed in memory in such a way that one can read in $\mathcal{O}(1)$ time any $\Theta(\log_\sigma n)$ consecutive letters of the input packed into one $\Theta(\log n)$-bit machine word, where $\{0, 1, \ldots, \sigma-1\}$ is the input alphabet. In this case the $\mathcal{O}(n)$ construction time is not necessarily optimal for the sparse suffix tree and the LCE index and one might expect to have $\mathcal{O}(n/\log_\sigma n)$ time. As was shown by Kempa and Kociumaka [25], this is indeed possible for LCE indexes in $\mathcal{O}(n/\log_\sigma n)$ space. It remains open whether one can improve our results for the $\mathcal{O}(b)$-space construction in this setting; note that the lower bound of [26] does not apply here due to its assumption of single-letter input memory cells.

Techniques. The core of our solution is a version of locally consistent parsing developed by Birenzwige et al. [5], the so-called τ-partitioning sets (unfortunately, we could not adapt the more neat τ-synchronizing sets from [25] for the deterministic case). It was shown by Birenzwige et al. that the $\mathcal{O}(b)$-space construction of a sparse suffix tree or an LCE index can be performed in linear time provided a τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ is given. We define a variant of τ-partitioning sets and, for completeness, repeat the argument of Birenzwige et al. with minor adaptations to our case. The main bulk of the text is devoted to the description of an $\mathcal{O}(b)$-space algorithm that builds a (variant of) τ-partitioning set of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ in $\mathcal{O}(n \log_b n)$ time provided $b \ge \Omega(\log^2 n)$, which is the main result of the present paper. In comparison Birenzwige et al.'s algorithm for their τ-partitioning sets runs in $\mathcal{O}(n)$ *expected* time (so that it is a Las Vegas construction) and $\mathcal{O}(b)$ space; their deterministic algorithm takes $\mathcal{O}(n \log \tau)$ time but the resulting set is only $\tau \log^* n$-partitioning. Concepts very similar to partitioning sets appeared also in [31, 33].

Our solution combines two well-known approaches to deterministic locally consistent parsings: the *deterministic coin tossing* introduced by Cole and Vishkin [7] and developed in [1, 11, 12, 13, 15, 28, 29, 32], and the *recompression* invented by Jeż [19] and studied in [17, 21, 20, 22]. The high level idea is first to use Cole and Vishkin's technique that constructs a τ-partitioning set of size $\mathcal{O}(b \log^* n)$ where $\tau = \frac{n}{b}$ (in our algorithm the size is actually $\mathcal{O}(b \log\log\log n)$ since we use a "truncated" version of Cole and Vishkin's bit reductions); second, instead of storing the set explicitly, which is impossible in $\mathcal{O}(b)$ space, we construct a string R of length $\mathcal{O}(b \log^* n)$ in which every letter corresponds to a position of the set and occupies $o(\log\log n)$ bits so that R takes $o(b \log^* n \log\log n)$ bits in total and, thus, can be stored into $\mathcal{O}(b)$ machine words of size $\mathcal{O}(\log n)$ bits; third, Jeż's recompression technique is iteratively applied to the string R until R is shortened to length $\mathcal{O}(b)$; finally, the first technique generating a τ-partitioning set is performed again but this time we retain and store explicitly those positions that correspond to surviving letters of the string R. There are many hidden obstacles on this path and because of them our solution is only of purely theoretical value in its present form due to numerous internal complications in the actual scheme (in contrast, randomized results on synchronizing sets [9, 25] seem quite practical).

The paper is organized as follows. In Section 2 we define τ-partitioning sets and show how one can use them to build an LCE index. Section 3 describes the first stage of the construction of a τ-partitioning set that is based on a modification of Cole and Vishkin's technique. Section 4 improves the running time of this stage from $\mathcal{O}(n \log \tau)$ to $\mathcal{O}(n \log_b \tau)$. In Section 5 the second stage based on a modification of Jeż's recompression technique is presented. Appendix C in the full version [27] describes separately the case of very small τ.

2 Partitioning Sets with Applications

Let us fix a readonly string s of length n whose letters $s[0], s[1], \dots, s[n{-}1]$ are from a polynomially bounded alphabet $\{0, 1, \dots, n^{\mathcal{O}(1)}\}$. We use s as the input in our algorithms. As is standard, the algorithms are in the word-RAM model, their space is measured in $\Theta(\log n)$-bit machine words, and each $s[i]$ occupies a separate word. We write $s[i..j]$ for the *substring* $s[i]s[i{+}1] \cdots s[j]$, assuming it is empty if $i > j$; $s[i..j]$ is called a *suffix* (resp., *prefix*) of s if $j = n-1$ (resp., $i = 0$). For any string t, let $|t|$ denote its length. We say that t *occurs* at position i in s if $s[i..i{+}|t|{-}1] = t$. Denote $[i..j] = \{k \in \mathbb{Z} \colon i \le k \le j\}$, $(i..j] = [i..j] \setminus \{i\}$, $[i..j) = [i..j] \setminus \{j\}$, $(i..j) = [i..j) \cap (i..j]$. A number $p \in [1..|t|]$ is called a *period* of t if $t[i] = t[i-p]$ for each $i \in [p..|t|)$. For brevity, denote $\log\log\log n$ by $\log^{(3)} n$. We assume that n, the length of s, is sufficiently large: larger than $2^{\max\{16,c\}}$, where c is a constant such that n^c upper-bounds the alphabet.

Given an integer $\tau \in [4..n/2]$, a set of positions $S \subseteq [0..n)$ is called a *τ-partitioning set* if it satisfies the following properties:

(a) if $s[i{-}\tau..i{+}\tau] = s[j{-}\tau..j{+}\tau]$ for $i, j \in [\tau..n{-}\tau)$, then $i \in S$ iff $j \in S$;

(b) if $s[i..i{+}\ell] = s[j..j{+}\ell]$, for $i, j \in S$ and some $\ell \ge 0$, then, for each $d \in [0..\ell{-}\tau)$, $i + d \in S$ iff $j + d \in S$;

(c) if $i, j \in S \cup \{0, n{-}1\}$ with $j{-}i > \tau$ and $(i..j) \cap S = \emptyset$, then the period of $s[i..j]$ is at most $\tau/4$.

Our definition is inspired by the *forward synchronized* (τ, τ)*-partitioning sets* from [5, Def. 3.1 and 6.1] but slightly differs; nevertheless, we retain the term "partitioning" to avoid inventing unnecessary new terms for very close concepts. In the definition, (a), (b), and (c) state, respectively, that S is locally consistent, forward synchronized, and dense: the choice of positions depends only on short substrings around them, long enough equal right "contexts" of positions from S are "partitioned" identically, and S has a position every τ letters unless a long range with small period is encountered. In our construction of S a certain converse of (c) will also hold: whenever a substring $s[i..j]$ has a period at most $\tau/4$, we will have $S \cap [i + \tau..j - \tau] = \emptyset$ (see Lemma 17). This converse is not in the definition since it is unnecessary for our applications and we will use auxiliary τ-partitioning sets not satisfying it. The definition also implies the following convenient property of "monotonicity".

▶ **Lemma 1.** *For any $\tau' \ge \tau$, every τ-partitioning set is also τ'-partitioning.*

Due to (c), all τ-partitioning sets in some strings have size at least $\Omega(n/\tau)$. In the remaining sections we devise algorithms that construct a τ-partitioning set of s with size $\mathcal{O}(n/\tau)$ (matching the lower bound) using $\mathcal{O}(n/\tau)$ space on top of s; for technical reasons, we assume that $\Omega(\log^2 n)$ space is always available, i.e., $n/\tau \ge \Omega(\log^2 n)$, which is a rather mild restriction. Thus, we shall prove the following main theorem.

▶ **Theorem 2.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $\tau \in [4..\mathcal{O}(n/\log^2 n)]$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string a τ-partitioning set of size $\mathcal{O}(b)$, for $b = n/\tau$.*

Let us sketch how one can construct indexes with the τ-partitioning set of Theorem 2.

LCE index and sparse suffix tree. An LCE index is a data structure on s that, given a pair of positions p and q, answers the *LCE query* $\mathsf{lce}(p, q)$ computing the length of the longest common prefix of $s[p..n{-}1]$ and $s[q..n{-}1]$. Such index can be stored in $\mathcal{O}(b)$ space on top of s with $\mathcal{O}(\frac{n}{b})$ query time [3] and this trade-off is optimal, at least for $b \ge \Omega(\frac{n}{\log n})$ [26].

Given b suffixes $s[i_1..n{-}1], s[i_2..n{-}1], \ldots, s[i_b..n{-}1]$, their *sparse suffix tree* [24] is a compacted trie on these suffixes in which all edge labels are stored as pointers to corresponding substrings of s. Thus, the tree occupies $\mathcal{O}(b)$ space.

Our construction scheme for these two indexes is roughly as follows: given a τ-partitioning set S with $\tau = \frac{n}{b}$ and size $\mathcal{O}(b) = \mathcal{O}(n/\tau)$, we first build the sparse suffix tree T for the suffixes $s[j..n{-}1]$ with $j \in S$, then we use it to construct an LCE index, and, using the index, build the sparse suffix tree for arbitrarily chosen b suffixes. We elaborate on this scheme below; our exposition, however, is rather sketchy and some details are omitted since the scheme is essentially the same as in [5] and is given here mostly for completeness.

To construct the sparse suffix tree T for all $s[j..n{-}1]$ with $j \in S$, we apply the following lemma. Its cumbersome formulation is motivated by its subsequent use in Section 4. In the special case when $m = n$ and $\sigma = n^{\mathcal{O}(1)}$, which is of primary interest for us now, the lemma states that T can be built in $\mathcal{O}(n)$ time: this case implies that $m \log_b \sigma = \mathcal{O}(n \log_b n)$ is $\mathcal{O}(n)$ if $b > n/\log n$, and $b \log b$ is $\mathcal{O}(n)$ if $b \le n/\log n$, and, therefore, $\min\{m \log_b \sigma, b \log b\} = \mathcal{O}(n)$. The proof essentially follows arguments of [5] and is given in Appendix A in the full version [27].

▶ **Lemma 3.** *Given an integer $\tau \ge 4$ and a read-only string s of length m over an alphabet $[0..\sigma)$, let S be an "almost" τ-partitioning set of size $b = \Theta(m/\tau)$: it satisfies properties (a) and (b), but not necessarily (c). The sparse suffix tree T for all suffixes $s[j..m{-}1]$ with $j \in S$ can be built in $\mathcal{O}(m + \min\{m \log_b \sigma, b \log b\})$ time and $\mathcal{O}(m/\tau)$ space on top of the space required for s.*

For our LCE index, we equip T with the lowest common ancestor (LCA) data structure [16], which allows us to compute $\mathsf{lce}(p, q)$ in $\mathcal{O}(1)$ time for $p, q \in S$, and we preprocess an array $N[0..b{-}1]$ such that $N[i] = \min\{j \ge i\tau \colon j \in S\}$ for $i \in [0..b)$, which allows us to calculate $\min\{j \ge p \colon j \in S\}$, for any p, in $\mathcal{O}(\tau)$ time by traversing $j_k, j_{k+1}, \ldots$ in S, for $j_k = N[\lfloor p/\tau \rfloor]$. In order to answer an arbitrary query $\mathsf{lce}(p, q)$, we first calculate $p' = \min\{j \ge p + \tau \colon j \in S\}$ and $q' = \min\{j \ge q{+}\tau \colon j \in S\}$ in $\mathcal{O}(\tau)$ time. If either $p' - p \le 2\tau$ or $q' - q \le 2\tau$, then by the local consistency of S, $s[p..n{-}1]$ and $s[q..n{-}1]$ either differ in their first 3τ positions, which is checked naïvely, or $s[p..p'] = s[q..q']$ and the answer is given by $p' - p + \mathsf{lce}(p', q')$ using T. If $\min\{p' - p, q' - q\} > 2\tau$, then the strings $s[p{+}\tau..p']$ and $s[q{+}\tau..q']$ both have periods at most $\tau/4$ due to property (c); we compare $s[p..p{+}2\tau]$ and $s[q..q{+}2\tau]$ naïvely and, if there are no mismatches, therefore, due to periodicity, $s[p{+}\tau..p']$ and $s[q{+}\tau..q']$ have a common prefix of length $\ell = \min\{p' - p, q' - q\} - \tau$; hence, the problem is reduced to $\mathsf{lce}(p + \ell, q + \ell)$, which can be solved as described above since either $p' - (p + \ell) \le 2\tau$ or $q' - (q + \ell) \le 2\tau$. We thus have proved the following theorem.

▶ **Theorem 4.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $b \ge \Omega(\log^2 n)$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string an LCE index that can answer LCE queries in $\mathcal{O}(n/b)$ time.*

Let us consider the construction of the SST for b suffixes $s[i_1..n{-}1]$, $s[i_2..n{-}1], \ldots, s[i_b..n{-}1]$. Denote by j_k the kth position in a given τ-partitioning set S of size $\mathcal{O}(b)$ with $\tau = \frac{n}{b}$ (so that $j_1 < \cdots < j_{|S|}$). For each suffix $s[i_\ell..n{-}1]$, we compute in $\mathcal{O}(\tau)$ time using the array N an index k_ℓ such that $j_{k_\ell} = \min\{j \geq i_\ell + \tau \colon j \in S\}$. It takes $\mathcal{O}(b\tau) = \mathcal{O}(n)$ time in total. Then, we sort all strings $s[i_\ell..i_\ell{+}4\tau]$ in $\mathcal{O}(n)$ time as in the proof of Lemma 3 and assign to them ranks r_ℓ (equal strings are of equal ranks). For each $k \in [1..|S|]$, we obtain from the tree T the rank $\bar{r}_k$ of $s[j_k..n{-}1]$ among the suffixes $s[j..n{-}1]$ with $j \in S$. Suppose that $j_{k_\ell} \leq i_\ell + 3\tau$, for all $\ell \in [1..b]$. By property (a), the equality $r_\ell = r_{\ell'}$, for any $\ell, \ell' \in [1..b]$, implies that $j_{k_\ell} - i_\ell = j_{k_{\ell'}} - i_{\ell'}$ when $j_{k_\ell} - i_\ell \leq 3\tau$. Then, we sort the suffixes $s[i_\ell..n{-}1]$ with $\ell \in [1..b]$ in $\mathcal{O}(b)$ time using the radix sort on the corresponding pairs $(r_\ell, \bar{r}_{j_{k_\ell}})$. The SST can be assembled from the sorted suffixes in $\mathcal{O}(b\tau) = \mathcal{O}(n)$ time using the LCE index to calculate longest common prefixes of adjacent suffixes.

The argument is more intricate when the condition $j_{k_\ell} > i_\ell + 3\tau$ does not hold. Suppose that $j_{k_\ell} > i_\ell + 3\tau$, for some $\ell \in [1..b]$. Then, by property (c), the minimal period of $s[i_\ell{+}\tau..j_{k_\ell}]$ is at most $\tau/4$. Denote this period by p_ℓ. We compute p_ℓ in $\mathcal{O}(\tau)$ time using a linear $\mathcal{O}(1)$-space algorithm [8] and, then, we find the leftmost position $t_\ell > j_{k_\ell}$ breaking this period: $s[t_\ell] \neq s[t_\ell{-}p_\ell]$. As $j_{k_\ell}{-}p_\ell > i_\ell{+}2\tau > j_{k_\ell - 1}$, we obtain $s[j_{k_\ell}{-}\tau..j_{k_\ell}{+}\tau] \neq s[j_{k_\ell}{-}p_\ell{-}\tau..j_{k_\ell}{-}p_\ell{+}\tau]$ (since otherwise $j_{k_\ell}{-}p_\ell \in S$ by property (a)) and, hence, $t_\ell \in (j_{k_\ell}..j_{k_\ell}{+}\tau]$. Therefore, the computation of t_ℓ takes $\mathcal{O}(\tau)$ time. Thus, all p_ℓ and t_ℓ can be calculated in $\mathcal{O}(b\tau) = \mathcal{O}(n)$ total time. We then sort the strings $s[t_\ell..t_\ell{+}\tau]$ in $\mathcal{O}(n)$ time and assign to them ranks $\tilde{r}_\ell$. For each suffix $s[i_\ell..n{-}1]$ with $\ell \in [1..b]$, we associate the tuple $(r_\ell, 0, 0, \bar{r}_{j_{k_\ell}})$ if $j_{k_\ell} \leq i_\ell + 3\tau$, and the tuple $(r_\ell, d_\ell, \tilde{r}_\ell, \bar{r}_{j_{k_\ell}})$ if $j_{k_\ell} > i_\ell{+}3\tau$, where $d_\ell = \pm(t_\ell{-}i_\ell - n)$ with plus if $s[t_\ell] < s[t_\ell{-}p_\ell]$ and minus otherwise. We claim that the order of the suffixes $s[i_\ell..n{-}1]$ is the same as the order of their associated tuples and, hence, the suffixes can be sorted by sorting the tuples in $\mathcal{O}(n)$ time using the radix sort. We then assemble the SST as above using the LCE index. We do not dive into the proof of the claim since it essentially repeats similar arguments in [5]; see [5] for details.

▶ **Theorem 5.** *For any string of length n over an alphabet $[0..n^{\mathcal{O}(1)}]$ and any $b \geq \Omega(\log^2 n)$, one can construct in $\mathcal{O}(n \log_b n)$ time and $\mathcal{O}(b)$ space on top of the string the sparse suffix tree for arbitrarily chosen b suffixes.*

3 Refinement of Partitioning Sets

In this section we describe a process that takes the trivial partitioning set $[0..n)$ and iteratively refines it in $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases removing some positions so that, after the kth phase, the set is $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning and has size $\mathcal{O}(n/2^k)$; moreover, it is "almost" 2^{k+3}-partitioning, satisfying properties (a) and (b) but not necessarily (c) (for $\tau = 2^{k+3}$). In particular, the set after the last phase is $\frac{\tau}{2}$-partitioning (and, thus, τ-partitioning by Lemma 1) and has size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$. Each phase processes all positions of the currently refined set from left to right and, in an almost online fashion, chooses which of them remain in the set. Rather than performing the phases one after another, which requires $\mathcal{O}(n)$ space, we run them simultaneously feeding the positions generated by the kth phase to the $(k{+}1)$th phase. Thus, the resulting set is produced in one pass. The set, however, has size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$, which is still too large to be stored in $\mathcal{O}(n/\tau)$ space; this issue is addressed in Section 5. Let us elaborate on the details of this process.

Throughout this section, we assume that $\tau \geq 2^5 \log^{(3)} n$ and, hence, the number of phases is non-zero; the case $\tau < 2^5 \log^{(3)} n$ is addressed in Appendix E in the full version [27]. Consider the kth phase, for $k \geq 1$. Its input is a set S_{k-1} produced by the $(k{-}1)$th phase; for $k = 1$, $S_0 = [0..n)$. Denote by j_h the hth position in S_{k-1} (so that $j_1 < \cdots < j_{|S_{k-1}|}$). The phase processes $j_1, j_2, \ldots$ from left to right and decides which of them to put into the new set $S_k \subseteq S_{k-1}$ under construction. The decision for j_h is based on the distances $j_h - j_{h-1}$ and $j_{h+1} - j_h$, on the substrings $s[j_{h+\ell}..j_{h+\ell}{+}2^k]$ with $\ell \in [-1..4]$, and on certain numbers v_{h-1}, v_h, v_{h+1} computed for j_{h-1}, j_h, j_{h+1}, which we define below. For technical reasons, we also assume $j_0 = -\infty$ and $j_{|S_{k-1}|+1} = \infty$, so $j_1 - j_0 = \infty$ and $j_{|S_{k-1}|+1} - j_{|S_{k-1}|} = \infty$.

For any distinct integers $x, y \geq 0$, denote by $\mathsf{bit}(x, y)$ the index of the lowest bit in which the bit representations of x and y differ (the lowest bit has index 0); e.g., $\mathsf{bit}(1, 0) = 0$, $\mathsf{bit}(2, 8) = 1$, $\mathsf{bit}(8, 0) = 3$. It is well known that $\mathsf{bit}(x, y)$ can be computed in $\mathcal{O}(1)$ time provided x and y occupy $\mathcal{O}(1)$ machine words [35]. Denote $\mathsf{vbit}(x, y) = 2\,\mathsf{bit}(x, y) + a$, where a is the bit of x with index $\mathsf{bit}(x, y)$; e.g., $\mathsf{vbit}(8, 0) = 7$ and $\mathsf{vbit}(0, 8) = 6$. Note that the bit representation of the number $\mathsf{vbit}(x, y)$ is obtained from that of $\mathsf{bit}(x, y)$ by appending a.

Let w be the number of bits in an $\mathcal{O}(\log n)$-bit machine word sufficient to represent letters from the alphabet $[0..n^{\mathcal{O}(1)}]$ of s. For each j_h, denote $s_h = \sum_{i=0}^{2^k} s[j_h{+}i]2^{wi}$. Each number s_h takes $(2^k{+}1)w$ bits and its bit representation coincides with that of the string $s[j_h..j_h{+}2^k]$, when we treat this string as a number stored in memory in the little endian format. The numbers s_h are introduced merely for convenience of the exposition, they are never discerned from their corresponding substrings $s[j_h..j_h{+}2^k]$ in the algorithm. For each j_h, define $v'_h = \mathsf{vbit}(s_h, s_{h+1})$ if $j_{h+1} - j_h \leq 2^{k-1}$ and $s_h \neq s_{h+1}$, and $v'_h = \infty$ otherwise. Observe that $\mathsf{bit}(s_h, s_{h+1}) = w\ell + \mathsf{bit}(s[j_h{+}\ell], s[j_{h+1}{+}\ell])$, where $\ell = \mathsf{lce}(j_h, j_{h+1})$; i.e., $\mathsf{bit}(s_h, s_{h+1})$ is given by an LCE query in the bit string of length wn obtained from s by substituting each letter with its w-bit representation. Define $v''_h = \mathsf{vbit}(v'_h, v'_{h+1})$, $v'''_h = \mathsf{vbit}(v''_h, v''_{h+1})$, $v_h = \mathsf{vbit}(v'''_h, v'''_{h+1})$, assuming $\mathsf{vbit}(x, y) = \infty$ if either $x = \infty$ or $y = \infty$.

For each j_h, denote by $R(j_h)$ a predicate that is true iff $j_{h+1} {-} j_h \leq 2^{k-1}$ and $s_h = s_{h+1}$; to verify whether $R(j_h)$ holds, we always check the former condition first and only then the latter if the former condition is satisfied.

Refinement rule. *The kth phase decides to put a position j_h into S_k either if $\infty > v_{h-1} > v_h$ and $v_h < v_{h+1}$ (i.e., $v_{h-1} \neq \infty$ and v_h is a local minimum of the sequence $v_1, v_2, \ldots$), or in three "boundary" cases: (i) $j_{h+1} - j_h > 2^{k-1}$ or $j_h - j_{h-1} > 2^{k-1}$; (ii) $R(j_{h-1})$ does not hold while $R(j_h)$, $R(j_{h+1})$, $R(j_{h+2})$ hold; (iii) $R(j_h)$ holds but $R(j_{h+1})$ does not.*

Note that we always have $j_1, j_{|S_{k-1}|} \in S_k$ since $j_1 {-} j_0 = \infty > 2^{k-1}$ and $j_{|S_{k-1}|+1} {-} j_{|S_{k-1}|} = \infty > 2^{k-1}$. For now, assume that the numbers $\mathsf{bit}(s_h, s_{h+1})$, required to calculate v'_h and $R(j_h)$, are computed by the naïve comparison of $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ in $\mathcal{O}(2^k)$ time (we will change it later). Thus, the process is well defined. The trick with local minima and vbit reductions is, in essence, as in the deterministic approach of Cole and Vishkin to locally consistent parsings [7]. In what follows we derive some properties of this approach in order to prove that the kth phase indeed produces a $(2^{k+3}\lfloor\log^{(3)} n\rfloor)$-partitioning set.

It is convenient to interpret the kth phase as follows (see Fig. 1): the sequence $j_1, j_2, \ldots$ is split into maximal disjoint contiguous regions such that, for any pair of adjacent positions j_h and j_{h+1} inside each region, the distance $j_{h+1} - j_h$ is at most 2^{k-1} and $R(j_h) = R(j_{h+1})$. Thus, the regions are of two types: all-R ($\{j_{16}, \ldots, j_{20}\}$ in Fig. 1) and all-non-R ($\{j_1, \ldots, j_{15}\}$ or $\{j_{21}, \ldots, j_{25}\}$ in Fig. 1). By case (i), for each long gap $j_{h+1} - j_h > 2^{k-1}$ between regions, we put both j_h and j_{h+1} into S_k. In each all-R region, we put into S_k its last position due to case (iii) and, if the length of the region is at least 3, its first position by case (ii). In each all-non-R region, we put into S_k all local minima v_h such that $v_{h-1} \neq \infty$. Only all-non-R regions have positions j_h with $v_h \neq \infty$; moreover, as it turns out, only the last three or four their positions j_h have $v_h = \infty$ whereas, for other j_h, $v_h \neq \infty$ and $v_h \neq v_{h+1}$. Lemmas 8, 9 describe all this formally; their proof is deferred to Appendix B.1 in the full version [27].

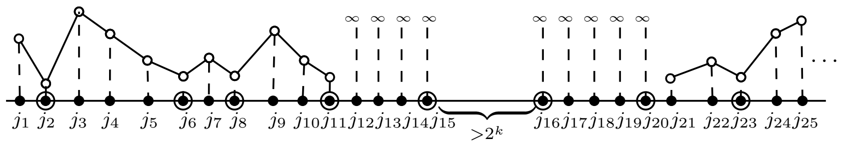

∞
j_1 j_2 j_3 j_4 j_5 j_6 j_7 j_8 j_9 j_{10} j_{11} j_{12} j_{13} j_{14} j_{15}
$>2^k$
j_{16} j_{17} j_{18} j_{19} j_{20} j_{21} j_{22} j_{23} j_{24} j_{25}

Figure 1 The kth phase. The heights of the dashed lines over j_h are equal to v_h. Encircled positions are put into S_k: they are local minima of v_h, or are at the "boundaries" of all-R regions, or form a gap of length $>2^k$. In the figure $R(j_{16}), \dots, R(j_{20})$ hold and $R(j_{21})$ does not hold.

The goal of the fourfold vbit reduction for v_h is to make v_h small enough so that local minima occur often and, thus, the resulting set S_k is not too sparse. This is the key observation of Cole and Vishkin [7] and it is stated in Lemma 7 and directly follows from the construction of v_h and Lemma 6.

▶ **Lemma 6** (see [7]). *Given a string $a_1a_2\cdots a_m$ over an alphabet $[0..2^u)$ such that $a_i \neq a_{i+1}$ for any $i \in [1..m)$, the string $b_1b_2\cdots b_{m-1}$ such that $b_i = \mathsf{vbit}(a_i, a_{i+1})$, for $i \in [1..m)$, satisfies $b_i \neq b_{i+1}$, for any $i \in [1..m{-}1)$, and $b_i \in [0..2u)$.*

Proof. Consider b_i and b_{i+1}. Denote $\ell = \mathsf{bit}(a_i, a_{i+1})$ and $\ell' = \mathsf{bit}(a_{i+1}, a_{i+2})$. As $a_i, a_{i+1} \in [0..2^u)$, we have $\ell \in [0..u)$. Hence, $b_i \le 2\ell{+}1 \le 2u-1$, which proves $b_i \in [0..2u)$. If $b_i = b_{i+1}$, then $\ell = \ell'$ and the bits with indices ℓ and $\ell' = \ell$ in a_i and a_{i+1} coincide; however, by the definition of $\ell = \mathsf{bit}(a_i, a_{i+1})$, a_i and a_{i+1} must differ in this bit, which is a contradiction. ◀

▶ **Lemma 7** (see [7]). *For any $v_h \neq \infty$ in the kth phase, we have $v_h \in [0..2\log^{(3)} n{+}3)$.*

Proof. Since $v'_h \in [0..2nw) = [0..\mathcal{O}(n\log n))$, we deduce from Lemma 6 that $v''_h \in [0..\mathcal{O}(\log n))$, $v'''_h \in [0..2\log\log n{+}\mathcal{O}(1))$, and, due to the inequality $\log(x{+}\delta) \le \log x{+}\frac{\delta \log e}{x}$, we finally obtain $v_h \in [0..2\log^{(3)} n + 3)$, for sufficiently large n. ◀

The refinement rule implies that, for contiguous regions $j_p, j_{p+1}, \dots, j_q$ where $R(j_h)$ holds, only j_p and j_q may be in S_k and the period of $s[j_p..j_q + 2^k]$ is $\le 2^{k-1}$; for "dense" contiguous regions $j_p, j_{p+1}, \dots, j_q$ where $R(j_h)$ does not hold, Lemma 6 ensure frequent local minima. This is summarized in Lemmas 8, 9 (the proofs are in Appendix B.1 in the full version [27]).

▶ **Lemma 8.** *Let $j_p, j_{p+1}, \dots, j_q$ be a maximal contiguous region of $j_1, j_2, \dots$ such that, for all $h \in [p..q]$, $R(j_h)$ holds. Then, we have $j_q \in S_k$. Further, if $q - p \ge 2$ or $j_p - j_{p-1} > 2^{k-1}$, we have $j_p \in S_k$. All other positions j_h in the region do not belong to S_k. The string $s[j_p..j_q{+}2^k]$ has a period at most 2^{k-1}.*

▶ **Lemma 9.** *Let $j_p, j_{p+1}, \dots, j_q$ be a maximal contiguous region of $j_1, j_2, \dots$ such that, for all $h \in [p..q]$, $R(j_h)$ does not hold and, for $h \in [p..q)$, we have $j_{h+1} - j_h \le 2^{k-1}$. Then, $v_h \neq \infty$ for $h \in [p..q{-}4]$, $v_h = \infty$ for $h \in (q{-}3..q]$, and v_{q-3} may be ∞ or not. Further, for $h \in [p..q{-}3]$, we have $v_h \neq v_{h+1}$ whenever $v_h \neq \infty$. For $h \in (p..q)$, $j_h \in S_k$ iff $\infty > v_{h-1} > v_h$ and $v_h < v_{h+1}$; $j_p \in S_k$ iff $j_p - j_{p-1} > 2^{k-1}$; $j_q \in S_k$ iff $j_{q+1} - j_q > 2^{k-1}$.*

By Lemmas 9 and 7, any sequence of $8\log^{(3)} n + 12$ numbers v_h all of which are not ∞ contains a local minimum v_h and j_h will be put in S_k. Thus, we obtain the following lemma.

▶ **Lemma 10.** *Let S_{k-1} and S_k be the sets generated by the $(k{-}1)$th and kth phases. Then, any range $j_\ell, j_{\ell+1}, \dots, j_m$ of at least $8\log^{(3)} n + 12$ consecutive positions from S_{k-1} such that $v_h \neq \infty$, for all $h \in [\ell..m]$, has a position from S_k.*

The following intuitive lemma is very non-trivial; see Appendix B.2 in the full version [27].

▶ **Lemma 11.** *For any $i, i' \in [0..n]$, $|S_k \cap [i..i')| \le 2^6 \lceil (i'-i)/2^k \rceil$; in particular, $|S_k| \le n/2^{k-6}$.*

Now we are able to prove that S_k is a $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning set and, moreover, it is almost a 2^{k+3}-partitioning set, in a sense. The proof technique is very similar to the one in [5]; for brevity, we defer its detailed proof to Appendix B.2 in the full version [27].

▶ **Lemma 12.** *The kth phase generates a $(2^{k+3}\lfloor \log^{(3)} n \rfloor)$-partitioning set S_k. Moreover, S_k is almost 2^{k+3}-partitioning: for $\tau = 2^{k+3}$, it satisfies properties (a) and (b) but not (c), i.e., if $(i..j) \cap S_k = \emptyset$, for $i, j \in S_k$ such that $2^{k+3} < j-i \le 2^{k+3}\lfloor \log^{(3)} n \rfloor$, then $s[i..j]$ does not necessarily have period $\le 2^{k+2}$.*

4 Speeding up the Refinement Procedure

Since, for any k, $|S_k| \le n/2^{k-6}$ by Lemma 11, it is evident that the algorithm of Section 3 takes $\mathcal{O}(|S_0| + |S_1| + \cdots) = \mathcal{O}(n)$ time plus the time needed to calculate the numbers v'_h, for all positions (from which the numbers v_h are derived). For a given $k \ge 1$, denote by j_h the hth position in S_{k-1}. For each j_h, the number v'_h can be computed by checking whether $j_{h+1} - j_h > 2^{k-1}$ (in this case $v'_h = \infty$), and, if $j_{h+1} - j_h \le 2^{k-1}$, by the naïve comparison of $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ in $\mathcal{O}(2^k)$ time. Thus, all numbers v'_h for the set S_{k-1} can be computed in $\mathcal{O}(2^k|S_{k-1}|) = \mathcal{O}(n)$ time, which leads to $\mathcal{O}(n \log \tau)$ total time for the whole algorithm. This naïve approach can be sped up if one can perform the LCE queries that compare $s[j_h..j_h{+}2^k]$ and $s[j_{h+1}..j_{h+1}{+}2^k]$ faster; in fact, if one can do this in $\mathcal{O}(1)$ time, the overall time becomes linear. To this end, we exploit the online nature of the procedure. Let us briefly outline the procedure again on a high level.

The algorithm runs simultaneously $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases: the kth phase takes positions from the set S_{k-1} produced by the $(k{-}1)$th phase and decides which of them to feed to the $(k{+}1)$th phase, i.e., to put into S_k (the “top” phase feeds the positions to an external procedure described in the next section). To make the decision for $j_h \in S_{k-1}$, the kth phase needs to know the distance $j_h{-}j_{h-1}$ and the distances $j_{h+\ell}{-}j_h$ to the positions $j_{h+\ell}$ with $\ell \in [1..5]$ such that $j_{h+\ell}{-}j_h \le 5 \cdot 2^{k-1}$. Then, the kth phase calculates $\min\{2^k{+}1, \mathsf{lce}(j_{h+\ell-1}, j_{h+\ell})\}$, for all $\ell \in [0..5]$ such that $j_{h+\ell}{-}j_{h+\ell-1} \le 2^{k-1}$ and $j_{h+\ell}{-}j_h \le 5 \cdot 2^{k-1}$, and, based on the distances and the LCE values, computes v_{h-1}, v_h, v_{h+1} and decides the fate of j_h.

The key for our optimization is the locality of the decision making in the phases that is straightforward for the described process: for any prefix $s[0..d]$, once the positions $S_{k-1} \cap [0..d]$ are known to the kth phase, it reports all positions from the set $S_k \cap [0..d{-}5 \cdot 2^{k-1}]$ and no position from the set $S_{k-1} \cap [0..d{-}6 \cdot 2^{k-1}]$ will be accessed by an LCE query of the kth phase in the future. Thus, we can discard all positions $S_{k-1} \cap [0..d{-}6 \cdot 2^{k-1}]$ and have to focus only on positions $S_{k-1} \cap (d{-}6 \cdot 2^{k-1}..\infty]$ and LCE queries on them in the future. We deduce from this that after processing the prefix $s[0..d]$ by the whole algorithm, the kth phase reports all positions from the set $S_k \cap [0..d{-}5\sum_{k'=0}^{k-1} 2^{k'}] \supseteq S_k \cap [0..d{-}5 \cdot 2^k]$ and no LCE query in the kth phase accesses positions from the set $S_{k-1} \cap [0..d{-}6 \cdot 2^k]$ in the future.

This locality of the decision procedure guarantees that, at the time we processed a length-ℓ prefix of the string s, for some $\ell \ge 0$, all positions from the set $S_k \cap [0..\ell{-}5 \cdot 2^k]$ are reported and no position from the set $S_{k-1} \cap [0..\ell{-}5 \cdot 2^k]$ will be accessed by an LCE query of the kth phase in the future. Let us summarize this as follows.

▶ **Lemma 13.** *Suppose we run the described $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases on a string s of length n from left to right. Then, for any $k \ge 1$ and $d \ge 0$, after processing the prefix $s[0..d]$, the kth phase reports all positions from $S_k \cap [0..d{-}5 \cdot 2^k]$ to the $(k+1)$th phase and will not perform queries $\mathsf{lce}(j, j')$ on positions $j, j' \in S_{k-1}$ such that $\min\{j, j'\} \le d - 6 \cdot 2^k$ in the future.*

Recall that we have $\mathcal{O}(\log \tau)$ phases and at least $\Omega(\log \tau)$ space. Let us sketch main techniques to speed up the algorithm. Details are given in Appendix C in the full version [27].

Suppose that $\tau < \sqrt{n}$. We have $b = \Theta(\frac{n}{\tau}) \geq \Omega(\sqrt{n})$ additional space for the algorithm in this case. To answer all required LCE queries in constant time, when the algorithm processes a letter $s[d]$, the classical LCE data structure from [16] is maintained for the leftmost substring $C_i = s[i\lfloor\sqrt{n}\rfloor..(i+3)\lfloor\sqrt{n}\rfloor - 1]$ whose middle part contains the position d (i.e. $d \in (i+1)\lfloor\sqrt{n}\rfloor..(i+2)\lfloor\sqrt{n}\rfloor - 1]$). By Lemma 13, we can use the data structure to correctly handle all queries because all LCE queries performed by the algorithm at the step d lie within the substring C_i. Since we must build the LCE data structure for every C_i once, the overall running time is $\mathcal{O}(n + \sum_i |C_i|) = \mathcal{O}(n)$ and the occupied space is $\mathcal{O}(\sqrt{n}) = \mathcal{O}(b)$.

Let us generalize this idea to the case $\tau \geq \sqrt{n}$. Denote $b = \frac{n}{\tau}$. We have $\mathcal{O}(b) < \mathcal{O}(\sqrt{n})$ space and cannot use the scheme described above since LCE data structures for substrings of length $\mathcal{O}(b)$ are not enough to answer queries of the form $\min\{2^k{+}1, \mathsf{lce}(j, j')\}$ when $2^k > \Omega(b)$. The key idea is to group contiguous phases into "levels" and maintain SST for a sliding window of positions in each level (in the case $\tau < \sqrt{n}$ we had a single "level" and a sliding window of size $\mathcal{O}(\sqrt{n})$). We must choose "level" size to be large enough to build less SSTs and fit in the $\mathcal{O}(n \log_b n)$ running time, but also the "levels" must be small to efficiently reduce the number of positions in each level and fit all supporting data structures in the $\mathcal{O}(b)$ space. To achieve this, we split evenly all $\lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$ phases into "levels", each containing $\Theta(\log \hat{b})$ phases, where $\hat{b} = \lfloor \frac{b}{\log n} \rfloor$. For each "level", we maintain a window of $\mathcal{O}(\hat{b})$ positions from S_k, where k is the lowest phase in the "level"; one window spans a substring of length $\mathcal{O}(2^k \hat{b})$ and the windows change $\mathcal{O}(\frac{n}{2^k \hat{b}})$ times in total. Overall we use $\mathcal{O}(\hat{b} \log \hat{b}) = \mathcal{O}(b)$ space. By Lemma 12, the set S_k is "almost" 2^k-partitioning, so we can build SST for each "level" as in Lemma 3 in time $\mathcal{O}(2^k \hat{b} + \min\{2^k \hat{b} \log_{\hat{b}} n, \hat{b} \log \hat{b}\})$, which simplifies to $\mathcal{O}(\hat{b} \log_{\hat{b}} n)$ for the first "level" and to $\mathcal{O}(2^k \hat{b})$ for subsequent "levels". (Note that there are no vicious circles here: Lemma 3 is self-contained and builds its SST using only the radix sort for strings related to its input partitioning set.) Overall we can upperbound the running time with $\mathcal{O}(n \log_b n)$ for all $\mathcal{O}(\log_b n)$ "levels". Thus, the described routine builds partitioning sets S_k in time $\mathcal{O}(n \log_b n)$ and space $\mathcal{O}(b)$. The described sketch of the algorithm is elaborated in details in Appendix C in the full version [27].

5 Recompression

Let S be the set produced by the last phase of the procedure from Sections 3 and 4. By Lemma 12, S is a $\frac{\tau}{2}$-partitioning set of size $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$. Throughout this section, we assume that $\tau \geq (\log^{(3)} n)^4$ so that the size of S is at most $\mathcal{O}(\frac{n}{(\log^{(3)} n)^3})$; the case $\tau < (\log^{(3)} n)^4$ is discussed in Appendix E in the full version [27]. In what follows we describe an algorithm that removes positions from S transforming it into a τ-partitioning set of size $\mathcal{O}(n/\tau)$.

Instead of storing S explicitly, which is impossible in $\mathcal{O}(n/\tau)$ space, we construct a related-to-S string R of length $\mathcal{O}(\frac{n}{\tau} \log^{(3)} n)$ over a small alphabet such that R can be packed into $\mathcal{O}(n/\tau)$ machine words. Positions of S are represented, in a way, by letters of R. The construction of R is quite intricate, which is necessary in order to guarantee that letters of R corresponding to close positions of S (namely, positions at a distance at most $\tau/2^5$) are necessarily distinct even if the letters are not adjacent in R. This requirement is stronger than the requirement of distinct adjacent letters that was seen, for instance, in Lemma 6 but it is achieved by similar means using vbit reductions as in Section 3. We then apply to R a variant of the iterative process called *recompression* [19] that removes some letters thus shrinking the length of R to $\mathcal{O}(n/\tau)$. Then, the whole procedure of Sections 3–4 that generated S is performed again but this time we discard all positions of S corresponding to removed positions of the string R and store the remaining positions explicitly in a set $S^* \subseteq S$. We show that S^* is τ-partitioning and has size $\mathcal{O}(n/\tau)$. Let us elaborate on the details.

The algorithm starts with an empty string R and receives positions of S from left to right appending to the end of R new letters corresponding to the received positions. It is more convenient to describe the algorithm as if it acted in two stages: the first stage produces a $\frac{3}{4}\tau$-partitioning set $S' \subseteq S$, for which a condition converse to property (c) holds (thus, some positions of S are discarded already in this stage), and the second stage, for each position of S', appends to the end of R a letter of size $\mathcal{O}((\log^{(3)} n)^2)$ bits. Both stages act in an almost online fashion and, hence, can be actually executed simultaneously in one pass without the need to store the auxiliary set S'. The separation is just for the ease of the exposition.

The first stage. The goal is to construct set $S' \subseteq S$ by excluding from S all positions h for which there exist $i, j \in S$ such that $i < h \le j$, $j - i \le \tau/4$, and $s[i..i{+}\tau/2] = s[j..j{+}\tau/2]$. The algorithm generating S' is as follows.

We consider all positions of S from left to right and, for each $i \in S$, process every $j \in (i..i{+}\tau/4] \cap S$ by comparing $s[i..i{+}\tau/2]$ with $s[j..j{+}\tau/2]$. If $s[i..i{+}\tau/2] = s[j..j{+}\tau/2]$, then we traverse all positions of the set $(i..j] \cap S$ from right to left marking them for removal until an already marked position is encountered. Since the marking procedure works from right to left, every position is marked at most once. The position i is put into S' iff it was not marked previously. During the whole process, we maintain a "look-ahead" queue that stores the positions $(i..i{+}\tau/4] \cap S$ and indicates which of them were marked for removal.

Due to Lemma 11, the size of the set $(i..i{+}\tau/4] \cap S$ is $\mathcal{O}(\log^{(3)} n)$. Therefore, the look-ahead queue takes $\mathcal{O}(\log^{(3)} n)$ space, which is $\mathcal{O}(n/\tau)$ since $n/\tau \ge \log^2 n$, and $\mathcal{O}(\log^{(3)} n)$ comparisons are performed for each i. Hence, if every comparison takes $\mathcal{O}(1)$ time, the set S' is constructed in $\mathcal{O}(|S| \log^{(3)} n) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^2)$ time, which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$. Thus, it remains to explain how the comparisons can be performed.

Similar to the algorithm of Section 4, we consecutively consider substrings $C'_i = s[i\tau..(i+3)\tau)$, for $i \in [0..n/\tau{-}3]$: when all positions from a set $S \cap [i\tau..(i{+}3)\tau)$ are collected, we use the algorithm of Lemma 3 to build a SST for all suffixes of the string C'_i whose starting positions are from S; the tree, endowed with an LCA data structure [16], is used in the procedure for deciding which of the positions from the set $S \cap [(i+\frac{1}{2})\tau..(i+\frac{3}{2})\tau)$ (or $S \cap [0..\frac{3}{2}\tau)$ if $i = 0$) should be marked for removal. Thus, after processing the last string C'_i, all positions of S are processed and S' is generated. By Lemma 11, the number of suffixes in the SST for C'_i is $\mathcal{O}(\log^{(3)} n)$ and, therefore, the tree occupies $\mathcal{O}(\log^{(3)} n) \le \mathcal{O}(n/\tau)$ space and its construction takes $\mathcal{O}(\tau + \log^{(3)} n \cdot \log\log^{(3)} n)$ time by Lemma 3, which is $\mathcal{O}(\tau)$ since $\tau \ge (\log^{(3)} n)^4$. Thus, the total construction time for all the trees in the stage is $\mathcal{O}(\frac{n}{\tau}\tau) = \mathcal{O}(n)$ and the space used is $\mathcal{O}(\log^{(3)} n)$ since, at every moment, at most one tree is maintained.

The following lemma shows that the transformation within the first stage does not break τ-partitioning properties. Its proof is deferred to Appendix D.1 in the full version [27].

▶ **Lemma 14.** *The set S' is τ-partitioning and satisfies a converse of property (c): if a substring $s[i..j]$ has a period at most $\tau/4$, then $S' \cap [i{+}\frac{3}{4}\tau..j{-}\frac{3}{4}\tau] = \emptyset$. Moreover, S' is almost $\frac{3}{4}\tau$-partitioning, meeting properties (a) and (b) with $\frac{3}{4}\tau$ in place of τ, but not necessarily (c).*

The second stage. We consider all positions of S' from left to right and, for each $p \in S'$, append to the end of the (initially empty) string R a new carefully constructed letter a_p occupying $\mathcal{O}((\log^{(3)} n)^2)$ bits. Thus, the string R will have length $|S'|$ and will take

$\mathcal{O}(|S'|(\log^{(3)} n)^2) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^3)$ bits of space, which can be stored into $\mathcal{O}(n/\tau)$ machine words of size $\mathcal{O}(\log n)$ bits. The crucial property of R for us is that any two letters of R corresponding to close positions of S' are distinct, namely the following lemma will be proved:

▶ **Lemma 15.** *For any $p, \bar{p} \in S'$, if $0 < \bar{p} - p \le \tau/2^5$, then $a_p \ne a_{\bar{p}}$.*

Consider $p \in S'$. We are to describe an algorithm generating an $\mathcal{O}((\log^{(3)} n)^2)$-bit letter a_p for p that will be appended to the string R.

Denote by $p_1, p_2, \ldots, p_m$ all positions of $S' \cap (p..p{+}\tau/2^5]$ in the increasing order. By Lemma 11, $m \le \mathcal{O}(\log^{(3)} n)$ and, hence, there is enough space to store them. By construction, $s[p..p{+}\frac{\tau}{2}] \ne s[p_j..p_j{+}\frac{\tau}{2}]$, for each $j \in [1..m]$. One can compute the longest common prefix of $s[p..p{+}\frac{\tau}{2}]$ and $s[p_j..p_j{+}\frac{\tau}{2}]$, for any $j \in [1..m]$, in $\mathcal{O}(1)$ time using a SST with an LCA data structure [16] built in the first stage for a substring $C'_i = s[i\tau..(i+3)\tau - 1]$ such that $p \in [i\tau..(i+\frac{3}{2})\tau)$. (In order to have $p_1, p_2, \ldots, p_m$ prepared, we handle p, which was reported by the first stage after processing C'_i, only when C'_{i+1} was processed too; thus, the first stage maintains two SSTs: one for a substring C'_{i+1} currently under analysis and one for C'_i, retained for its use in the second stage.)

Denote $\ell = 2^6\lceil \log^{(3)} n\rceil$. Recall that S is produced by the kth phase of the procedure of Section 3, for $k = \lfloor \log \frac{\tau}{2^4 \log^{(3)} n}\rfloor$, and hence, by Lemma 11, the size of any set $S \cap [i..j]$, for $i \le j$, is at most $2^6\lceil (j-i+1)/2^k\rceil$. Therefore, since $S' \subseteq S$ and m is the size of the set $S' \cap (p..p{+}\tau/2^5]$, we obtain $m \le 2^6(\tau/2^5)/\frac{\tau}{2\cdot 2^4 \log^{(3)} n} \le \ell$.

Let w be the number of bits in an $\mathcal{O}(\log n)$-bit machine word sufficient to represent letters from the alphabet $[0..n^{\mathcal{O}(1)}]$ of s. For each p_j, denote $t_j = \sum_{i=0}^{\tau/2} s[p_j{+}i]2^{wi}$; similarly, for p, denote $t = \sum_{i=0}^{\tau/2} s[p{+}i]2^{wi}$. As in an analogous discussion in Section 3, we do not discern the numbers t_j and t from their corresponding substrings in s and use them merely in the analysis. The intuition behind our construction is that the numbers $t, t_1, t_2, \ldots, t_m$, in principle, could have been used for the string R as letters corresponding to the positions $p, p_1, p_2, \ldots, p_m$ since $t, t_1, t_2, \ldots, t_m$ are pairwise distinct (due to the definition of S') but, unfortunately, they occupy too much space ($\mathcal{O}(w\tau)$ bits each). One has to reduce the space for the letters retaining the property of distinctness. The tool capable to achieve this was already developed in Section 3: it is the vbit reduction, a trick from Cole and Vishkin's deterministic locally consistent parsing [7].

We first generate for p a tuple of ℓ numbers $\langle w'_1, w'_2, \ldots, w'_\ell\rangle$: for $j \in [1..\ell]$, $w'_j = \mathsf{vbit}(t, t_j)$ if $j \le m$, and $w'_j = \infty$ otherwise. Since the longest common prefix of substrings $s[p..p{+}\frac{\tau}{2}]$ and $s[p_j..p_j{+}\frac{\tau}{2}]$, for $j \in [1..m]$, can be calculated in $\mathcal{O}(1)$ time, the computation of the tuple takes $\mathcal{O}(\ell) = \mathcal{O}(\log^{(3)} n)$ time. By Lemma 6, each number w'_j occupies less than $\lceil \log w + \log \tau + 1\rceil$ bits. Thus, we can pack the whole tuple into $\ell\lceil \log w + \log \tau + 1\rceil$ bits encoding each value w'_j into $\lceil \log w + \log \tau + 1\rceil$ bits and representing ∞ by setting all bits to 1. We denote this chunk of $\ell\lceil \log w + \log \tau + 1\rceil$ bits by $\bar{t}$. In the same way, for each p_i with $i \in [1..m]$, we generate a tuple $\langle w'_{i,1}, w'_{i,2}, \ldots, w'_{i,\ell}\rangle$ comparing $s[p_i..p_i{+}\tau/2]$ to $s[q..q{+}\tau/2]$, for each $q \in S' \cap (p_i..p_i{+}\tau/2^5]$, and using the vbit reduction; the tuple is packed into a chunk $\bar{t}_i$ of $\ell\lceil \log w + \log \tau + 1\rceil$ bits. See Figure 2. For each $j \in [1..m]$, the number w'_j is not equal to ∞ and, thus, due to Lemma 6, differs from the number $w'_{j,j}$ (the jth element of the tuple $\langle w'_{i,1}, w'_{i,2}, \ldots, w'_{i,\ell}\rangle$). Therefore, all the tuples – and, hence, their corresponding numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \ldots, \bar{t}_m$ – are pairwise distinct.

The numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \ldots, \bar{t}_m$, like the numbers $t, t_1, t_2, \ldots, t_m$, could have been used, in principle, as letters for the string R but they still are too large. We therefore repeat the same vbit reduction but now for the numbers $\bar{t}, \bar{t}_1, \bar{t}_2, \ldots, \bar{t}_m$ in place of $t, t_1, t_2, \ldots, t_m$ thus generating a tuple $\langle w''_1, w''_2, \ldots, w''_\ell\rangle$: for $j \in [1..\ell]$, $w''_j = \mathsf{vbit}(\bar{t}, \bar{t}_j)$ if $j \le m$, and $w''_j = \infty$

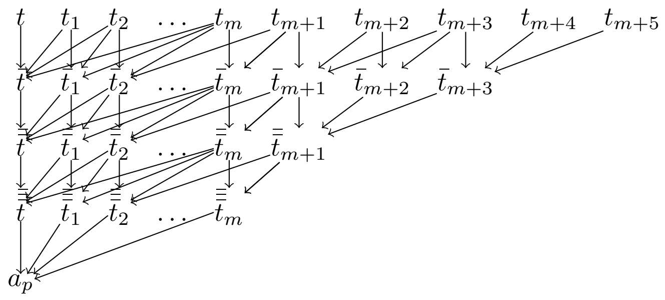

t t_1 t_2 $\dots$ t_m t_{m+1} t_{m+2} t_{m+3} t_{m+4} t_{m+5}
$\bar{t}$ $\bar{t}_1$ $\bar{t}_2$ $\dots$ $\bar{t}_m$ $\bar{t}_{m+1}$ $\bar{t}_{m+2}$ $\bar{t}_{m+3}$
$\bar{\bar{t}}$ $\bar{\bar{t}}_1$ $\bar{\bar{t}}_2$ $\dots$ $\bar{\bar{t}}_m$ $\bar{\bar{t}}_{m+1}$
$\bar{\bar{\bar{t}}}$ $\bar{\bar{\bar{t}}}_1$ $\bar{\bar{\bar{t}}}_2$ $\dots$ $\bar{\bar{\bar{t}}}_m$
a_p

Figure 2 The scheme generating a_p via vbit reductions. If a node $\hat{t}$ has ingoing edges labeled with $\tilde{t}, \tilde{t}_1, \tilde{t}_2, \dots, \tilde{t}_r$ (from left to right), then $\hat{t}$ encodes a tuple $\langle \tilde{w}_1, \tilde{w}_2, \dots, \tilde{w}_\ell \rangle$ such that, for $j \in [1..r]$, $\tilde{w}_j = \mathsf{vbit}(\tilde{t}, \tilde{t}_j)$ and, for $j \in (r..\ell]$, $\tilde{w}_j = \infty$. In the figure, the numbers $t, t_1, t_2, \dots, t_{m+5}$ correspond to consecutive positions $p, p_1, p_2, \dots, p_{m+5}$ in the set S', respectively. By looking at which of the ingoing edges are present and which are not, one can deduce that here we have $S' \cap (p..p{+}\tau/2^5] = \{p_1, \dots, p_m\}$, $S' \cap (p_1..p_1{+}\tau/2^5] = \{p_2, \dots, p_m\}$, $S' \cap (p_2..p_2{+}\tau/2^5] = \{p_3, \dots, p_m, p_{m+1}\}$, $S' \cap (p_m..p_m{+}\tau/2^5] = \{p_{m+1}\}$, $S' \cap (p_{m+1}..p_{m+1}{+}\tau/2^5] = \{p_{m+2}, p_{m+3}\}$, $S' \cap (p_{m+2}..p_{m+2}{+}\tau/2^5] = \{p_{m+3}\}$, $S' \cap (p_{m+3}..p_{m+3}{+}\tau/2^5] = \{p_{m+4}, p_{m+5}\}$.

otherwise. The computation of $\mathsf{vbit}(\bar{t}, \bar{t}_j)$ takes $\mathcal{O}(\ell)$ time since $\bar{t}$ occupies ℓ machine words of size $\mathcal{O}(\log n)$ bits. It follows from Lemma 6 that the tuple $\langle w''_1, w''_2, \dots, w''_\ell \rangle$ can be packed into a chunk $\bar{\bar{t}}$ of $\ell\lceil \log \ell + \log\lceil \log w + \log \tau + 1\rceil + 1\rceil$ bits (i.e., $\mathcal{O}(\log^{(3)} n \cdot \log\log n)$ bits), which already fits into one machine word. We perform analogous reductions for the positions $p_1, p_2, \dots, p_m$ generating m tuples $\langle w''_{i,1}, w''_{i,2}, \dots, w''_{i,\ell} \rangle$, for $i \in [1..m]$, packed into new chunks $\bar{\bar{t}}_1, \bar{\bar{t}}_2, \dots, \bar{\bar{t}}_m$, respectively. Note that, in order to produce a tuple $\langle w''_{i,1}, w''_{i,2}, \dots, w''_{i,\ell} \rangle$, for $i \in [1..m]$, that is packed into $\bar{\bar{t}}_i$, we use not only the numbers $\bar{t}_i, \bar{t}_{i+1}, \dots, \bar{t}_m$ corresponding to positions $p_i, p_{i+1}, \dots, p_m$ but also similarly computed numbers at other positions from $S' \cap (p_i..p_i{+}\tau/2^5]$, if any. See Figure 2 for a clarification: it can be seen that the "top" numbers include not only $t, t_1, \dots, t_m$ precisely because of this.

By the same argument that proved the distinctness of $\bar{t}, \bar{t}_1, \bar{t}_2, \dots, \bar{t}_m$, one can easily show that $\bar{\bar{t}}, \bar{\bar{t}}_1, \bar{\bar{t}}_2, \dots, \bar{\bar{t}}_m$ are pairwise distinct. But they are still too large to be used as letters of R. Then again, we repeat the same reductions at positions $p, p_1, p_2, \dots, p_m$ but now for the numbers $\bar{\bar{t}}, \bar{\bar{t}}_1, \bar{\bar{t}}_2, \dots, \bar{\bar{t}}_m$ in place of $\bar{t}, \bar{t}_1, \bar{t}_2, \dots, \bar{t}_m$, thus generating new chunks $\bar{\bar{\bar{t}}}, \bar{\bar{\bar{t}}}_1, \bar{\bar{\bar{t}}}_2, \dots, \bar{\bar{\bar{t}}}_m$. Finally, once more, we do the vbit reduction for $\bar{\bar{\bar{t}}}, \bar{\bar{\bar{t}}}_1, \bar{\bar{\bar{t}}}_2, \dots, \bar{\bar{\bar{t}}}_m$ generating a tuple $\langle w_1, w_2, \dots, w_\ell \rangle$ such that, for $j \in [1..\ell]$, w_j is $\mathsf{vbit}(\bar{\bar{\bar{t}}}, \bar{\bar{\bar{t}}}_j)$ if $j \le m$, and ∞ otherwise.

Using the same reasoning as in the proof of Lemma 7, one can deduce from Lemma 6 that the tuple $\langle w_1, w_2, \dots, w_\ell \rangle$ fits into a chunk of $\ell \cdot 2 \log\log^{(3)} n \le 2^6 \lceil \log^{(3)} n \rceil^2$ bits (the inequality holds provided $n > 2^{16}$) encoding each value w_j into $\lceil \log^{(3)} n \rceil$ bits and representing ∞ by setting all $\lceil \log^{(3)} n \rceil$ bits to 1. Denote by a_p this chunk of $2^6 \lceil \log^{(3)} n \rceil^2$ bits that encodes the tuple. We treat a_p as a new letter of R that corresponds to the position p and we append a_p to the end of R. Lemma 15 follows then straightforwardly by construction.

Given $p \in S'$, the calculation of the numbers $\bar{t}, \bar{\bar{\bar{t}}}, a_p$ takes $\mathcal{O}(\ell^2)$ time. The calculation of $\bar{\bar{t}}$ requires $\mathcal{O}(\ell^3)$ time since each reduction $\mathsf{vbit}(\bar{t}, \bar{t}_j)$ for it takes $\mathcal{O}(\ell)$ time. Hence, the total time for the construction of R is $\mathcal{O}(|S'|\ell^3) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^4)$, which is $\mathcal{O}(n)$ as $\tau \ge (\log^{(3)} n)^4$.

Recompression. If the distance between any pair of adjacent positions of S' is at least $\tau/2^6$, then $|S'| \le 2^6 n/\tau$ and, by Lemma 14, S' can be used as the resulting τ-partitioning set of size $\mathcal{O}(n/\tau)$. Unfortunately, in general, this is not the case and we have to "sparsify" S'.

There is a one-to-one correspondence between S' and positions of R. Using a technique of Jeż [19] called *recompression* , we can remove in $\mathcal{O}(|R|)$ time some letters of R reducing by a fraction $\frac{4}{3}$ the number of pairs of adjacent letters $R[i], R[i{+}1]$ whose corresponding positions in S' are at a distance at most $\tau/2^6$. We perform such reductions until the length of R becomes at most $2^{14} \cdot n/\tau$. The positions of S' corresponding to remaining letters will constitute a τ-partitioning set of size $\mathcal{O}(n/\tau)$. In order to guarantee that this subset of S' is τ-partitioning, we have to execute the recompression reductions gradually increasing the distances that are of interest for us: first, we get rid of adjacent pairs with distances at most $\tau/\log^{(3)} n$ between them, then the threshold is increased to $2\tau/\log^{(3)} n$, then $2^2\tau/\log^{(3)} n$, and so on until (most) adjacent pairs with distances at most $2^{\log^{(4)} n - 6}\tau/\log^{(3)} n = \tau/2^6$ between them are removed in last recompression reductions. The details follow.

Since it is impossible to store in $\mathcal{O}(n/\tau)$ space the precise distances between positions of S', the information about distances needed for recompression is encoded as follows. For each $i \in [0..|R|)$ and a position $p \in S'$ corresponding to the letter $R[i]$, we store an array of numbers $M_i[0..\lceil\log^{(4)} n\rceil]$ such that, for $j \in [0..\lceil\log^{(4)} n\rceil]$, $M_i[j]$ is equal to the size of the set $S' \cap (p..p{+}\tau/2^j]$. By Lemma 11, we have $|S' \cap (p..p{+}\tau]| \le \mathcal{O}(\log^{(3)} n)$ and, hence, each number $M_i[j]$ occupies $\mathcal{O}(\log^{(4)} n)$ bits. Therefore, all the arrays M_i can be stored in $\mathcal{O}(|R|(\log^{(4)} n)^2) \le \mathcal{O}(\frac{n}{\tau}\log^{(3)} n \cdot (\log^{(4)} n)^2)$ bits, which fits into $\mathcal{O}(\frac{n}{\tau})$ machine words of size $\mathcal{O}(\log n)$ bits. All arrays M_i are constructed in a straightforward way in $\mathcal{O}(|R|\log^{(3)} n) = \mathcal{O}(\frac{n}{\tau}(\log^{(3)} n)^2)$ time (which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$) during the left-to-right pass over S' that generated the string R.

Our algorithm consecutively considers all numbers $j \in [6..\lceil\log^{(4)} n\rceil]$ in decreasing order, starting from $j = \lceil\log^{(4)} n\rceil$. For each j, it iteratively performs a recompression procedure reducing the number of adjacent letters $R[i], R[i{+}1]$ whose corresponding positions from S' are at a distance at most $\tau/2^j$, until R shrinks to a length at most $2^{j+10} \cdot \frac{n}{\tau}$. Thus, $|R| \le 2^{16} \cdot \frac{n}{\tau}$ after last recompression reductions for $j = 6$. Let us describe the recompression procedure.

Fix $j \in [6..\lceil\log^{(4)} n\rceil]$. To preserve property (c) of the τ-partitioning set S' during the sparsifications, we impose an additional restriction: a letter $R[i]$ cannot be removed if either $i = 0$ or the distance between the position $p \in S'$ corresponding to $R[i]$ and the predecessor of p in S' is larger than $\tau/2^5$, i.e., if $M_{i-1}[5] = 0$. The rationale is as follows: the position p might be the right boundary of a gap in S' of length $> \tau$ and it is dangerous to break the gap since, once p is removed, the gap might not satisfy property (c) (the range of the string s corresponding to the gap should have a period that is at most $\tau/4$).

The processing of the number j starts with checking whether $|R| \le 2^{j+10} \cdot \frac{n}{\tau}$. If so, we skip the processing of j and move to $j - 1$ (provided $j > 6$). Suppose that $|R| > 2^{j+10} \cdot \frac{n}{\tau}$. Denote $\sigma = 2^{2^6\lceil\log^{(3)} n\rceil^2}$, the size of the alphabet $[0..\sigma)$ of R. Then, the algorithm creates an array $P[0..\sigma{-}1][0..\sigma{-}1]$ filled with zeros, which occupies $\mathcal{O}(\sigma^2) = \mathcal{O}(2^{2^7(\log^{(3)} n)^2}) = o(\log n)$ space, and collects in P statistics on pairs of adjacent letters of R whose corresponding positions in S' are at a distance at most $\tau/2^j$ and whose first letter may be removed: namely, we traverse all $i \in [1..|R|)$ and, if $M_i[j] \ne 0$ and $M_{i-1}[5] \ne 0$, then we increase by one the number $P[R[i]][R[i{+}1]]$. By Lemma 15, $R[i] \ne R[i+1]$ when $M_i[j] \ne 0$.

The core tool of the recompression technique proposed by Jeż [19] is an algorithm for multidigraph without self-loops $G = (V, E)$ that constructs a directed cut of size at least $\lceil\frac{|E|}{4}\rceil$ edges in time $\mathcal{O}(|V|^2)$ if the graph is given by an adjacency matrix. If we interpret P as an adjacency matrix, we can use Jeż's technique (there are no self-loops because $R[i] \ne R[i+1]$ when $M_i[j] \ne 0$ due to Lemma 15) and split the alphabet into two disjoint subsets correspoding to the cut: $[0..\sigma) = \acute{\Sigma} \sqcup \grave{\Sigma}$. After that we mark for removal from R

all indices $i \in [1..|R|-1)$ for which the following conditions hold: $M_i[j] \neq 0$, $M_{i-1}[5] \neq 0$, $R[i] \in \acute{\Sigma}$, and $R[i{+}1] \in \grave{\Sigma}$. Once the sets $\acute{\Sigma}$ and $\grave{\Sigma}$ are computed in time $\mathcal{O}(\sigma^2) = o(\log^2 n)$, the marking takes $\mathcal{O}(|R|)$ time and can be organized using a bit array of length $|R|$.

After the marking step we update values in all arrays M_i according to removal marks in one right to left pass: for each $i \in [0..|R|)$ and $j' \in [0..\lceil\log^{(4)} n\rceil]$, the new value for $M_i[j']$ is the number of indices $i+1, i+2, \ldots, i+M_i[j']$ that were not marked for removal, i.e., $M_i[j']$ is the number of positions in the set $S' \cap (p..p{+}\tau/2^{j'}]$ whose corresponding letters $R[i']$ will remain in R, where $p \in S'$ is the position corresponding to $R[i]$. Since $M_i[j'] \leq M_{i+1}[j'] + 1$, for $i \in [0..|R|-1)$, the pass updating M can be executed in $\mathcal{O}(|R| \log^{(4)} n)$ time.

Finally, we delete letters $R[i]$ and arrays M_i, for all indices i marked for removal, thus shrinking the length of R and the storage for M_i. We call this procedure, which marks letters of R and removes them and their corresponding arrays M_i, the recompression. One recompression iteration takes $\mathcal{O}(|R| \log^{(4)} n)$ time, where $|R|$ is the length of R before shrinking.

The next lemma states that the recompression shrinks the string R by a constant factor.

▶ **Lemma 16.** *If, for $j \in [6..\lceil\log^{(4)} n\rceil]$, before the recompression procedure there were d non-zero numbers $M_i[j]$ with $i \in [1..|R|)$ such that $M_{i-1}[5] \neq 0$, then the arrays M_i modified by the procedure, for all i corresponding to unremoved positions of R, contain at most $\frac{3}{4}d$ non-zero numbers $M_i[j]$ such that $M_{i-1}[5]{\neq}0$.*

Proof. The proof repeats an argument from [19] and [17, Lemma 7]. Consider an undirected weighted graph G corresponding to the digraph encoded in the adjacency matrix P. By construction of P, we have $d = \sum_{a \neq b} P[a][b]$, which follows from Lemma 15 that guarantees $R[i] \neq R[i+1]$ when $M_i[j] \neq 0$. Thus, d is the sum of weights of all edges in G. Putting a letter a into either $\acute{\Sigma}$ or $\grave{\Sigma}$, we add to the cut at least half of the total weight of all edges connecting a to the letters $0, 1, \ldots, a{-}1$. Therefore, the cut of G induced by $\acute{\Sigma}$ and $\grave{\Sigma}$ has a weight at least $\frac{1}{2}d$. The edges in the cut might be directed both from $\acute{\Sigma}$ to $\grave{\Sigma}$ and in the other direction. Switching $\acute{\Sigma}$ and $\grave{\Sigma}$, if needed, we ensure that the direction from $\acute{\Sigma}$ to $\grave{\Sigma}$ has a maximal total weight, which is obviously at least $\frac{1}{4}d$. According to this cut, we mark for removal from R at least $\frac{1}{4}d$ letters $R[i]$ such that $M_i[j] \neq 0$. Hence, the number of non-zero values $M_i[j]$ such that $M_{i-1}[5] \neq 0$ is reduced by $\frac{1}{4}d$, which gives the result of the lemma since new non-zero values could not appear after the deletions. ◀

Suppose, for a fixed $j \in [6..\lceil\log^{(4)} n\rceil]$, the algorithm has performed one iteration of the recompression. Denote by S'' the set of all positions from S' that "survived" the recompression for $j \in [6..\lceil\log^{(4)} n\rceil]$ and, thus, have a corresponding letter in the updated string R. There is a one-to-one correspondence between S'' and letters of R. For each $i \in [0..|R|)$ and $j' \in [0..\lceil\log^{(4)} n\rceil]$, the number $M_i[j']$ in the modified arrays M_i is the size of the set $S'' \cap (p..p{+}\tau/2^{j'}]$, for a position $p \in S''$ corresponding to i. We therefore can again apply the recompression procedure thus further shrinking the length of R. The algorithm first again checks whether $|R| > 2^{j+10} \cdot \frac{n}{\tau}$ and, if so, repeats the recompression. For the given fixed j, we do this iteratively until $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$. During this process, the number of zero values $M_i[j]$ in the arrays M_i is always at most $2^j \cdot \frac{n}{\tau}$ since the equality $M_i[j] = 0$ implies that $S''' \cap (p..p{+}\tau/2^j] = \emptyset$, for a set $S''' \subseteq S'$ of size $|R|$ defined by analogy to the definition of S'' and for a position $p \in S'''$ corresponding to i. Therefore, due to Lemma 16, the condition $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$ eventually should be satisfied. Furthermore, as we are to show, for each j, the condition $|R| \leq 2^{j+10} \cdot \frac{n}{\tau}$ holds after at most three recompression iterations.

Given $j \in [6..\lceil\log^{(4)} n\rceil)$, the length of R before the first iteration of the recompression for j is at most $2^{j+11} \cdot \frac{n}{\tau}$ since this is a condition under which shrinking iterations stopped for $j+1$. The same bound holds for $j = \lceil\log^{(4)} n\rceil$: the initial length of R is at most $2^{11} \cdot \frac{n}{\tau} \log^{(3)} n$ (which is upper-bounded by $2^{j+11} \cdot \frac{n}{\tau}$) since $S' \subseteq S$ and S is produced by the kth phase of the procedure of Section 3, for $k = \lfloor \log \frac{\tau}{2^4 \log^{(3)} n} \rfloor$, so that the size of S, by Lemma 11, is at most $2^6\lceil n/2^k\rceil \le 2^6 n / \frac{\tau}{2 \cdot 2^4 \log^{(3)} n} = 2^{11}\frac{n}{\tau}\log^{(3)} n$. Fix $j \in [6..\lceil\log^{(4)} n\rceil]$. Since the number of zero values $M_i[j]$ is always at most $2^j \cdot n/\tau$ and the number of zero values $M_{i-1}[5] = 0$ is at most $2^5 \cdot \frac{n}{\tau}$, three iterations of the recompression for j performed on a string R with initial length r shrink the length of R to a length at most $(\frac{3}{4})^3 r + 2^j \cdot \frac{n}{\tau} + 2^5 \cdot \frac{n}{\tau} \le (\frac{3}{4})^3 r + 2 \cdot 2^j \cdot \frac{n}{\tau}$, by Lemma 16. Putting $r = 2^{j+11} \cdot \frac{n}{\tau}$, we estimate the length of R after three iterations for j from above by $((\frac{3}{4})^3 2^{11} + 2) 2^j \cdot \frac{n}{\tau} < 2^{j+10} \cdot \frac{n}{\tau}$. That is, for each j, three iterations are enough to reduce the length of R to at most $2^{j+10} \cdot \frac{n}{\tau}$.

Thus, the total running time of all recompression procedures is $\mathcal{O}(\sum_{j=\lceil\log^{(4)} n\rceil}^{6} 2^{j+11} \cdot \frac{n}{\tau}\log^{(4)} n) = \mathcal{O}(\frac{n}{\tau}\log^{(4)} n)$, which is $\mathcal{O}(n)$ since $\tau \ge (\log^{(3)} n)^4$. Observe that the most time consuming part is in recalculations of the arrays M_i, each taking $\mathcal{O}(|R| \log^{(4)} n)$ time, all other parts take $\mathcal{O}(|R|)$ time, i.e., $\mathcal{O}(\sum_{j=\lceil\log^{(4)} n\rceil}^{6} 2^{j+11} \cdot \frac{n}{\tau}) = \mathcal{O}(\frac{n}{\tau})$ time is needed for everything without the recalculations. The length of R in the end is at most $2^{16} \cdot n/\tau$, which is a condition under which shrinking iterations stopped for $j = 6$.

Finally, we create a bit array E of the same length as the original string R that marks by 1 those letters that survived all iterations. Additional navigational structures for linear-time E construction are straightforward. We then re-run whole "semi-online" algorithm that generates the set S' (from which the string R was constructed) but, in this time, we discard all positions of S' that correspond to unmarked indices in E and we store all positions corresponding to marked indices of E explicitly in an array S^*. Since at most $2^{16} \cdot n/\tau$ indices in E are marked by 1, the size of S^* is $\mathcal{O}(n/\tau)$.

Finally, we have all required instruments to prove the main lemma. The proof is rather technical and, in a way, similar to the proof of Lemma 12; it is detailed in Appendix D.2 in the full version [27].

▶ **Lemma 17.** *The set S^* is τ-partitioning; also a converse of property (c) holds for S^*: if a substring $s[i..j]$ has a period at most $\tau/4$, then $S^* \cap [i+\tau..j-\tau] = \emptyset$.*

References

1 S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 819–828. SIAM, 2000.

2 P. Bille, J. Fischer, I.L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse text indexing in small space. *ACM Transactions on Algorithms*, 12(3):1–19, 2016. doi:10.1145/2836166.

3 P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *LNCS*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.

4 P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/j.jda.2013.06.003.

5 O. Birenzwige, S. Golan, and E. Porat. Locally consistent parsing for text indexing in small space. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 607–626. SIAM, 2020. doi:10.1137/1.9781611975994.37.

6 G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, volume 6347 of *LNCS*, pages 171–182. Springer, 2010. doi:10.1007/s00453-011-9499-0.

7 R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

8 M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

9 P. Dinklage, J. Fischer, A. Herlez, T. Kociumaka, and F. Kurpicz. Practical performance of space efficient data structures for longest common extensions. In *Proc. 28th Annual European Symposium on Algorithms (ESA)*, volume 173 of *LIPIcs*, pages 39:1–39:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.39.

10 M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.

11 J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Proc. 12th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 9644 of *LNCS*, pages 483–496. Springer, 2016. doi:10.1007/978-3-662-49529-2_36.

12 M. Gańczorz, P. Gawrychowski, A. Jeż, and T. Kociumaka. Edit distance with block operations. In *Proc. 26th Annual European Symposium on Algorithms (ESA)*, volume 112 of *LIPIcs*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.33.

13 P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

14 P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 425–439. SIAM, 2017. doi:10.1137/1.9781611974782.27.

15 A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 315–324. ACM, 1987. doi:10.1145/28395.28429.

16 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

17 T. I. Longest common extensions with recompression. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPIcs*, pages 18:1–18:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2017.18.

18 T. I, J. Kärkkäinen, and D. Kempa. Faster sparse suffix sorting. In *Proc. 31st International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 25 of *LIPIcs*, pages 386–396, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2014.386.

19 A. Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015. doi:10.1016/j.tcs.2015.05.027.

20 A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20, 2015. doi:10.1145/2631920.

21 A. Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. doi:10.1016/j.tcs.2015.12.032.

22 A. Jeż. Recompression: a simple and powerful technique for word equations. *Journal of the ACM*, 63(1):4, 2016. doi:10.1145/2743014.

23 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

24 J. Kärkkäinen and E. Ukkonen. Lempel–Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

25 D. Kempa and T. Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

26 D. Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. doi:10.1016/j.ipl.2017.05.003.

27 D. Kosolobov and N. Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. *arXiv preprint arXiv:2105.03782*, 2021.

28 K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

29 T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. *Discrete Applied Mathematics*, 274:116–129, 2020. doi:10.1016/j.dam.2019.01.014.

30 N. Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Transactions on Algorithms*, 17(1):1–23, 2020. doi:10.1145/3426870.

31 M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. doi:10.1093/bioinformatics/bth408.

32 S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proc. 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 300–309. ACM, 1994. doi:10.1145/195058.195164.

33 S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 76–85, 2003. doi:10.1145/872757.872770.

34 Y. Tanimura, T. I, H. Bannai, S. Inenaga, S.J. Puglisi, and M. Takeda. Deterministic sub-linear space LCE data structures with efficient construction. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 1:1–1:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2016.1.

35 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

BAT-LZ out of hell

Zsuzsanna Lipták
Dipartimento di Informatica, University of Verona, Italy

Francesco Masillo
Dipartimento di Informatica, University of Verona, Italy

Gonzalo Navarro
Center for Biotechnology and Bioengineering (CeBiB), Department of Computer Science, University of Chile, Chile

Abstract

Despite consistently yielding the best compression on repetitive text collections, the Lempel-Ziv parsing has resisted all attempts at offering relevant guarantees on the cost to access an arbitrary symbol. This makes it less attractive for use on compressed self-indexes and other compressed data structures. In this paper we introduce a variant we call BAT-LZ (for Bounded Access Time Lempel-Ziv) where the access cost is bounded by a parameter given at compression time. We design and implement a linear-space algorithm that, in time $O(n \log^3 n)$, obtains a BAT-LZ parse of a text of length n by greedily maximizing each next phrase length. The algorithm builds on a new linear-space data structure that solves 5-sided orthogonal range queries in rank space, allowing updates to the coordinate where the one-sided queries are supported, in $O(\log^3 n)$ time for both queries and updates. This time can be reduced to $O(\log^2 n)$ if $O(n \log n)$ space is used.

We design a second algorithm that chooses the sources for the phrases in a clever way, using an enhanced suffix tree, albeit no longer guaranteeing longest possible phrases. This algorithm is much slower in theory, but in practice it is comparable to the greedy parser, while achieving significantly superior compression. We then combine the two algorithms, resulting in a parser that always chooses the longest possible phrases, and the best sources for those. Our experimentation shows that, on most repetitive texts, our algorithms reach an access cost close to $\log_2 n$ on texts of length n, while incurring almost no loss in the compression ratio when compared with classical LZ-compression. Several open challenges are discussed at the end of the paper.

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

Keywords and phrases Lempel-Ziv parsing, data compression, compressed data structures, repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.21

Supplementary Material *Software (Source Code)*: `https://github.com/fmasillo/BAT-LZ`

Funding *Zsuzsanna Lipták*: Partially funded by the MUR PRIN project Nr. 2022YRB97K "PINC" (Pangenome INformatiCs. From Theory to Applications) and by the INdAM-GNCS Project CUP_E53C23001670001 (Compressione, indicizzazione, analisi e confronto di dati biologici).
Gonzalo Navarro: Funded by Basal Funds FB0001, Mideplan, Chile, and Fondecyt Grant 1-230755, Chile.

1 Introduction

The sharply growing sizes of text collections, particularly repetitive ones, has raised the interest in compressed data structures that can maintain the texts all the time in compressed form [43, 42, 41]. For archival purposes, the original Lempel-Ziv (LZ) compression format [36] is preferred because it yields the least space among the methods that support compression and decompression in polynomial time – actually, Lempel-Ziv compresses and decompresses a text $T[1..n]$ in $O(n)$ time [48]. For using a compression format as a compressed data structure, however – in particular, to build a compressed text self-index on it [34] –, we need that arbitrary text snippets $T[i\,..\,i+\ell]$ can be extracted efficiently, without the need of decompressing the whole text up to the desired snippet. Grammar compression formats [31] allow extracting such text snippets in time $O(\ell + \log n)$ [5, 22], which is nearly optimal [51]. So, although the compression they achieve is always lower-bounded by the size of the LZ parse [49, 8], grammar compression algorithms are preferred over LZ compression in the design of text indexes [42, 12], and of compressed data structures in general.

The LZ compression algorithm parses the text T into a sequence of so-called phrases, where each phrase points backwards to a previous occurrence of it in T and stores the next symbol in explicit form. While this yields a simple linear-time left-to-right decompression algorithm, consider the problem of accessing a particular symbol $T[i]$. Unless it is the final explicit symbol of a phrase, we must determine the text position $j < i$ where $T[i] = T[j]$ was copied from. We must then determine $T[j]$, which again may be – with low chance – the end of a phrase, or it may – most likely – refer to an earlier symbol $T[j] = T[k]$, with $k < j$. The process continues until we hit an explicit symbol. The cost of extracting $T[i]$ is then proportional to the length of that *referencing chain* $i \to j \to k \to \ldots$ Despite considerable interest in algorithms to access arbitrary text positions from the LZ compression format, and apart from some remarkable results on restricted versions of LZ [30], there has been no progress on the original LZ parse (which yields the strongest compression).

In this paper we introduce and study an LZ variant we call *Bounded Access Time Lempel-Ziv (BAT-LZ)*, which takes a compression parameter c and produces a parse where no symbol has a referencing chain longer than c, thereby guaranteeing $O(c)$ access time.[1] As opposed to classical LZ, BAT-LZ parses allow very fast access to the text, indeed, like a bat out of hell.

We design a *Greedy BAT-LZ parser*, which at each step of the compression chooses the longest possible phrase. Finding such a phrase boils down to solving a 4-sided orthogonal range query in a 3-dimensional grid (in rank space), where one of the coordinates undergoes updates as the parsing proceeds. We design such a data structure, which turns out to handle 5-sided queries and support updates on the coordinate where the query is one-sided. Our data structure handles queries and updates in time $O(\log^3 n)$, yielding a greedy BAT-LZ parsing in time $O(n \log^3 n)$ and space $O(n)$. We then design another BAT-LZ parser, referred to as *Minmax*, which runs on an enhanced suffix tree. It looks for the “best” possible sources of the chosen phrases, that is, with symbols having shorter referencing chains, while not necessarily choosing the longest possible phrase. Finally, we combine the two ideas, resulting in our *Greedier parser*, which runs again on an enhanced suffix tree. These last two algorithms, while their running time is upper bounded by $O(n^3 \log n)$, both run in decent time in practice.

We implemented and tested our three BAT-LZ parsers on various repetitive texts of different sorts, comparing them with the original LZ parse and with two simple baselines that ensure BAT-LZ parses without any optimization. The results show that all three algorithms run in a few seconds per megabyte and produce much better parses than the baselines. For values of $c = O(\log n)$ with a small constant, they produce just a small fraction of extra phrases on top of LZ. In particular, Greedier increases the size of the LZ parse by less than 1% with c values that are about $\log_2 n$ (i.e., 20–30 in our texts).

We note that, unlike the original LZ parse, a greedy parsing does not guarantee obtaining the minimal BAT-LZ parse. Indeed, finding the optimal BAT-LZ parse has recently been shown to be NP-hard for all constant c, and also hard to approximate for any constant approximation ratio [10]. Our results show that, on repetitive texts, a polylog-linear time

[1] A parsing like BAT-LZ was described as a baseline in the experimental results in previous work [33] of one of the authors, but without a parsing algorithm, see Sec. 3 for more details.

greedy algorithm can nonetheless achieve good compression while guaranteeing fast access to text snippets. The other two algorithms are still polynomial time and offer fast access with almost no loss in compression compared to the classical LZ-compression. In our scenarios of interest (i.e., accessing the compressed text at random) the data is compressed only once and accessed many times, so slower compression algorithms can be afforded in exchange for faster access. We discuss at the end this and some other problems our work opens.

2 Basic Data Structures

A string (or text) T is a finite sequence of characters from an alphabet Σ. We write $T = T[1..n]$ for a string T of length n, and assume that the final character is a unique end-of-string marker \$. We index strings from 1 and write $T[i..j]$ for the substring $T[i]\,..\,T[j]$, $T[i\,..]$ for the suffix starting in position i, and $T[..\,j]$ for the prefix ending in position i.

Bitvectors and Wavelet Matrices. A bitvector $B[1\,..\,n]$ can be stored using n bits, or actually $\lceil n/w \rceil$ words on a w-bit word machine, while providing access and updates to arbitrary bits in constant time. If the bitvector is static (i.e., does not undergo updates) then it can be preprocessed to answer *rank* queries in $O(1)$ time using $o(n)$ further bits [11, 39]: $rank_b(B, i)$, where $b \in \{0, 1\}$ and $0 \le i \le n$, is the number of times bit b occurs in $B[1\,..\,i]$.

A wavelet matrix [13] is a data structure that can be used, in particular, to represent a discrete $[1, n] \times [1, n]$ grid, with exactly one point per column, using $n \log_2 n + o(n \log_2 n)$ bits. Let $S[1\,..\,n]$ be such that $S[i]$ is the row of the point at column i. The first wavelet matrix level contains a bitvector $B_1[1\,..\,n]$ with the highest (i.e., $\lceil \log_2 n \rceil$th) bit of every value in S. For the second level, the sequence values are stably sorted by their highest bit, and the wavelet matrix stores a bitvector $B_2[1\,..\,n]$ with the second highest bits in that order. To build the third level, the values are stably sorted by their second highest bit, and so on. Every level i also stores the number $z_i = rank_0(B_i, n)$ of zeros in its bitvector.

The value $S[i]$ can be retrieved from the wavelet matrix in $O(\log n)$ time. Its highest bit is $b_1 = B_1[i_1]$, with $i_1 = i$. The second highest bit is $b_2 = B_2[i_2]$, with $i_2 = rank_0(B_1, i_1)$ if $b_1 = 0$ and $i_2 = z_1 + rank_1(B_1, i_1)$ if $b_1 = 1$. The other bits are obtained analogously.

The wavelet matrix can also obtain the grid points that fall within a rectangle $[x_1, x_2] \times [y_1, y_2]$ (i.e., the values $(i, S[i])$ such that $x_1 \le i \le x_2$ and $y_1 \le S[i] \le y_2$) in time $O(\log n)$, plus $O(\log n)$ per point reported. We start at the first level, in the range $B_1[sp_1, ep_1] = B_1[x_1, x_2]$. We then map the range into two ranges of the second level: the positions i where $B_1[i] = 0$ are all mapped to the range $B_2[sp_2, ep_2] = B_2[rank_0(B_1, sp_1-1)+1, rank_0(B_1, ep_1)]$, and those where $B_1[i] = 1$ are mapped to $B_2[sp'_2, ep'_2] = B_2[z_1 + rank_1(B_1, sp_1 - 1) + 1, z_1 + rank_1(B_1, ep_1)]$. The recursive process stops when the range becomes empty; when the sequence of highest bits makes the possible set of values either disjoint with $[y_1, y_2]$ or included in $[y_1, y_2]$; or when we reach the last level. It can be shown that the recursion ends in $O(\log n)$ ranges, at most two per level, so that every value in those ranges is an answer. The corresponding y values can be obtained by tracking them downwards as explained.

These data structures, and our results, hold in the RAM model with computer word size $w = \Theta(\log n)$. The wavelet matrix is then said to use $O(n)$ space – i.e., linear space –, which is counted in w-bit words. The wavelet matrix is easily built in $O(n \log n)$ time, and less [40].

Another relevant functionality that can be offered within $2n + o(n)$ bits is the so-called *range maximum query (RMQ)*: given a static array $A[1\,..\,n]$, we preprocess it in $O(n)$ time so that we can answer RMQs in $O(1)$ time [19]: $rmq(A, i, j)$ is a position p, $i \le p \le j$, such that $A[p] = \max\{A[k],\ i \le k \le j\}$. The data structure does not need to maintain A. In this paper we will use RMQs where A can undergo updates, see Sec. 5.

Suffix Arrays and Trees. The suffix tree [52] is a classic data structure on texts which is able to answer efficiently many different kinds of string processing queries [24, 1], which uses linear space and can be built in linear time [52, 38, 17, 50]. We give a brief recap; see Gusfield [24] for more details.

The suffix tree $\mathsf{ST}(T)$ of a text T is the compact trie of the suffixes of T; it is a rooted tree whose edges are labeled by substrings of T (stored as two pointers into T), and whose inner nodes are branching. The *label* $L(v)$ of a node v is the concatenation of the labels of the edges on the root-to-v path. There is a one-to-one correspondence between leaves and suffixes of T; leaf_i is then the unique leaf whose label equals the ith suffix $T[i\,..]$. The *stringdepth* $\mathit{sd}(v)$ of a node v is the length of its label, and we assume $\mathit{sd}(v)$ is stored in v.

The suffix array SA of T is a permutation of the index set $\{1, \ldots, n\}$ such that $\mathsf{SA}[i] = j$ if the jth suffix of T is the ith in lexicographic order among all suffixes. The suffix array can be computed from the suffix tree, or directly from the text, in linear time and space [47, 45]. The inverse suffix array, denoted ISA, can be computed in linear time using $\mathsf{ISA}[\mathsf{SA}[i]] = i$.

3 The Lempel-Ziv (LZ) Parsing and its Bounded Version (BAT-LZ)

The Lempel-Ziv (LZ) parsing of a text $T[1\,..\,n]$ [36] produces a sequence of z "phrases", which are substrings of T whose concatenation is T. Each phrase is formed by the longest substring that has an occurrence starting earlier in T, plus the character that follows it.

▶ **Definition 1.** *A* leftward parse *of* $T[1\,..\,n]$ *is a sequence of substrings* $T[i\,..\,i+\ell]$ *(called* phrases*) whose concatenation is* T *and such that there is an occurrence of each* $T[i\,..\,i+\ell-1]$ *starting before* i *in* T *(the occurrence is called the* source *of the phrase). The LZ parse of* T *is the leftward parse of* T *that, in a left-to-right process, chooses the longest possible phrases.*

The algorithm moves a pointer i along T, from $i = 1$ to $i = n$. At each step, the algorithm has already processed $T[1\,..\,i-1]$, and it must form the next phrase. As said, the phrase is formed by (1) the longest prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ that has an occurrence in T starting before position i, and (2) the next symbol $T[i+\ell]$. If $\ell > 0$, then the occurrence of (1), $T[s\,..\,s+\ell-1] = T[i\,..\,i+\ell-1]$ with $s < i$, is called the source of $T[i\,..\,i+\ell-1]$. Once suitable s and ℓ have been determined, the next phrase is $T[i\,..\,i+\ell]$ and the algorithm proceeds from $i \leftarrow i+\ell+1$ onwards. The phrase $T[i\,..\,i+\ell]$ is encoded as the triple $(s, \ell, T[i+\ell])$, and if $\ell = 0$ we can encode just the character $(T[i+\ell])$.

This greedy parsing, which maximizes the phrase length at each step, turns out to be optimal [36], that is, it produces the least number z of phrases among all the leftward parses of T. Further, it can be computed in $O(n)$ time [48, 9, 46, 25, 26, 23, 20, 32, 3, 27, 21].

Note that phrases can overlap their sources, as sources must start – but not necessarily end – before i. For example, the LZ parse of $T = \mathtt{a}^{n-1}\$$ is $(\mathtt{a})$ $(0, n-1, \$)$. For illustrative purposes, we describe the parsings by writing bars, "|", between the formed phrases. The parsing of the example is then written as $\mathtt{a}|\mathtt{a}^{n-1}\$$. To illustrate the access problem, consider the LZ parsing of the text `alabaralalabarda$` (disregard for now the numbers below):

a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	$
0	0	1	0	1	0	1	1	2	0	2	1	2	1	0	1	0

Assume we want to extract $T[11] = \mathtt{a}$. The position is the first of the 6th phrase, `abard`, and it is copied from the third phrase, `ab`. In turn, the first position of that phrase is copied from the first phrase, where `a` is stored in explicit form. We need then to follow a *chain* of length two in order to extract $T[11]$, so the length of that chain is the access cost. The numbers we wrote below the symbols in the parse are the lengths of their chains.

Bounded Access Time Lempel-Ziv (BAT-LZ). We define a leftward parse we call Bounded Access Time Lempel-Ziv (BAT-LZ), which takes as a parameter the maximum length c any chain can have. A BAT-LZ parse is a leftward parse where no chain is longer than c. Note that we do not require a BAT-LZ parse to be of minimal size. For example, a BAT-LZ parse for the above text with $c = 1$ is as follows:

a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	$
0	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0

When the LZ parse produces the phrase $T[i\,..\,i+\ell]$ from the source $T[s\,..\,s+\ell-1]$ and the extra symbol $T[i+\ell]$, the character $T[i+\ell]$ is stored in explicit form, and thus its chain is of length zero. The chain length of every other phrase symbol, $T[i+l]$ for $0 \le l < \ell$, is one more than the chain length of its source symbol, $T[s+l]$.

A special case occurs when sources and targets overlap. If we want to extract $T[n-1]$ from $T = \mathtt{a}^{n-1}\$$, we could note that it is copied from $T[n-2]$, which is in turn copied from $T[n-3]$, and so on, implying a chain of length $n-1$. Instead, we can note that our phrase $T[2\,..\,n]$ overlaps its source $T[1\,..\,n-2]$. In general, when the phrase $T[i\,..\,i+\ell-1]$ overlaps its source $T[s\,..\,s+\ell-1]$ by $0 < b = i-s$ characters, this implies that the word $S = T[s\,..\,s+\ell-1] = T[i\,..\,i+\ell-1]$ has a *border* (a prefix which is also a suffix) of length b. It is well known that if S has a border of length b, then S has a period $p = |S| - b$, see [37, Ch. 8]. Therefore, S can be written in the form $S = U^{\lfloor |S|/p \rfloor} V$, where U is the p-length prefix of S and V a proper prefix of U, and thus, for all $l > p$, $S[l] = S[l \bmod p]$.

▶ **Definition 2** (Chain length). *Let $T[i\,..\,i+\ell]$ be a phrase in a leftward parse of $T[1\,..\,n]$, whose source is $T[s\,..\,s+\ell-1]$. The chain length of the explicit character is $C[i+\ell] = 0$. If $\ell \le i-s$ (i.e., there is no overlap between the source and the phrase), then for all $0 \le l < \ell$, $C[i+l] = C[s+l]+1$. Otherwise, for $0 \le l < i-s$, the chain length is $C[i+l] = C[s+l]+1$, and for $i-s \le l < \ell$, the chain length is $C[i+l] = C[i+(l \bmod (i-s))]$.*

We remark that a parsing like BAT-LZ is described as a baseline in the experimental results of one of the current authors' previous work [33], under the name LZ-Cost, but as no efficient parsing algorithm was devised for it, it could be tested only on the tiny texts of the Canterbury Corpus (`https://corpus.canterbury.ac.nz`). It also did not handle overlaps between sources and targets, so it did not perform well on the text $T = \mathtt{a}^n$. For testing the BAT-LZ parsing on large repetitive text collections we need an efficient parsing algorithm.

4 A Greedy Parsing Algorithm for BAT-LZ

In this section we describe an algorithm that, using $O(n)$ space and $O(n \log^3 n)$ time, produces a BAT-LZ parse of a text $T[1\,..\,n]$ by maximizing the next phrase length at each step. We then show how to reduce the time to $O(n \log^2 n)$ at the price of increasing the space to $O(n \log n)$. Of course, unlike in LZ, this greedy algorithm does not in general produce an optimal BAT-LZ parse, since the problem is NP-hard.

▶ **Definition 3.** *A BAT-LZ parse of $T[1\,..\,n]$ with maximum chain length c is a leftward parse of T where the chain length of no position exceeds c. A* greedy BAT-LZ parse *is a BAT-LZ parse where each phrase, processed left to right, is as long as possible.*

Let $T[1\,..\,i-1]$ be already processed. We call a prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ *valid* if $C[j] \le c$ for all $j = i, \ldots, i+\ell-1$. A leftward parse of T is therefore a BAT-LZ parse if and only if all phrases are valid. Our Greedy BAT-LZ parser proceeds then analogously to the original LZ parser. At each step, it has already processed $T[1\,..\,i-1]$, and it must find the next phrase, which is formed by (1) the longest valid prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ that has an occurrence $T[s\,..\,s+\ell-1]$ with $s<i$, and (2) the next symbol $T[i+\ell]$. In other words, the algorithm enforces that every symbol in $T[s\,..\,s+\ell-1]$ must have a chain length less than c, the maximum chain length allowed. The phrase $T[i\,..\,i+\ell]$ is encoded just as in the standard LZ, as a triple $(s,\ell,T[i+\ell])$.

To efficiently find s and ℓ, our BAT-LZ parsing algorithm stores the following structures:

1. The suffix array $\mathsf{SA}[1\,..\,n]$ of T, represented as a wavelet matrix [13].
2. The inverse suffix array $\mathsf{ISA}[1\,..\,n]$ of T, represented in plain form.
3. An array $C[1\,..\,n]$, where $C[i]$ is the chain length of i. Note that $C[i]$ is defined only for the already parsed positions of T.
4. An array $D[1\,..\,n]$, where $D[s]$ is the minimum $d\geq 0$ such that $C[s+d]=c$. If no such a d exists (in particular, because $C[i]$ is defined only for the parsed prefix), then $D[s]=\infty$ (which holds initially for all s).
5. For each level of the wavelet matrix of SA, a special *dynamic RMQ* data structure to track the text positions that can be used. This structure is related to the values of D and therefore it changes along the parsing.

Note that the definition of BAT-LZ implies that, if the source of $T[i\,..\,i+\ell-1]$ is $T[s\,..\,s+\ell-1]$, then it must be that $\ell\leq D[s]$. This motivates the following observation:

▶ **Observation 4.** *Let $T[1\,..\,i-1]$ be already processed. A prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ is valid if and only if there exists a source $T[s\,..\,s+\ell-1]$ such that*

(i) *its lexicographic position satisfies $\mathsf{ISA}[s]\in[sp\,..\,ep]$, where $[sp\,..\,ep]$ is the suffix array range of $T[i\,..\,i+\ell-1]$ (i.e., $T[s\,..\,s+\ell-1]=T[i\,..\,i+\ell-1]$);*

(ii) *its starting position in T is $s<i$; and*

(iii) *it does not use forbidden text positions, that is, $\ell\leq D[s]$.*

The parsing then must find the longest valid prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$. We do so by testing the consecutive values $\ell=1,2,\ldots$. Note that, once we have determined the next phrase $T[i\,..\,i+\ell]$, we must update C and D as follows: (1) $C[i+l]\leftarrow C[s+l]+1$ for all $0\leq l<\ell$, and $C[i+\ell]\leftarrow 0$.[2], and (2) Every time we obtain $C[t]=c$ in the previous point, we set $D[k]\leftarrow t-k$ for all $k'<k\leq t$, where k' is the last position where $D[k']<\infty$ (so $k'=0$ in the beginning and we reset $k'\leftarrow t$ after this process).

Note that points (i) and (ii) above correspond to the classic LZ parsing problem. In particular, they correspond to determining whether there are points in the range $[sp,ep]\times[1,i-1]$ of the grid represented by our wavelet matrix, which represents the points $(j,\mathsf{SA}[j])$. As the wavelet matrix answers this query in time $O(\log n)$, this yields an $O(n\log n)$ LZ parsing algorithm. Point (iii), however, is exclusive to BAT-LZ. It can be handled by converting the grid into a three-dimensional mesh, where we store the values $(j,\mathsf{SA}[j],D[\mathsf{SA}[j]])$ and look for the existence of points in the range $[sp,ep]\times[1,i-1]\times[\ell,n]$. Note that we need to determine whether the range is empty and, if it is not, retrieve a point from it (whose second coordinate is the desired s). In addition, as the array D is modified along the parsing, we need a dynamic 3-dimensional data structure: every time we modify D in point 2 above, our data structure changes (this occurs up to n times). See Fig. 1.

[2] Recall that a special case occurs if $T[i\,..\,i+\ell-1]$ overlaps $T[s\,..\,s+\ell-1]$: we start copying from $k=s$ and increasing k and, whenever $k=s+l=i$, we restart copying from $k=s$.

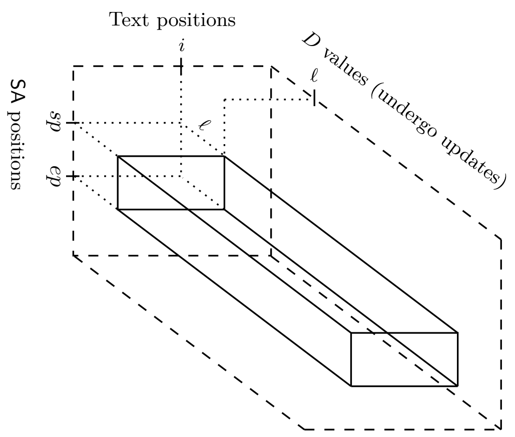


Figure 1 General scheme of our translation of queries onto a 3-dimensional data structure.

Our 3-dimensional problem, then, (a) is essentially a range emptiness query (where we must return one point if there are any), (b) the search is 4-sided (though our solution handles 5-sided queries), and (c) the updates in D occur only to convert some $D[k] = \infty$ into a smaller value, so each value $D[k]$ changes at most once along the parsing process (yet, our solution handles arbitrary updates along the coordinate where the query is one-sided). We have found no linear-space solutions to this problem in the literature; only solutions to less general ones or using super-linear space (indeed, more than $O(n \log n)$): (1) linear space for *two dimensions*, with $O(\log n)$ query time and $O(\log^{3+\epsilon} n)$ update time [44]; (2) linear space for three dimensions with *no updates*, with $O(\log n / \log \log n)$ query time [6]; (3) *super-linear* space (at least $O(n \log^{1.33} n)$ for three dimensions), with $O((\log n / \log \log n)^2)$ query time and $O(\log^{1.33+\epsilon} n)$ update time [7]. In the next section we describe our data structures for this problem: one uses linear space and $O(\log^3 n)$ query and update time; the other uses $O(n \log n)$ space and $O(\log^2 n)$ query time. This yields our first main result.

▶ **Theorem 5.** *A Greedy BAT-LZ parse of a text $T[1..n]$ can be computed using $O(n)$ space and $O(n \log^3 n)$ time, or $O(n \log n)$ space and $O(n \log^2 n)$ time.*

5 A Geometric Data Structure

To solve the 3-dimensional search problem we associate, with each level of the wavelet matrix, a data structure that represents the sequence of values $D[k]$ in the order the text positions k are listed in that level. Because in linear space we cannot store the actual values in every wavelet matrix level, we store only a dynamic RMQ data structure on the internal levels, and store the explicit values only in (the order corresponding to) the last level (in a wavelet matrix, that final level is not the text order, thus we need another array to map it to D).

Let D_l be the array D permuted in the way it corresponds to level l of the wavelet matrix. The dynamic RMQ structure for level l is then a heap-shaped perfectly balanced tree $H_l[1..n]$ whose leaves (implicitly) point to the entries of D_l. The nodes $H_l[p]$ store only one bit, 0 indicating that the maximum in the subtree is to the left and 1 indicating that it is to the right. By navigating H_l from the root p of any subtree, moving to $H_l[2p]$ if $H_l[p] = 0$ and $H_l[2p+1]$ if $H_l[p] = 1$, we arrive in $O(\log n)$ time at the position p where $D_l[p]$ is maximum below that subtree. The actual value $D_l[p]$ is obtained in other $O(\log n)$ time by tracking position p downwards in the wavelet matrix, from level l until the last level, where the values of D are explicitly stored. See Fig. 2 (right); ignore the query for now.

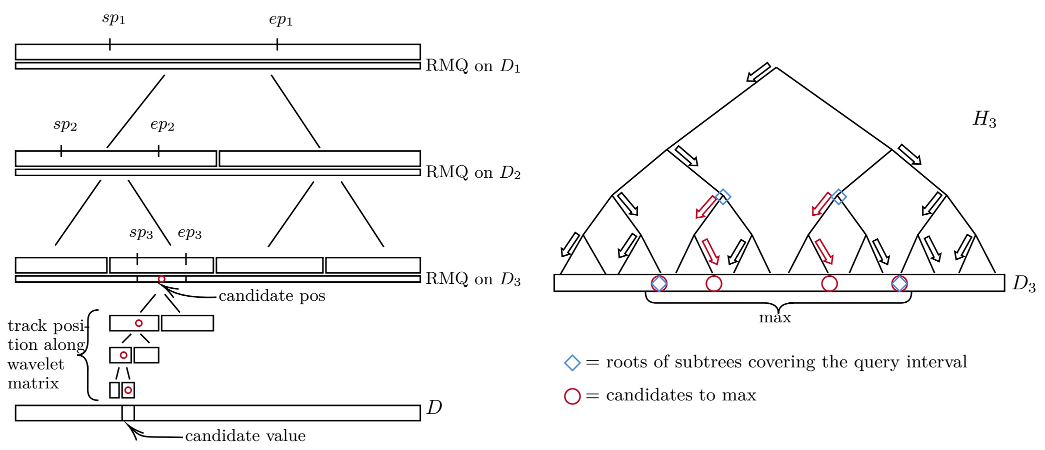


Figure 2 On the left, we reach a candidate area $[sp_3, ep_3]$ of the wavelet matrix and must obtain its maximum D value using the (dynamic) RMQ data structure for D_3. The tree H_3 for this RMQ structure is shown on the right. Arrows point to the child holding the maximum value in D_3. Blue diamonds are the roots $v_3^1, \ldots, v_3^4$ of the subtrees that cover the query area $[sp_3, ep_3]$ and red circles are the candidates in the range. The left plot shows how we find the actual value of one of those circles by tracking it down in the wavelet matrix.

Updates. When a value $D[k]$ decreases from ∞, we obtain its position in the top-level of the wavelet matrix as $p = \mathsf{ISA}[k]$; thus we must reflect in H_1 the decrease in the value of $D_1[p]$. By halving p successively we arrive at its ancestors, $H_1[p_h]$ for $p_h = \lfloor p/2^h \rfloor$, $h = 1, 2, \ldots$ We traverse the path upwards, recomputing the maximum value m below p_h and modifying accordingly the bits of $H_1[p_h]$. Initially, this new maximum is $m = D_1[p] = D[k]$. At any point in the traversal, if the parent $H_1[p_h]$ of the current node indicates that the maximum below p_h descends from the *other* child of p_h, then we can stop updating of H_1, because decreasing $D_1[p]$ does not require further changes. Otherwise, we must obtain the maximum value m' below the other child of $H_1[p_h]$ and compare it with m. The value m' is obtained in $O(\log n)$ time as explained in the previous paragraph. We set $H_1[p_h]$ depending on which is larger between m and m', update $m \leftarrow \max(m, m')$, and continue upwards. This process takes $O(\log^2 n)$ time as we traverse all the levels of H_1. We then track position p downwards to the second level of the wavelet matrix, update H_2 in the same way, and continue updating H_l on all the wavelet matrix levels l, for a total update time of $O(\log^3 n)$.

Searches. The search for a range $[sp, ep] \times [1, i-1] \times [\ell, n]$ first determines, as in the normal wavelet matrix search algorithm, the $O(\log n)$ maximal ranges that cover $[1, i-1]$ along the wavelet matrix levels l (there is at most one range per level because the range $[1, i-1]$ is one-sided; otherwise there could be two), and maps $[sp, ep]$ to $[sp_l, ep_l]$ on each such range (see Sec. 2), all in time $O(\log n)$. We then need to determine if there is some value $D_l[p] \geq \ell$ below some of the ranges $[sp_l, ep_l]$ (see the top-left part of Fig. 2). Each such range is then, again, decomposed into $O(\log n)$ maximal nodes $v_l^1, v_l^2, \ldots$ of H_l (see the right of Fig. 2). We find, in $O(\log n)$ time, the maximum value of D_l below each node v_l^j, stopping as soon as we find some value $\geq \ell$. Note that we use $O(\log n)$ time to find the *position* of the maximum in D_l using H_l, and then $O(\log n)$ time to find the *value* of that maximum by tracking the position down in the wavelet matrix (see the bottom left of Fig. 2). Since we have $O(\log n)$ ranges $[sp_l, ep_l]$, each yielding $O(\log n)$ candidates v_l^i, and the maximum of each candidate is computed in $O(\log n)$ time, the whole search process takes time $O(\log^3 n)$.

Generalizations. Though not necessary for our problem, we remark that our update process can be extended to arbitrary updates on the third coordinate, $D[k]$, not only to reductions in value. Further, our search could support five-sided ranges, not only four-sided, because we would still have $O(\log n)$ ranges $[sp_l, ep_l]$ if the range of the second coordinate was two-sided. Only the range of the third coordinate (the one supporting the updates) must be one-sided.

Faster and larger. By storing the values of D_l in each node of H_l for each wavelet matrix level l, the space increases to $O(n \log n)$ but the time of updates and searches decreases to $O(\log^2 n)$, as we have now the maximum below any $H_l[p_h]$ readily available in $O(1)$ time.

6 The Minmax Parsing Algorithm

We note that our Greedy BAT-LZ algorithm does not necessarily produce the smallest greedy parse, because it may fail in choosing the best *source* for the longest phrase. Consider, say, the text $T =$ `alabaralalabarda$` and $c = 2$. Our implementation parses it into 8 phrases as

$$\begin{array}{c|c|cc|cc|cccc|ccc|cc|cc|}
\texttt{a} & \texttt{l} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{r} & \texttt{a} & \texttt{l} & \texttt{a} & \texttt{l} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{r} & \texttt{d} & \texttt{a} & \texttt{\$} \\
0 & 0 & 1 & 0 & 2 & 0 & 1 & 1 & 2 & 0 & 2 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}$$

because it chooses $T[3]$ as the source for the 4th phrase, `ar`, and then $T[5]$ has a chain of length two and cannot be used again. If, instead, we choose $T[1]$ as the source of the 4th phrase, the chain of $T[5]$ will be of length 1 and we could parse T into 7 phrases, just as the first parse shown in Sec. 3.

Our second algorithm, the Minmax parser, always chooses a source that minimizes the maximum chain length in the phrase, among all possible sources. It compromises however on the *length* of the phrase, by not always choosing the longest admissible phrase. As we will see, this is well worth it: Minmax always produces a much better compression than Greedy.

High-level description of the Minmax parser. Let $T[1..i-1]$ be already processed. We will call a prefix $T[i..i+\ell-1]$ of $T[i..]$ *admissible* if it has a source $T[s..s+\ell-1]$ with $\max C[s..s+\ell-1] < c$. We would ideally like to find the longest admissible prefix of $T[i..]$, and then choose its best source if there is more than one. We will use an enhanced suffix tree of the text; this will allow us to store additional information in the nodes. Navigating in the suffix tree, we will then be able to choose the longest admissible prefix *which ends in some node* (i.e., not necessarily the longest), and then choose the best source of this prefix. In order to do this, we will match the current suffix $T[i..]$ in the usual way in the suffix tree, using the desired information written in the nodes. As this information is dynamic, however, we will have to update it during the algorithm. The algorithm thus proceeds by (1) matching the suffix $T[i..]$ in the suffix tree and returning the next phrase and its source, and (2) updating the annotation.

Annotation of the suffix tree. On the suffix tree of T, we annotate each node v with three variables $\mathsf{minmax}(v)$, $\mathsf{txtpos}(v)$, and a Boolean $\mathsf{real}(v)$, initializing $\mathsf{minmax}(v)$ to $+\infty$, $\mathsf{txtpos}(v)$ to -1, and $\mathsf{real}(v)$ to 0. Recall that $L(v)$ is the label of v and $sd(v)$ its length. The variables $\mathsf{minmax}(v)$ and $\mathsf{txtpos}(v)$ will point to the current best candidate of an occurrence of $L(v)$, with $\mathsf{txtpos}(v)$ its starting position and $\mathsf{minmax}(v)$ the maximum C-value within this occurrence. The Boolean $\mathsf{real}(v)$ indicates whether this value is *realistic* ($\mathsf{real}(v) = 1$), i.e., a full occurrence with this value has already been seen, or only *optimistic* ($\mathsf{real}(v) = 0$), meaning that no full occurrence has yet been seen. More formally, let i be the current position, and let us first assume that $\mathsf{real}(v) = 1$. Then $\mathsf{minmax}(v) = x$ if $x = \min\{\max C[s\,..\,s + sd(v) - 1] : T[s\,..\,s + sd(v) - 1] = L(v)$ and $s + sd(v) - 1 < i\}$, and $\mathsf{txtpos}(v) = s_0$ for one such s_0, i.e., (i) $T[s_0\,..\,s_0 + sd(v) - 1] = L(v)$, (ii) $s_0 + sd(v) - 1 < i$, and (iii) $\max C[s_0\,..\,s_0 + sd(v) - 1] = x$.

Now let us look at the case $\mathsf{real}(v) = 0$, we have yet to see an occurrence of $L(v)$. Initially, $\mathsf{minmax}(v) = +\infty$; when we encounter a non-empty prefix of $L(v)$, of length $0 < d \leq sd(v)$, starting, say, in position s_0, we update $\mathsf{minmax}(v)$ to $\max C[s_0\,..\,s_0 + d - 1]$ and $\mathsf{txtpos}(v)$ to s_0. Thus, we have seen an occurrence of a prefix of $L(v)$ but not yet a full occurrence of $L(v)$, and we are optimistic since we are hoping to find a full occurrence whose max does not exceed the current one. However, as soon as we find the first full occurrence (and set $\mathsf{real}(v) = 1$), from that point on we only update $\mathsf{minmax}(v)$ and $\mathsf{txtpos}(v)$ if we see another full occurrence. Therefore, $\mathsf{real}(v)$ is updated exactly once during the algorithm.

Finding an admissible phrase and choosing its source. Let us now assume that we have processed $T[1\,..\,i-1]$ and want to find the next phrase and source. We match $T[i\,..]$ in the suffix tree, making sure during navigation that we only get admissible prefixes of $T[i\,..]$. In particular, if we are in node v and should go to child u of v next (because $T[i + sd(v)]$ is the first character of the edge label (v, u)), then we first check if $\mathsf{minmax}(u) < c$. If so, then we can descend to u and continue from there, skipping over the next $sd(u) - sd(v)$ positions in T. Otherwise, $\mathsf{minmax}(u) \geq c$ and we return the new phrase $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$. Moreover, the C-array for $j = i, \ldots, i + \ell$ is set according to Def. 2.

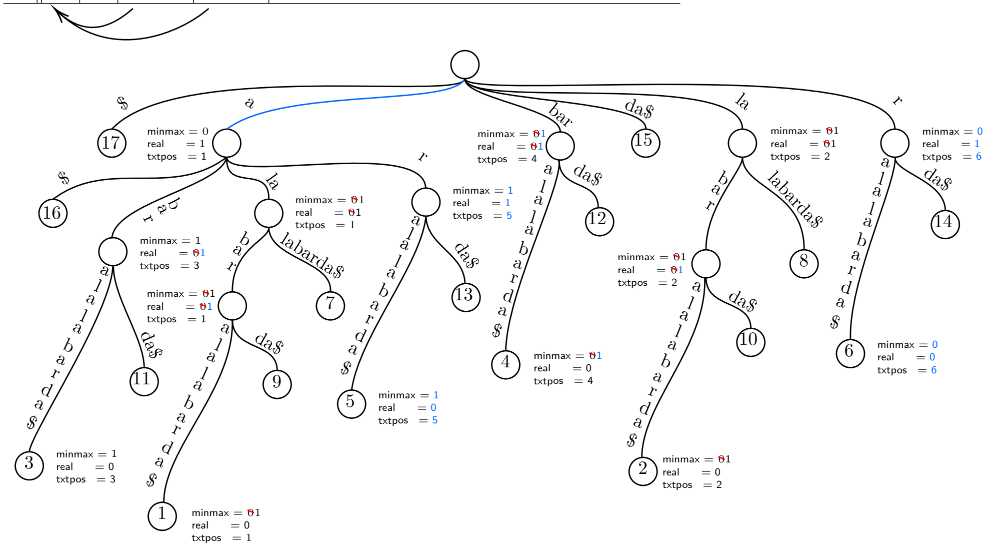
Updating the suffix tree annotation. After the new phrase has been computed, we need to update the annotations in the suffix tree. For $j \leq i + \ell$, going backward in the string, we will update the nodes on the leaf-to-root path from leaf j. The idea is the following.

Fix $j \leq i + \ell$. The prefix $T[j\,..\,i + \ell]$ of $T[j\,..]$ now has the C-array filled in, so its max-value $m = \max C[j\,..\,i + \ell]$ is known. This may or may not necessitate updates in the nodes on the path from leaf $leaf_j$ to the root. First, for the leaf j itself, if $i \leq j$, then the minmax is still $+\infty$, so we set $\mathsf{minmax}(leaf_j) \leftarrow m$. Otherwise, we are seeing a longer prefix of $T[j\,..]$ than before, so we update $\mathsf{minmax}(leaf_j) \leftarrow \max(\mathsf{minmax}(leaf_j), m)$. Regarding the nodes v on the path from $leaf_j$ to the root: their labels are increasingly shorter prefixes of suffix $T[j\,..]$, so they need to be updated only as long as $j + sd(v) - 1 \geq i$, since otherwise, the prefix $L(v)$ does not overlap with the newly assigned subinterval $C[i\,..\,i + \ell]$.

So let $j + sd(v) \geq i$, there are two cases. First, if $j + sd(v) - 1 \leq i + \ell$, then m is a realistic value, since the entire corresponding C-array interval has been filled in. Therefore, we can then compute m in a more clever way by using an RMQ on C, i.e., $m = RMQ(C, j, j + sd(v) - 1)$. So if $\mathsf{real}(v) = 0$, then we update $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{real}(v) \leftarrow 1$. Otherwise (if $\mathsf{real}(v) = 1$), an update is needed only if $\mathsf{minmax}(v) > m$, in which case we set $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{txtpos}(v) \leftarrow j$; since $\mathsf{real}(v) = 1$, we have seen the label $L(v)$ before and already had a realistic value for its minmax value. Second, if $j + sd(v) - 1 > i + \ell$, then m is an optimistic value only, and therefore, we update the annotation of v only if $\mathsf{real}(v) = 0$; in that case, we set $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{txtpos}(v) \leftarrow j$.

Finally, we use the following criterion for how far back in the string we need to go with j. If no node in the path from $leaf_j$ to the root can be effected by the new phrase, then we do not need to consider position j at all in the current iteration. This holds if the label of the parent node does not reach i, i.e., if $j + sd(parent(leaf_j)) - 1 < i$. We compute an auxiliary array $E[1\,..\,n]$ s.t. $E[j] = j + sd(parent(leaf_j)) - 1$. It is easy to see that $E[j] \leq E[j']$ if $j < j'$. This means that, moving back-to-front, we can stop at the first j for which $E[j] < i$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$
C	0	0	1	0	1	0											



\$
a
bar
da\$
la
r
17
minmax = 0 real = 1 txtpos = 1
\$
16
abar
la
r
minmax = ~~0~~1 real = ~~0~~1 txtpos = 4
da\$
15
minmax = ~~0~~1 real = ~~0~~1 txtpos = 2
minmax = 0 real = 1 txtpos = 6
minmax = 1 real = ~~0~~1 txtpos = 3
minmax = ~~0~~1 real = ~~0~~1 txtpos = 1
minmax = 1 real = 1 txtpos = 5
12
bar
labarda\$
8
14
11
7
13
minmax = ~~0~~1 real = ~~0~~1 txtpos = 2
minmax = ~~0~~1 real = ~~0~~1 txtpos = 1
9
10
6
minmax = 0 real = 0 txtpos = 6
4
minmax = ~~0~~1 real = 0 txtpos = 4
5
minmax = 1 real = 0 txtpos = 5
3
minmax = 1 real = 0 txtpos = 3
2
minmax = ~~0~~1 real = 0 txtpos = 2
1
minmax = ~~0~~1 real = 0 txtpos = 1

Figure 3 Example of the Minmax algorithm using the suffix tree of $T =$ `alabaralalabarda$`. The vertical bars are for delimiting already parsed phrases. See Example 6 for more details.

A worst-case time complexity for a Minmax parse producing z' phrases is $O(z'n^2) \subseteq O(n^3)$, as in principle one can consider every $j \in [1..i-1]$ for every new phrase $T[i..i+\ell]$, traverse the $O(n)$ ancestors of $leaf_j$, and run an RMQ operation on each. While the RMQ structure we use on C is dynamic, it only undergoes appends to the right, in which case it is possible to support updates in $O(1)$ amortized time and queries in $O(1)$ time [18, p. 5]. We do not know if this cubic complexity is tight, however. In practice we expect z' to be much less than n on highly repetitive texts, and the height of the suffix tree to be logarithmic, yielding a time complexity of $O(z'n \log n)$, which thus becomes practical on repetitive data.

▶ **Example 6.** In Fig. 3, we can see the suffix tree ST for $T =$ `alabaralalabarda$` with some additional annotations in some nodes. In this example, the first three phrases, i.e., `a`, `l`, and `ab`, have been already computed, with the corresponding chain lengths in C and annotations in ST. The annotations exhibit non-trivial updates using the colour red, namely updates that are different than changing the starting value for minmax, i.e., changing minmax from $+\infty$ to a finite value. Nodes whose annotation is not shown have not been updated yet, therefore, they have minmax $= +\infty$, txtpos $= -1$, and real $= 0$. The updates caused by the new phrase `ar` are highlighted in blue. First, to find the longest previous factor we have to descend to the child with label `a`, then we check whether the child with label `ar` has minmax $< c$. In this case, it was not less than c (minmax $= +\infty$), so we stopped the search and output the new phrase `ar`. Then all suffixes j with $1 \le j \le 6$ undergo an update starting from the corresponding leaf; e.g., leaves 5 and 6 and corresponding ancestors get updated to some non-initial value, whereas inner nodes with label `abar`, `alabar`, `bar` and `labar` change real to 1 because $j + sd(v) - 1 \le i + \ell$.

7 The Greedier Parser: Combining Greedy with Minmax

We now combine the ideas of the Greedy and the Minmax parsers, using the enhanced suffix tree to consider only longest admissible phrases. Consider when the Minmax algorithm stops in a node v and returns $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$. It did not descend to the next child u because $\mathsf{minmax}(u) = c$, i.e., every occurrence of $L(v)$ seen so far has a value c somewhere in the C-array. However, it is possible that in one of these occurrences, the position of this c is after $L(v)$; in other words, that we could have gone down the edge some way towards u.

To check this, we will use the D-array from Sec. 4, in addition to the C-array and the enhanced suffix tree. Let v and u be as before, i.e., v is parent of u, $\mathsf{minmax}(v) < c$, $\mathsf{minmax}(u) \geq c$, $L(v)$ is a prefix of $T[j\,..]$ and $T[j + sd(v)]$ is the first character of the label of (v, u). Let d be the maximum value of $D[k]$ for some occurrence of $L(u)$ that we have already processed, so d is the largest distance from the start of an occurrence of $L(u)$ to the next c in the C-array. We return $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$, as before, if $d \leq sd(v)$, and $(k, D[k], T[i+D[k]])$, where k is a leaf in u's subtree with $D[k] = d$, otherwise. As for updating the annotations, if node v has $\mathsf{minmax}(v) = c$ and some $\mathsf{txtpos}(v) = x$, then, when performing the traversal from $leaf_j$ up to the root, we want to change $\mathsf{txtpos}(v) \leftarrow j$ if $D[j] > D[\mathsf{txtpos}(v)]$. It is easy to see that the Greedier algorithm now returns the longest admissible phrases; otherwise, it works similarly to the Minmax algorithm. The time complexity increases to $O(z'n^2 \log n) \subseteq O(n^3 \log n)$, because we need dynamic RMQs on array D as well, which undergoes updates at arbitrary positions.

8 Experiments

We implemented the BAT-LZ parsing algorithms in C, and ran our experiments on an AMD EPYC 7343, with 32 cores at 1.5 GHz, with a 32 MB cache and 1 TB of RAM. We used the repetitive files from Pizza&Chili (`http://pizzachili.dcc.uchile.cl`) and compared the number of phrases produced by BAT-LZ using different maximum values c for the chains, with the number of phrases produced by LZ (i.e., with no limit c). We used a classic LZ implementation [26] where the source of each phrase is its lexicographically closest suffix.[3]

As a reference point, we also implemented two simple baselines that obtain a BAT-LZ parse. The first, called BAT-LZ1, runs the classic LZ parse and then cuts the phrases at the points where the chain lengths reach $c + 1$. Since the symbol becomes explicit, its chain length becomes zero and the chain lengths of the symbols referencing it decrease by $c + 1$. We should then find the new positions that reach $c + 1$, and so on. It is not hard to see that this laborious postprocessing can be simulated by just adding, to the original z value of LZ, the number of positions i where $C[i] \bmod (c + 1) = 0$.

The second baseline, BAT-LZ2, is slightly stronger: when it detects that it has produced a text position exceeding the maximum c, it cuts the phrase there (making the symbol explicit), and restarts the LZ parse from the next position. This gives the chance of choosing a better phrase starting after the cut, unlike BAT-LZ1, which maintains the original source.

Despite some optimizations, our Greedy BAT-LZ parser consistently reaches the $\Theta(\log^3 n)$ time complexity per text symbol, making it run at about 3 MB per minute. The Greedier and the Minmax parsers, despite their cubic worst-case time complexity, run at a similar pace: 1.9–4.7 MB per minute: our upper bound is utterly pessimistic, and perhaps not tight.

[3] It is likely that using the variant called "rightmost LZ parse", which chooses the rightmost source, gives better results because it tends to distribute the uses of the sources more uniformly. Such a parse seems to be nontrivial to compute [4, 15], however, and we are not aware of practical implementations.

Table 1 Our repetitive text collections and some statistics: alphabet size σ, length n, number z of phrases in the LZ parse, average phrase length n/z, maximum chain length in our LZ parse, size g of a balanced grammar, and height h of that grammar.

File	σ	n	z	n/z	max c	g	h
`coreutils`	236	205,281,779	1,286,070	160	66	2,409,429	28
`kernel`	160	257,961,616	705,791	365	70	1,374,651	32
`einstein`	139	467,626,545	75,779	6,171	1,736	212,902	47
`leaders`	89	46,968,181	155,937	301	60	399,667	27
`para`	5	429,265,758	1,879,635	228	38	5,344,477	26
`influenza`	15	154,808,555	557,349	278	63	1,957,370	26

Table 1 shows the main characteristics of the collections chosen. We included two versioned software repositories (`coreutils` and `kernel`, where the versioning has a tree structure), two versioned documents (`einstein` and `leaders`, where the versioning has a linear structure), and two biological sequence collections (`para` and `influenza`, where all the sequences are pairwise similar). The average phrase length is in the range 160–365 and the maximum chain length of a symbol is in the range 38–70. The exception is `einstein`, which is extremely compressible and also has a very large c value.

As a point of comparison, the table also includes the grammar size and height obtained with a balanced version of RePair [35].[4] We modified the RePair grammar so as to remove the nonterminals that are referenced only once, inserting their right-hand side in that unique referencing place. The maximum grammar height is comparable with c as a measure of access cost in the grammar-compressed text. We can see that the height is considerably smaller than c, for the price of a weaker compression method.

Fig. 4 shows how the quotient between the number of phrases generated by the BAT-LZ parsers and by the optimal number of LZ phrases evolves as we allow longer chains. It can be seen that our Greedy BAT-LZ parser sharply outperforms the baselines in terms of compression performance. Our Greedy parser is, in turn, outperformed by Minmax, and Minmax is outperformed by our Greedier parser. The last one reaches a number of phrases that is only 1% over the optimal for c as low as 20–30, which is 0.7–1.1 times $\log_2 n$.

We also show in the figures the balanced grammar method, using the values of Table 1.[5] We can see that grammars are competitive, in some cases, with the simple baselines, but not with our new algorithms, which yield much better tradeoffs. The only exception to this analysis is `einstein`, which features a huge maximum c value of 1,736 and whose (extremely low) z value is approached only with c values near 700 using our BAT-LZ parsers. On this text, the balanced grammar offers an access time that is not achievable with our techniques.

Fig. 5 (left) zooms in the area where Greedier BAT-LZ reaches less than 10% extra space on top of standard LZ (excluding `einstein`).

[4] From `www.dcc.uchile.cl/gnavarro/software/repair.tgz`, directory `bal/`.

[5] For a fair comparison of space, we consider a tight space needed to support fast extraction: For each of the z phrases we count $\log_2 n$ bits to point to the source, $\log_2(n/z)$ bits for the length (as there are z lengths adding up to n; the cumulative sequence of lengths also allow finding the desired phrase using Elias-Fano codes [14, 16]), and 8 bits for the final symbol. For a grammar of size g and r symbols, we count $g \log_2 r$ bits for the right-hand sides, $g \log_2 n$ bits for the expansion lengths (cumulative on the right-hand sides to binary search them), and $r \log(g/r)$ bits to encode the rule lengths with Elias-Fano.

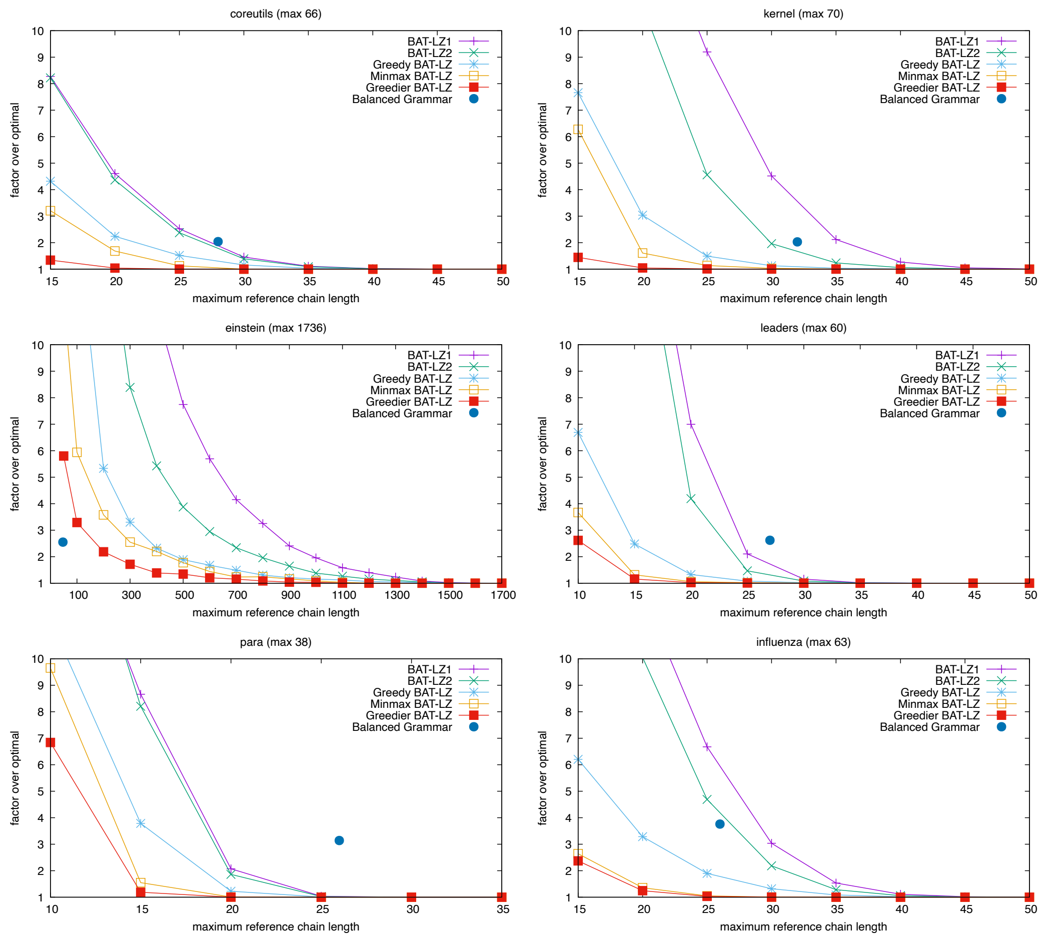


Figure 4 Overhead factor of number of BAT-LZ versus LZ phrases as a function of the maximum length c of a chain, for our different BAT-LZ parsers and a balanced grammar.

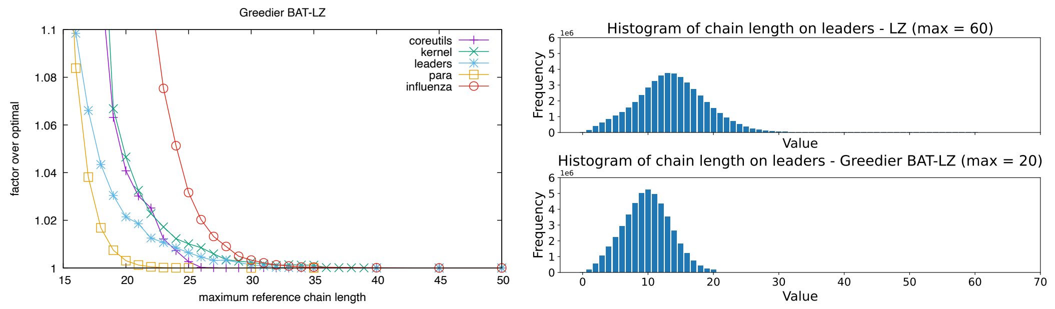


Figure 5 Left: detail of Fig. 4, for the Greedier BAT-LZ parser, focusing on the overheads below 10% over the LZ parse. Right: a comparison of histograms with shared x and y axis representing the chain length values on `leaders`; LZ on top and Greedier BAT-LZ with $c = 20$ on the bottom.

9 Discussion and Future Work

A first question is whether a Greedy BAT-LZ parsing can be produced in $o(n \log^3 n)$ time within linear space, either by solving our geometric problem faster or without recasting the parse into a geometric problem. This question seems to be answered in a very recent work, simultaneous with ours, that gives an $O(n \log \sigma)$-time greedy algorithm [2] based on simulating a suffix tree construction.[6] This algorithm is likely to be faster than ours in practice, but also to use much more space, which is relevant when compressing large repetitive texts. They also propose a parse similar to our BAT-LZ2, along with others that are incomparable to ours (in particular to Greedier, our best performing BAT-LZ parse).

Besides our reduction to a geometric problem being of independent interest, we believe that its flexibility can be exploited to compute more sophisticated parses in $O(n \log^3 n)$ time. For example, it might compute the Greedier parse if we extend the RMQ data structure to incorporate the additional optimization criterion (minmax of sources).

Other heuristics may also be of interest: there may be better ways to rank sources, other than their maximum chain length. Further, we have so far focused on reducing the worst case access time, but we might prefer to reduce the *average* access time. Our parsings do reduce it (Fig. 5 right), but this is just a side effect and has not been our main aim. So we pose as an open problem to efficiently build a leftward parse with bounded average reference chain length whose number of phrases is minimal, or in practice close to that of classical LZ.

Another intriguing line of work is to study the compression performance of BAT-LZ. An important result by Bannai et al. [2] shows that, letting g_{rl} be the size of the smallest run-length context-free grammar that generates a text T, there exists a BAT-LZ parse for T of size $O(g_{rl})$ if we let $c = \Theta(\log n)$ with some convenient multiplying constant. This bound is nearly optimal, because existing bounds [51] forbid the existence of BAT-LZ parses of size $O(g)$ – where $g \geq g_{rl}$ is the size of the smallest context-free grammar – with access time $c = O(\log^{1-\epsilon} n)$ for a constant $\epsilon > 0$. A relevant question is whether there is a BAT-LZ parse of size $O(z)$ – where $z \leq g_{rl}$ is the size of the Lempel-Ziv parse of T – with $c = \Theta(\log n)$.

Finally, from an application viewpoint, it would be interesting to incorporate BAT-LZ in the construction of the LZ-index [34] and measure how much its time performance improves at the price of an insignificant increase in space. Obtaining an efficient bounded version of the LZ-End parsing described in the same article [34] is also an interesting problem since efficient parsings for unrestricted LZ-End have appeared only recently [29, 28].

References

1 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

2 Hideo Bannai, Mitsuru Funakoshi, Diptarama Hendrian, Myuji Matsuda, and Simon J. Puglisi. Height-bounded Lempel-Ziv encodings. *CoRR*, abs/2403.08209, 2024.

3 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.

4 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.

[6] They use a slightly modified definition of Lempel-Ziv parses, which has no explicit character at the end of the phrases. The precise consequences of this difference are not totally clear to us.

5 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.

6 Timothy M. Chan, Yakov Nekrich, Saladi Rahul, and Konstantinos Tsakalidis. Orthogonal point location and rectangle stabbing queries in 3-d. *Journal of Computational Geometry*, 13(1), 2022.

7 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the RAM, revisited. *Journal of Computational Geometry*, 9(2):45–66, 2018.

8 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

9 Gang Chen, Simon J. Puglisi, and William F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1:605–623, 2008.

10 Ferdinando Cicalese and Francesca Ugazio. On the complexity and approximability of bounded access Lempel Ziv coding. *CoRR*, abs/2403.15871, 2024. Submitted.

11 David R. Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.

12 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

13 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

14 P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.

15 Jonas Ellert, Johannes Fischer, and Max Rishøj Pedersen. New advances in rightmost Lempel-Ziv. In *Proc. 30th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 188–202, 2023.

16 R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

17 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE Computer Society, 1997.

18 J. Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

19 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

20 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 172–184, 2015.

21 Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.

22 Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):article 27, 2021.

23 Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. 23rd Data Compression Conference (DCC)*, pages 133–142, 2013.

24 D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

25 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 139–150, 2013.

26 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 189–200, 2013.

27 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lazy Lempel-Ziv factorization algorithms. *ACM Journal of Experimental Algorithmics*, 21(1):2.4:1–2.4:19, 2016.

28 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in compressed space. In *Proc. 27th Data Compression Conference (DCC)*, pages 350–359, 2017.

29 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in linear time. In *Proc. 25th Annual European Symposium on Algorithms (ESA)*, pages 53:1–53:14, 2017.

30 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-End parsing. In *Proc. 33rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2847–2866, 2022.

31 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.

32 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. 26th Data Compression Conference (DCC)*, pages 3–12, 2016.

33 Sebastian Kreft and Gonzalo Navarro. Lz77-like compression with fast random access. In *Proc. 20th Data Compression Conference (DCC)*, pages 239–248, 2010.

34 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

35 J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

36 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

37 M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, 2002.

38 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

39 J. Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

40 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016.

41 Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

42 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.

43 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

44 Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

45 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

46 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 15–26, 2011.

47 Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):article 4, 2007.

48 Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

49 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

50 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

51 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 247–258, 2013.

52 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE Computer Society, 1973.

Subsequences with Generalised Gap Constraints: Upper and Lower Complexity Bounds

Florin Manea
Computer Science Department and CIDAS, Universität Göttingen, Germany

Jonas Richardsen
Computer Science Department and CIDAS, Universität Göttingen, Germany

Markus L. Schmid
Humboldt-Universität zu Berlin, Berlin, Germany

Abstract

For two strings u, v over some alphabet A, we investigate the problem of embedding u into w as a subsequence under the presence of generalised gap constraints. A generalised gap constraint is a triple $(i, j, C_{i,j})$, where $1 \leq i < j \leq |u|$ and $C_{i,j} \subseteq A^*$. Embedding u as a subsequence into v such that $(i, j, C_{i,j})$ is satisfied means that if $u[i]$ and $u[j]$ are mapped to $v[k]$ and $v[\ell]$, respectively, then the induced gap $v[k+1..\ell-1]$ must be a string from $C_{i,j}$. This generalises the setting recently investigated in [Day et al., ISAAC 2022], where only gap constraints of the form $C_{i,i+1}$ are considered, as well as the setting from [Kosche et al., RP 2022], where only gap constraints of the form $C_{1,|u|}$ are considered.

We show that subsequence matching under generalised gap constraints is NP-hard, and we complement this general lower bound with a thorough (parameterised) complexity analysis. Moreover, we identify several efficiently solvable subclasses that result from restricting the interval structure induced by the generalised gap constraints.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Formal languages and automata theory

Keywords and phrases String algorithms, subsequences with gap constraints, pattern matching, fine-grained complexity, conditional lower bounds, parameterised complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.22

Funding *Florin Manea*: Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) in the Heisenberg programme, project number 466789228.
Markus L. Schmid: Supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) – project number 522576760 (gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 522576760).

1 Introduction

For a string $v = v_1 v_2 \dots v_n$, where each v_i is a single symbol from some alphabet Σ, any string $u = v_{i_1} v_{i_2} \dots v_{i_k}$ with $k \leq n$ and $1 \leq i_1 < i_2 < \dots < i_k \leq n$ is called a *subsequence* (or *scattered factor* or *subword*) of v (denoted by $u \preceq v$). This is formalised by the *embedding* from the positions of u to the positions of v, i. e., the increasing mapping $e : \{1, 2, \dots, k\} \to \{1, 2, \dots, n\}$ with $j \mapsto i_j$ (we use the notation $u \preceq_e v$ to denote that u is a subsequence of v via embedding e). For example, the string `abacbba` has among its subsequences `aaa`, `abca`, `cba`, and `ababba`. With respect to `aaa`, there exists just one embedding, namely $1 \mapsto 1$, $2 \mapsto 3$, and $3 \mapsto 7$, but there are two embeddings for `cba`.

This classical concept of subsequences is employed in many different areas of computer science: in formal languages and logics (e. g., piecewise testable languages [54, 55, 29, 30, 31, 47], or subword order and downward closures [26, 38, 37, 59]), in combinatorics on words [49, 23, 40, 39, 52, 44, 50, 51], for modelling concurrency [48, 53, 12], in database theory (especially event stream processing [4, 25, 60, 33, 34, 24]). Moreover, many classical algorithmic problems are based on subsequences, e. g., longest common subsequence [6] or shortest common supersequence [43] (see [3, 22] and the survey [36], for recent results on string problems concerned with subsequences). Note that the longest common subsequence problem, in particular, has recently regained substantial interest in the context of fine-grained complexity (see [10, 11, 1, 2]).

In this paper, we are concerned with the following special setting of subsequences recently introduced in [17]. If a string u is a subsequence of a string v via an embedding e, then this embedding e also induces $|u| - 1$ so-called *gaps*, i. e., the (possibly empty) factors $v_{e(i)+1}v_{e(i)+2}\dots v_{e(i+1)-1}$ of v that lie strictly between the symbols where u is mapped to. For example, $\mathtt{acb} \preceq_e \mathtt{abacbba}$ with e being defined by $1 \mapsto 1$, $2 \mapsto 4$, and $3 \mapsto 6$ induces the gaps $\mathtt{ba}$ and $\mathtt{b}$. We can now restrict the subsequence relation by adding *gap constraints* as follows. A string u is accompanied by $|u| - 1$ gap constraints $C_1, C_2, \dots, C_{|u|-1} \subseteq \Sigma^*$, and u is a valid subsequence of a string v under these gap constraints, if $u \preceq_e v$ for an embedding e that induces gaps from the gap constraints, i. e., the i^{th} gap is in C_i.

Such gap-constrained subsequences allow to model situations for which classical subsequences are not expressive enough. For example, if we model concurrency by shuffling together strings that represent threads on a single processor, then fairness properties of a scheduler usually imply that the gaps of these subsequences are not huge. Or assume that we compute an alignment between two strings by computing a long common subsequence. Then it is not desirable if roughly half of the positions of the common subsequence are mapped to the beginning of the strings, while the other half is mapped to the end of the strings, with a huge gap (say thousands of symbols) in between. In fact, an overall shorter common subsequence that does not contain such huge gaps seems to induce a more reasonable alignment (this setting is investigated in [3]). Another example is complex event processing: Assume that a log-file contains a sequence of events of the run of a large system. Then we might query this string for the situation that between some events of a job A only events associated to a job B appear (e. g., due to unknown side-effects this leads to a failure of job A). This can be modeled by embedding a string as a subsequence such that the gaps only contain symbols from a certain subset of the alphabet, i. e., the events associated to job B (such subsequence queries are investigated in [33, 34, 24]).

In [17], two types of gap constraints are considered: Length constraints $C = \{w \in \Sigma^* \mid \ell \leq |w| \leq k\}$, and regular constraints where C is just a regular language over Σ^*, as well as combinations of both. In a related paper, [35], the authors went in a slightly different direction, and were interested in subsequences appearing in bounded ranges, which is equivalent to constraining the length of the string occurring between the first and last symbol of the embedding, namely $v_{e(1)+1}v_{e(i)+2}\dots v_{e(m)-1}$. In this paper, we follow up on the work of [17, 35], but significantly generalise the concept of gap constraints. Assume that $u \preceq_e v$. Instead of only considering the gaps given by the images of two consecutive positions of u, we consider each string $v_{e(i)+1}v_{e(i)+2}\dots v_{e(j)-1}$ of v as a gap, where $i, j \in \{1, 2, \dots, |u|\}$ with $i < j$ (note that these general gaps also might contain symbols from v that correspond to images of e, namely $e(i+1), e(i+2), \dots, e(j-1)$). For example, $\mathtt{abac} \preceq_e \mathtt{baabbcaccab}$ with e defined by $1 \mapsto 2$, $2 \mapsto 5$, $3 \mapsto 7$ and $4 \mapsto 9$ induces the following gaps: The $(1,2)$-gap $\mathtt{ab}$, the $(2,3)$-gap $\mathtt{c}$, the $(3,4)$-gap $\mathtt{c}$, the $(1,3)$-gap $\mathtt{abbc}$, the $(2,4)$-gap $\mathtt{cac}$, and the $(1,4)$-gap $\mathtt{abbcac}$. In this more general setting, we can now add gap-constraints in an analogous way as before. For example, the gap constraint $C_{2,4} = \{\mathtt{a}, \mathtt{c}\}^*$ for the $(2,4)$-gap, the gap constraint $C_{1,4} = \{w \in \Sigma^* \mid 3 \leq |w| \leq 5\}$ for the $(1,4)$-gap and the gap constraint

$C_{2,3} = \{\mathtt{c}^n \mid n \geq 1\}$ for the $(2,3)$-gap. Under these gap-constraints, the embedding e defined above is not valid: The gap constraints $C_{2,4}$ and $C_{2,3}$ are satisfied, but the $(1,4)$-gap $\mathtt{abbcac}$ is too long for gap constraint $C_{1,4}$. However, changing $4 \mapsto 9$ into $4 \mapsto 8$ yields an embedding that satisfies all gap constraints.

Our Contribution. We provide an in-depth analysis of the complexity of the *matching problem* associated with the setting explained above, i. e., for given strings u, v and a set $\mathcal{C}$ of generalised gap-constraints for u, decide whether or not $u \preceq_e v$ for an embedding e that satisfies all constraints in $\mathcal{C}$. We concentrate on two different kinds of constraints: semilinear constraints of the form $\{w \in \Sigma^* \mid |w| \in S\}$, where S is a semilinear set, and regular constraints.

In general, this matching problem is NP-complete for both types of constraints (demonstrating a stark contrast to the simpler setting of gap constraints investigated in [17, 35]), and this even holds for binary alphabets and if each semilinear constraint has constant size, and also if every regular constraint is represented by an automaton with a constant number of states. On the other hand, if the number of constraints is bounded by a constant, then the matching problem is solvable in polynomial-time, but, unfortunately, we obtain W[1]-hardness even if the complete size $|u|$ is a parameter (also for both types of constraints). An interesting difference in complexity between the two types of constraints is pointed out by the fact that for regular constraints the matching problem is fixed-parameter tractable if parameterised by $|u|$ and the maximum size of the regular constraints (measured in the size of a DFA), while for semilinear constraints this variant stays W[1]-hard.

We then show that structurally restricting the interval structure induced by the given constraints yields polynomial-time solvable subclasses. Moreover, if the interval structure is completely non-intersecting, then we obtain an interesting subcase for which the matching problem can be solved in time $O(n^\omega|\mathcal{C}|)$, where $O(n^\omega)$ is the time needed to multiply two $n \times n$ Boolean matrices. We complement this result by showing that an algorithm with running time $\mathcal{O}(|w|^g|\mathcal{C}|^h)$ with $g + h < 3$ would refute the strong exponential time hypothesis. While this is not a tight lower bound, we wish to point out that, due to the form of our algorithm, which boils down to performing $O(|\mathcal{C}|)$ matrix multiplications, a polynomially stronger lower bound would have proven that matrix multiplication in quadratic time is not possible.

Related Work. Our work extends [17, 35]. However, subsequences with various types of gap constraints have been considered before, mainly in the field of combinatorial pattern matching with biological motivations (see [7, 41, 42, 27] and [5, 13] for more practical papers).

2 Preliminaries

Let $\mathbb{N} = \{1, 2, \dots\}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. For $m, n \in \mathbb{N}_0$ let $[m, n] = \{k \in \mathbb{N}_0 \mid m \leq k \leq n\} = \{m, \dots, n\}$ and $[n] = [1, n]$. For some alphabet Σ and some length $n \in \mathbb{N}_0$ we define Σ^n as the set of all words of length n over Σ (with Σ^0 only containing the empty word ε). Furthermore $\Sigma^* := \bigcup_{n \in \mathbb{N}_0} \Sigma^n$ is the set of all words over Σ. For some $w \in \Sigma^*$, $|w|$ is the length of w, $w[i]$ denotes the i-th character of w and $w[i..j] := w[i] \dots w[j]$ is the substring of w from the i-th to the j-th character (where $i, j \in [|w|], i \leq j$).

We use deterministic and nondeterministic finite automata (DFA and NFA) as commonly defined in the literature; as a particularity, for the sake of having succinct representations of automata, we allow DFAs to be incomplete: given a state q of a DFA and a letter a, the transition from q with a may be left undefined, which means that the computations of the DFA on the inputs which lead to the respective transition are not-accepting. For a DFA or NFA A, we denote by $\mathsf{size}(A)$ its total size, and by $\mathsf{states}(A)$ its number of states. Note that if A is a DFA over alphabet Σ, then we have that $\mathsf{size}(A) = O(\mathsf{states}(A)|\Sigma|)$.

A subset $L \subseteq \mathbb{N}$ is called *linear*, if there are $m \in \mathbb{N}_0$ and $x_0 \in \mathbb{N}_0$, $x_1, \dots, x_m \in \mathbb{N}$, such that $L = L(x_0; x_1, \dots, x_m) := \{x_0 + \sum_{i=1}^{m} k_i x_i \,|\, k_1, \dots, k_m \in \mathbb{N}_0\}$. For $m = 0$, we write $L(x_0) = \{x_0\}$. We can assume without loss of generality that $x_i \neq x_j$ for $i \neq j, i, j \in [m]$. A set S is *semilinear*, if it is a finite union of linear sets (see also [46]).

We assume that each integer involved in the representation of a linear set fits into constant memory (see our discussion about the computational model at the end of this section). Consequently, we measure the size of a linear set $L = L(x_0; x_1, \dots, x_m)$ as $\mathsf{size}(L) = m + 1$, and the size of a semilinear set $S = L_1 \cup L_2 \cup \ldots \cup L_k$ is measured as $\mathsf{size}(S) = \sum_{i=1}^{k} \mathsf{size}(L_i)$. In other words, $\mathsf{size}(S)$ is the number of integers used for defining S.

Computational Model. For the complexity analysis of the algorithmic problems described in this paper we assume the *unit-cost RAM model with logarithmic word size* (see [15]). This means that for input size N, the memory words of the model can store $\log N$ bits. Thus, if we have input words of length N, they are over an alphabet which has at most $\sigma \leq N$ different characters, which we can represent using the *integer alphabet* $\Sigma = [\sigma]$. As such, we can store each character within one word of the model. Then, it is possible to read, write and compare single characters in one unit time.

Complexity Hypotheses. Let us consider the *Satisfiability problem for formulas in conjunctive normal form*, or CNF-SAT for short. Here, given a boolean formula F in conjunctive normal form, i.e., $F = \{c_1, \dots, c_m\}$ and $c_i \subseteq \{v_1, \dots, v_n, \neg v_1, \dots, \neg v_n\}$ for variables $v_1, \dots, v_n$, it is to be determined whether F is satisfiable. This problem was shown to be NP-hard [14]. By restricting $|c_i| \leq k$ for all $i \in [m]$ we obtain the problem of k-CNF-SAT. We will base our lower bound on the following algorithmic hypothesis:

▶ **Hypothesis 1** (Strong Exponential Time Hypothesis (SETH) [28]). *For any $\varepsilon > 0$, there exists a $k \in \mathbb{N}$, such that k-CNF-SAT cannot be solved in $\mathcal{O}(2^{n(1-\varepsilon)}\,\mathrm{poly}(m))$ time, where $\mathrm{poly}(n)$ is an arbitrary (but fixed) polynomial function.*

The *Clique problem*, CLIQUE, asks, given a graph G and a number $k \in \mathbb{N}$, whether G has a k-clique. Hereby, a k-clique is a subset of k pairwise adjacent vertices, i.e., there is an edge between any pair of vertices in the subset. Since CNF-SAT can be reduced to CLIQUE [32], the latter is also NP-hard.

The *k-Orthogonal Vectors problem*, k-OV, receives as inputs k sets $V_1, \dots, V_k$ each containing n elements from $\{0,1\}^d$ for some $d \in \omega(\log n)$, i.e., d-dimensional boolean vectors. The question is, whether one can select vectors $\vec{v}_i \in V_i$ for $i \in [k]$ such that the vectors are orthogonal: $\sum_{j=1}^{n} \prod_{i=1}^{k} \vec{v}_i[j] = 0$. It is possible to show the following lemma ([58, 57]):

▶ **Lemma 2.** *k-OV cannot be solved in $n^{k-\varepsilon}\,\mathrm{poly}(d)$ time for any $\varepsilon > 0$, unless SETH fails.*

This lemma will later form the basis for the conditional lower bound in the case of non-intersecting constraints.

3 Subsequences with Gap Constraints

An *embedding* is any function $e : [k] \to [\ell]$ for some $k, \ell \in \mathbb{N}$ with $k \leq \ell$, such that $e(1) < e(2) < \ldots < e(k)$ (note that this also implies that $1 \leq e(1)$ and $e(k) \leq \ell$). Let Σ be some alphabet. For a string $v = v_1 v_2 \ldots v_n$, where $v_i \in \Sigma$ for every $i \in [n]$, any string $u = v_{i_1}v_{i_2}\dots v_{i_k}$ with $k \leq n$ and $1 \leq i_1 < i_2 < \dots < i_k \leq n$ is called a *subsequence* (or, altternatively, *scattered factor* or *subword*) of v (denoted by $u \preceq v$). Every embedding $e : [k] \to [|v|]$ with $k \leq |v|$ *induces* the subsequence $u_e = v_{e(1)}v_{e(2)}\dots v_{e(k)}$ of v. If u is a subsequence of v induced by an embedding e, then we denote this by $u \preceq_e v$; we also say that an embedding e *witnesses* $u \preceq v$ if $u \preceq_e v$. When embedding a substring $u[s..t]$ for some $s, t \in [|u|]$, $s \leq t$, we use a partial embedding $e : [s, t] \to [n]$ and write $u[s..t] \preceq_e w$.

For example, the string $\mathtt{abacbba}$ has among its subsequences $\mathtt{aaa}$, $\mathtt{abca}$, $\mathtt{cba}$, and $\mathtt{ababba}$. With respect to $\mathtt{aaa}$, there exists just one embedding, namely $1 \mapsto 1$, $2 \mapsto 3$, and $3 \mapsto 7$, but there are two different embeddings for $\mathtt{cba}$.

Let $v \in \Sigma^*$ and let $e : [k] \to [|v|]$ be an embedding. For every $i, j \in [k]$ with $i < j$, the string v and the embedding e induces the (i, j)-*gap*, which is the factor of v that occurs strictly between the positions corresponding to the images of i and j under the embedding e, i. e., $\mathsf{gap}_{v,e}[i, j] = v[e(i) + 1..e(j) - 1]$. If v and e are clear from the context, we also drop this dependency in our notation, i. e., we also write $\mathsf{gap}[i, j]$.

As an example, consider $v = \mathtt{abcbcabcabac}$ and $u = \mathtt{acaba}$. There are several embeddings $e : [|u|] \to [|v|]$ that witness $u \preceq v$. Each such embedding also induces an (i, j)-gap for every $i, j \in [5]$ with $i < j$. For the embedding e with $e(1) = 1$, $e(2) = 3$, $e(3) = 6$, $e(4) = 7$, $e(5) = 11$ (that satisfies $u \preceq_e v$), some of these gaps are illustrated below (note that e is also indicated by the boxed symbols of v):

$$v = \boxed{\mathtt{a}}\,\underbrace{\overbrace{\mathtt{b}\,\boxed{\mathtt{c}}\,\overbrace{\mathtt{b}\,\mathtt{c}}^{\mathsf{gap}[2,3]}\,\boxed{\mathtt{a}}}^{\mathsf{gap}[1,4]}\,\boxed{\mathtt{b}}\,\overbrace{\mathtt{c}\,\mathtt{a}\,\mathtt{b}}^{\mathsf{gap}[4,5]}}_{\mathsf{gap}[1,5]}\,\boxed{\mathtt{a}}\,\mathtt{c}$$

A *gap-constraint* for a string $u \in \Sigma^*$ is a triple $C = (i, j, L)$ with $1 \leq i < j \leq |u|$ and $L \subseteq \Sigma^*$. A gap-constraint $C = (i, j, L)$ is also called an (i, j)-*gap-constraint*. The component L is also called the *gap-constraint language* of the gap-constraint (i, j, L). We say that a string v and some embedding $e : [|u|] \to [|v|]$ satisfies the gap-constraint C if and only if $\mathsf{gap}_{v,e}[i, j] \in L$.

As an example, let us define some gap-constraints for the string $u = \mathtt{acaba}$: $(1, 4, \Sigma^*)$, $(1, 5, \{w_1\,\mathtt{c}\,w_2\,\mathtt{c}\,w_3 \mid w_1, w_2, w_3 \in \Sigma^*\})$, $(2, 3, \{w \in \Sigma^* \mid |w| \geq 5\})$ and $(4, 5, \{w \in \Sigma^* \mid |w| \leq 4\})$. It can be easily verified that the gaps induced by the string v and the embedding e defined above (and illustrated by the figure) satisfy all of these gap constraints, except $(2, 3, \{w \in \Sigma^* \mid |w| \geq 5\})$ since $|\mathsf{gap}_{v,e}[2, 3]| = 2 < 5$. However, the embedding e' defined by $e'(1) = 1$, $e'(2) = 3$, $e'(3) = 9$, $e'(4) = 10$ and $e'(5) = 11$ is such that v and e' satisfy all of the mentioned gap-constraints (in particular, note that $|\mathsf{gap}_{v,e'}[2, 3]| = 5$ and that $\mathsf{gap}_{v,e'}[4, 5] = \varepsilon$).

A *set of gap-constraints* for u is a set $\mathcal{C}$ that, for every $i, j \in \mathbb{N}$ with $i < j$, may contain at most one (i, j)-gap-constraint for u. A string v and some embedding $e : [|u|] \to [|v|]$ satisfy $\mathcal{C}$ if v and e satisfy every gap-constraint of $\mathcal{C}$.

For strings $u, v \in \Sigma^*$ with $|u| \leq |v|$, and a set $\mathcal{C}$ of gap-constraints for u, we say that u is a $\mathcal{C}$-*subsequence of* v, if $u \preceq_e v$ for some embedding e such that v and e satisfy $\mathcal{C}$. We shall also write $u \preceq_{\mathcal{C}} v$ to denote that u is a $\mathcal{C}$-subsequence of v.

The Matching Problem. The central decision problem that we investigate in this work is the following matching problem, Match, for subsequences with gap-constraints:

- *Input:* Two strings $w \in \Sigma^*$ (also called text), with $|w| = n$, and $p \in \Sigma^*$ (also called pattern), with $|p| = m \leq n$, and a non-empty set $\mathcal{C}$ of gap-constraints.
- *Question:* Is p a $\mathcal{C}$-subsequence of w?

Obviously, for this matching problem it is vital how we represent gap-constraints (i, j, L), especially the gap-constraint language L. Moreover, since every possible language membership problem "$v \in L$?" can be expressed as the matching problem instance $w = \#v\#$, $p = \#\#$ and $\mathcal{C} = (1, 2, L)$, where $\# \notin \Sigma$, we clearly should restrict our setting to constraints (i, j, L) with sufficiently simple languages L. These issues are discussed next.

Types of Gap-Constraints. A gap-constraint $C = (i, j, L)$ (for some string) is a

- *regular constraint* if $L \in \text{REG}$. We represent the gap-constraint language of a regular constraint by a deterministic finite automaton (for short, DFA). In particular, $\mathsf{size}(C) = \mathsf{size}(L) = \mathsf{size}(A)$ and $\mathsf{states}(C) = \mathsf{states}(L) = \mathsf{states}(A)$.
- *semilinear length constraint* if there is a semi-linear set S, such that $L = \{w \in \Sigma^* \mid |w| \in S\}$. We represent the gap-constraint language of a semi-linear length constraint succinctly by representing the semilinear set S in a concise way (i. e., as numbers in binary encoding). In particular, $\mathsf{size}(C) = \mathsf{size}(L) = \mathsf{size}(S)$.

For a set $\mathcal{C}$ of gap constraints, let $\mathsf{size}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \mathsf{size}(C)$ and $\mathsf{gapsize}(\mathcal{C}) = \max\{\mathsf{size}(C) \mid C \in \mathcal{C}\}$. We have $\mathsf{size}(\mathcal{C}) \leq |\mathcal{C}|\mathsf{gapsize}(\mathcal{C})$.

Obviously, $\{v \in \Sigma^* \mid |v| \in S\}$ is regular for any semilinear set S. However, due to our concise representation, transforming a simple length constraint into a semilinear length constraints, or transforming a semilinear length constraint into a DFA representation may cause an exponential size increase.

By $\text{MATCH}_{\text{REG}}$ and $\text{MATCH}_{\text{SLS}}$ we denote the problem MATCH, where all gap constraints are regular constraints or semilinear length constraints, respectively.

For semilinear length constraints, we state the following helpful algorithmic observation.

▶ **Lemma 3.** *For a semilinear set S and an $n \in \mathbb{N}$, we can compute in time $\mathrm{O}(n\,\mathsf{size}(S))$ a data structure that, for every $x \in [n]$, allows us to answer whether $x \in S$ in constant time.*

A similar result can be stated for regular constraints.

▶ **Lemma 4.** *For a regular language $L \subseteq \Sigma^*$, given by a DFA A, accepting L, and a word $w \in \Sigma^*$, of length n, we can compute in time $\mathrm{O}(n^2 \log\log n + \mathsf{size}(A))$ a data structure that, for every $i, j \in [n]$, allows us to answer whether $w[i..j] \in L$ in constant time.*

▶ Remark 5. For every instance $(p, w, \mathcal{C})$ of $\text{MATCH}_{\text{SLS}}$, we assume that $\mathsf{size}(C) \leq |w|$ for every $C \in \mathcal{C}$. This is justified, since without changing the solvability of the $\text{MATCH}_{\text{SLS}}$ instance, any semilinear constraint defined by some semilinear set S can be replaced by a semilinear constraint defined by the semilinear set $S \cap \{0, 1, 2, \ldots, |w|\}$, which is represented by at most $|w|$ integers.

Moreover, we assume $\mathsf{size}(C) \leq |w|^2$ for every regular constraint C for similar reasons. More precisely, a regular constraint defined by a DFA M can be replaced by a constraint defined by a DFA M' that accepts $\{w' \in L(M) \mid w' \text{ is a factor of } w\}$. The number of states and, in fact, the size of such a DFA M' can be upper bounded by $|w|^2$. For instance, the DFA M' can be the trie of all suffixes of w (constructed as in [19]), with the final states used to indicate which factors of w are valid w.r.t. C.

We emphasise that working under these assumptions allows us to focus on the actual computation done to match constrained subsequences rather than on how to deal with over-sized constraints.

4 Complexity of the Matching Problem: Initial Results

As parameters of the problem Match, we consider the length $|p|$ of the pattern p to be embedded as a subsequence, the number $|\mathcal{C}|$ of gap constraints, the gap description size $\mathsf{gapsize}(\mathcal{C}) = \max\{\mathsf{size}(C) \mid C \in \mathcal{C}\}$, and the alphabet size $|\Sigma|$. Recall that, by assumption, $\mathcal{C}$ is always non-empty, which means that neither $|\mathcal{C}|$ nor $\mathsf{gapsize}(\mathcal{C})$ can be zero.

For any Match-instance, we have that $|\mathcal{C}| \leq |p|^2$. Consequently, if $|p|$ is constant or considered a parameter, so is $|\mathcal{C}|$. This means that an upper bound with respect to parameter $|\mathcal{C}|$ also covers the upper bound with respect to parameter $|p|$, and a lower bound with respect to parameter $|p|$ also covers the lower bound with respect to parameter $|\mathcal{C}|$. Consequently, it is always enough to just consider at most one of these parameters.

For all possible parameter combinations, we can answer the respective complexity for Match$_{\mathrm{REG}}$ and Match$_{\mathrm{SLS}}$ (both when the considered parameters are bounded by a constant, or treated as parameters in terms of parameterised complexity).

From straightforward brute-force algorithms, we can conclude the following upper bounds.

▶ **Theorem 6.** *Match$_{\mathrm{REG}}$ and Match$_{\mathrm{SLS}}$ can be solved in polynomial time for constant $|\mathcal{C}|$. Moreover, Match$_{\mathrm{REG}}$ is fixed parameter tractable for the combined parameter $(|p|, \mathsf{gapsize}(\mathcal{C}))$.*

These upper bounds raise the question whether Match$_{\mathrm{REG}}$ and Match$_{\mathrm{SLS}}$ are fixed parameter tractable for the single parameter $|p|$, or whether Match$_{\mathrm{SLS}}$ is at least also fixed parameter tractable for the combined parameter $(|p|, \mathsf{gapsize}(\mathcal{C}))$, as in the case of Match$_{\mathrm{REG}}$. Both these questions can be answered in the negative by a reduction from the Clique problem.

For the k-Clique problem, we get an undirected graph $G = (V, E)$ and a number $k \in [|V|]$, and we want to decide whether there is a clique of size at least k, i. e., a set $K \subseteq V$ with $|K| \geq k$ and, for every $u, v \in K$ with $u \neq v$, we have that $\{u, v\} \in E$. It is a well-known fact that k-Clique is W[1]-hard. We will now sketch a parameterised reduction from k-Clique to $|p|$-Match$_{\mathrm{SLS}}$ and to $|p|$-Match$_{\mathrm{REG}}$.

Let $G = (V, E)$ with $|V| = n$ be a graph represented by its adjacency matrix $A = (a_{i,j})_{1 \leq i,j \leq n}$ (we assume that $a_{i,i} = 1$ for every $i \in [n]$), and let $k \in [|V|]$. It can be easily seen that G has a k-clique, if the $k \times k$ matrix containing only 1's is a principal submatrix of A, i.e., a submatrix where the set of remaining rows is the same as the set of remaining columns. This can be described as embedding $p = 01^{k^2}0$ as a subsequence into $w = 0a_{1,1}a_{1,2} \cdots a_{1,n}a_{2,1} \cdots a_{2,n} \cdots a_{n,1} \cdots a_{n,n}0$. However, the corresponding embedding e must be such that the complete i^{th} (1^k)-block of p is embedded in the same $a_{s_i,1}a_{s_i,2} \cdots a_{s_i,n}$ block of w, for some s_i. Furthermore, for every $i \in [k]$, the first 1 of the i^{th} (1^k)-block must be mapped to the $(s_1)^{\text{th}}$ 1 of $a_{s_i,1}a_{s_i,2} \cdots a_{s_i,n}$, the second 1 of the i^{th} (1^k)-block must be mapped on the $(s_2)^{\text{th}}$ 1 of $a_{s_i,1}a_{s_i,2} \cdots a_{s_i,n}$, and so on. In other words, $1 \leq s_1 < s_2 < \ldots < s_k \leq n$ are the rows and columns where we map the "all-1"-submatrix; thus, $\{v_{s_1}, v_{s_2}, \ldots, v_{s_k}\}$ is the clique. Obviously, we have to use the semilinear length constraints to achieve this synchronicity.

We first force $e(1) = 1$ and $e(k^2 + 2) = n^2 + 2$ by constraint $(1, k^2 + 2, L(n^2))$. In the following, we use $(i, j)_k$ and $(i, j)_n$ to refer to the position of the entries in the i-th row and j-th column of the flattened matrices in p or w respectively (e. g. $w[(i, j)_n] = a_{ij}$). In order to force that $e((i, i)_k) = (s_i, s_i)_n$ for every $i \in [k]$ and some $s_i \in [n]$, we use constraints $(1, (i, i)_k, L(0; n + 1))$, $i \in [k]$ (i. e., the first 0 of p is mapped to the first 0 of w, and then we allow only multiples of $n + 1$ between the images of $(i, i)_k$ and $(i + 1, i + 1)_k$). Next, we establish the synchronicity between the columns by requiring that the gap between $e((i, j)_k)$

and $e((i+1,j)_k)$ has a size that is one smaller than a multiple of n, which is done by constraints $((i,j)_k, (i+1,j)_k, L(n-1;n))$, $i \in [k-1]$, $j \in [k]$. Finally, the constraints $((i,1)_k, (i,k)_k, \{0\} \cup [n-1])$, $i \in [k]$, force all $e((i,1)_k), e((i,2)_k), \dots, e((i,k)_k)$ into the same block $a_{s_i,1}a_{s_i,2}\cdots a_{s_i,n}$. Note that in order to show the last step, we also have to argue with the previously defined constraints for synchronising the columns (more precisely, we have to show that the $e((i,1)_k), \dots, e((i,k)_k)$ cannot overlap from one row in the next one).

This is a valid reduction from k-Clique to $|p|$-Match$_{\text{SLS}}$ (note that $|p| = k^2+2$). We can strengthen the reduction in such a way that all constraints have even constant size (note that the constraints $((i,1)_k, (i,k)_k, \{0\} \cup [n-1])$ are the only non-constant sized constraints, since $\{0\} \cup [n-1]$ is a semilinear set of size n). To this end, we observe that for fixed s_1 and s_k, all gap sizes $e((i,k)_k) - e((i,1)_k)$ are the same, independent from i, namely $s_k - s_1$. Thus, we can turn the reduction into a Turing reduction by guessing this value $d = s_k - s_1$ and then replace each *non-constant* $((i,1)_k, (i,k)_k, \{0\} \cup [n-1])$ by $((i,1)_k, (i,k)_k, L(d))$.

In order to obtain a reduction to Match$_{\text{REG}}$, we can simply represent all semilinear constraints as regular constraints. Obviously, the corresponding DFAs are not of constant size anymore. In summary, this yields the following result.

▶ **Theorem 7.** *Match$_{\text{SLS}}$ parameterised by $|p|$ is* W[1]*-hard, even for constant* $\mathsf{gapsize}(\mathcal{C})$ *and binary alphabet Σ, and Match$_{\text{REG}}$ parameterised by $|p|$ is* W[1]*-hard, even for binary alphabet Σ.*

The lower bound for Match$_{\text{REG}}$ is weaker, since the parameter $\mathsf{gapsize}(\mathcal{C})$ is not constant in the reduction. At least for a reduction from k-Clique, this is to be expected, due to the fact that Match$_{\text{REG}}$ is fixed parameter tractable with respect to the combined parameter $(|p|, \mathsf{gapsize}(\mathcal{C}))$. So this leaves one relevant question open: Can Match$_{\text{REG}}$ be solved in polynomial time, if the parameter $\mathsf{gapsize}(\mathcal{C})$ is bounded by a constant? We can answer this in the negative by a reduction from a variant of the SAT-problem, which we shall sketch next.

For the problem 1-in-3-3SAT, we get a set $A = \{x_1, x_2, \dots, x_n\}$ of variables and clauses $c_1, c_2, \dots, c_m \subseteq A$ with $|c_i| = 3$ for every $1 \le i \le m$. The task is to decide whether there is a subset $B \subseteq A$ such that $|c_i \cap B| = 1$ for every $1 \le i \le m$. For the sake of concreteness, we also set $c_j = \{x_{\ell_j,1}, x_{\ell_j,2}, x_{\ell_j,3}\}$ for every $j \in [m]$, and with $x_{\ell_j,1} < x_{\ell_j,2} < x_{\ell_j,3}$ for some order "$<$" on A.

We transform such an 1-in-3-3SAT-instance into two strings $u_A = (\mathtt{b}\,\#)^n(\mathtt{b}\,\#)^m$ and $v_A = (\mathtt{b}^2\,\#)^n(\mathtt{b}^3\,\#)^m$. For every $i \in [n]$, the i^{th} $\mathtt{b}$-factor of u_A and the i^{th} $\mathtt{b}^2$-factor of v_A are called *x_i-blocks*. Analogously, we denote the last m $\mathtt{b}$-factors of u_A and the last m $\mathtt{b}^3$-factors of v_A as *c_j-blocks* for $j \in [m]$.

Obviously, if $u_A \preceq_e v_A$ for some embedding $e : [|u_A|] \to [|v_A|]$, then the single $\mathtt{b}$ of u_A's x_i-block is mapped to either the first or the second $\mathtt{b}$ of v_A's x_i-block, and the single $\mathtt{b}$ of u_A's c_j-block is mapped to either the first or the second or the third $\mathtt{b}$ of v_A's c_j-block. Thus, the embedding e can be interpreted as selecting a set $B \subseteq A$ (where mapping u_A's x_i-block to the *second* $\mathtt{b}$ of v_A's x_i-block is interpreted as $x_i \in B$), and selecting either the first or second or third element of c_j (depending on whether u_A's c_j-block is mapped to the first, second or third $\mathtt{b}$ of v_A's c_j-block). We can now introduce a set of regular gap constraints that enforce the necessary synchronicity between B and the elements picked from the clauses: Assume that x_i is the p^{th} element of c_j. If e maps u_A's x_i-block to the second $\mathtt{b}$ of v_A's x_i-block, then e must map u_A's c_j-block to the p^{th} $\mathtt{b}$ of v_A's c_j-block, and if e maps u_A's x_i-block to the first $\mathtt{b}$ of v_A's x_i-block, then e must map u_A's c_j-block to the q^{th} $\mathtt{b}$ of v_A's c_j-block for some $q \in \{1,2,3\} \setminus \{p\}$. For example, if $x_{\ell_j,2} = x_i$ for some $i \in [n]$ and $j \in [m]$, i.e., the second element of c_j is x_i, then we add a regular gap constraint $(i', j', L_{i',j'})$, where i' and j' are the positions of u_A's x_i-block and u_A's c_j-block, and $L_{i',j'} = \{\mathtt{b}\,w\#, \#w\#\,\mathtt{b}, \mathtt{b}\,w\#\,\mathtt{b}\,\mathtt{b} \mid w \in \{\mathtt{b}, \#\}^*\}$. If e maps u_A's x_i-block to the second $\mathtt{b}$ of v_A's x_i-block, then the gap between positions i' and j' must start with $\#$; thus, due to the gap constraint, it must be of the form $\#w\#\,\mathtt{b}$, which is only possible if e maps u_A's c_j-block to the second $\mathtt{b}$ of v_A's c_i-block. If e maps u_A's x_i-block to the first $\mathtt{b}$ of v_A's x_i-block, then the gap must be of the form $\mathtt{b}\,w\#$ or $\mathtt{b}\,w\#\,\mathtt{b}\,\mathtt{b}$, which means that e must map u_A's c_j-block to the first or third $\mathtt{b}$ of v_A's c_i-block. Similar gap constraints can be defined for the case that x_i is the first or third element of c_j. Hence, we can define gap constraints such that there is an embedding $e : [|u_A|] \to [|v_A|]$ with $u_A \preceq_e v_A$ and e satisfies all the gap constraints if and only if the 1-in-3-3SAT-instance is positive. Independent of the actual 1-in-3-3SAT-instance, the gap languages can be represented by DFAs with at most 8 states. This shows the following result.

▶ **Theorem 8.** *MATCH*$_{\text{REG}}$ *is* NP*-complete, even for binary alphabet* Σ *and with gap-constraints that can be represented by* DFA*s with at most* 8 *states.*

5 Complexity of the Matching Problem: A Finer Analysis

Two representations of constraints. We start by defining two (strongly related) natural representations of the set of constraints which is given as input to the matching problem. Intuitively, both these representations facilitate the understanding of a set of constraints. Then, we see how these representations can be used to approach the MATCH problem.

The Interval Structure of Sets of Constraints. For a constraint $C = (a, b, L)$ we define interval$(C) := [a, b-1]$. If we have another constraint $C' = (a', b', L')$ (with $a \neq a'$ or $b \neq b'$), we say that C is contained in C', written $C \sqsubset C'$, if interval$(C) \subsetneq$ interval(C'). Because this order is derived from the inclusion order $\subsetneq$, it is also a strict partial order. We denote the corresponding covering relation with $\sqsubset\!\!\cdot$. Furthermore, we say that C and C' intersect, if interval$(C) \cap$ interval$(C') \neq \emptyset$, and they are not comparable w.r.t. $\sqsubset$, i.e., neither of them contains the other. Importantly, because we have $b \notin$ interval(C), in the case $b = a'$, the constraints C and C' do not intersect.

The Graph Structure of Sets of Constraints. For a string p, of length m, and a set of constraints $\mathcal{C}$ on p, we define a graph $G_p = (V_p, E_p)$ as follows. The set of vertices V_p of G_p is the set of numbers $\{1, \ldots, m\}$, corresponding to the positions of p. We define the set of undirected edges E_p as follows: $E_p = \{(a, b) \mid (a, b, L) \in \mathcal{C}\} \cup \{(i, i+1) \mid i \in [m-1]\} \cup \{(1, m)\}$. Note that in the case when we have a constraint $C = (i, i+1, L)$, for some $i \in [m-1]$ and L (respectively, a constraint $C(1, m, L)$) we will still have a single edge connecting the nodes i and $i+1$ (respectively, 1 and m) as E_p is constructed by a set union of two sets, so the elements in the intersection of the two sets are kept only once in the result. Moreover, we can define a labelling function on the edges of G by the following three rules: label$((i, j)) = C$, if there exists $C \in \mathcal{C}$ with $C = (i, j, L)$; label$((i, i+1)) = (i, i+1, \Sigma^*)$, for $i \in [m-1]$, if there exists no $C \in \mathcal{C}$ such that $C = (i, i+1, L)$; and label$((1, m)) = (1, m, \{w \in \Sigma^* | |w| \geq m-2\})$, if there exists no $C \in \mathcal{C}$ such that $C = (1, m, L)$. Clearly, all respective labels can be expressed trivially both as regular languages or as semilinear sets. Intuitively, the edges of G which correspond to constraints of $\mathcal{C}$ are labelled with the respective constraints. The other edges have trivial labels: the label of the edges of the form $(i, i+1)$ express that, in an embedding of p in the string v (as required in MATCH), the embedding of position $i+1$ is to the right of the embedding of position i; the label of edge $(1, m)$ simply states that at least $m - 2$ symbols should occur between the embedding of position 1 and the embedding of position m, therefore allowing for the entire pattern to be embedded.

Further, it is easy to note that this graph admits a Hamiltonian cycle, which traverses the vertices $1, 2, \ldots, m$ in this order. In the following, we also define two-dimensional drawing of the graph G_p as a *half-plane arc diagram*: the vertices $1, 2, \ldots, m$ are represented as points on a horizontal line ℓ, with the edges $(i, i+1)$, for $i \leq m-1$ being segments of unit-length on that line, spanning between the respective vertices i and $i+1$; all other edges (i, j) are drawn as semi-circles, whose diameter is equal to the length of $(j - i)$, drawn in the upper half-plane with respect to the line ℓ. In the following, we will simply call this diagram arc-diagram, without explicitly recalling that all semicircles are drawn in the same half-plane with respect to ℓ. In this diagram associated to G_p, we say that two edges cross if and only if they intersect in a point of the plane which is not a vertex of G_p. See also [18] and the references therein for a discussion on this representation of graphs.

In Figure 1 we see an example: We have $p = \texttt{xyzyx}$ and $\mathcal{C} = \{C = (1, 3, L_1), C' = (1, 4, L_1), C'' = (3, 5, L_2)\}$. Thus, $|p| = 5$ and G_p has the vertices $\{1, \ldots, 5\}$ and the edges $(1, 3), (1, 4), (3, 5)$ corresponding to the constraints of $\mathcal{C}$, as well as the edges $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 5)$, $(1, 5)$ (note that the edges are undirected, and the trivial labels are omitted for the sake of readability). The figure depicts the arc diagram associated to this graph (with the semi-circles flattened a bit, for space reasons). In the interval representation of these constraints, we can see that C is contained in C' (as interval$(C) = [1, 2] \subsetneq$ interval$(C') = [1, 3]$). Furthermore, C' and C'' intersect ($[1, 3] \cap [3, 4] \neq \emptyset$), while C and C'' do not ($[1, 2] \cap [3, 4] = \emptyset$). Note that two constraints (such as C and C'' in our example) might not intersect (according to the interval representation), although the edges that correspond to them in the graph representation share a common vertex; in particular, two constraints intersect if and only if the corresponding edges cross.

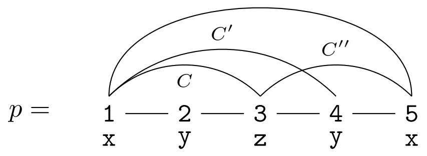


Figure 1 Relations between constraints.

The two representations defined above are clearly very strongly related. However, the graph-representation allows us to define a natural structural parameter for subsequences with constraints, while the interval-representation will allow us to introduce a class of subsequences with constraints which can be matched efficiently.

In the following, by $\textsc{Match}_{\mathcal{L},\mathcal{G}}$ we denote the problem $\textsc{Match}$, where all gap constraints are from the class of languages $\mathcal{L}$, with $\mathcal{L} \in \{\text{REG}, \text{SLS}\}$, and the graphs corresponding to the input gap constraints are all from the class $\mathcal{G}$.

Vertex separation number and its relation to Match. Given a linear ordering $\sigma = (v_1, \ldots, v_m)$ of the vertices of a graph G with m vertices, the vertex separation number of σ is the smallest number s such that, for each vertex v_i (with $i \in [m]$), at most s vertices of $v_1, \ldots, v_{i-1}$ have some v_j, with $j \geq i$, as neighbour. The vertex separation number $\text{vsn}(G)$ of G is the minimum vertex separation number over all linear orderings of G. The vertex separation number was introduced in [20] (see also [21] and the references therein) and was shown (e.g., in [9]) to be equal to the pathwidth of G.

Let us briefly overview the problem of computing the vertex separation number of graphs. Firstly, we note that checking, given a graph G with n vertices and a number k, whether $\mathrm{vsn}(G) \leq k$ is NP-complete, as it is equivalent to checking whether the pathwidth of G is at most k. We can show that this problems remains intractable even when we restrict it to the class of graphs with a Hamiltonian cycle, and this cycle is given as input as well.

However, this problem is linear fixed parameter tractable w.r.t. the parameter k: deciding, for a given graph G with n vertices and a constant number k, if $\mathrm{vsn}(G) \leq k$ and, if so, computing a linear ordering of the vertices with vertex separation number at most k can be solved in $O(n)$ time, where the constant hidden by the O-notation depends superexponentially on k. This follows from the results of [9], where the relation between pathwidth and vertex separation number is established, and [8], where it is shown that, for constant k, one can check in linear time $O(n)$ if a graph with n vertices has pathwidth at most k, and, if so, produce a path decomposition of that graph of width at most k.

For a constant k, let $\mathcal{V}_k$ be the class of all graphs G with $\mathrm{vsn}(G) \leq k$. We can show the following meta-theorem.

▶ **Theorem 9.** *Let $k \geq 1$ be a constant integer and let $\mathcal{L} \in \{\mathrm{SLS}, \mathrm{REG}\}$. Then, $\textsc{Match}_{\mathcal{L},\mathcal{V}_k}$ can be solved in polynomial time: $O(m^2 n^{k+1})$, in the case of SLS-constraints, and $O(m^2 n^{k+1} + m^2 n^2 \log\log n)$, in the case of REG-constraints. Moreover, $\textsc{Match}_{\mathcal{L},\mathcal{V}_k}$ parameterised by k is W[1]-hard.*

Due to space restrictions, we only sketch the proof. The matching algorithm implements a dynamic programming approach. We choose an ordering of the vertices of the graph representing $\mathcal{C}$, with vsn at most k. These vertices are, in fact, positions of p, so we find, for q from 1 to m, embeddings for the first q positions of this ordering in w, such that all the constraints involving only these positions are fulfilled. Given that the vsn of the respective ordering is bounded by k, we can compute efficiently the embeddings of the first q positions by extending the embeddings for the first $q-1$ positions, as, when considering a new position of our ordering, and checking where it can be embedded, only k of the previously embedded positions are relevant. As such, the embeddings of the first $q-1$ positions of the ordering, which are relevant for computing the embeddings of its first q positions, can be represented using an $O(n^k)$ size data-structure, and processed in $O(n^k \operatorname{poly}(n, m))$ time. This leads to a polynomial time algorithm, with a precise runtime as stated above. The lower bound follows from the reduction showing Theorem 7.

Non-intersecting constraints and Match. We have shown that $\textsc{Match}$ can be solved efficiently if the input gap constraints are represented by graphs with bounded vsn. However, while this condition is sufficient to ensure that $\textsc{Match}$ can be solved efficiently (as long as the constraint-languages are in P), it is not necessary. We will exhibit in the following a class $\mathcal{H}$ of gap constraints, which contains graphs with arbitrarily large vsn, and for which $\textsc{Match}_{\mathcal{L},\mathcal{H}}$ can be solved in polynomial time, for $\mathcal{L} \in \{\mathrm{REG}, \mathrm{SLS}\}$.

More precisely, in the following, we will consider the class $\mathcal{H}$ of non-intersecting gap constraints. That is, we consider $\textsc{Match}$ where the input consists of two strings $v \in \Sigma^*$ and $p \in \Sigma^*$ and a non-empty set $\mathcal{C}$ of gap constraints, where for any $C, C' \in \mathcal{C}$ we have that $\mathrm{interval}(C) \cap \mathrm{interval}(C') \in \{\mathrm{interval}(C), \mathrm{interval}(C'), \emptyset\}$. It is immediate that $\mathcal{H}$, the class of non-intersecting gap constraints, can be described as the class of gap constraints which are represented by outerplanar graphs which have a Hamiltonian cycle: the arc diagram constructed for these gap constraints is already outerplanar: if $C = (a, b, L)$ and $C' = (a', b', L')$ are two non-intersecting constraints of some set of constraints $\mathcal{C}$, then, in the graph representation of this set of constraints based on arc diagrams, the edges (a, b) and (a', b') do not cross (although they might share a common vertex).

Moreover, an outerplanar graph which admits a Hamiltonian cycle can be represented canonically as an arc diagram of a set of non-intersecting gap constraints. It is a folklore result that if an outerplanar graph has a Hamiltonian cycle then the outer face forms its unique Hamiltonian cycle. Moreover, every drawing of a graph in the plane may be deformed into an arc diagram without changing its number of crossings [45], and, in the case of outerplanar graphs this means the following. For an outerplanar graph G, we start with a drawing of G witnessing its outerplanarity. Assume that, after a potential renaming, there exists a traversal of the Hamiltonian cycle of the outerplanar graph (i.e., of its outer face) which consists of the vertices $1, 2, \ldots, n$, in this order. We simply reposition these vertices, in the same order $1, 2, \ldots, n$, on a horizontal line ℓ, such that consecutive vertices on the line are connected by edges of length 1, and then the edge $(1, n)$ is deformed so that it becomes a semicircle of diameter $n-1$ connecting the respective vertices, in the upper half-plane w.r.t. ℓ. Further, each edge (a, b) is deformed to become a semicircle above the line of the vertices, whose diameter equals the distance between vertex a and vertex b. By the result of [45], the edges of this graph do not cross in this representation, as the initial graph was outerplanar. But the resulting diagram is, clearly, the arc diagram corresponding to a set of non-intersecting gap constraints.

Based on the above, we can make a series of observations. Firstly, as one can recognize outerplanar graphs in linear time [56], we can also decide in linear time whether a set of constraints is non-crossing. Secondly, according to [16], there are outerplanar graphs with arbitrarily large pathwidth, so with arbitrarily large vsn. This means that there are sets of non-intersecting gap constraints whose corresponding graph representations have arbitrarily large vsn. Thirdly, the number of constraints in a set of non-intersecting gap constraints is linear in the length of the string constrained by that set (as the number of edges in an outerplanar graph with n vertices is at most $2n-3$).

We can show the following theorem (here we just sketch the proof, and only in the case of SLS-constraints, as the REG-constraints case is identical.

▶ **Theorem 10.** *$\textsc{Match}_{\mathrm{SLS},\mathcal{H}}$ can be solved in time $O(n^\omega K)$ and $\textsc{Match}_{\mathrm{REG},\mathcal{H}}$ can be solved in time $O(n^\omega K + n^2 K \log\log n)$, where K is the number of constraints in the input set of constraints $\mathcal{C}$ and $O(n^\omega)$ is the time needed to multiply two boolean matrices of size $n \times n$.*

As said, we sketch the algorithm solving $\textsc{Match}_{\mathrm{SLS},\mathcal{H}}$ in the stated complexity. Assume that the input words are $w \in \Sigma^n$ and $p \in \Sigma^m$, and $\mathcal{C} = (C_1, \ldots, C_K)$, with $C_i = (a_i, b_i, L_i)$ for $i \in [K]$. Firstly, we add a constraint $C = (1, m, L(m-2, 1))$ to $\mathcal{C}$ if it does not contain any constraint having the first two components $(1, m)$. So, in the following, we will assume w.l.o.g. that such a constraint $(1, m, \cdot)$ always exists in $\mathcal{C}$. Moreover, the number of constraints in $\mathcal{C}$ is $O(m)$ (as the graph representation of $\mathcal{C}$ is outerplanar).

As a first phase in our algorithm, we build the data structures from Lemma 3. Hence, by Remark 5, after an $O(n^2 K)$-time preprocessing we can answer queries "is $w[i..j] \in L_k$?" in $O(1)$ time, for all $i, j \in [n], k \in [K]$.

After this, the algorithm proceeds as follows. Because the set of constraints $\mathcal{C}$ is non-intersecting, one can build in linear time the Hasse-diagram of the set of intervals associated with the set of constraints $\mathcal{C}$ (w.r.t. the interval-inclusion relation), and this diagram is a tree, whose root corresponds to the single constraint of the form $(1, m, \cdot)$. Further, the algorithm uses a dynamic programming strategy to find matches for the constraints of $\mathcal{C}$ in a bottom-up fashion with respect to the Hasse-diagram of this set. The algorithm maintains the matches for each constraint $C = (a, b, L)$ as a Boolean $n \times n$ matrix, where the element on position (i, j) of that matrix is true if and only if there exists a way to embed $p[a..b]$ in $w[i..j]$, such that $p[a]$ is mapped to $w[i]$ and $p[b]$ to $w[j]$ in the respective embedding, and this embedding also fulfils C and all the constraints occurring in the sub-tree of root C in the Hasse-diagram. This matrix can be computed efficiently, by multiplying the matrices corresponding to the children of C (and a series of matrices corresponding to the unconstrained parts of $p[a..b]$). As the number of nodes in this tree is $O(K)$, the whole process of computing the respective matrices for all nodes of the tree requires $O(K)$ matrix multiplications, i.e., $O(n^\omega K)$ time in total. Finally, one needs to see if there is a match of $p[1..m]$ to some factor of v, which can be checked in $O(n^2)$ by simply searching in the matrix computed for the root of the diagram.

The following lower bound is shown by a reduction from 3-OV.

▶ **Theorem 11.** *For $\mathcal{L} \in \{\mathrm{REG}, \mathrm{SLS}\}$, then $\textsc{Match}_{\mathcal{L},\mathcal{H}}$, cannot be solved in $\mathcal{O}(|w|^g|\mathcal{C}|^h)$ time with $g + h < 3$, unless SETH fails.*

The reduction proving this hardness result works as follows. In 3-OV, we are given three sets $A = \{\vec{a}_1, \ldots, \vec{a}_n\}$, $B = \{\vec{b}_1, \ldots, \vec{b}_n\}$ and $C = \{\vec{c}_1, \ldots, \vec{c}_n\}$ with elements from $\{0,1\}^d$, and want to determine whether there exist $i^*, j^*, k^* \in [n]$, such that $\sum_{\ell=1}^{j} \vec{a}_{i^*}[\ell] \cdot \vec{b}_{j^*}[\ell] \cdot \vec{c}_{k^*}[\ell] = 0$. This is achieved by encoding our input sets over a constant size alphabet, via two functions C_p and C_w, into a pattern p and a text w, respectively, as well as a set of constraints $\mathcal{C}$, such that the answer to the 3-OV problem is positive if and only if p is a $\mathcal{C}$-subsequence of w. Basically, the encoding of each d-dimensional vector of A (respectively, B and C) is done via C_p (respectively, C_w), in such a way that (when no constraints are considered) $\mathsf{C}_p(\vec{v})$ is a subsequence of $\mathsf{C}_w(\vec{v}')$ for any $\vec{v}, \vec{v}' \in \{0,1\}^d$. Further, $\overline{\mathsf{C}}_p$ and $\overline{\mathsf{C}}_w$ are mirrored versions of these encodings (both for bits and vectors), where the order of the characters in the output is inverted. We can then use the original encoding for one part of the pattern and the text and the mirrored encoding for the other part. Then, we encode the set A in $p := \overline{\mathsf{C}}_p(\vec{a}_n) \ldots \overline{\mathsf{C}}_p(\vec{a}_1) \,\S\, \mathsf{C}_p(\vec{a}_1) \ldots \mathsf{C}_p(\vec{a}_n)$ and the sets B and C in $w := \overline{w}_0 \,\#\, \overline{\mathsf{C}}_w(\vec{c}_n) \ldots \overline{\mathsf{C}}_w(\vec{c}_1) \,\#\, \overline{w}_0 \,\S\, w_0 \,\#\, \mathsf{C}_w(\vec{b}_1) \ldots \mathsf{C}_w(\vec{b}_n) \,\#\, w_0$ (where $\overline{x}$ denotes the mirror image of x, and w_0 is a suitably choosen padding). To finalise our construction, we can define constraints which ensure that an embedding of p in w is possible if and only if there exist some i^*, j^*, k^* such that $\mathsf{C}_p(a_{i^*})$ is embedded in $\mathsf{C}_w(b_{j^*})$, and $\overline{\mathsf{C}}_p(a_{i^*})$ in $\overline{\mathsf{C}}_p(c_{k^*})$, while all the other strings $\overline{\mathsf{C}}_p(a_t)$ are embedded in the paddings. Moreover, additional constraints ensure that the simultaneous embedding of $\mathsf{C}_p(a_{i^*})$ in $\mathsf{C}_w(b_{j^*})$ and of $\overline{\mathsf{C}}_p(a_{i^*})$ in $\overline{\mathsf{C}}_p(c_{k^*})$ is only possible if and only if for each component $u \in [d]$ with $a_{i^*}[u] \neq 0$, we have that $b_{j^*}[u] = 0$ or $c_{k^*}[u] = 0$.

We conclude by noting that, while this is not a tight lower bound with respect to the upper bound shown in Theorem 10, finding a polynomially stronger lower bound (i.e., replacing in the statement of Theorem 11 the condition $g + h < 3$ with $g + h < 3 + \delta$, for some $\delta > 0$) would show that matrix multiplication in quadratic time is not possible, which in turn would solve a well-researched open problem. Indeed, the algorithm from Theorem 10 consists in a reduction from $\textsc{Match}_{\mathcal{L},\mathcal{H}}$ to $O(|C|)$ instances of matrix multiplication, for quadratic matrices of size $|w|$, so a better lower bound would mean that at least one of these multiplications must take more than quadratic time.

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78, 2015. `doi:10.1109/FOCS.2015.14`.

2 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, pages 39–51, 2014. `doi:10.1007/978-3-662-43948-7_4`.

3 Duncan Adamson, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Longest common subsequence with gap constraints. In *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, pages 60–76, 2023. doi:10.1007/978-3-031-33180-0_5.

4 Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex event recognition languages: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 7–10, 2017. doi:10.1145/3093742.3095106.

5 Johannes Bader, Simon Gog, and Matthias Petri. Practical variable length gap pattern matching. In *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 1–16, 2016. doi:10.1007/978-3-319-38851-9_1.

6 Ricardo A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.

7 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theor. Comput. Sci.*, 443:25–34, 2012. doi:10.1016/j.tcs.2012.03.029.

8 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.

9 Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.*, 209(1-2):1–45, 1998. doi:10.1016/S0304-3975(97)00228-4.

10 Karl Bringmann and Bhaskar Ray Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In *Proc. FSTTCS 2018*, volume 122 of *LIPIcs*, pages 40:1–40:16, 2018.

11 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proc. SODA 2018*, pages 1216–1235, 2018.

12 Sam Buss and Michael Soltys. Unshuffling a square is NP-hard. *J. Comput. Syst. Sci.*, 80(4):766–776, 2014. doi:10.1016/j.jcss.2013.11.002.

13 Manuel Cáceres, Simon J. Puglisi, and Bella Zhukova. Fast indexes for gapped pattern matching. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*, pages 493–504, 2020. doi:10.1007/978-3-030-38919-2_40.

14 Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi:10.1145/800157.805047.

15 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

16 David Coudert, Florian Huc, and Jean-Sébastien Sereni. Pathwidth of outerplanar graphs. *J. Graph Theory*, 55(1):27–41, 2007. doi:10.1002/JGT.20218.

17 Joel D. Day, Maria Kosche, Florin Manea, and Markus L. Schmid. Subsequences with gap constraints: Complexity bounds for matching and analysis problems. In *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, pages 64:1–64:18, 2022. doi:10.4230/LIPICS.ISAAC.2022.64.

18 Hristo N. Djidjev and Imrich Vrto. Crossing numbers and cutwidths. *J. Graph Algorithms Appl.*, 7(3):245–251, 2003. doi:10.7155/JGAA.00069.

19 Shiri Dori and Gad M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *Inf. Process. Lett.*, 98(2):66–72, 2006. doi:10.1016/J.IPL.2005.11.019.

20 John A. Ellis, Ivan Hal Sudborough, and Jonathan S. Turner. Graph separation and search number. In *Proc. 1983 Allerton Conf. on Communication, Control, and Computing*, 1983.

21 John A. Ellis, Ivan Hal Sudborough, and Jonathan S. Turner. The vertex separation and search number of a graph. *Inf. Comput.*, 113(1):50–79, 1994. doi:10.1006/INCO.1994.1064.

22 Pamela Fleischmann, Sungmin Kim, Tore Koß, Florin Manea, Dirk Nowotka, Stefan Siemer, and Max Wiedenhöft. Matching patterns with variables under simon's congruence. In *Reachability Problems - 17th International Conference, RP 2023, Nice, France, October 11-13, 2023, Proceedings*, pages 155–170, 2023. doi:10.1007/978-3-031-45286-4_12.

23 Dominik D. Freydenberger, Pawel Gawrychowski, Juhani Karhumäki, Florin Manea, and Wojciech Rytter. Testing k-binomial equivalence. In Multidisciplinary Creativity*, a collection of papers dedicated to G. Păun 65th birthday*, pages 239–248, 2015. available in CoRR abs/1509.00622.

24 André Frochaux and Sarah Kleest-Meißner. Puzzling over subsequence-query extensions: Disjunction and generalised gaps. In *Proceedings of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023), Santiago de Chile, Chile, May 22-26, 2023*, 2023. URL: https://ceur-ws.org/Vol-3409/paper3.pdf.

25 Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. Complex event recognition in the big data era: a survey. *VLDB J.*, 29(1):313–352, 2020. doi:10.1007/s00778-019-00557-w.

26 Simon Halfon, Philippe Schnoebelen, and Georg Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *Proc. LICS 2017*, pages 1–12, 2017.

27 Costas S. Iliopoulos, Marcin Kubica, M. Sohel Rahman, and Tomasz Walen. Algorithms for computing the longest parameterized common subsequence. In *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, pages 265–273, 2007. doi:10.1007/978-3-540-73437-6_27.

28 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

29 Prateek Karandikar, Manfred Kufleitner, and Philippe Schnoebelen. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015.

30 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *Proc. CSL 2016*, volume 62 of *LIPIcs*, pages 37:1–37:22, 2016.

31 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Log. Methods Comput. Sci.*, 15(2), 2019.

32 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

33 Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. Discovering event queries from traces: Laying foundations for subsequence-queries with wildcards and gap-size constraints. In *25th International Conference on Database Theory, ICDT 2022, 29th March-1st April, 2022 Edinburgh, UK*, 2022.

34 Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. Discovering multi-dimensional subsequence queries from traces - from theory to practice. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings*, pages 511–533, 2023. doi:10.18420/BTW2023-24.

35 Maria Kosche, Tore Koß, Florin Manea, and Viktoriya Pak. Subsequences in bounded ranges: Matching and analysis problems. In Anthony W. Lin, Georg Zetzsche, and Igor Potapov, editors, *Reachability Problems - 16th International Conference, RP 2022, Kaiserslautern, Germany, October 17-21, 2022, Proceedings*, volume 13608 of *Lecture Notes in Computer Science*, pages 140–159. Springer, 2022. doi:10.1007/978-3-031-19135-0_10.

36 Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Combinatorial algorithms for subsequence matching: A survey. In Henning Bordihn, Géza Horváth, and György Vaszil, editors, *Proceedings 12th International Workshop on Non-Classical Models of Automata and Applications, NCMA 2022, Debrecen, Hungary, August 26-27, 2022*, volume 367 of *EPTCS*, pages 11–27, 2022. doi:10.4204/EPTCS.367.2.

37 Dietrich Kuske. The subtrace order and counting first-order logic. In *Proc. CSR 2020*, volume 12159 of *Lecture Notes in Computer Science*, pages 289–302, 2020.

38 Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In *Proc. FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364, 2019.

39 Marie Lejeune, Julien Leroy, and Michel Rigo. Computing the k-binomial complexity of the Thue-Morse word. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 278–291, 2019.

40 Julien Leroy, Michel Rigo, and Manon Stipulanti. Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.*, 24(1.44):36 pp., 2017.

41 Chun Li and Jianyong Wang. Efficiently mining closed subsequences with gap constraints. In *SDM*, pages 313–322. SIAM, 2008.

42 Chun Li, Qingyan Yang, Jianyong Wang, and Ming Li. Efficient mining of gap-constrained subsequences and its various applications. *ACM Trans. Knowl. Discov. Data*, 6(1):2:1–2:39, 2012.

43 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

44 Alexandru Mateescu, Arto Salomaa, and Sheng Yu. Subword histories and Parikh matrices. *J. Comput. Syst. Sci.*, 68(1):1–21, 2004.

45 T.A.J. Nicholson. Permutation procedure for minimising the number of crossings in a network. *Proceedings of the Institution of Electrical Engineers*, 115:21–26(5), January 1968.

46 Rohit J Parikh. Language generating devices. *Quarterly Progress Report*, 60:199–212, 1961.

47 M. Praveen, Philippe Schnoebelen, Julien Veron, and Isa Vialard. On the piecewise complexity of words and periodic words. In *SOFSEM 2024: Theory and Practice of Computer Science - 48th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2024, Cochem, Germany, February 19-23, 2024, Proceedings*, pages 456–470, 2024. doi:10.1007/978-3-031-52113-3_32.

48 William E. Riddle. An approach to software system modelling and analysis. *Comput. Lang.*, 4(1):49–66, 1979. doi:10.1016/0096-0551(79)90009-2.

49 Michel Rigo and Pavel Salimov. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 601:47–57, 2015.

50 Arto Salomaa. Connections between subwords and certain matrix mappings. *Theoret. Comput. Sci.*, 340(2):188–203, 2005.

51 Philippe Schnoebelen and Julien Veron. On arch factorization and subword universality for words and compressed words. In *Combinatorics on Words - 14th International Conference, WORDS 2023, Umeå, Sweden, June 12-16, 2023, Proceedings*, pages 274–287, 2023. doi:10.1007/978-3-031-33180-0_21.

52 Shinnosuke Seki. Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.*, 418:116–120, 2012. doi:10.1016/J.TCS.2011.10.017.

53 Alan C. Shaw. Software descriptions with flow expressions. *IEEE Trans. Software Eng.*, 4(3):242–254, 1978. doi:10.1109/TSE.1978.231501.

54 Imre Simon. *Hierarchies of events with dot-depth one — Ph.D. thesis.* University of Waterloo, 1972.

55 Imre Simon. Piecewise testable events. In *Autom. Theor. Form. Lang., 2nd GI Conf.*, volume 33 of *LNCS*, pages 214–222, 1975.

56 Manfred Wiegers. Recognizing outerplanar graphs in linear time. In Gottfried Tinhofer and Gunther Schmidt, editors, *Graphtheoretic Concepts in Computer Science, International Workshop, WG '86, Bernried, Germany, June 17-19, 1986, Proceedings*, volume 246 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 1986. doi:10.1007/3-540-17218-1_57.

57 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

58 Virginia Vassilevska Williams. *On some fine-grained questions in algorithms and complexity*, pages 3447–3487. World Scientific, 2018. doi:10.1142/9789813272880_0188.

59 Georg Zetzsche. The complexity of downward closure comparisons. In *Proc. ICALP 2016*, volume 55 of *LIPIcs*, pages 123:1–123:14, 2016.

60 Haopeng Zhang, Yanlei Diao, and Neil Immerman. On complexity and optimization of expensive queries in complex event processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 217–228, 2014. doi:10.1145/2588555.2593671.

The *Rational* Construction of a Wheeler DFA

Giovanni Manzini
Dept. of Computer Science, University of Pisa, Italy

Alberto Policriti
Dept. of Mathematics, Computer Science and Physics, University of Udine, Italy

Nicola Prezza
Dept. of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Italy

Brian Riccardi
Dept. of Informatics, Systems and Communication, University of Milano-Bicocca, Italy

Abstract

Deterministic Finite Wheeler Automata are a natural generalisation to regular languages of the theory of compressed data structures originated by the introduction of the Burrows-Wheeler transform. Indeed, if we can find a Wheeler automaton recognizing a given language $\mathcal{L}$, such automaton can be used to design time and space efficient algorithms for representing and searching $\mathcal{L}$.

In this paper we introduce an alternative representation of Deterministic Wheeler Automata by showing that a natural map between strings and rational numbers in $\mathbb{Q}[0,1)$ can be extended to represent the automaton's states as *intervals* in $\mathbb{Q}[0,1)$. Using this representation it emerges a natural relationship between automata properties and some properties of real numbers. In addition, such representation enables us to formulate problems related to automata in a numerical setting. Although at the moment the numerical approach does not lead to time efficient algorithms, we believe this new perspective deserves further consideration.

As a further demonstration of the convenience of this new representation, we use it to provide a simple proof of an unexpected result on regular languages. More precisely, we compare the size of the smallest *Wheeler* automaton recognizing a given language $\mathcal{L}$ with respect to the size of the smallest automaton, possibly non-Wheeler, recognizing the same language. We show settings in which there can be an exponential gap between the two sizes, and we discuss the implications of this result on the problem of representing regular languages.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Formal languages and automata theory

Keywords and phrases String Matching, Deterministic Finite Automata, Wheeler languages, Graph Indexing, Co-lexicographical Sorting

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.23

Funding *Giovanni Manzini*: Funded by the Italian Ministry of Health, POS 2014-2020, project ID T4-AN-07, CUP I53C22001300001, by INdAM-GNCS Project CUP E53C23001670001 and by PNRR ECS00000017 Tuscany Health Ecosystem, Spoke 6 CUP I53C22000780001.
Alberto Policriti: project funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.4 - Call for tender No. 3138 of 16 December 2021, rectified by Decree n.3175 of 18 December 2021 of Italian Ministry of University and Research funded by the European Union - NextGenerationEU. **Award Number:** project code *CN*_00000033, Concession Decree No. 1034 of 17 June 2022 adopted by the Italian Ministry of University and Research, CUP G23C22001110007, Project title "National Biodiversity Future Center – NBFC".
Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Brian Riccardi: Received grants from the European Union's Horizon 2020 Research and Innovation Programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539, and from MUR 2022YRB97K, PINC, Pangenome INformatiCs: from Theory to Applications.

1 Introduction

A (deterministic) automaton is a simple version of a Turing Machine, operating just moving from left to right and using the tape just for reading a pattern (no writing). It encodes a (very simple, testing) algorithm, operating either accepting or rejecting its input pattern. A *Wheeler* automaton [1, 6] is an automaton equipped with a *total* order $\prec$ on its set of states and constrained by two simple axioms that, ultimately, cast an order on the entire collection of prefixes of accepted strings. As a matter of fact, a Wheeler automaton operates generalising to a *collection* of strings the computation performed to produce the Burrows-Wheeler transform of a string – that is, a linear and invertible permutation turning a string α into a highly compressible and searchable equivalent [4]. Many important, practical byproducts become available, starting with the ability to store and search the language accepted by a Wheeler automaton in little space and time (see [5]).

A remarkable property of regular languages, not present in other settings, is that a given language can be accepted by different automata with different properties. Hence, a language accepted by a Wheeler automaton can be accepted also by a non-Wheeler automaton. For an automaton $\mathcal{A}$ we define the *width of* $\mathcal{A}$, width($\mathcal{A}$), as the minimum *width of a partial order*[1] on $\mathcal{A}$ satisfying Wheeler axioms (details given below). Since (by definition) a Wheeler automaton $\mathcal{A}_w$ admits a total order, it is always width($\mathcal{A}_w$) = 1. In [5] it is shown that width($\mathcal{A}$) measures the "hardness" of representing and searching $\mathcal{A}$. For example, if width($\mathcal{A}$) = p the automaton can be represented in $\Theta(\log p)$ bits per transition and there exists a linear-space data structure solving regular expression matching in $O(p^2)$ time per matched character.

Let $\mathcal{L}$ be a language accepted by a minimal (in terms of number of states) Deterministic Finite Automaton (DFA) $\mathcal{D}$ as well as by a minimal *Wheeler DFA* (WDFA, see also Section 2) $\mathcal{D}_w$. Since either $\mathcal{D}$ or $\mathcal{D}_w$ can be used to represent the language $\mathcal{L}$ it is worthwhile to compare their effectiveness for this task. To this end, in this paper we consider the problem of bounding the size of $\mathcal{D}_w$ in terms of the size of $\mathcal{D}$ and of the width width($\mathcal{D}$). We prove that even for width($\mathcal{D}$) = 2, a minimal Wheeler automaton $\mathcal{D}_w$ can have exponentially more states than $\mathcal{D}$. This result has the immediate consequence that the Wheeler automata representation is not always the more effective: it can be algorithmically more convenient to deal with a non-Wheeler automaton with a small width rather than working with a (minimal) Wheeler automaton for the same language.

To provide a simple proof of the above result, we introduce a new general method for *representing* automaton $\mathcal{D}$ (and $\mathcal{D}_w$), proving that the co-lexicographic order of strings and the ordering of a Wheeler automaton can be conveniently presented using rational numbers and convex subsets of rational numbers in $[0, 1)$. This representation provides a different perspective on some properties of automata, highlighting their connection with established properties of real numbers. In addition, it suggests a new view for a number of problems that we illustrate and discuss, concluding by showing that some such problems can also be approached in an *arithmetic way*.

[1] The width of a partial order is the maximum length of any of its anti-chains.

2 Basics

Let $\Sigma = \{a_1, \dots, a_\sigma\}$ denote a finite ordered alphabet of size σ. We denote by Σ^* the set of finite strings over Σ. The character ϵ denotes the empty string. We assume that the elements of Σ^* are ordered according to the co-lexicographic (co-lex) order defined as follows: given $\alpha, \beta \in \Sigma^*$, we say that α is co-lex smaller than β ($\alpha \prec \beta$) if and only if α is a suffix of β or there exist $\gamma, \alpha', \beta' \in \Sigma^*$ and $a, b \in \Sigma$ with $a < b$ such that $\alpha = \alpha' a \gamma$ and $\beta = \beta' b \gamma$.

A Deterministic Finite-State Automaton (DFA) $\mathcal{A} = (Q, s, \delta, F)$ consists of a finite set of states Q, an initial state $s \in Q$, a set of final states $F \subseteq Q$, and a transition function $\delta : Q \times \Sigma \to Q$. We extend the transition function to words $\alpha \in \Sigma^*$ as follows: for $a \in \Sigma$, $\alpha \in \Sigma^*$, and $q \in Q$: $\delta(q, a \cdot \alpha) = \delta(\delta(q, a), \alpha)$ and $\delta(q, \epsilon) = q$. For $q \in Q$ we write I_q to denote the set of strings reaching q from the initial state: $I_q = \{\alpha \in \Sigma^* \mid q = \delta(s, \alpha)\}$. The language $\mathcal{L} \subseteq \Sigma^*$ recognised by $\mathcal{A}$ is the set of strings reaching a final state from the initial state: $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} I_q$. We denote by $\mathrm{Pref}(\mathcal{L})$ the collection of prefixes of strings in $\mathcal{L}$.

▶ Remark 1. Since we are interested in $\mathcal{L}$ rather than in the structure of $\mathcal{A}$, we tacitly discarded from $\mathcal{A}$ all states that are not relevant for the definition of $\mathcal{L}$. That is, we assume that all states of $\mathcal{A}$ are reachable from the initial state s and reaching at least a final state $f \in F$. These assumptions imply that for all $q \in Q$ it is $I_q \neq \emptyset$, and $I_q \subseteq \mathrm{Pref}(\mathcal{L})$.

Following the literature [1, 2], we assume that the initial state s has no incoming arcs and that $\mathcal{A}$ is *input-consistent*: $(\forall u, v \in Q)(\delta(u, a_1) = \delta(v, a_2) \to a_1 = a_2)$. These assumptions are not too restrictive since any automaton can be converted into an equivalent input-consistent automaton by just multiplying its size by a factor of $|\Sigma|$ (it is sufficient, for each $a \in \Sigma$ and $q \in Q$, to replace q by a copy q_a duplicating out-going arcs and redirecting all a-arcs entering q to q_a – possibly none).

▶ Remark 2. In an input-consistent automaton all δ-arcs reaching a given state are labelled by the same character. Thus we may *shift* labels from arcs to states, obtaining an equivalent *state*-labelled automaton. In the following, we will denote by $\lambda(q) \in \Sigma$ the character labelling state q. For the initial state s, which does not have any incoming arc, we set $\lambda(s) = \#$, where $\# \notin \Sigma$ is smaller than any character in Σ.

▶ Remark 3. If $\mathcal{A} = (Q, s, \delta, F)$ is input-consistent, on the grounds of the above observation the second argument of the transition function δ can be safely ignored assuming that $\delta(q) = q'$ stands for $\delta(q, \lambda(q')) = q'$.

▶ **Definition 4.** *A Wheeler DFA (WDFA) $\mathcal{A} = (Q, s, \delta, F, \prec)$ is a DFA endowed with a binary relation $\prec$ such that $(Q, \prec)$ is a total order having the initial state s as minimum, and the following two (Wheeler) properties are satisfied. Let $v_1 = \delta(u_1)$, and $v_2 = \delta(u_2)$:*

i $v_1 \prec v_2 \;\Rightarrow\; \lambda(v_1) \leqslant \lambda(v_2)$;
ii $(\lambda(v_1) = \lambda(v_2) \wedge v_1 \prec v_2) \Rightarrow u_1 \prec u_2$.

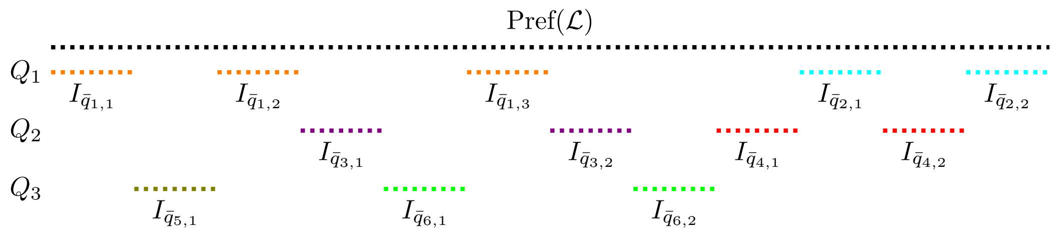
Let $\mathcal{L} \subseteq \Sigma^*$ be a Wheeler language, that is, a language accepted by a deterministic Wheeler automaton. In the rest of the paper we will consider $\mathcal{D} = (Q, s, \delta, F)$ defined as the DFA with the minimum number of states accepting $\mathcal{L}$, and to $\mathcal{D}_w = (Q_w, s, \delta_w, F_w, \prec)$ defined as the WDFA with the minimum number of states accepting $\mathcal{L}$. Uniqueness of $\mathcal{D}$ follows from Myhill-Nerode theorem [8, 9], while uniqueness of $\mathcal{D}_w$ is proven in [2]. By definition it is always $|Q| \leqslant |Q_w|$, but very little else is known about the relative sizes of Q and Q_w; intuitively the ratio $|Q_w|/|Q|$ is the price one has to pay for representing the language $\mathcal{L}$ with a Wheeler automaton.

Definition 4 requires $\prec$ to be a *total* (linear) ordering of the collection of the automaton's states. However, in general, $\mathcal{D}$ does not admit such a total order. Nevertheless $\mathcal{D}$ always admits a *partial* order satisfying (i) and (ii) of Definition 4. Let $p = \mathrm{width}(\mathcal{D})$ (the *width* of

$\mathcal{D}$) be the the minimum number of linear components of a *partial* order satisfying (i) and (ii) of Definition 4. In [5], plenty of arguments are given to illustrate p as a good measure for the distance of $\mathcal{D}$ from being Wheeler.

As established in [2, Lemma 3.4], being $\mathcal{D}_w$ a Wheeler automaton, given $\bar{q} \in Q_w$, the set $I_{\bar{q}} \subseteq \text{Pref}(\mathcal{L})$ of strings reaching $\bar{q}$ from the initial state is an interval $I_{\bar{q}}$ in the linear order $(\text{Pref}(\mathcal{L}), <)$ and the collections of intervals $\{I_{\bar{q}} \mid \bar{q} \in Q_w\}$ constitutes an equivalence relation $\sim_{\mathcal{D}_w}$ partitioning $\text{Pref}(\mathcal{L})$. As a matter of fact, also $\{I_q \mid q \in Q\}$ constitutes an equivalence relation $\sim_{\mathcal{D}}$ partitioning $\text{Pref}(\mathcal{L})$ – even though the I_q's are not, in general, intervals in $(\text{Pref}(\mathcal{L}), <)$ – and $\sim_{\mathcal{D}_w}$ is a *refinement* of $\sim_{\mathcal{D}}$. Hence, for any $\bar{q} \in Q_w$ there exists a unique $q \in Q$, such that $I_{\bar{q}} \subseteq I_q$. In other words, even though I_q for $q \in Q$ might not be an interval, it is always decomposable into a finite collection of intervals $I_{\bar{q}}$'s.

▶ **Example 5.** The following is an example of $\mathcal{D}$ of width 3 where states of $\mathcal{D}_w$ are indexed in such a way that $I_{\bar{q}_{i,j}} \subseteq I_{q_i}$. The partial order of $\mathcal{D}$'s states is: $q_1 \prec q_2; q_3 \prec q_4; q_5 \prec q_6$, with $Q_1 = \{q_1, q_2\}, Q_2 = \{q_3, q_4\}$, and $Q_3 = \{q_5, q_6\}$ being a partition of Q into linearly ordered subsets. Notice that every state of $\mathcal{D}_w$ is reached by interval of strings in $(\text{Pref}(\mathcal{L}), <)$ and any state of $\mathcal{D}$ is reached by a finite collection of intervals of strings.



▶ Remark 6. *Not* being Wheeler for a language means that, for some $q \in Q$, any attempt to produce the previously mentioned decomposition of I_q would result in the introduction of *infinitely many* sub-intervals.

3 The Rational Embedding

In this section we introduce a very simple formal tool, the *rational* embedding, easing the representation and analysis of automata. We begin by embedding Σ^* into $\mathbb{Q}[0, 1)$, the half-open interval of rational numbers between 0 and 1. In what follows, we assume, without loss of generality, that $\Sigma = \{1, 2, \ldots, \sigma\}$ (with the usual order of the integers).

▶ **Definition 7** (The Rational Embedding of Σ^*). *The* Rational Embedding *of* Σ^* *is the map* $\shortparallel : \Sigma^* \to \mathbb{Q}[0, 1)$ *defined as follows. For any* $\alpha = \alpha_1 \ldots \alpha_m \in \Sigma^*$*:*

$$\shortparallel(\alpha) = \sum_{i=1}^{m} \alpha_i \cdot (\sigma + 2)^{-(m-i+1)}.$$

The above embedding sends any non-empty Σ-string to a rational in $(0, 1)$ and the empty word to 0. In the rest of the paper we will always write the values $\shortparallel(\alpha)$ in base $\sigma + 2 = |\Sigma| + 2$; note that by construction the representation will never contain the digit 0 or the digit $\sigma + 1$ to the right of the dot sign.

▶ **Example 8.** Consider the string $\alpha = \alpha_1\alpha_2 \cdots \alpha_m \in \Sigma^*$. The value $\shortparallel$ on α is the rational number $\shortparallel(\alpha)$ written in base $(\sigma + 2)$ as $\shortparallel(\alpha) = 0.\alpha_m \cdots \alpha_2\alpha_1$. Notice that when $\alpha \in \text{Pref}(\mathcal{L})\backslash\{\epsilon\}$, for some $\mathcal{L} = \mathcal{L}(\mathcal{A})$ with $\mathcal{A}$ input-consistent, the most significant digit of $\shortparallel(\alpha)$ is the label of the state reached on $\mathcal{A}$ reading α.

▶ Remark 9. Avoiding 0 and $\sigma+1$ – i.e. the smallest and the largest digits in base $\sigma+2$ – will turn out convenient in order to make the map $\shortparallel(\cdot)$ injective. This assumption is used in Corollary 21 and, clearly, it does not reduce the overall applicability of the embedding.

The fundamental property of the map $\shortparallel(\cdot)$, is that the co-lex order on Σ^* corresponds to the order among elements of the rational embeddings of Σ^*. In formulae, denoting by $<$ also the (standard) natural order on $\mathbb{Q}$:

$$\alpha < \beta \text{ (in co-lex order)} \quad \text{if and only if} \quad \shortparallel(\alpha) < \shortparallel(\beta) \text{ (as rational numbers).}$$

Based on the rational embedding of strings we can define the rational embedding of (the states of) a DFA.

▶ **Definition 10.** *Let $I_{\mathbb{Q}[0,1)}$ be the collection of non-empty convex sets of rationals in $\mathbb{Q}[0,1)$:*

$$I_{\mathbb{Q}[0,1)} = \{J \subseteq \mathbb{Q}[0,1) \mid J \neq \varnothing \wedge (\forall a, c \in J)(\forall b \in \mathbb{Q})(a \leqslant b \leqslant c \Rightarrow b \in J)\}.$$

▶ **Definition 11** (The Rational Embedding of a DFA)**.** *The* Rational Embedding *of $\mathcal{A} = (Q, s, \delta, F)$ is the map $I^{\shortparallel} : Q \to I_{\mathbb{Q}[0,1)}$ defined as follows: for any $q \in Q$,*

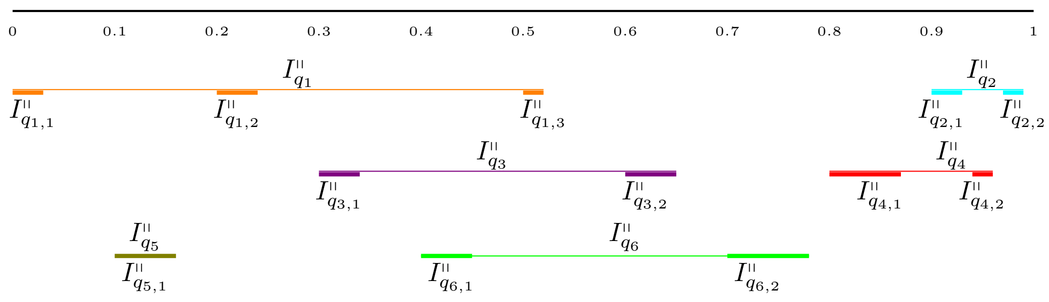
$$I^{\shortparallel}(q) = \bigcap \{J \in I_{\mathbb{Q}[0,1)} \mid (\forall \alpha \in I_q)(\shortparallel(\alpha) \in J)\}.$$

In other words, $I^{\shortparallel}(q)$ is the convex closure (or convex hull) of I_q.

In the following $I^{\shortparallel}(q)$ will also be denoted by $I^{\shortparallel}_q$ and we will denote by ℓ_q (respectively r_q) the inf (respectively the sup) of $I^{\shortparallel}_q$.

Even though determinism guarantees that $q \neq q'$ implies $I_q \cap I_{q'} = \varnothing$, it might be the case that $q \neq q'$ and $I^{\shortparallel}_q \cap I^{\shortparallel}_{q'} \neq \varnothing$. However, [2, Theorem 4.3] implies that $\mathcal{A}$ is Wheeler if and only if $q \neq q'$ implies $I^{\shortparallel}_q \cap I^{\shortparallel}_{q'} = \varnothing$. This is shown by the following example.

▶ **Example 12.** As already shown in Example 5, given a $\mathcal{D}$-state $q \in Q$ the co-lexicographically ordered words in I_q can be decomposed in a finite sequence of sub-intervals that will constitute the $\mathcal{D}_w$-states. By embedding words and states in $\mathbb{Q}[0,1)$ we simply reproduce this situation on the rationals (as landscape). Below we depict the example, with intervals above referring to states in Q and below to states in Q_w:



0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
$I^{\shortparallel}_{q_1}$
$I^{\shortparallel}_{q_{1,1}}$
$I^{\shortparallel}_{q_{1,2}}$
$I^{\shortparallel}_{q_{1,3}}$
$I^{\shortparallel}_{q_2}$
$I^{\shortparallel}_{q_{2,1}}$
$I^{\shortparallel}_{q_{2,2}}$
$I^{\shortparallel}_{q_3}$
$I^{\shortparallel}_{q_{3,1}}$
$I^{\shortparallel}_{q_{3,2}}$
$I^{\shortparallel}_{q_4}$
$I^{\shortparallel}_{q_{4,1}}$
$I^{\shortparallel}_{q_{4,2}}$
$I^{\shortparallel}_{q_5}$
$I^{\shortparallel}_{q_{5,1}}$
$I^{\shortparallel}_{q_6}$
$I^{\shortparallel}_{q_{6,1}}$
$I^{\shortparallel}_{q_{6,2}}$

Mapping each I_q to a set of real numbers $I^{\shortparallel}_q$ makes it possible to study automata using tools from elementary calculus. As a first example we show that the notion of accumulation point is related to the concept of *entanglement* introduced in [5, Definition 4.7]. Intuitively, two states q and q' are entangled when there exists an infinite co-lex-monotone sequence of strings reaching alternatively q and q'. Below a formalization of this important notion in a more general setting.

▶ **Definition 13.** *Let $\mathcal{D}$ be a DFA with set of states Q. A subset $Q' \subseteq Q$ is* entangled *if there exists a monotone sequence $(\alpha_i)_{i\in\mathbb{N}}$ in Pref($\mathcal{L}(\mathcal{D})$) such that for all $u' \in Q'$ it holds $\delta(s, \alpha_i) = u'$ for infinitely many i's.*

▶ **Definition 14.** *We say that $x \in \mathbb{R}$ is a* left-accumulation point *for a set U if there exists a sequence of elements $u_i \in U$ strictly greater than x and converging to x. Similarly, we say that x is a* right-accumulation point *for U if the elements u_i converging to x are all strictly smaller than x.*

▶ **Lemma 15.** *If a value x is a left-accumulation point (resp. right-accumulation point) for both the sets $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ then q and q' are entangled.*

Proof. If x is a left-accumulation point for both I_q and $I_{q'}$ from elementary calculus we know that there exists an infinite sequence $u_1 > v_1 > \cdots u_i > v_i > \cdots$ converging to x with $u_i \in I^{\shortparallel}_q$ and $v'_i \in I^{\shortparallel}_{q'}$. Hence there is a sequence of strings $\alpha_1 \succ \beta_1 \cdots \alpha_i \succ \beta_i \succ \cdots$ with $\alpha_i \in I_q$ and $\beta_i \in I_{q'}$ and the states q and q' are entangled according to Definition 4.7 in [5]. The case when x a right-accumulation point for both $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ is analogous. ◀

▶ **Theorem 16.** *If $\mathcal{D}$ is the minimum DFA accepting $\mathcal{L}$, and x is a left-accumulation point (resp. right-accumulation point) for two distinct sets $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ then $\mathcal{L}$ is not Wheeler.*

Proof. By Lemma 15 the states q and q' are entangled. Since $\mathcal{D}$ is the minimum DFA accepting $\mathcal{L}$ by [5, Theorem 4.21] any DFA recognizing $\mathcal{L}$ has width at least 2. ◀

▶ **Example 17.** Consider the two automata in Figure 1.

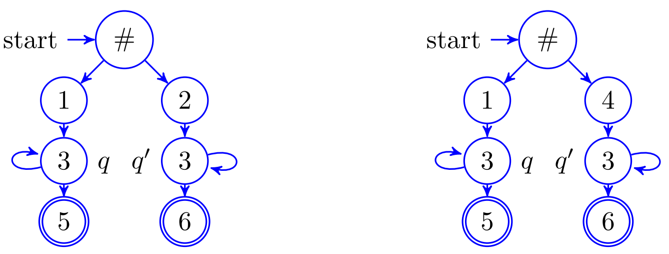


Figure 1 The language accepted by the automaton on the right is Wheeler, while the one accepted by the automaton on the left is not.

The automaton on the right accepts a Wheeler language, while the one on the left does not, the reason being that on the left $0.\bar{3}$ is a right-accumulation point for both $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ because of the sequences $0.31, 0.331, 0.3331, \ldots$ and $0.32, 0.332, 0.3332, \ldots$ (reaching q'): by Theorem 16 the corresponding language is non Wheeler. In the automaton on the right $0.\bar{3}$ is a right-accumulation point for $I^{\shortparallel}_q$ and a left-accumulation point for $I^{\shortparallel}_{q'}$ and the automaton is Wheeler with $q \prec q'$. Note that, if in the left automaton we remove states 5 and 6 and make q and q' final, then $0.\bar{3}$ is still a right-accumulation point for both $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ but the resulting automaton is not minimum so Theorem 16 do not apply: indeed the resulting language is Wheeler.

We are particularly interested in the study of the extreme values $\ell_q = \inf I^{\shortparallel}_q$ and $r_q = \sup I^{\shortparallel}_q$ as defined in Definition 11. We have the following preliminary results.

▶ **Lemma 18.** *If $\mathcal{L} = \mathcal{L}(\mathcal{D})$ is Wheeler and $\mathcal{D}$ is the minimum DFA accepting $\mathcal{L}$, then for all pairwise distinct $q, q' \in Q$, we have:*

$$\ell_q = \ell_{q'} \rightarrow (\ell_q \in I^{\shortparallel}_q \vee \ell_{q'} \in I^{\shortparallel}_{q'}), \qquad r_q = r_{q'} \rightarrow (r_q \in I^{\shortparallel}_q \vee r_{q'} \in I^{\shortparallel}_{q'}).$$

The same property holds if $\mathcal{D}$ is a WDFA accepting $\mathcal{L}$.

Proof. Assume $\mathcal{D}$ is the minimum DFA for the Wheeler language $\mathcal{L}$. If there exist $q, q' \in Q$ such that $\ell_q = \ell_{q'}$ and $(\ell_q \notin I^{\shortparallel}_q \wedge \ell_{q'} \notin I^{\shortparallel}_{q'})$ then ℓ_q would be a left-accumulation point for both $I^{\shortparallel}_q$ and $I^{\shortparallel}_{q'}$ which is impossible by Theorem 16.

If instead $\mathcal{D}$ is a WDFA accepting $\mathcal{L}$ observe that $\ell_q = \ell_{q'}$ and $(\ell_q \notin I^{\shortparallel}_q \wedge \ell_{q'} \notin I^{\shortparallel}_{q'})$ implies $I^{\shortparallel}_q \cap I^{\shortparallel}_{q'} \neq \emptyset$, which would contradict the hypothesis that $\mathcal{D}$ is Wheeler. The case $r_q = r_{q'}$ is entirely analogous for both kind of automata. ◀

▶ **Lemma 19.** *If $\lambda_q \in \mathcal{L}$ is such that $\shortparallel(\lambda_q) = \ell_q$, then: $\ell_q \in I^{\shortparallel}_q$ if and only if $\lambda_q \in I_q$. The same result holds for r_q as well.*

Proof. See Appendix. ◀

Using Lemma 18 we can order the intervals $I^{\shortparallel}_q$'s according to their left ends ℓ_q's (or their right ends r_q's) breaking ties, when $\ell_q = \ell_{q'}$, by setting $I^{\shortparallel}_q$ less than $I^{\shortparallel}_{q'}$ if $\ell_q \in I^{\shortparallel}_q$ and $\ell_{q'} \notin I^{\shortparallel}_{q'}$ (we cannot have $\ell_q \in I^{\shortparallel}_q \wedge \ell_q \in I^{\shortparallel}_{q'}$ since by Lemma 19 we would have $I_q \cap I'_q \neq \emptyset$). The rationale for this tie-breaking rule is that $\ell_q \in I^{\shortparallel}_q$ ensures that there is an element in $I^{\shortparallel}_q$ which is strictly smaller than all elements in $I^{\shortparallel}_{q'}$. However, we will see below (Corollary 21) that, in fact, breaking ties will never be necessary.

Using the above ordering of the intervals we can derive a procedure to determine the values ℓ_q and r_q, for all $q \in Q$.

▶ **Lemma 20.** *Let $\mathcal{L} = \mathcal{L}(\mathcal{D})$, with $\mathcal{L}$ Wheeler and $\mathcal{D}$ either minimum or Wheeler. For any $q \in Q$ we have:*

$$\ell_q = 0.a_{q,1} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}},$$

with $h + j \leqslant |Q|$, and $j = 0$ meaning that ℓ_q is not periodic. Moreover, $j > 0$ if and only if $\ell_q \notin I^{\shortparallel}_q$. An analogous characterisation holds for r_q.

Proof. Let $q_0 = s < q_1 < \ldots < q_n$ be the total order of Q induced by the order of the intervals $I^{\shortparallel}_q$ mentioned above, that is

$$q_i < q_{i'} \overset{\text{def}}{\iff} \left(\ell_{q_i} < \ell_{q_{i'}} \vee (\ell_{q_i} = \ell_{q_{i'}} \wedge \ell_{q_{i'}} \notin I^{\shortparallel}_{q_{i'}})\right). \tag{1}$$

Algorithm 1 determines the digits and the (possible) periodicity of ℓ_q for any state q.

After a call of `left_dd`(q), let $P = \{q_{i_1}, \ldots, q_{i_{k-1}}\}$. It is easily seen by induction that the digits determined $a_{q,1} \cdots a_{q,k-1}$ are, in fact, the first $k-1$ digits of ℓ_q. Upon exit of `left_dd`(q), $k - 1 = h + j < |Q|$ and the algorithm stops in one of the following two cases:

1. $q_{i_k} = s$, or
2. $q_{i_k} = q_{i_{k'}}$, for some $k' \in \{1, \ldots, k-1\}$.

In the first case $h = k - 1$, $j = 0$, and $\ell_q = 0.a_{q,1} \cdots a_{q,h}$, as determined by `left_dd`$(q)$. In the second case $h = k' - 1$, $j = k - k'$, and our claim is that

$$\ell_q = 0.a_{q,1} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}}.$$

In fact, in this case it is easy to produce a sequence of strings reaching q whose rational embeddings converge to ℓ_q. Take, for example, β labelling a simple path from s to $q_{i_{k'}}$ and consider the following infinite sequence of words reaching q: for $i \in \mathbb{N}$,

$$\beta(a_{q,h+j} \cdots a_{q,h+1})^i a_{q,h} \cdots a_{q,1}.$$

Algorithm 1 `left_digits_detector` (q) (`left_dd` (q)).

```
$k \leftarrow 1$;   // initialise a counter for visited states (and for the digits)
$q_{i_k} \leftarrow q$;   // set the first state (and digit)
$P \leftarrow \varnothing$;   // $P$ will store visited states
while $q_{i_k} \neq s$ and $q_{i_k} \notin P$ do   // stop when $s$ or a state in $P$ is reached
    $P \leftarrow P \cup \{q_{i_k}\}$;
    $a_{q,k} \leftarrow \lambda(q_{i_k})$;
    $k \leftarrow k + 1$;
    $i_k \leftarrow \min\{k' \mid \delta(q_{k'}) = q_{i_{k-1}}\}$;   // use the ordering (1)
if $q_{i_k} = s$ then   // $\ell_q$ is not periodic
    $h \leftarrow k - 1$;
    $j \leftarrow 0$;
else   // $q_{i_k}$ is a previously visited state: set periodicity
    let $k' < k$ such that $q_{i_k} = q_{i_{k'}}$
    $h \leftarrow k' - 1$
    $j \leftarrow k - k'$;
```

By construction we have that, for any $i \in \mathbb{N}$,

$$\shortparallel(\beta(a_{q,h+j} \cdots a_{q,h+1})^{i+1} a_{q,h} \cdots a_{q,1}) < \shortparallel(\beta(a_{q,h+j} \cdots a_{q,h+1})^{i} a_{q,h} \cdots a_{q,1}),$$

and that:

$$\ell_q = \lim_{i \to \infty} \shortparallel(\beta(a_{q,h+j} \cdots a_{q,h+1})^{i} a_{q,h} \cdots a_{q,1}). \qquad \blacktriangleleft$$

Using the characterisation of ℓ_q and r_q provided by Lemma 20 we can strengthen Lemma 18 and prove that different $I_q^{\shortparallel}$'s have always different left and right limits.

▶ **Corollary 21.** *Let $\mathcal{L} = \mathcal{L}(\mathcal{D})$, with $\mathcal{L}$ Wheeler and $\mathcal{D}$ either minimum or Wheeler. Then, for any pairwise distinct $q, q' \in Q$ we have: $\ell_q \neq \ell_{q'}$ and $r_q \neq r_{q'}$.*

Proof. See Appendix. ◀

The above corollary clarifies that any state $q \in Q$ can be uniquely characterised by a rational number or, equivalently, by a string of at most $|Q| - 1$ characters.

▶ **Theorem 22.** *If $\mathcal{L} = \mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}_w)$, with $\mathcal{L}$ Wheeler and $\mathcal{D}$ either minimum or Wheeler, then for all $q \in Q$, we have $\ell_q, r_q \in \mathbb{Q}$.*

Proof. Follows directly from Lemma 20. ◀

▶ Remark 23. We can give examples of automata $\mathcal{D}$ such that, for some $q \in Q$, $I_q^{\shortparallel}$ includes Cauchy sequences of rational embeddings of strings converging to irrational numbers – as a matter of fact, this easily follows from the fact that the language Σ^* is Wheeler. However, Theorem 22 ensures that such irrational limits of (encodings of) words will never occur as endpoints of $I_q^{\shortparallel}$'s. In fact, Lemma 20, exploiting the linear order of the reals, used by algorithm `left_dd` to direct the search, shows that – not surprisingly, being $\mathcal{D}$ a finite automaton – a *finite* representation of the bounding elements of $I_q^{\shortparallel}$'s, can be given.

Corollary 21 immediately implies that the existence of distinct states with equal left (right) limits guarantees non-Wheelerness. The converse of the above result is, in general, not true as illustrated by the following example.

▶ **Example 24.** The automaton in Figure 2 is such that $I^{\shortparallel}_q = [0.51, 0.56]$ and $I^{\shortparallel}_{q'} = [0.52, 0.57]$. However, the value $0.\overline{5}$ is a left-accumulation point for both q and q'. By Theorem 16 the accepted language is not Wheeler even if all the left and right limit are distinct.

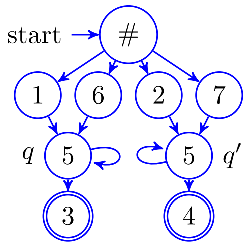


Figure 2 States q and q' in the above automaton are entangled by Lemma 15; however $\ell_q \neq \ell_{q'}$ and $r_q \neq r_{q'}$.

4 On the number of states of the minimum WDFA

Given $\mathcal{D}_w = (Q_w, s_w, \delta_w, F_w, <)$ minimum (in the number of states) WDFA accepting a Wheeler language $\mathcal{L} = \mathcal{L}(\mathcal{D})$, with $\mathcal{D} = (Q, s, \delta, F)$ minimum DFA accepting $\mathcal{L}$, we want to study the relationship between the size of $\mathcal{D}_w$ and $\mathcal{D}$.

Consider the collection of intervals $\{I^{\shortparallel}_q \mid q \in Q_w\}$. Since $\mathcal{D}_w$ is Wheeler, as already observed we have that for pairwise distinct $q, q' \in Q_w$, $I^{\shortparallel}_q \cap I^{\shortparallel}_{q'} = \emptyset$. This is, in general, not the case for $\mathcal{D}$ and below we prove that the size of $\mathcal{D}_w$ can be exponential in the size of $\mathcal{D}$, even in case $\text{width}(\mathcal{D}) = 2$.

Below we give a simple example of automaton $\mathcal{D}^1$ accepting a Wheeler language but such that the minimum WDFA $\mathcal{D}^1_w$ has size exponential in the size of $|\mathcal{D}^1|$. Let $\mathcal{D}^1$ be the automaton in Figure 3.

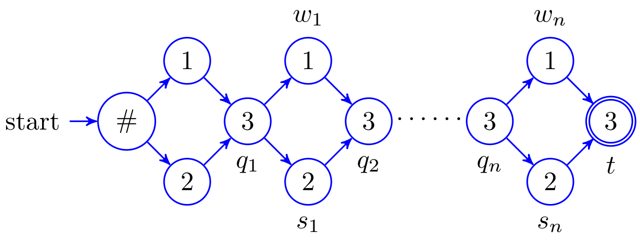
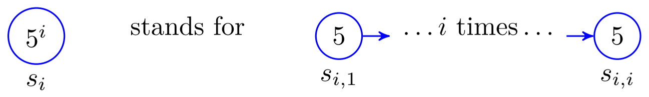


Figure 3 The depicted DFA is accepting a Wheeler (finite) language and the minimum accepting Wheeler DFA accepting the same language has size exponential in n.

$\mathcal{L} = \mathcal{L}(\mathcal{D}^1)$, being finite, is Wheeler [2]. Moreover, any Wheeler automaton accepting $\mathcal{L}$ must have a number of states exponential in n. In fact, given any pair of strings $\alpha, \gamma \in I_t$ such that $\shortparallel(\alpha) < \shortparallel(\gamma)$, it is easy to find a $\beta \in I_{q_n}$ such that $\shortparallel(\alpha) < \shortparallel(\beta) < \shortparallel(\gamma)$. Since there are exponentially many pairwise distinct strings reaching state t, in a Wheeler automaton the set I_t must be partitioned into an exponential number of sub-intervals. Hence, state t must be "split" into exponentially many states of $\mathcal{D}_w$ and the size of $\mathcal{D}_w$ must be exponential in n.

▶ Remark 25. The automaton $\mathcal{D}^1$ of Figure 3 has $\text{width}(\mathcal{D}^1) = n$. Hence, one could think that the explosion in the number of states is exponential in the width of the minimum automaton accepting a given language. Below we show that this is not the case, providing an example of automaton whose width is just 2 but the explosion still occurs.

Consider the DFA $\mathcal{D}^2$ of Figure 4, where, for example, state s_i labeled 5^i stands for a sequence of i states $s_{i,1}, \dots, s_{i,i}$, all labeled 5. That is:



and, analogously, for states s'_i, w_i and w'_i, for $i \in \{1, \dots, n\}$. Furthermore, we say that states $s_{i,j}$ and $s'_{i,j}$ are *twins* – the same goes for all other states and their *primed* version.

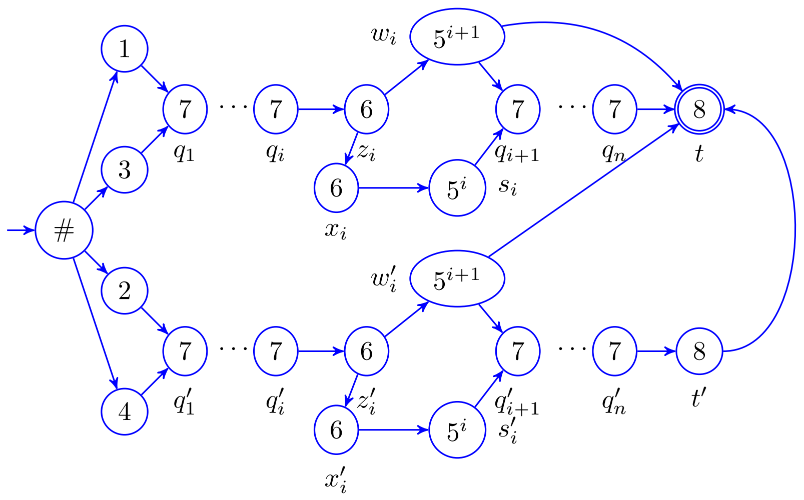


Figure 4 The automaton $\mathcal{D}^2$. The gadget between q_i and q_{i+1}, consisting of states z_i, x_i, w_i, s_i, (and, analogously, the gadget between q'_i and q'_{i+1}) is deployed for $i = 1, \dots, n-1$. Notice that the i-th copy consists of $2i + 4$ states overall since each label 5^i expands to i states.

Table 1 Left and right limits of $I^{\shortparallel}$ for different kinds of states from automaton $\mathcal{D}^2$. Intervals of states denoted by different letters are clearly non-intersecting except in the case of t with t'.

State type	**Left limit**	**Right limit**
$s_{i,j}$	$0.5^j6675^i67\dots$	$0.5^j6675^{i-1}667\dots$
$w_{i,j}$	$0.5^j675^i67\dots$	$0.5^j675^{i-1}667\dots$
x_i	$0.6675^i67\dots$	$0.6675^{i-1}667\dots$
z_i	$0.675^i67\dots$	$0.675^{i-1}667\dots$
q_i	$0.75^i67\dots$	$0.75^{i-1}667\dots$
t	$0.85^n67\dots$	$0.8875^{n-1}667\dots$
t'	$0.875^n67\dots$	$0.875^{n-1}667\dots$

Our goal is to show that $\mathcal{D}^2$ has width equal to 2. The following two lemmas, whose proofs can be found in the appendix, ensure that non-empty intersection of intervals happens only between twin states.

▶ **Lemma 26.** *Let u, v states of any automaton. If $\lambda(u) \neq \lambda(v)$, then $I^{\shortparallel}_u \cap I^{\shortparallel}_v = \emptyset$.*

▶ **Lemma 27.** *Let $\mathcal{D}^2$ be the automaton in Figure 4. Let u, v be a pair of distinct states of $\mathcal{D}^2$ such that $\lambda(u) = \lambda(v)$. Then, $I^{\shortparallel}_u \cap I^{\shortparallel}_v \neq \emptyset$ if and only if u and v are twins.*

By above lemmas it follows:

▶ **Lemma 28.** *Let $\mathcal{D}^2$ be the automaton of Figure 4. Then,* $\text{width}(\mathcal{D}^2) = 2$.

$\mathcal{D}^2$ accepts a (finite) Wheeler language, but its minimum *Wheeler* automaton has exponential size in n. The fact that $\mathcal{D}^2_w$ has size exponential in n is verified observing that strings reaching t and t' are interleaved, analogously to $\mathcal{D}^1$.

▶ **Theorem 29.** *Let $\mathcal{L} = \mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{D}_w)$, with $\mathcal{L}$ Wheeler, $\mathcal{D}$ minimum, $\mathcal{D}_w$ minimum Wheeler, and let $f(\cdot,\cdot)$ be such that $|\mathcal{D}_w| = O(\, f(|\mathcal{D}|, \text{width}(\mathcal{D}))\,)$. Then, for any $k, p \in \mathbb{N}$, $f(n, p) \notin O(n^k + 2^p)$.*

5 Left and right limits: the arithmetic way

In this section we describe an alternative way to determine the left and right limit of the intervals defining the rational embedding of the automaton. Our starting point is the following lemma (proof in Appendix) establishing an arithmetic relationship between the left values ℓ_q's. An analogous result holds for the right values r_q's.

▶ **Lemma 30.** *Given $\mathcal{D} = (Q, s, \delta, F)$, DFA accepting $\mathcal{L}$ Wheeler, and $q \in Q\backslash\{s\}$, there exists a unique $q' \in Q$ such that $\delta(q') = q$ and $(\sigma + 2) \cdot \ell_q = \lambda(q) + \ell_{q'}$.*

In the following we use $\mathbb{R}^Q$ to denote the set of real-valued vectors indexed by elements of Q. Given $x \in \mathbb{R}^Q$ and $q \in Q$ we write x_q to denote the entry associated to q. Similarly we use $\mathbb{Q}^Q$ to denote the set of rational-valued vectors. We write ℓ to denote the vector in $\mathbb{Q}^Q$ containing the left limits ℓ_q with $q \in Q$.

Lemma 30 suggests a way of computing left (and right) limits through constraint programming [3, 11]. Formally, for the *left* case, we consider the problem of finding the set of all real-valued vectors $x \in \mathbb{R}^Q$ that satisfy the following constraint satisfaction program, that we name $\mathcal{P}_{\mathit{Left}}$:

$$\begin{array}{lll} (1) & x_s = 0, & \\ (2) & 0 < x_q < 1, & (\forall q \in Q\backslash\{s\}) \\ (3) & (\sigma + 2) \cdot x_q = \lambda(q) + \min\left\{x_{q'} \mid \delta(q') = q\right\}, & (\forall q \in Q\backslash\{s\}) \end{array}$$

We now prove that the vector $\ell \in \mathbb{Q}^Q$ of left limits is the only solution of the above program. As a first step, we make sure that $\mathcal{P}_{\mathit{Left}}$ is complete, that is, the vector ℓ satisfies constraints (1–3).

▶ **Lemma 31.** *Let $\mathcal{L}$ be a Wheeler language, and $\mathcal{D} = (Q, s, \delta, F)$ be either minimum or Wheeler accepting $\mathcal{L}$, and let $\ell \in \mathbb{Q}^Q$ be the vector of left limits. Then, ℓ is a solution of $\mathcal{P}_{\mathit{Left}}$.*

Proof. First of all, notice that constraints (1) and (2) of $\mathcal{P}_{\mathit{Left}}$ are clearly satisfied by ℓ. Consider the order $<_Q$ of the states of $\mathcal{D}$ defined by: $q <_Q q' \overset{\text{def}}{\iff} \ell_q < \ell_{q'}$. The order is well-defined and total in virtue of Corollary 21. By Lemma 30, for every state $q \neq s$ there exists a *unique* $q' \in \delta^{-1}(q)$ such that $(\sigma + 2) \cdot \ell_q = \lambda(q) + \ell_{q'}$. Moreover, from the proof of Lemma 20 we know that $q' = \min_{<_Q} \delta^{-1}(q)$. By definition of $<_Q$ we have:

$$q' = \min_{<_Q} \delta^{-1}(q) \iff \ell_{q'} = \min\left\{\ell_{q''} \mid q'' \in \delta^{-1}(q)\right\},$$

thus satisfying constraint (3). ◀

To prove that ℓ is the only solution we need the notion of (x, q)-min-path.

▶ **Definition 32.** *Let $\mathcal{D} = (Q, s, \delta, F)$ be a DFA, $x \in \mathbb{R}^Q$, and $q \in Q$. We say that an infinite sequence of states $(q_i)_{i \geqslant 1}$ is a* (x, q)-min-path in $\mathcal{D}$ *if the following hold:*

1. $q_1 = q$,
2. $(\forall i \geqslant 1)(\delta(q_{i+1}) = q_i \vee q_i = q_{i+1} = s)$,
3. $(\forall i \geqslant 1)(x_{q_{i+1}} = \min\{x_{q'} \mid \delta(q') = q_i\} \vee q_i = q_{i+1} = s)$.

Roughly speaking, a (x, q)-min-path is a path in the automaton that follows (backward) states whose associated x-value is minimum. It does not come as a surprise that if $(q_1, q_2, \dots)$ is a (x, q)-min-path, then for every $j \geqslant 1$ we have that $(q_j, q_{j+1}, \dots)$ is a (x, q_j)-min-path as well: the proof of this simple fact follows directly from Definition 32.

Furthermore, when $x \in \mathbb{R}^Q$ is a solution of $\mathcal{P}_{Left}$, (x, q)-min-paths spell out precisely x_q's digits. Formally:

▶ **Lemma 33.** *Let $x \in \mathbb{R}^Q$ be a solution of $\mathcal{P}_{Left}$, $q \in Q$, and let $(q_i)_{i \geqslant 1}$ be any (x, q)-min-path. Then, for every $j \geqslant 1$, the j-th digit of x_q is $\lambda(q_j)$.*

Proof. See Appendix. ◀

▶ **Corollary 34.** *If $x \in \mathbb{R}^Q$ is a solution of $\mathcal{P}_{Left}$, then for every $q \in Q$ the first digit of x_q is $\lambda(q)$.*

Proof. Follows immediately from Definition 32 and Lemma 33. ◀

We can now state the main result of this section.

▶ **Theorem 35.** *Let $\mathcal{D} = (Q, s, \delta, F)$ be either minimum or Wheeler accepting $\mathcal{L}$ Wheeler, and $\ell \in \mathbb{Q}^Q$ be the vector of left limits. Then, $\mathcal{P}_{Left}$ always admits ℓ as its* unique *solution.*

Proof. By Lemma 31 we know that ℓ is a solution of $\mathcal{P}_{Left}$. Let x be a generic solution of $\mathcal{P}_{Left}$, and denote by $x_{q,j}$ the j-th digit of x_q. Suppose, for the sake of contradiction, that there exists some state $q \in Q$ such that $x_q \neq \ell_q$, let q be any for which x_q and ℓ_q have the shortest prefix of digits in common, and let j be the length of such prefix. By Corollary 34, $j \geqslant 1$. Let $(q_i)_{i \geqslant 1}$ and $(q'_i)_{i \geqslant 1}$ be, respectively, any (x, q)-min-path and (ℓ, q)-min-path. By Lemma 33 and hypothesis on q, it holds:
1. $(\forall i \leqslant j)(\lambda(q_i) = x_{q,i} = \ell_{q,i} = \lambda(q'_i))$, and
2. $\lambda(q_{j+1}) = x_{q,j+1} \neq \ell_{q,j+1} = \lambda(q'_{j+1})$.

Consider the case $\lambda(q_{j+1}) < \lambda(q'_{j+1})$ (the other case is symmetric). By minimality of the choice of q, the first j digits of both x_{q_2} and ℓ_{q_2} are the same. Similarly, the first j digits of both $\ell_{q'_2}$ and $x_{q'_2}$ are the same. Therefore, $\ell_{q_2} < \ell_{q'_2}$ contradicting the hypothesis that $(q'_i)_{i \geqslant 1}$ was a (l, q)-min-path. ◀

Program $\mathcal{P}_{Left}$ can be implemented as a Mixed Integer Program, whose solution is, in general, computationally hard to obtain [10]. The fact that a graph-oriented approach can compute the left limits in polynomial time [7], justifies the following:

▶ **Conjecture 36.** *There exists a linear programming model equivalent to $\mathcal{P}_{Left}$.*

We give a partial answer to Conjecture 36. Let $\pi : Q \to Q$ be the *parent* function, *i.e.* $\pi(q) = q'$ if and only if q' is the unique state mandated by Lemma 30 (with $\pi(s) = s$). Indeed, π is precisely what has been computed in [7, Section 4] in the form of a pruned automaton. Consider matrix $\Pi \in \{0, 1\}^{|Q| \times |Q|}$ such that $\Pi_{i,j} = 1$ if and only if $\pi(i) = j$. Combining $\mathcal{P}_{Left}$ and Theorem 35, we express the problem of computing vector ℓ of left limits given vector λ of characters as the unique solution of:

$$\begin{cases} x_s = 0, \\ (\forall q \neq s)(0 < x_q < 1), \\ (\sigma + 2) \cdot x = \lambda + \Pi^{\intercal} \cdot x \end{cases}$$

The third equation reminds of the condition for x to be an eigenvector of $\Pi^{\intercal}$, with λ acting as a *corrective* term. This intuition can be made formal. In fact, if we denote by I the $|Q| \times |Q|$ identity matrix, it is easily verified that:

$$\begin{pmatrix} \Pi^{\intercal} & I \\ 0 & (\sigma + 2)I \end{pmatrix} \cdot \begin{pmatrix} x \\ \lambda \end{pmatrix} = (\sigma + 2) \begin{pmatrix} x \\ \lambda \end{pmatrix}$$

As already stated, we consider the above model unsatisfactory since we would need to know Π in advance. Finally, for the right limits case, we define the constraint satisfaction program $\mathcal{P}_{Right}$ by substituting max for min in (3) of $\mathcal{P}_{Left}$:

$$(3^*) \quad (\sigma + 2) \cdot x_q = \lambda(q) + \max \left\{ x_{q'} \mid \delta(q') = q \right\}, \qquad (\forall q \in Q \backslash \{s\})$$

The discussion made for $\mathcal{P}_{Left}$ computing ℓ can be translated into a discussion for $\mathcal{P}_{Right}$ computing the vector of right limits r by exchanging min-arguments into max-arguments. The key observation is that Lemma 33, rephrased in terms of *max-paths*, still holds.

6 Conclusions

One of the goals of this paper was to study the problem of building, given a Wheeler language presented by its minimum accepting automaton, a Wheeler accepting automaton of minimum size. Such minimum Wheeler DFA is proved to be, in general, exponential in size with respect to the size of the minimum input DFA. The lower bound is proved by exhibiting an example of DFA whose size explodes exponentially when we perform the "splits" necessary to guarantee the order characterizing Wheeler-ness. Moreover, and most importantly, this happens even when the *width* of the input DFA is just 2. We point out that the latter phenomenon is a sort of exception: for most classic operations, once the width is fixed we are able to put polynomial bounds on their complexity.

The above result is illustrated while introducing a simple view on DFAs and WDFAs, that starts from a mapping of strings into rational numbers. According to this *rational* embedding, automaton's states can be (over)approximated by convex sets (intervals) of rational numbers and the basic Wheeler properties turn out to be translated into ordering and non-intersecting constraint on the collection of states-intervals. Moreover, a characterisation of the number of digits necessary to identify left and right limits of states-intervals can be carried out analysing the underlying automaton's transition function and using the Wheeler order of its states.

The latter technique suggests also that the infinite alternation of strings reaching different states (the so-called *entanglement* of states) can be linked with the existence, position, and distribution of accumulation points of the collection of embedding of prefixes of strings on the $[0, 1)$ half-open interval of the real line. An interesting further direction of study is the characterisation of order-types obtainable by rationals corresponding to embedding of prefixes of general, not necessarily Wheeler, languages. The final section is devoted to propose a further angle from which the problem of determining digits of limiting rationals can be approached, namely constraint programming.

References

1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.

2 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Inf. Comput.*, 281:104820, 2021. doi:10.1016/j.ic.2021.104820.

3 Krzysztof Apt. *Principles of constraint programming.* Cambridge university press, 2003. doi:10.1017/CBO9780511615320.

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

5 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Co-Lexicographically Ordering Automata and Regular Languages - Part I. *J. ACM*, 70(4), August 2023. doi:10.1145/3607471.

6 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo). doi:10.1016/j.tcs.2017.06.016.

7 Sung-Hwan Kim, Francisco Olivares, and Nicola Prezza. Faster prefix-sorting algorithms for deterministic finite automata. In *Proc. CPM 23.* Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CPM.2023.16.

8 John Myhill. Finite automata and the representation of events. *WADD Technical Report*, 57:112–137, 1957.

9 Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

10 Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981. doi:10.1145/322276.322287.

11 Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008. doi:10.1016/S1574-6526(07)03004-0.

A Proofs

Proof of Lemma 19. Since $I_q^{\shortparallel}$ is the convex closure of I_q, it is $I_q \subseteq I_q^{\shortparallel}$ and the "if" implication is immediate. If it were $\ell \notin I_q \wedge \ell_q \in I_q^{\shortparallel}$, by considering $U = I^{\shortparallel} \backslash \{\ell_q\}$ we would get a convex set $U \supseteq I_q$ strictly contained in $I_q^{\shortparallel}$ which is a contradiction. ◀

Proof of Corollary 21. We deal with left limits only since the argument for right limits is entirely similar. If both ℓ_q and $\ell_{q'}$ are not periodic, by Lemma 20 and the fact that $\mathcal{D}$ is deterministic it must be the case that $\ell_q \neq \ell_{q'}$.

Otherwise, by Lemma 18 we have that, say, $\ell_q \in I_q^{\shortparallel}$ and $\ell_{q'} \notin I_{q'}^{\shortparallel}$. By Lemma 20 this means that $\ell_{q'}$ is a periodic rational while ℓ_q is not. Since the largest digit of Σ will never label a state (see Remark 9), the two rational numbers ℓ_q and $\ell_{q'}$ cannot possibly be equal. ◀

Proof of Lemma 26. Suppose, without loss of generality, that $\lambda(u) < \lambda(v)$. It is clear that:

$$r_u = 0.\lambda(u)\cdots \; < \; 0.\lambda(v)\cdots \; = \ell_v.$$

Thus, their respective intervals do not intersect. ◀

Proof of Lemma 27. ($\Leftarrow$) The case for t and t' is clearly true (see Table 1). Consider two twin states u and u', respectively from the top and the bottom level of $\mathcal{D}^2$. By construction, there exist $\alpha, \beta \in \Sigma^*$ such that:

$$\begin{array}{ccccccc} 0.\alpha 1 & < & 0.\alpha 2 & < & 0.\beta 3 & < & 0.\beta 4 \\ \parallel & & \parallel & & \parallel & & \parallel \\ \ell_u & & \ell_{u'} & & r_u & & r_{u'} \end{array}$$

Thus, $I_u^{=} \cap I_{u'}^{=} \neq \varnothing$.

($\Rightarrow$) We prove the contrapositive. The case for states denoted by different letters is simple (see again Table 1). Let u and v be two non-twin states denoted by the same letter and different indexes, and suppose, without loss of generality, that $u = s_{i,j}$ and $v = s_{i',j'}$ (all other cases are proved in a similar way). We have two cases. If $j = j'$ and $i < i'$, then:

$$r_v = 0.5^j 6675^i 5^{i'-i-1} 667 \cdots < 0.5^j 6675^i 67 \cdots = \ell_u$$

and their respective intervals do not intersect. Otherwise, if $j < j'$, then:

$$r_v = 0.5^j 5^{j'-j} 667 \cdots < 0.5^j 667 \cdots = \ell_u$$

and, again, their respective intervals do not intersect. ◀

Proof of Lemma 30. By Lemma 20, we have:

$$\ell_q = 0.a_{q,1} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}} = 0.\lambda(q) a_{q,2} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}}.$$

Let q' be the first state visited after q by `left_dd` (q), we have:

$$\ell_{q'} = \begin{cases} 0.a_{q,2} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}}, & \text{if } h > 0, \\ 0.\overline{a_{q,2} \cdots a_{q,j-1}\lambda(q)}, & \text{if } h = 0. \end{cases}$$

Since all the above values are expressed in base $\sigma + 2$ it is

$$(\sigma+2) \cdot \ell_q \;=\; \lambda(q).a_{q,2} \cdots a_{q,h}\overline{a_{q,h+1} \cdots a_{q,h+j}} \;=\; \lambda(q) + \ell_{q'}$$

as claimed. The uniqueness of q' follows from Corollary 21. ◀

Proof of Lemma 33. First of all, the unique (x, s)-min-path is $(s, s, \dots)$. Therefore, the j-th digit of x_s is $\lambda(s)$ for every j. If $q \neq s$, we prove the lemma by induction on $j \geqslant 1$. In what follows, we denote by $x_{q,j}$ the j-th digit of x_q, and we let $(q_i)_{i \geqslant 1}$ be any (x, q)-min-path.

Base. By Definition 32 and constraints of $\mathcal{P}_{Left}$ we have $\lambda(q_1) \leqslant (\sigma+2) \cdot x_q < \lambda(q_1) + 1$. Thus, $x_{q,1} = \lambda(q_1)$.

Step. Let $j > 1$, and suppose the property holds for every state and every $j' < j$. Sequence $(q_2, q_3, \dots)$ is a (x, q_2)-min-path. Thus:

$$\begin{aligned} x_{q,j} &= x_{q_2,j-1} && \text{(Constraint 3 of } \mathcal{P}_{Left} \text{ and Def. 32)} \\ &= \lambda(q_j) && \text{(Induction hypothesis on } q_2 \text{ and } j-1) \end{aligned}$$

◀

Shortest Cover After Edit

Kazuki Mitani
Graduate School of Information Science and Technology, Hokkaido University, Japan

Takuya Mieno
Department of Computer and Network Engineering, University of Electro-Communications, Tokyo, Japan

Kazuhisa Seto
Faculty of Information Science and Technology, Hokkaido University, Japan

Takashi Horiyama
Faculty of Information Science and Technology, Hokkaido University, Japan

Abstract

This paper investigates the (quasi-)periodicity of a string when the string is edited. A string C is called a cover (as known as a quasi-period) of a string T if each character of T lies within some occurrence of C. By definition, a cover of T must be a border of T; that is, it occurs both as a prefix and as a suffix of T. In this paper, we focus on the changes in the longest border and the shortest cover of a string when the string is edited only once. We propose a data structure of size $O(n)$ that computes the longest border and the shortest cover of the string in $O(\ell \log n)$ time after an edit operation (either insertion, deletion, or substitution of some string) is applied to the input string T of length n, where ℓ is the length of the string being inserted or substituted. The data structure can be constructed in $O(n)$ time given string T.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases string algorithm, border, cover, quasi-periodicity, dynamic string

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.24

Funding *Takuya Mieno*: JSPS KAKENHI Grant Numbers JP22K21273 and JP23H04381
Kazuhisa Seto: JSPS KAKENHI Grant Number JP22H00513
Takashi Horiyama: JSPS KAKENHI Grant Number JP20H05964

1 Introduction

Periodicity and repetitive structure in strings are important concepts in the field of stringology and have applications in various areas, such as pattern matching and data compression. A string u is called a *period-string* (or simply a *period*) of string T if $T = u^k u'$ holds for some positive integer k and some prefix u' of u. While periods accurately capture the repetitive structure of strings, the definition is too restrictive. In contrast, alternative concepts that capture a sort of periodicity with relaxed conditions have been studied. A *cover* (a.k.a. *quasi-period*) of a string is a typical example of such a concept [5, 6]. A string v is called a cover of T if every character in T lies within some occurrence of v. In other words, T can be written as a repetition of occurrences of v that are allowed to overlap. By definition, a cover of T must occur as both a prefix and a suffix of T, and such string is called a *border* of T. Therefore, a cover of T is necessarily a border of T. For instance, $v =$ `aba` is a cover for $S =$ `abaababa`, and v is both a prefix and a suffix of T. Then, the string $v =$ `aba` of length 3 can be regarded as an "almost" period-string in S while the shortest period-string of S is `abaab` of length 5. Thus, covers can potentially discover quasi-repetitive structures not captured by periods. The concept of covers (initially termed quasi-periods) was introduced by Apostolico and Ehrenfeucht [5, 6]. Subsequently, an algorithm to compute the shortest cover offline in linear time was proposed by Apostolico et al. [7]. Furthermore, an online and linear-time method was presented by Breslauer [9]. Gawrychowski et al. explored cover computations in streaming models [14]. In their problem setting, the computational complexity is stochastic. Other related work on covers can be found in the survey paper by Mhaskar and Smyth [20].

In this paper, we investigate the changes in the shortest cover of a string T when T is edited and design algorithms to compute it. As mentioned above, the shortest cover of T is necessarily a border of T, so we first consider how to compute borders when T is edited. To the best of our knowledge, there is only one explicitly-stated result on the computation of borders in a dynamic setting: the longest border of a string S (equivalently, the smallest period of S) can be maintained in $O(|S|^{o(1)})$ time per character substitution operation (Corollary 19 of [2]). Also, although is not stated explicitly, an $O(\log^3 n)$-time (w.h.p.) algorithm can be obtained by using the results on the *PILLAR model* in dynamic strings [11]. We are unsure whether their results can be applied to compute the shortest cover in a dynamic string. Instead, we focus on studying the changes in covers when a *factor* is edited only once. We believe that this work will be the first step towards the computation of covers for a fully-dynamic string. We now introduce two problems: the LBAE (longest border after-edit) query and the SCAE (shortest cover after-edit) query for the input string T of length n. The LBAE query (resp., the SCAE query) is, given an edit operation *on the original string* T as a query, to compute the longest border (resp., the shortest cover) of the edited string. We note that, after we answer a query, the edit operation is discarded. That is, the following edit operations are also applied to the original string T. This type of problem is called the *after-edit model* [3]. Also, in our problems, the edit operation includes insertion, deletion, or substitution of strings of length one or more. Our main contribution is designing an $O(n)$-size data structure that can answer both LBAE and SCAE queries in $O(\ell \log n)$ time, where ℓ is the length of the string being inserted or substituted. The data structures can be constructed in $O(n)$ time.

Related Work on After-Edit Model

The after-edit model was formulated by Amir et al. [3]. They proposed an algorithm to compute the *longest common factor* (LCF) of two strings in the after-edit model. This problem allows editing operations on only one of the two strings. Abedin et al. [1] subsequently improved their results. Later, Amir et al. [4] generalized this problem to a *fully-dynamic* model and proposed an algorithm that maintains the LCF in $\tilde{O}(n^{\frac{2}{3}})$ time[1] per edit operation. Charalampopoulos et al. [10] improved the maintenance time to amortized $\tilde{O}(1)$ time with high probability per substitution operation. Urabe et al. [23] addressed the problem of computing the *longest Lyndon factor* (LLF) of a string in the after-edit model. The insights gained from their work were later applied to solve the problem of computing the LLF of a fully-dynamic string [4]. Problems of computing the *longest palindromic factor* and *unique palindromic factors* in a string were also considered in the after-edit model [13, 12, 21].

2 Preliminaries

2.1 Basic Definitions and Notations

Strings. Let Σ be an *alphabet*. An element in Σ is called a *character*. An element in $\Sigma^\star$ is called a *string*. The length of a string S is denoted by $|S|$. The string of length 0 is called the *empty string* and is denoted by ε. If a string S can be written as a concatenation of

[1] The $\tilde{O}(\cdot)$ notation hides poly-logarithmic factors.

three strings p, f and s, i.e., $S = pfs$, then p, f and s are called a *prefix*, a *factor*, and a *suffix* of S, respectively. Also, if $|p| < |S|$ holds, p is called a *proper* prefix of S. Similarly, s is called a proper suffix of S if $|s| < |S|$ holds. For any integer i, j with $1 \leq i \leq j \leq |S|$, we denote by $T[i]$ the i-th character of S, and by $T[i..j]$ the factor of S starting at position i and ending at position j. For convenience, let $T[i'..j'] = \varepsilon$ for any i', j' with $i' > j'$. For two strings S and T, we denote by $LCP(S, T)$ the *longest common prefix* of S and T. Also, we denote by $lcp(S, T)$ the length of $LCP(S, T)$. If $f = S[i..i + |f| - 1]$ holds, we say that f *occurs* at position i in S. Let $occ_S(f) = \{i \mid f = S[i..i + |f| - 1]\}$ be the set of occurrences of f in S. Further let $cover_S(f) = \{p \mid p \in [i, i + |f| - 1]$ for some $i \in occ_S(f)\}$ be the set of positions in S that are covered by some occurrence of f in S. A string f is called a *cover* of S if $cover_S(f) = \{1, \ldots, |S|\}$ holds. A string b is called a *border* of a non-empty string S if b is both a proper prefix of S and a proper suffix of S. We say that S has a border b when b is a border of S. By definition, any non-empty string has a border ε. If a string S has a border b, integer $p = |S| - |b|$ is called a *period* of S. We sometimes call the smallest period of S *the* period of S. Similarly, we call the longest border of S *the* border of S, and the shortest cover of S *the* cover of S. Also, we denote by $\mathsf{per}(S)$, $\mathsf{bord}(S)$, and $\mathsf{cov}(S)$ the period of S, the border of S, and the cover of S, respectively. The rational number $|S|/\mathsf{per}(S)$ is called the *exponent* of S. We say that S is *periodic* if $\mathsf{per}(S) \leq |S|/2$. A string S is said to be *superprimitive* if $\mathsf{cov}(S) = S$.

After-edit Model. The *after-edit model* is, given an edit operation on the input string T as a query, to compute the desired objects on the edited string T' that is obtained by applying the edit operation to T. Note that in the after-edit model, each query, namely each edit operation, is discarded after we finish computing the desired objects on T', so the next edit operation will be applied to the original string T. In this paper, edit operations consist of inserting a string and substituting a factor with another string. Note that factor substitutions contain factor deletions since substituting a factor with the empty string ε is identical to deleting the factor. We denote an edit operation as $\phi(i, j, w)$ where $1 \leq j \leq |T|$, $1 \leq i \leq j + 1$ and $w \in \Sigma^\star$: if $i \leq j$, $\phi(i, j, w)$ means to substitute $T[i..j]$ for w. If $i = j + 1$, $\phi(i, j, w)$ means to insert w just after $T[i-1]$. In both cases, the resulting string is $T' = T[1..i-1]wT[j+1..|T|]$ and thus $T'[i..i + |w| - 1] = w$. For a given query $\phi(i, j, w)$, let $L_{i,j} = T[1..i - 1]$ and $R_{i,j} = T[j + 1..|T|]$. We will omit the subscripts when they are clear from the context. Thus, $T' = LwR$. We consider the two following problems with the after-edit model:

LBAE (Longest Border After-Edit) query

Preprocess: A string T of length n.
Query: An edit-operation $\phi(i, j, w)$.
Output: The longest border of $T' = L_{i,j}wR_{i,j}$.

SCAE (Shortest Cover After-Edit) query

Preprocess: A string T of length n.
Query: An edit-operation $\phi(i, j, w)$.
Output: The shortest cover of $T' = L_{i,j}wR_{i,j}$.

In the following, we fix the input string T of arbitrary length $n > 0$. Also, we assume that the computation model in this paper is the word-RAM model with word size $\Omega(\log n)$. We further assume that the alphabet Σ is *linearly-sortable*, i.e., we can sort n characters from the input string in $O(n)$ time.

2.2 Combinatorial Properties of Borders and Covers

This subsection describes known properties of borders and covers.

Periodicity of Borders

Let us consider partitioning the set $\mathcal{B}_T$ of borders of a string T of length n. Let $G_1, G_2, \ldots, G_m$ be the sets of borders of T such that $\{G_1, G_2, \ldots, G_m\}$ is a partition of $\mathcal{B}_T$ and for each set, all borders in the same set have the same smallest period. Let p_k be the period of borders belonging to G_k. Without loss of generality, we assume that they are indexed so that $p_k > p_{k+1}$ for every $1 \le k < m$. We call G_k the *k-th group*. Then, the following fact is known:

▶ **Proposition 1** ([17, 16]). *For the partition $\{G_1, G_2, \ldots, G_m\}$ of $\mathcal{B}_T$ defined above, the following statements hold:*
1. *For each $1 \le k \le m$, the lengths of borders in the k-th group can be represented as a single arithmetic progression with common difference p_k.*
2. *If a group contains at least three elements, the borders in the group except for the shortest one are guaranteed to be periodic.*
3. *The number m of sets is in $O(\log n)$.*

Properties of Covers

The following lemma summarizes some basic properties of covers, which we will use later.

▶ **Lemma 2** ([9, 22]). *For any string T, the following statements hold.*
1. *The cover $\mathsf{cov}(T)$ of T is either $\mathsf{cov}(\mathsf{bord}(T))$ or T itself.*
2. *$\mathsf{cov}(T)$ is superprimitive and non-periodic.*
3. *Let v be a cover of T, and u be a factor of T which is shorter than v. Then u is a cover of T if and only if u is a cover of v.*

2.3 Algorithmic Tools

This subsection shows algorithmic tools we will use later.

Border Array and Border-group Array

The *border array* B_T of a string T is an array of length n such that $\mathsf{B}_T[i]$ stores the length of the border of $T[1..i]$ for each $1 \le i \le n$. Also, for convenience, let $\mathsf{B}_T[0] = 0$ for any string T. There is a well-known *online* algorithm for linear-time computation of the border array (e.g., see [15]). While the worst-case running time of the algorithm is $O(n)$ per a character, it can be made $O(\log n)$ by constructing B with the *strict border array* proposed in [17].

▶ **Lemma 3** ([15, 17]). *For each $1 \le i \le n$, if we have $T[1..i-1]$ and $\mathsf{B}_{T[1..i-1]}$, then we can compute $\mathsf{B}_{T[1..i]}$ in worst-case $O(\log n)$ time and amortized $O(1)$ time given the next character $T[i]$.*

Next, we introduce a data structure closely related to the border array. The *border-group array* BG_T of a string T is an array of length n such that, for each $1 \le i \le n$, $\mathsf{BG}_T[i]$ stores the length of the shortest border of $T[1..i]$ whose smallest period equals $\mathsf{per}(T[1..i])$ if such a border exists, and $\mathsf{BG}_T[i] = i$ otherwise. By definition, if $T[1..i]$ is a border of T and it belongs to group G_k, then $\mathsf{BG}_T[i]$ stores the first (the smallest) term of the arithmetic

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i]$		a	b	a	b	a	b	a	a	b	a	b	a	b	a	a	b	a
$\mathsf{B}_T[i]$	0	0	0	1	2	3	4	5	1	2	3	4	5	6	7	8	9	10
$\mathsf{per}(T[1..\,i])$		1	2	2	2	2	2	2	7	7	7	7	7	7	7	7	7	7
$\mathsf{per}(\mathsf{bord}(T[1..\,i]))$		-	-	1	2	2	2	2	1	2	2	2	2	2	2	7	7	7
$\mathsf{BG}_T[i]$		1	2	3	2	3	2	3	8	9	10	11	12	13	14	8	9	10

Figure 1 An example of a border array and a border-group array. For position $i = 7$, the period of $T[1..7]$ is 2. All borders of $T[1..7]$ are `ababa`, `aba`, `a`, and ε. Also, their smallest periods are 2, 2, 1, and 0, respectively. Thus $\mathsf{BG}_T[7] = |\texttt{aba}| = 3$. For position $i = 8$, the period of $T[1..8]$ is 7. Any border of $T[1..8]$ does not have period 7, and thus $\mathsf{BG}_T[8] = i = 8$.

progression representing the lengths of borders in G_k. This is why we named BG_T the border-group array. Also, the common difference $p_k = i - \mathsf{B}_T[i]$ can be obtained from B_T if $\mathsf{BG}_T[i] \neq i$. See Figure 1 for an example. We can compute the border-group array in linear time in an online manner together with B_T. Before proving it, we note a fact about periods.

▶ **Proposition 4.** *If u is a factor of v, then* $\mathsf{per}(u) \leq \mathsf{per}(v)$.

▶ **Lemma 5.** *For each $1 < i \leq n$, if we have $T[1..i-1]$, $\mathsf{B}_{T[1..i-1]}$, and $\mathsf{BG}_{T[1..i-1]}$, then we can compute $\mathsf{BG}_{T[1..i]}$ in amortized $O(1)$ time given the next character $T[i]$.*

Proof. By definition, $\mathsf{B}_{T[1..1]} = [0]$ and $\mathsf{BG}_{T[1..1]} = [1]$. Assume that we have $\mathsf{B}_{T[1..i-1]}$, $\mathsf{BG}_{T[1..i-1]}$, and $T[1..i]$ for $i \geq 2$. By Lemma 3, we can obtain $\mathsf{B}_{T[1..i]}$ in amortized $O(1)$ time. Now let $p = i - \mathsf{B}_{T[1..i]}[i]$ and $q = \mathsf{B}_{T[1..i]}[i] - \mathsf{B}_{T[1..i]}[\mathsf{B}_{T[1..i]}[i]]$, meaning that $p = \mathsf{per}(T[1..i])$ and $q = \mathsf{per}(\mathsf{bord}(T[1..i]))$. If $p = q$, then we set $\mathsf{BG}_{T[1..i]}[i] = \mathsf{BG}_{T[1..i-1]}[\mathsf{B}_{T[1..i]}[i]]$ since $\mathsf{per}(T[1..i]) = \mathsf{per}(\mathsf{bord}(T[1..i]))$. Otherwise, $\mathsf{per}(T[1..i]) > \mathsf{per}(\mathsf{bord}(T[1..i]))$, and thus, the period of *any* border of $T[1..i]$ is smaller than $\mathsf{per}(T[1..i])$ by Proposition 4. Hence we set $\mathsf{BG}_{T[1..i]}[i] = i$. The running time of the algorithm is (amortized) $O(1)$. ◀

Longest Common Extension Query

The *longest common extension* query (in short, *LCE* query) is, given positions i and j within T, to compute $lcp(T[i..|T|], T[j..|T|])$. We denote the answer of the query as $lce_T(i, j)$. We heavily use the following result in our algorithms.

▶ **Lemma 6** (E.g., [8]). *We can answer any LCE query in $O(1)$ time after $O(n)$-time and space preprocessing on the input string T.*

Prefix Table

The *prefix table* Z_S of a string S of length m is an array of length m such that $\mathsf{Z}_S[i] = LCP(S, S[i..m])$ for each $1 \leq i \leq m$.

▶ **Lemma 7** ([19, 15]). *Given a string S of length m over a general unordered alphabet, we can compute the prefix table Z_S in $O(m)$ time*[2].

We emphasize that this linear-time algorithm does not require linearly-sortability of the alphabet.

[2] The algorithm described in [15] is known as *Z-algorithm*, so we use Z to represent the prefix table.

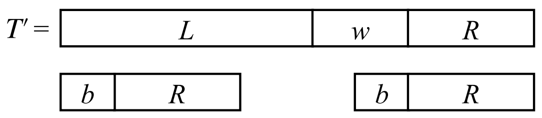


Figure 2 A border of T' which is longer than R is written as bR where b is a border of Lw.

Internal Pattern Matching

The *internal pattern matching* query (in short, *IPM* query) is, given two factors u, v of T with $|v| \leq 2|u|$, to compute the occurrences of u in v. The output is represented as an arithmetic progression due to the lengths constraint and periodicity [18]. If u occurs in v, we denote by $\mathsf{rightend}(u, v)$ the ending position of the rightmost occurrence of u in v.

▶ **Lemma 8** ([18]). *We can answer any IPM query in $O(1)$ time after $O(n)$-time and space preprocessing on the input string T,*

3 Longest Border After Edit

This section proposes an algorithm to solve the LBAE problem. In the following, we assume that $|L| \geq |R|$ and $|w| \leq |L|/2$ for a fixed query $\phi(i, j, w)$. Because, when $|L| < |R|$, running our algorithm on the reversal inputs can answer LBAE queries without growing complexities. Also, if $|w| > |L|/2$, then $|w| > |T'|/5$ holds since $T' = LwR$ and $|L| \geq |R|$. We can obtain the border of T' in $O(|T'|) = O(|w|)$ time by computing the border array of T' from scratch.

We compute the border of T' in the following two steps. **Step 1**: Find the longest border of T' which is longer than R. **Step 2**: Find the longest border of T' of length at most $|R|$ if nothing is found in step 1. Step 2 can be done in constant time by pre-computing all borders of T and the longest border of T of length at most k for each k with $1 \leq k \leq n$. Thus we focus on step 1, i.e., how to find the longest border of T' which is longer than R. We observe that such a border is the concatenation of some border of Lw and R (see Figure 2). By pre-computing the border array B_T, the border array B_{Lw} of Lw can be computed in $O(|w| \log n)$ time starting from $\mathsf{B}_L = \mathsf{B}_T[1..|L|]$ (Lemma 3). Let b_{Lw} be the border of Lw. There are two cases: (i) $|b_{Lw}| \leq |w|$ or (ii) $|b_{Lw}| > |w|$. We call the former case the short border case and the latter case the long border case.

3.1 Short Border Case

In this case, the length of the border of $T' = LwR$ is at most $|b_{Lw}R| \leq |wR| \leq |Lw|$, so its prefix-occurrence ends within Lw. Also, for any border b of Lw, string bR is a border of T' if and only if $lce_{T'}(|b| + 1, |Lw| + 1) = |R|$ holds. Thus, we pick up each border b of Lw in descending order of length and check whether bR is a border of T' by computing $lce_{T'}(|b| + 1, |Lw| + 1)$. Since Lw has at most $|w|$ borders, constant-time LCE computation results in a total of $O(|w|)$ time. If $|b| + |R| \leq |L|$ then we can use the LCE data structure on the original string T since $lce_{T'}(|b| + 1, |Lw| + 1) = lce_T(|b| + 1, j + 1)$ holds. Otherwise, we may compute the longest common prefix of w and some suffix of R, which cannot be computed by applying LCE queries on T naïvely. To resolve this issue, we compute the prefix table Z_W of W in $O(|W|) = O(|w|)$ time where $W = w \cdot R[|R| - |w| + 1..|R|]$ is the concatenation of w and the length-$|w|$ suffix of R. Note that $|R| > |w|$ holds here since $|R| > |L| - |b| \geq |L| - |w| \geq |w|$ by the assumptions in this case. Then the longest common prefix of w and any suffix of R of length at most $|w|$ is obtained in constant time, and so is $lce_{T'}(|b| + 1, |Lw| + 1)$. Therefore, we can compute $lce_{T'}(|b| + 1, |Lw| + 1)$ for all borders b of Lw in a total of $O(|w| \log n)$ time.

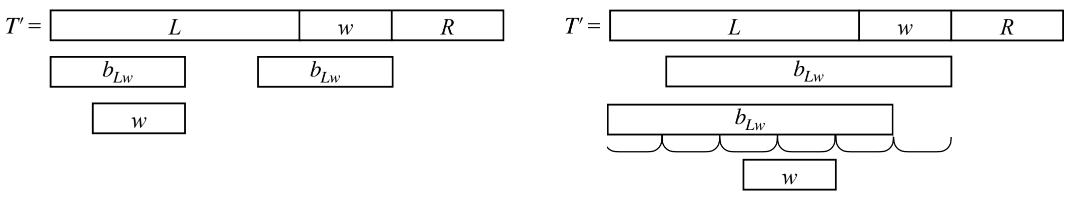


Figure 3 Left: If b_{Lw} is not longer than L, then w occurs within L as a suffix of the prefix-occurrence of b_{Lw} in L. Right: If b_{Lw} is longer than L, then w occurs within L because of the periodicity of b_{Lw}.

3.2 Long Border Case

Firstly, we give some observations for the long border case; $|b_{Lw}| > |w|$. If $|b_{Lw}| \leq |L|$, then $w = L[|b_{Lw}| - |w| + 1..|b_{Lw}|]$ holds. If $|b_{Lw}| > |L|$, then the period p_{Lw} of Lw is $p_{Lw} = |Lw| - |b_{Lw}| < |Lw| - |L| = |w|$. Let k be the smallest integer such that $kp_{Lw} \geq |w|$. Since $k \geq 2$, $kp_{Lw} \leq 2(k-1)p_{Lw} < 2|w| \leq |L|$ holds. Thus $w = L[|L| - kp_{Lw} + 1..|L| - kp_{Lw} + |w|]$ holds (see also Figure 3). Thus, in both cases, w occurs *within* L, which is a factor of the original T. From this observation, any single LCE query on T' can be simulated by constant times LCE queries on T because any LCE query on $w = T'[|L|+1..|L|+|w|]$ can be simulated by a constant number of LCE queries on another occurrence of w within L. Therefore, in the following, we use the fact that any LCE query on $T' = LwR$ can be answered in $O(1)$ time as a black box.

Now, we show some properties of the border of T'. As we mentioned in Proposition 1, the sets of borders of Lw can be partitioned into $m \in O(\log n)$ groups w.r.t. their smallest periods. Let $G_1, G_2, \ldots, G_m$ be the groups such that $p_k > p_{k+1}$ for every $1 \leq k < m$, where p_k is the period of borders in G_k. Next, let us assume that there exists a border of T' which is longer than R. Let $b^\star$ be the border of Lw such that $b^\star R$ is the border of T'. Further let $k^\star$ be the index of the group to which $b^\star$ belongs. There are three cases: (i) $b^\star$ is periodic and $\mathsf{per}(b^\star) = p_{k^\star} = \mathsf{per}(b^\star R)$, (ii) $b^\star$ is periodic and $\mathsf{per}(b^\star) = p_{k^\star} \neq \mathsf{per}(b^\star R)$, or (iii) $b^\star$ is not periodic. The first two cases are illustrated in Figure 4. For the case (i), the following lemma holds. Here, for a group G_k, let α_k be the exponent of the longest prefix of T' with period p_k.

▶ **Lemma 9.** *If $b^\star$ is periodic and $p_{k^\star} = \mathsf{per}(b^\star R)$, then $|b^\star| \leq \alpha_{k^\star} p_{k^\star} - |R|$ holds.*

Proof. Assume the contrary that $|b^\star| > \alpha_{k^\star} p_{k^\star} - |R|$ holds. Then $|b^\star R|/p_{k^\star} > \alpha_{k^\star}$ holds. This contradicts the maximality of $\alpha_{k^\star}$ since $b^\star R$ occurs as a prefix of T' and $\mathsf{per}(b^\star R) = p_{k^\star}$. ◀

For the case (ii), the following lemma holds. Here, for a group G_k, let $r_k = lce_{T'}(|T'| - |R| - p_k + 1, |T'| - |R| + 1)$.

▶ **Lemma 10.** *If $b^\star$ is periodic and $p_{k^\star} \neq \mathsf{per}(b^\star R)$, then $|b^\star| = \alpha_{k^\star} p_{k^\star} - r_{k^\star}$ holds.*

Proof. Since the period of $b^\star$ is $p_{k^\star}$, the longest prefix of $T'[|T'| - |b^\star R| + 1..|T'|]$ with period $p_{k^\star}$ is of length $|b^\star| + r_{k^\star}$. Thus, by the definition of $\alpha_{k^\star}$, $\alpha_{k^\star} p_{k^\star} = |b^\star| + r_{k^\star}$ holds since $T'[1..|b^\star R|] = T'[|T'| - |b^\star R| + 1..|T'|]$. Therefore, $|b^\star| = \alpha_{k^\star} p_{k^\star} - r_{k^\star}$. ◀

Clearly, if $p_{k^\star} = \mathsf{per}(b^\star R)$, then $r_{k^\star} = |R|$ holds. Hence, by combining the two above lemmas, we obtain the next corollary:

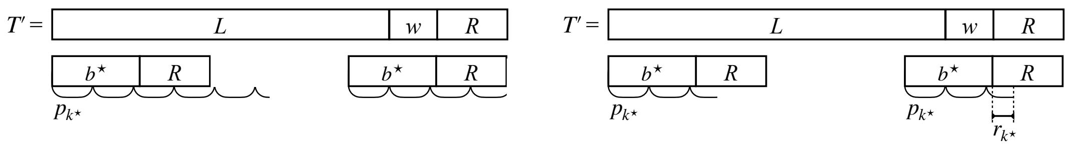


Figure 4 Left: Illustration for the case (i) $b^\star$ is periodic and $\mathsf{per}(b^\star) = p_{k^\star} = \mathsf{per}(b^\star R)$. The period $p_{k^\star}$ repeats five times and a little more in T'. Then $|b^\star|$ is at most the length of the repetition minus $|R|$ (Lemma 9). Right: Illustration for the case (ii) $b^\star$ is periodic and $\mathsf{per}(b^\star) = p_{k^\star} \neq \mathsf{per}(b^\star R)$. Since the maximal repetition of period $p_{k^\star}$ ends within R, the length $|b|$ is equal to the length of the maximal repetition minus $r_{k^\star}$ where $r_{k^\star}$ is the length of the suffix of the repetition that enters R (Lemma 10).

▶ **Corollary 11.** *If $b^\star$ is periodic, then $b^\star$ is the longest border of Lw whose length is at most $\alpha_{k^\star} p_{k^\star} - r_{k^\star}$.*

Based on this corollary, we design an algorithm to answer the LBAE queries.

Algorithm

The idea of our algorithm is as follows: given a query, we first initialize *candidates-set* $\mathcal{C} = \emptyset$, which will be a set of candidates for the length of the border of T'. Next, for each group of borders of Lw, we calculate a constant number of candidates from the group and add their lengths to the candidates-set $\mathcal{C}$ (the details are described below). In the end, we choose the maximum from $\mathcal{C}$ and output it.

Now we consider the k-th group G_k for a fixed k and how to calculate candidates. If $|G_k| \le 2$, we just try to extend each border in G_k to the right by using LCE queries on T', and if the extension reaches the right-end of T', we add its length to $\mathcal{C}$. Note that we do not care about the periodicity of borders here. Otherwise, we compute α_k and r_k by using LCE queries on T'. Let $\tilde{b}_k$ be the longest element in G_k whose length is at most $\alpha_k p_k - r_k$, if such a border exists. If $\tilde{b}_k$ is defined, we check whether $\tilde{b}_k R$ is a border of T' or not, again by using an LCE query on T'. If $\tilde{b}_k R$ is a border of T', we add its length to $\mathcal{C}$. Also, we similarly check whether $b_k^{\min} R$ is a border of T', and if so, add its length to $\mathcal{C}$, where $b_k^{\min}$ is the shortest element in G_k, which may be non-periodic.

Correctness

If a group G_k contains at least three elements, the borders in G_k except for the shortest one are periodic (Proposition 1). Namely, any non-periodic border of Lw is either an element of a group whose size is at most two, or the shortest element of a group whose size is at least three. Both cases are completely taken care of by our algorithm. For periodic borders, it is sufficient to check the longest border $\tilde{b}_k$ of Lw whose length is at most $\alpha_k p_k - r_k$ for each group G_k by Corollary 11. Therefore, the length of the border of T' must belong to the candidates-set $\mathcal{C}$ obtained at the end of our algorithm.

Running Time

Given a query $\phi(i, j, w)$, we can obtain the border array B_{Lw} and the border-group array BG_{Lw} in $O(|w| \log n)$ time from $\mathsf{B}_T[1..|L|]$ and $\mathsf{BG}_T[1..|L|]$ (Lemmas 3 and 5). Thus, by using those arrays, while we scan the groups $G_1, \ldots, G_m$, we can determine whether the current group G_k has at least three elements or not, and compute the first term and the common difference of the arithmetic progression representing the current group G_k both in constant time. All the other operations consist of LCE queries on T' and basic arithmetic operations, which can be done in constant time. Finally, we choose the maximum from $\mathcal{C}$, which can be done $O(|\mathcal{C}|)$ time. Since we add at most two elements to $\mathcal{C}$ when we process each group, the size of $\mathcal{C}$ is in $O(m)$. Thus the total running time is in $O(|w|\log n + m) \subseteq O(|w|\log n)$ since $m \in O(\log n)$.

To summarize this section, we obtain the following theorem.

▶ **Theorem 12.** *The longest border after-edit query can be answered in $O(\ell \log n)$ time after $O(n)$-time preprocessing, where ℓ is the length of the string to be inserted or substituted specified in the query.*

4 Shortest Cover After Edit

This section proposes an algorithm to solve the SCAE problem. Firstly, we give additional notations and tools. For a string S and an integer k with $1 \le k \le |S|$, $\mathsf{range}(S,k)$ denotes the largest integer r such that $S[1..k]$ can cover $S[1..r]$. Next, we give definitions of two arrays $\mathsf{C}(T)$ and $\mathsf{R}(T)$ introduced in [9]. The former $\mathsf{C}(T)$ is called the *cover array* and stores the length of the cover of each prefix of T, i.e., $\mathsf{C}(T)[k] = |\mathsf{cov}(T[1..k])|$ for each k with $1 \le k \le n$. For convenience, let $\mathsf{C}(T)[0] = 0$. The latter $\mathsf{R}(T)$ is called the *range array* that stores the values of range function only for *superprimitive prefixes* of T, i.e., for each k, $\mathsf{R}(T)[k] = \mathsf{range}(T,k)$ if $\mathsf{cov}(T[1..k]) = T[1..k]$, and otherwise $\mathsf{R}(T)[k] = 0$, meaning undefined. Cover array and range array can be computed in $O(n)$ time given T in an online manner [9]. In describing our algorithm, we use the next lemma:

▶ **Lemma 13.** *Assume that we already have data structure $\mathcal{D}_T$ consisting of the IPM data structure on T of Lemma 8, border array $\mathsf{B}(T)$, cover array $\mathsf{C}(T)$, range array $\mathsf{R}(T)$, and an array $\mathsf{R}^\star$ of size n initialized with $\mathbf{0}$. Given a query $\phi(i,j,w)$, we can enhance $\mathcal{D}_T$ in $O(|w|\log n)$ time so that we can obtain $\mathsf{cov}((Lw)[1..k])$ for any k with $1 \le k \le |Lw|$ and $\mathsf{range}(Lw,k')$ for any k' such that $1 \le k' \le |L|$ and $\mathsf{cov}(L[1..k']) = L[1..k']$ in $O(1)$ time.*

Due to lack of space, we only give the idea of the proof of Lemma 13. To prove the lemma, we first review Breslauer's algorithm [9] that computes $\mathsf{C}(T)$ and $\mathsf{R}(T)$ for a given string T in an online manner (Algorithm 1). By Lemma 3, we can compute $\mathsf{B}(Lw)$ in $O(|w|\log n)$ time if $\mathsf{B}(L)$ and w are given. Since Algorithm 1 runs in an online manner, if we have $\mathsf{C}(L)$ and $\mathsf{R}(L)$ in addition to $\mathsf{B}(Lw)$, then it is easy to obtain $\mathsf{C}(Lw)$ and $\mathsf{R}(Lw)$ in $O(|w|)$ time by running Algorithm 1 starting from the $(|L|+1)$-th iteration. However, we only have $\mathsf{C}(T)$ and $\mathsf{R}(T)$, not $\mathsf{C}(L)$ and $\mathsf{R}(L)$. The idea of our algorithm for Lemma 13 is to simulate $\mathsf{C}(L \cdot w[1..t-1])$ and $\mathsf{R}(L \cdot w[1..t-1])$ while iterating the while-loop of Algorithm 1 from $t = 1$ ($idx = |L|+1$) to $t = |w|$ ($idx = |L|+|w|$). We also show a complete pseudocode in Algorithm 2.

Note that we can prepare the input $\mathcal{D}_T$ of Lemma 13 in $O(n)$ time for a given T.

Overview of Our Algorithm for SCAE Queries

To compute the cover of T', we first run the LBAE algorithm of Section 3. Then, there are two cases: (i) The *non-periodic case*, where the length of $\mathsf{bord}(T')$ is smaller than $|T'|/2$, or (ii) the *periodic case*, the other case.

Algorithm 1 Algorithm to compute $\mathsf{C}(T)$ proposed in [9].

```
Require: The border array $\mathsf{B}(T)$ of string $T$, and two arrays $C[0..n] = R[0..n] = \mathbf{0}$.
Ensure: $C = \mathsf{C}(T)$ and $R = \mathsf{R}(T)$
 1: $idx \leftarrow 1$
 2: while $idx \leq n$ do
 3:     $clen \leftarrow C[\mathsf{B}(T)[idx]]$                ▷ $clen < idx$ always holds.
 4:     if $clen > 0$ and $R[clen] \geq idx - clen$ then
 5:         $C[idx] \leftarrow clen$
 6:         $R[clen] \leftarrow idx$ ▷ When $T[1..idx]$ is not superprimitive, $R[clen]$ is updated to $idx$.
 7:     else
 8:         $C[idx] \leftarrow idx$
 9:         $R[idx] \leftarrow idx$      ▷ When $T[1..idx]$ is superprimitive, $R[idx]$ is newly defined.
10:     end if
11:     $idx \leftarrow idx + 1$
12: end while
```

4.1 Non-periodic Case

Let $b = \mathsf{bord}(T')$ and $c = \mathsf{cov}(b)$. By the first statement of Lemma 2, $\mathsf{cov}(T') = \mathsf{cov}(\mathsf{bord}(T'))$ if $\mathsf{cov}(\mathsf{bord}(T'))$ can cover T', and $\mathsf{cov}(T') = T'$ otherwise. In the following, we consider how to determine whether $c = \mathsf{cov}(\mathsf{bord}(T'))$ is a cover of T' or not.

Let s be the maximum length of the prefix of Lw that c can cover. Further let t be the maximum length of the suffix of wR that c can cover if $|c| \leq |wR|$, and $t = |c|$ otherwise. By Lemma 13, the values of s and t can be obtained in $O(|w| \log n)$ time by computing values of $\mathsf{range}(Lw, |c|)$ and $\mathsf{range}((wR)^R, |c|)$ since $c = \mathsf{cov}(b)$ is superprimitive (by the second statement of Lemma 2), where $(wR)^R$ denotes the reversal of wR. If $s + t \geq |T'|$ then c is a cover of T'. Thus, $\mathsf{cov}(T') = c$, and the algorithm is terminated.

We consider the other case, where $s + t < |T'|$. The inequality $s + t < |T'|$ means that the occurrences of c within Lw or wR cannot cover the middle factor $T'[s + 1..|T'| - t]$ of T'. Thus, if c is a cover of T' when $s + t < |T'|$, then c must have an occurrence that starts in L and ends in R. Such an occurrence can be written as a concatenation of some border of Lw which is longer than w and some (non-empty) prefix of R. Similar to the method in Section 3.2, we group the borders of Lw using their periods and process them for each group. Again, let $G_1, \ldots, G_m$ be the groups sorted in descending order of their smallest periods.

Let us fix a group G_k arbitrarily. If $|G_k| \leq 2$, we simply try to extend each border in G_k to the right by using LCE queries. Now we use the following claim:

▷ Claim 14. For a border z of Lw with $|z| > |w|$, the value of $lce_{T'}(|z| + 1, |Lw| + 1)$ can be computed in constant time by using the LCE data structure of Lemma 6 on T.

This claim can be proven by similar arguments as in the first paragraph of Section 3.2. Thus, the case of $|G_k| \leq 2$ can be processed in constant time. If $|G_k| > 2$, we use the period p_k of borders in G_k. Let α_k be the exponent of the longest prefix of T' with period p_k. Further let $r_k = lce_{T'}(|T'| - |R| - p_k + 1, |T'| - |R| + 1)$. Note that $\alpha_k p_k < |c|$ since c is a prefix of T' and is non-periodic. See also Figure 5. By using LCE queries on T', α_k and r_k can be computed in constant time. For a border z in G_k, $T'[|z| + 1..|c|] = T'[|Lw| + 1..|Lw| + |c| - |z|]$ holds only if $|z| = \alpha_k p_k - r_k$. Thus, the only candidate for a border in G_k which can be extended to the right enough is of length exactly $\alpha_k p_k - r_k$ if it exists. The existence of such a border can be determined in constant time since the lengths of the borders in G_k are represented

Algorithm 2 Algorithm to compute data structures which can simulate $\mathsf{C}(Lw)$ and $\mathsf{R}(Lw)$.

```
Require: $\mathsf{B}(T)$, $\mathsf{C}(T)$, $\mathsf{R}(T)$, $\mathsf{R}^\star[1..n] = \mathbf{0}$, and $\phi(i, j, w)$.
Ensure: (i) $C_w[1..|w|] = \mathsf{C}(Lw)[|L|+1..|Lw|]$ and (ii) $\mathsf{R}^\star[k] = \mathsf{R}(Lw)[k]$ if $1 \le k \le |Lw|$ and
    $\mathsf{R}(T)[k] \ne \mathsf{R}(Lw)[k] > |L|$, and $\mathsf{R}^\star[k] = 0$ otherwise.
1:  $idx \leftarrow |L| + 1$                                  ▷ Starting from $(|L|+1)$-th position.
2:  while $idx \le |Lw|$ do
3:      $clen \leftarrow \mathsf{C}(T)[\mathsf{B}(T)[idx]]$
4:      if $clen > 0$ then                                    ▷ $T[1..clen]$ is superprimitive.
5:          if $\mathsf{R}^\star[clen] \ne 0$ then
6:              $r \leftarrow \mathsf{R}^\star[clen]$
7:          else if $\mathsf{R}(T)[clen] \le |L|$ then
8:              $r \leftarrow \mathsf{R}(T)[clen]$
9:          else                                              ▷ $\mathsf{R}^\star[clen] = 0$ and $\mathsf{R}(T)[clen] > |L|$
10:             $r \leftarrow \mathsf{rightend}(T[1..clen], T[|L| - 2clen + 2..|L|])$
11:         end if
12:         if $r \ge idx - clen$ then                        ▷ $r = \mathsf{R}(L \cdot w[1..idx - |L|])[clen]$
13:             $C_w[idx - |L|] \leftarrow clen$
14:             $\mathsf{R}^\star[clen] \leftarrow idx$                   ▷ $(Lw)[1..idx]$ is *not* superprimitive.
15:             $idx \leftarrow idx + 1$
16:             continue                                      ▷ Go to the next iteration.
17:         end if
18:     end if
19:     $C_w[idx - |L|] \leftarrow idx$
20:     $\mathsf{R}^\star[idx] \leftarrow idx$                           ▷ $(Lw)[1..idx]$ is superprimitive.
21:     $idx \leftarrow idx + 1$
22: end while
```

as an arithmetic progression. If such a border of length $\alpha_k p_k - r_k$ exists, then we check whether it can be extended to the desired string c by querying an LCE. Therefore, the total computation time is $O(1)$ for a single group G_k, and $O(\log n)$ time in total for all groups since there are $O(\log n)$ groups.

To summarize, we can compute $\mathsf{cov}(T')$ in $O(|w| \log n)$ for the non-periodic case.

4.2 Periodic Case

In this case, T' can be written as $(uv)^k u$ for some integer $k \ge 2$ and strings u, v with $|uv| = \mathsf{per}(T')$ since T' is periodic. By the third statement of Lemma 2, $\mathsf{cov}(T') = \mathsf{cov}(uvu)$ holds since uvu is a cover of T'. Thus, in the following, we focus on how to compute $\mathsf{cov}(uvu)$. We further divide this case into two sub-cases depending on the relation between the lengths of uvu and Lw.

If $|uvu| \le |Lw|$, then uvu is a prefix of Lw. Thus, by Lemma 13, $\mathsf{cov}(uvu) = \mathsf{cov}((Lw)[1..|uvu|])$ can be computed in $O(|w| \log n)$ time.

If $|uvu| > |Lw|$, then $T' = uvuvu$ since $|uvu| > n/2$. We call the factor $T[|uv|+1..|uvu|] = u$ the *second occurrence of* u. Also, since $|L| \ge |R| = |T'| - |Lw| > |T'| - |uvu| = |vu|$, both R and L are longer than uv. Thus vu is a suffix of R and uv is a prefix of L. Now let us consider the border of uvu.

▶ **Lemma 15.** *If the period of a string $T' = uvuvu$ is $|uv|$, then the border of uvu is not longer than $|uv|$.*

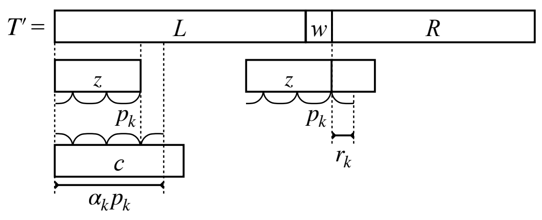


Figure 5 Illustration for the non-periodic case. Here, c is non-periodic, z is some border of Lw, and p_k is the period of z. If there is an occurrence of c starting in R and ending in R, then $|z| = \alpha_k p_k - r_k$ must hold since c is non-periodic.

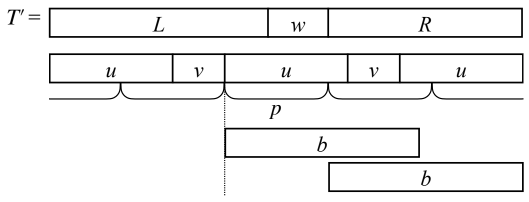


Figure 6 Illustration for a contradiction if we assume that uvu has a border which is longer than $|uv|$. Since $T' = uvuvu$, if uvu has a period which is smaller than $|u|$ then T' also has the same period.

Proof. If uvu has a border that is longer than $|uv|$, uvu has a period p which is smaller than $|u|$. Then the length-p prefix of the second occurrence of u repeats to the left and the right until it reaches both ends of T' (see Figure 6). This contradicts that $\mathsf{per}(T') = |uv|$. ◀

Therefore, the border of uvu is identical to the longest border of T whose length is at most $|uv|$, which can be obtained in constant time after $O(n)$-time preprocessing as in step 2 of Section 3. By the first statement of Lemma 2, $\mathsf{cov}(uvu)$ is either $\mathsf{cov}(\mathsf{bord}(uvu))$ or uvu. Since $|\mathsf{bord}(uvu)| \le |uv| < |Lw|$, $\mathsf{cov}(\mathsf{bord}(uvu))$ can be obtained in $O(|w| \log n)$ time by Lemma 13. Let $x = \mathsf{cov}(\mathsf{bord}(uvu))$. Thanks to Lemma 16 below, we do not have to scan $O(\log n)$ groups, unlike the non-periodic case.

▶ **Lemma 16.** *When $|uvu| > |Lw|$, string $x = \mathsf{cov}(\mathsf{bord}(uvu))$ covers uvu if and only if $\mathsf{range}(Lw, |x|) \ge |uvu| - \max\{|u|, |x|\}$ holds.*

Proof. Let $r = \mathsf{range}(Lw, |x|)$. We divide the proof into three cases.

The case when $|x| \le |u|/2$: In this case, x is a border of u and the occurrence of x as the prefix of the second occurrence of u ends within Lw. ($\Longrightarrow$) If x covers uvu, then $r \ge |uv| + |x| > |uv| = |uvu| - |u|$. ($\Longleftarrow$) If $r \ge |uvu| - |u| = |uv|$ holds, then x covers uvx and u. Hence x covers uvu (see the left figure of Figure 7).

The case when $|u|/2 < |x| \le |u|$: In this case, x is a border of u and its prefix-suffix occurrences in u share the center position $\lceil |u|/2 \rceil$ of u. ($\Longrightarrow$) Assume the contrary, i.e., x covers uvu and $r < |uv|$. Since x covers uvu, there exists an occurrence of x that covers position $r + 1$. Also, since $r < |uv|$, the occurrence does not end within Lw. Thus, the occurrence must cover the center position $\lceil |u|/2 \rceil$ of the second occurrence of u. Now, there are three distinct occurrences of x that cover the same position $\lceil |u|/2 \rceil$, however, it contradicts that $x = \mathsf{cov}(\mathsf{bord}(uvu))$ is non-periodic (the second statement of Lemma 2). ($\Longleftarrow$) Similar to the previous case, if $r \ge |uvu| - |u| = |uv|$ holds, then x covers uvx and u. Hence x covers uvu.

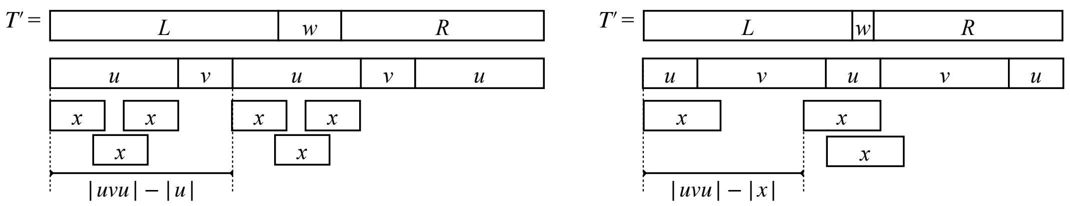


Figure 7 Left: Illustration for the case $|x| \le |u|$. Right: Illustration for the case $|x| > |u|$.

The case when $|x| > |u|$: Let $s = |uvu| - |x|$. In this case, x occurs at positions $s + 1$ and $|uv| + 1$. Thus, the occurrences share position $|uv| + 1$, which is the first position of the second occurrence of u (see the right figure of Figure 7). ($\Longrightarrow$) Assume the contrary, i.e., x covers uvu and $r < s$. Similar to the previous case, there must be an occurrence of x such that the occurrence covers position $r + 1$ and does not end within Lw. Then, there are three distinct occurrences of x that cover the same position $|uv| + 1$, which leads to a contradiction with the fact that x is non-periodic. ($\Longleftarrow$) This statement is trivial by the definitions and the conditions. ◀

Therefore, if $\mathsf{range}(Lw, |x|) \ge |uvu| - \max\{|u|, |x|\}$ then the cover of uvu is x. Otherwise, the cover of uvu is uvu itself. Further, by Lemma 13, the value of $\mathsf{range}(Lw, |x|)$ can be obtained in $O(|w| \log n)$ time since $x = \mathsf{cov}(\mathsf{bord}(uvu))$ is superprimitive and $|x| \le |uv| < |Lw|$.

To summarize, we can compute $\mathsf{cov}(T')$ in $O(|w| \log n)$ for the periodic case.

Finally, we have shown the main theorem of this paper:

▶ **Theorem 17.** *The shortest cover after-edit query can be answered in $O(\ell \log n)$ time after $O(n)$-time preprocessing, where ℓ is the length of the string to be inserted or substituted specified in the query.*

5 Conclusions and Discussions

In this paper, we introduced the problem of computing the longest border and the shortest cover in the after-edit model. For each problem, we proposed a data structure that can be constructed in $O(n)$ time and can answer any query in $O(\ell \log n)$ time where n is the length of the input string, and ℓ is the length of the string to be inserted or replaced.

As a direction for future research, we are interested in improving the running time. For LBAE queries, when the edit operation involves a single character, an $O(\log(\min\{\log n, \sigma\}))$ query time can be achieved by exploiting the periodicity of the border: we pre-compute all *one-mismatch borders* and store the triple of mismatch position, mismatch character, and the mismatch border length for each mismatch border. The number of such triples is in $O(n)$. Furthermore, the number of triples for each position is $O(\min\{\log n, \sigma\})$ due to the periodicity of borders. Thus, by employing a binary search on the triples for the query position, the query time is $O(\log(\min\{\log n, \sigma\}))$. However, this algorithm stores all mismatch borders and cannot be straightforwardly extended to editing strings of length two or more. It is an open question whether the query time of LBAE and SCAE queries can be improved to $O(\ell + \log \log n)$ for an edit operation of length-ℓ string in general. Furthermore, applying the results obtained in this paper to a more general problem setting, particularly the computation of borders/covers in a fully-dynamic string, is a future work that needs further exploration.

References

1 Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem revisited. In *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 20:1–20:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CPM.2018.20.

2 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.5.

3 Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2017. doi:10.1007/978-3-319-67428-5_2.

4 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

5 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Technical Report 90.5, The Leonadro Fibonacci Institute, Trento, Italy*, 1990.

6 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

7 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Inf. Process. Lett.*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

8 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *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.

9 Dany Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

10 Panagiotis Charalampopoulos, Pawel Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 27:1–27:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ICALP.2020.27.

11 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. *CoRR*, abs/2004.08350, 2020. arXiv:2004.08350.

12 Mitsuru Funakoshi and Takuya Mieno. Minimal unique palindromic substrings after single-character substitution. In *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2021. doi:10.1007/978-3-030-86692-1_4.

13 Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing longest palindromic substring after single-character or block-wise edits. *Theor. Comput. Sci.*, 859:116–133, 2021. doi:10.1016/J.TCS.2021.01.014.

14 Pawel Gawrychowski, Jakub Radoszewski, and Tatiana Starikovskaya. Quasi-periodicity in streams. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.22.

15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

16 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

17 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.

18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235, 2023. doi:10.48550/arXiv.1311.6235.

19 Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984. doi:10.1016/0196-6774(84)90021-X.

20 Neerja Mhaskar and W. F. Smyth. String covering: A survey. *Fundam. Informaticae*, 190(1):17–45, 2022. doi:10.3233/FI-222164.

21 Takuya Mieno and Mitsuru Funakoshi. Data structures for computing unique palindromes in static and non-static strings. *Algorithmica*, 2023. doi:10.1007/s00453-023-01170-8.

22 Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Inf. Process. Lett.*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

23 Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Longest Lyndon substring after edit. In *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 19:1–19:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CPM.2018.19.

Walking on Words

Ian Pratt-Hartmann
Department of Computer Science, University of Manchester, United Kingdom
Instytut Informatyki, Uniwersytet Opolski, Opole, Poland

Abstract

Any function f with domain $\{1, \ldots, m\}$ and co-domain $\{1, \ldots, n\}$ induces a natural map from words of length n to those of length m: the ith letter of the output word $(1 \leq i \leq m)$ is given by the $f(i)$th letter of the input word. We study this map in the case where f is a surjection satisfying the condition $|f(i+1) - f(i)| \leq 1$ for $1 \leq i < m$. Intuitively, we think of f as describing a "walk" on a word u, visiting every position, and yielding a word w as the sequence of letters encountered *en route*. If such an f exists, we say that u *generates* w. Call a word *primitive* if it is not generated by any word shorter than itself. We show that every word has, up to reversal, a unique primitive generator. Observing that, if a word contains a non-trivial palindrome, it can generate the same word via essentially different walks, we obtain conditions under which, for a chosen pair of walks f and g, those walks yield the same word when applied to a given primitive word. Although the original impulse for studying primitive generators comes from their application to decision procedures in logic, we end, by way of further motivation, with an analysis of the primitive generators for certain word sequences defined via morphisms.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases word combinatorics, palindrome, Rauzy morphism

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.25

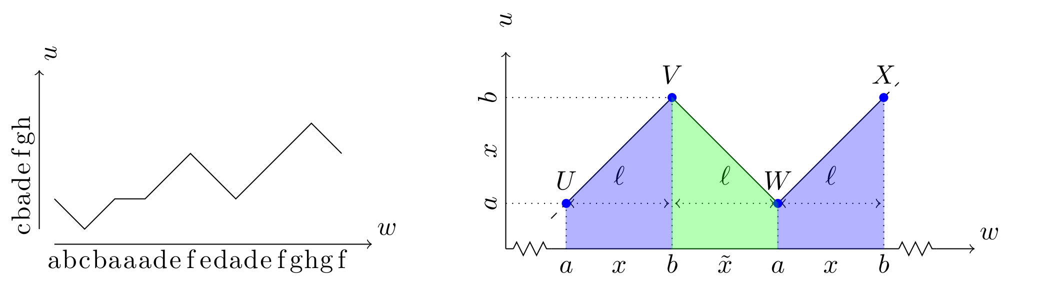
Related Version *Full Version*: `https://arxiv.org/abs/2208.08913`

Funding This work was supported by the Polish NCN, grant number 2018/31/B/ST6/03662.

Acknowledgements The author wishes to thank Prof. V. Berthé and Prof. L. Tendera for their valuable help, and Mr. D. Kojelis for his many suggestions, in particular the much-improved reformulation of Theorem 4.

1 Introduction

Take any word over some alphabet, and, if it is non-empty, go to any letter in that word. Now repeat the following any number of times (possibly zero): scan the current letter, and print it out; then either remain at the current letter, or move one letter to the left (if possible) or move one letter to the right (if possible). In effect, we are *going for a walk on* the input word. Since any unvisited prefix or suffix of the input word cannot influence the word we print out, they may as well be ablated; letting u be the factor of the input word comprising the scanned letters, and w the word printed out, we say that u *generates* w. It is obvious that every word generates itself and its reversal, and that all other words it generates are strictly longer than itself. We ask about the converse of generation. Given a word w, what words u generate it? Call a word *primitive* if it is not generated by any word shorter than itself. It is easy to see that every word must have a generator that is itself primitive. We show that this *primitive generator* is in fact unique up to reversal. On the other hand, while primitive generators are unique, the generating walks need not be, and this leads us to ask, for a given pair of walks, whether we can characterize those primitive words u for which they output the same word w. We answer this question in terms of the palindromes contained in u. Specifically, for a primitive word u, the locations and lengths of the palindromes it



(a) Example of generation. **(b)** The minimal leg $J = [V, W]$ (of length ℓ) of some walk.

Figure 1 Generation and minimal legs.

contains determine which pairs of walks yield identical outputs on u. As an illustration of the naturalness of the notion of primitive generator, we consider word sequences over the alphabet $\{1, \ldots, k\}$ generated by the generalized Rauzy morphism σ, which maps the letter k to the word 1, and any other letter i $(1 \leq i < k)$ to the two-letter word $1 \cdot (i{+}1)$. Setting $\alpha_1^{(k)} = 1$ and $\alpha_{n+1}^{(k)} = \sigma(\alpha_n^{(k)})$ for all $n \geq 1$, we obtain the word sequence $\{\alpha_n^{(k)}\}_{n \geq 1}$. We show that every word in this sequence from the kth onwards has the same primitive generator.

2 Principal results

Fix some alphabet Σ. We use $a, b, c \ldots$ to stand for letters of Σ, and $u, v, w, \ldots$ to stand for words over Σ. The concatenation of two words u and v is denoted uv, or sometimes, for clarity, $u \cdot v$. For integers i, k we write $[i, k]$ to mean the set $\{j \in \mathbb{Z} \mid i \leq j \leq k\}$. If $u = a_1 \cdots a_n$ is a (possibly empty) word over Σ, and $f\colon [1, m] \to [1, n]$ is a function, we write u^f to denote the word $a_{f(1)} \cdots a_{f(m)}$ of length m. We think of f as telling us where in the word u we should be at each time point in the interval $[1, m]$. Define a *walk* to be a surjection $f\colon [1, m] \to [1, n]$ satisfying $|f(i{+}1) - f(i)| \leq 1$ for all i $(1 \leq i < m)$. These conditions ensure that, as we move through the letters $a_{f(1)} \cdots a_{f(m)}$, we never change our position in u by more than one letter at a time, and we visit every position of u at least once. If $w = u^f$ for f a walk, we say that u *generates* w. We may picture a walk as a piecewise linear function, with the generat*ed* word superimposed on the abscissa and the generat*ing* word on the ordinate. Fig. 1a shows how $u =$ cbadefgh generates $w =$ abcbaaadefedadefghgf.

If $u = a_1 \cdots a_n$ is a word, denote the length of u by $|u| = n$, and the reversal of u by $\tilde{u} = a_n \cdots a_1$. Generation is evidently reflexive and reverse-reflexive: every word generates both itself and its reversal. Moreover, if u generates w, then $|u| \leq |w|$; in fact, u and $\tilde{u}$ are the only words of length $|u|$ generated by u. We call u *primitive* if it is not generated by any word shorter than itself – equivalently, if it is generated only by itself and its reversal. For example, *babcd* and *abcbcd* are not primitive, because they are generated by *abcd*; but *abcbda* is primitive. Since the composition of two walks is a walk, generation is transitive: if u generates v and v generates w, then u generates w. Define a *primitive generator* of w to be a primitive word that generates w. It follows easily from the above remarks that every word w has some primitive generator u, and indeed, $\tilde{u}$ as well, since the reversal of a primitive generator of w is obviously also a primitive generator of w. The principal result of this paper is that there are no others:

▶ **Theorem 1.** *The primitive generator of any word is unique up to reversal.*

As an immediate consequence, if u is the primitive generator of w, and v generates w, then u generates v. Theorem 1 is relatively surprising: let u and v be primitive words. Now suppose we go for a walk on u and, independently, go for a walk on v; recalling the stipulation that the two walks visit every position in the words they apply to, the theorem says that, provided only that $u \neq v$ and $u \neq \tilde{v}$, there is no possibility of coordinating these walks so that the sequences of visited letters are the same.

A *palindrome* is a word u such that $u = \tilde{u}$; a *non-trivial* palindrome is one of length at least 2. If u is a non-trivial palindrome, then it is not primitive. Indeed, if $|u|$ is even, then u has a double letter in the middle, and so is certainly not primitive (it is generated by the word in which one of the occurrences of the doubled letter is deleted); if $|u|$ is odd, then it is generated by its prefix of length $(|u|+1)/2 < |u|$ (start at the beginning, go just over half way, then return to the start). Call a word *uniliteral* if it is of the form a^n for some letter a and some $n \geq 0$. Note that the empty word ϵ counts as uniliteral.

▶ **Corollary 2.** *Every uniliteral word has precisely one primitive generator; all others have precisely two.*

Proof. By Theorem 1, if w is any word, its primitive generators are of the form u and $\tilde{u}$ for some word u. The first statement of the corollary is obvious: if $w = \epsilon$ then $u = \tilde{u} = \epsilon$; and if $u = a^n$ for some n ($n \geq 1$), then $u = \tilde{u} = a$. If w is not uniliteral, then $|u| > 1$. But since non-trivial palindromes cannot be primitive, $u \neq \tilde{u}$. ◀

Yet another way of stating Theorem 1 is to say that, for any fixed word w, the equation $u^f = w$ has exactly one primitive solution for u, up to reversal. The same is not true, however, of solutions for f, even if we fix the choice of primitive generator (either u or $\tilde{u}$). Indeed, $u = abcbd$ is one of the two primitive generators of $w = abcbcbd$, but we have $u^f = w$ for $f\colon [1, 7] \to [1, 5]$ given by either of the courses of values $[1, 2, 3, 4, 3, 4, 5]$ or $[1, 2, 3, 2, 3, 4, 5]$. Let u be a primitive word. Say that u is *perfect* if $u^f = u^g$ implies $f = g$ for any walks f and g on u. Thus, $abcbd$ is primitive but not perfect. On the other hand, it is easy to characterize those primitive words that are perfect:

▶ **Theorem 3.** *Let u be a word. Then u is perfect if and only if it contains no non-trivial palindrome as a factor.*

Theorem 3 tells us that generating walks are uniquely determined as long as the primitive generator u does not contain a non-trivial palindrome, but gives us little information if u does contain a non-trivial palindrome. In that case, we would like a characterization of *which* pairs of walks on u yield identical words. We answer this question in terms of the *positions* of the palindromes contained in u. Let $u = a_1 \cdots a_n$ be a word. We denote the ith letter of u by $u[i] = a_i$, and the factor of u from the ith to jth letters by $u[i, j] = a_i \cdots a_j$. If $u[i, j]$ is a non-trivial palindrome, call the ordered pair $\langle i, j\rangle$ a *defect* of u, and denote the set of defects of u by Δ_u. Regarding Δ_u as a binary relation on the set $[1, n]$, we write Δ^*_u for its equivalence closure, the smallest reflexive, symmetric and transitive relation including Δ_u. The interplay between defects and walks is then summed up in the following theorem.

▶ **Theorem 4.** *Let u be a primitive word of length n, and f, g walks with domain $[1, m]$ and co-domain $[1, n]$. Then $u^f = u^g$ if and only if $\langle f(i), g(i)\rangle \in \Delta^*_u$ for all $i \in [1, m]$.*

The motivation for the study of primitive generators comes from the study of the decision problem in (fragments of) first-order logic, in presentations where the logical variables are taken to be $x_1, x_2, \ldots$, and all signatures are assumed to be purely relational. Call a first-order formula φ *index-normal* if, for any quantified sub-formula $Qx_k\psi$ of φ, ψ is a Boolean combination of formulas that are either atomic with free variables among $x_1, \ldots, x_k$, or have as their major connective a quantifier binding x_{k+1}. By re-indexing variables, any first-order formula can easily be written as a logically equivalent index-normal formula. We call an index-normal formula *adjacent* if, in any atomic sub-formula, the indices of neighbouring arguments never differ by more than 1. For example, an atomic sub-formula $p(x_4, x_4, x_5, x_4, x_3)$ is allowed, but an atomic sub-formula $q(x_1, x_3)$ is not. It was shown in [1] that the problem of determining validity for adjacent formulas is decidable. A key notion in analysing this fragment is that of an *adjacent type.* Let $\mathfrak{A}$ be a structure interpreting some relational signature, and $\bar{a}$ a tuple of elements from its domain, A. Define the *adjacent type of* $\bar{a}$ *(in* $\mathfrak{A}$*)* to be the set of all adjacent atomic formulas $q(\bar{x})$ satisfied by $\bar{a}$ in $\mathfrak{A}$. If we think now of $\bar{a}$ as a word over the (possibly infinite) alphabet A, it can easily be shown that the adjacent type of $\bar{a}$ is determined by the adjacent type of its primitive generator. Thus, models of formulas can be unambiguously constructed by specifying only the adjacent types of *primitive* tuples, a crucial technique in establishing decidability of satisfiability.

Notwithstanding its logical genealogy, the concept of primitive generator may be of interest in its own right within the field of string combinatorics. For illustration, consider the sequences of words $\{\alpha_n^{(k)}\}_{n\geq 1}$ over the alphabet $\{1, \ldots, k\}$, defined by setting $\alpha_1^{(k)} = 1$ and $\alpha_{n+1}^{(k)} = \sigma(\alpha_n^{(k)})$, where $\sigma\colon \{1, \ldots, k\}^* \to \{1, \ldots, k\}^*$ is the monoid endomorphism given by

$$\sigma(i) = \begin{cases} 1 \cdot (i+1) & \text{if } i < k \\ 1 & \text{if } i = k. \end{cases}$$

(Here, the operator $\cdot$ represents string concatenation, not integer multiplication!) For $k = 2$, we obtain the so-called *Fibonacci word sequence* 1, 12, 121, 12112, …; for $k = 3$, we obtain the *tribonacci word sequence* 1, 12, 1213, 1213121, …; and so on. A simple induction shows that, for all $k \geq 2$ and all $n > k$, $\alpha_n^{(k)} = \alpha_{n-1}^{(k)}\alpha_{n-2}^{(k)} \cdots \alpha_{n-k}^{(k)}$. In other words, each element of the sequence $\{\alpha_n^{(k)}\}_{n\geq 1}$ after the kth is the concatenation, in reverse order, of the previous k elements; for this reason, the word sequence obtained is referred to as the *k-bonacci word sequence.* A simple proof also shows that $\alpha_n^{(k)}$ is always a left-prefix of $\alpha_{n+1}^{(k)}$, so that we may speak of the infinite word $\omega^{(k)}$ defined by taking the limit $\lim_{n\to\infty} \alpha_n^{(k)}$ in the obvious sense. Thus, the infinite word $\omega^{(2)} = 12112\cdots$ is the (infinite) *Fibonacci word*, and $\omega^{(3)} = 1213121\cdots$ the (infinite) *tribonacci word.* The Fibonacci word is an example of a *Sturmian word* (see, e.g. [3, Ch. 6] for an extensive treatment). The morphism yielding the tribonacci word is sometimes called the *Rauzy morphism* [6, p. 149] (see also [4, Secs. 10.7 and 10.8]). Intriguingly, for a fixed k, all but the first k elements of $\{\alpha_n^{(k)}\}_{n\geq 1}$ share the same primitive generator:

▶ **Theorem 5.** *For all $k \geq 2$, there exists a word γ_k such that, for all $n \geq k$, γ_k is the primitive generator of $\alpha_n^{(k)}$.*

The proof of Theorem 5 exploits a feature of the words $\alpha_n^{(k)}$ that is obvious when one computes a few examples: they are riddled with palindromes. As one might then expect in view of Theorem 4, for all k and all $n \geq k$, the primitive generator γ_k generates $\alpha_n^{(k)}$ via many different walks – in fact via walks beginning at any position of γ_k occupied by the letter 1.

3 Uniqueness of primitive generators

The following terminology will be useful. (Refer to Fig. 1a for motivation.) Let $f\colon [1,m] \to [1,n]$ be a walk, with $m > 1$. By a *leg* of f, we mean a maximal interval $[i,j] \subseteq [1,m]$ such that, for h in the range $i \leq h < j$, the difference $d = f(h{+}1){-}f(h)$ is constant. We speak of a *descending*, *flat* or *ascending* leg, depending on whether d is -1, 0 or 1. The *length* of the leg is $j{-}i$. A leg $[i,j]$ is *initial* if $i = 1$, *final* if $j = m$, *terminal* if it is either initial or final, and *internal* if it is not terminal. A number h which forms the boundary between two consecutive legs will be called a *waypoint*. We count the numbers 1 and m as waypoints by courtesy, and refer to them as *terminal waypoints*; all other waypoints are *internal*. Thus, a walk consists of a sequence of legs from one waypoint to another. If h is an internal waypoint where the change is from an increasing to a decreasing leg, we call h a *peak*; if the change is from a decreasing to an increasing leg, we call it a *trough*. Not all waypoints need be peaks or troughs, because some legs may be flat; however, it is these waypoints that will chiefly concern us in the sequel.

▶ **Lemma 6.** *A word u is not primitive if and only if it is of any of the following forms, where a, b are letters and x, y, z are words:* (i) *$xaay$,* (ii) *$b\tilde{x}axby$,* (iii) *$yb\tilde{x}axb$ or* (iv) *$yaxb\tilde{x}axbz$.*

Proof. Straightforward: see full version [5]. ◀

In the sequel, we shall primarily employ the if-direction of Lemma 6. It easily follows from Cases (i) and (ii) of Lemma 6 that, over the alphabet $\{1,2\}$, there are exactly five primitive words: ϵ, 1, 2, 12, and 21. However, over any larger alphabet, there are infinitely many. For example, over the alphabet $\{1,2,3\}$, the set of primitive words is easily seen to be given by the regular expression $[(\epsilon+3+23)(123)^*(\epsilon+1+12)]+[(\epsilon+2+32)(132)^*(\epsilon+1+13)]$. Over alphabets of any finite size, the set of primitive words is context-sensitive. This follows from the fact that the four patterns of Lemma 6 define context-sensitive languages, together with the standard Boolean closure properties of context-sensitive languages.

We shall occasionally need to consider a broader class of functions than walks. Define a *stroll* to be a function $f\colon [1,m] \to [1,n]$ satisfying $|f(i{+}1){-}f(i)| \leq 1$ for all i $(1 \leq i < m)$. Thus, a walk is a stroll which is surjective. Let $f\colon [1,m] \to [1,n]$ be a stroll. If $f(i) = f(j)$ for some i, j $(1 < i < j < m)$ define the function $f'\colon [1,m{-}j{+}i] \to [1,n]$ by setting $f'(h) = f(h)$ if $1 \leq h \leq i$, and $f'(h) = f(h{+}j{-}i)$ otherwise. Intuitively, f' is just like f, but with the interval $[i,j{-}1]$ – equivalently, the interval $[i{+}1,j]$ – removed. Evidently, f' is a also a stroll, and we denote it by $f/[i,j]$. For the cases $i = 1$ or $j = m$, we change the definition slightly, as no analogue of the condition $f(i) = f(j)$ is required. Specifically if $1 \leq i < j \leq m$, define the functions $f'\colon [1,m{-}j+1] \to [1,n]$ and $f''\colon [1,i] \to [1,n]$ by $f'(h) = f(j{+}h{-}1)$ and $f''(h) = f(h)$. Intuitively, f' is just like f, but with the interval $[1,j{-}1]$ removed, and f'' is just like f, but with the interval $[i{+}1,m]$ removed. Again f' and f'' are also strolls, and we denote them by $f/[1,j]$ and $f/[i,m]$, respectively.

With these preliminaries behind us, we give an outline sketch of the proof Theorem 1. The proof proceeds by contradiction, supposing that u and v are primitive words such that neither $u = v$ nor $u = \tilde{v}$, and w is a word generated from u by some walk f and from v by some walk g. Write $|w| = m$. Crucially, we may assume without loss of generality that w is a *shortest counterexample* – that is, a shortest word for which such u, v, f and g exist. Observe that, since u and v are primitive, they feature no immediately repeated letter. So suppose w does – i.e. is of the form $w = xaay$ for some words x, y and letter a. Letting $i = |x|{+}1$, we must therefore have $f(i) = f(i{+}1)$ and $g(i) = g(i{+}1)$. Now let $f' = f/[i,i{+}1]$, $g' = g/[i,i{+}1]$ and $w' = w[1,i] \cdot w[i{+}2,m]$. We see that f' is surjective if f is, and similarly for g', and moreover that $w' = u^{f'} = v^{g'}$, contrary to the assumption that w is shortest. Hence w contains no immediately repeated letters, whence all legs of f and g are either increasing or decreasing, and all internal waypoints are either peaks or troughs.

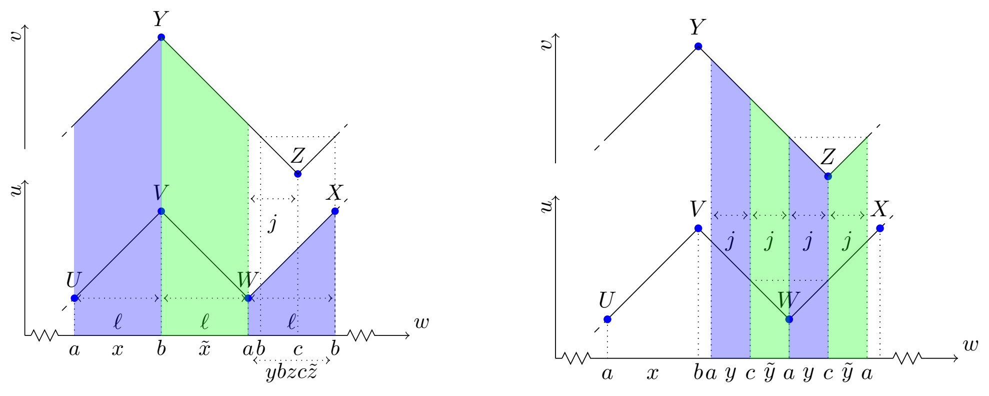
We claim first that at least one of f or g must have an internal waypoint. For if not, we have $w = u$ or $w = \tilde{u}$ and $w = v$ or $w = \tilde{v}$, whence $u = v$ or $u = \tilde{v}$, contrary to assumption. It then follows that *both f and g* have an internal waypoint. For suppose f has an internal waypoint (either a peak or a trough); then w is not primitive. But if g does not have an internal waypoint, $w = v$ or $w = \tilde{v}$, contrary to the assumption that v is primitive.

We use upper case letters in the sequel to denote integers in the range $[1, n]$ which are somehow significant for the walks f or g: note that these need not be waypoints. Let ℓ denote the minimal length of a leg on either of the walks f or g. Without loss of generality, we may take this minimum to be achieved on a leg of f, say $[V, W]$.

We suppose for the present that this leg is *internal*. Fig. 1b illustrates this situation where V is a peak and W a trough; but nothing essential would change if it were the other way around. Write $U = V - \ell$ and $X = W + \ell$. By the minimality of $[V, W]$ (assumed internal), $U \geq 1$ and $X \leq m$; moreover, f is monotone on $[U, V]$, $[V, W]$ and $[W, X]$. Now let $w[U] = a$, $w[V] = b$ and $w[U+1, V-1] = x$. Since V is a waypoint on f, $w[W] = a$ and $w[V+1, W-1] = \tilde{x}$. Similarly, $w[X] = b$ and $w[W+1, X-1] = \tilde{\tilde{x}} = x$. We see immediately that g must have a waypoint in the interval $[U+1, X-1]$, for otherwise, v (or $\tilde{v}$) contains a factor $axb\tilde{x}axb$, contrary to the assumption that v is primitive (Lemma 6, case (iv)). Let Y be the waypoint on g which is closest to either of V or W. Replacing w by its reversal if necessary, assume that $|Y - V| \leq |Y - W|$, and write $k = |Y - V|$. We consider possible values of $k \in [0, \ell-1]$ in turn, deriving a contradiction in each case.

Case (i): $k = 0$ (i.e. $Y = V$). For definiteness, let us suppose that Y is a peak, rather than a trough, but the reasoning is entirely unaffected by this determination. By the minimality of the leg $[V, W]$, g has no other waypoints in the interval $[U+1, W-1]$, and $g(U) = g(W)$. By inspection of Fig. 1b, it is also clear from the minimality of the leg $[V, W]$ that $f' = f/[U, W]$ is surjective (and hence a walk). We see immediately that the stroll $g' = g/[U, W]$ is not surjective. Indeed, if it were, writing $w' = w[1, U] \cdot w[W+1, n]$, we would have $w' = u^{f'} = v^{g'}$, contrary to the assumption that w is a shortest counterexample. In other words, there are positions of v which g reaches over the range $[U+1, W-1]$) that it does not reach outside this range. It follows that the position $g(V) = g(Y)$ in the string v is actually terminal. (Since we are assuming that Y is a peak, $g(Y) = |v|$; but the following reasoning is unaffected if Y is a trough and $g(Y) = 1$.) It also follows that W itself cannot be a waypoint of g. For otherwise, the leg following W, which is of length at least ℓ, covers all values in $g([U, W])$, thus ensuring that g' is surjective, which we have just shown to be false. However, g must have some waypoint in $[V+1, X-1]$. For if not, then g is decreasing between V and X (remember that $g(Y) = g(V) = |v|$), and thus v has a suffix $b\tilde{x}axb$, contrary to the assumption that v is primitive (Lemma 6 case (iii)). By the minimality of the leg $[V, W]$, we see that there is exactly one such waypoint, say Z. Since we have already shown that Y is the only waypoint on g in $[U+1, W-1]$, and that W is not a waypoint on g, it follows that $Z \in [W+1, X-1]$.

Now let $j = Z - W$. (Thus, $1 \leq j < \ell$.) If $j > \frac{1}{2}\ell$, we obtain the situation depicted in Fig. 2a. Since g has a waypoint at Z and remembering that $w[W+1, X-1] = x$ and $w[X] = b$, we see that x has the form $ybzc\tilde{z}$ for some strings y and z and some letter $c = w[Z]$. But we also know that $g(V) = g(Y) = |v|$, the final position of v, so that v has a suffix $xb = (ybzc\tilde{z})b$, and hence the suffix $bzc\tilde{z}b$, contrary to the assumption that v is primitive (Lemma 6 case (iii)). Furthermore, if $j = \frac{1}{2}\ell$, then, by the same reasoning, x has



(a) The condition $j = Z - W > \frac{1}{2}\ell$. **(b)** The condition $j = Z - W < \frac{1}{2}\ell$.

Figure 2 The walk g has waypoints at $Y = V$ and at Z.

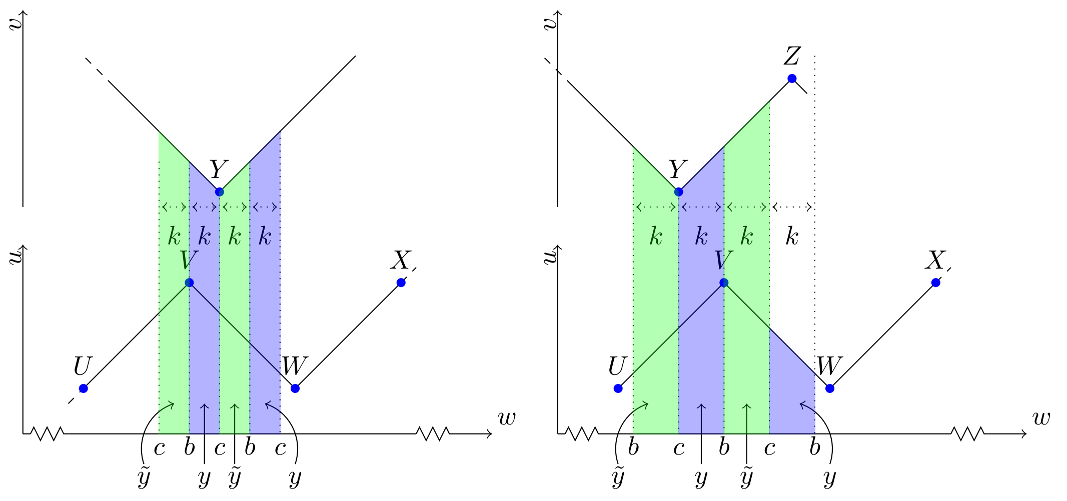
the form $zc\tilde{z}$ and $a = b$. Again then, v has a suffix $bzc\tilde{z}b$, contrary to the assumption that v is primitive. We conclude that $j < \frac{1}{2}\ell$, and we obtain the situation depicted in Fig. 2b. Now let $c = w[Z]$ and $y = w[W{+}1, Z{-}1]$. By considering the waypoint Z on g, we see that $w[Z, Z{+}j] = c\tilde{y}a$, whence $w[W, W{+}2j] = ayc\tilde{y}a$. By considering the waypoint W on f, we see that also $w[W{-}2j, W] = ayc\tilde{y}a$, whence $w[W{-}2j, W{+}j] = ayc\tilde{y}ayc$. But there are no waypoints of g strictly between $V = Y < W{-}2j$ and Z, whence $\tilde{v}$ contains the factor $ayc\tilde{y}ayc$, contrary to the supposition that v is primitive (Lemma 6 case (iv)).

Case (ii): $1 \leq k \leq \frac{1}{3}\ell$. We may have either $Y > V$ or $Y < V$: Fig. 3a shows the former case; however, the reasoning in the latter is almost identical. Let $w[V] = b$ and $w[Y] = c$. Furthermore, let $w[V{+}1, Y - 1] = y$. Since V is a waypoint of f, we have $w[V{-}k] = c$ and $w[V{-}k{+}1, V{-}1] = \tilde{y}$, whence $w[Y{-}2k, Y] = w[V{-}k, Y] = c\tilde{y}byc$. Since Y is a waypoint of g, we have $w[Y, Y{+}2k] = c\tilde{y}byc$, whence $w[V, V{+}3k] = byc\tilde{y}byc$. And since $\ell \geq 3k$, there is no waypoint on f in the interval $w[V{+}1, V{+}3k - 1]$, whence $\tilde{u}$ contains the factor $byc\tilde{y}byc$, contrary to the assumption that u is primitive (Lemma 6 case (iv)).

Case (iii): $\frac{1}{3}\ell < k < \frac{1}{2}\ell$. Again, in this case, we may have either $Y > V$ or $Y < V$. This time (for variety) assume the latter; however, the reasoning in the former case is almost identical. Thus, we have the situation depicted in Fig. 3b. Let $w[V] = b$, $w[Y] = c$ and $w[Y{+}1, V{-}1] = y$. Since Y is a waypoint on g, we see that $w[Y - k] = b$ and $w[Y{-}k{+}1, Y{-}1] = \tilde{y}$, whence $w[V{-}2k, V] = w[Y{-}k, V] = b\tilde{y}cyb$. Since V is a waypoint on f, we see that also $w[V, V{+}2k] = b\tilde{y}cyb$. Thus, u contains the factor $b\tilde{y}cyb$ and v contains the factor $cyb\tilde{y}c$; moreover $w[Y, Y{+}3k] = cyb\tilde{y}cyb$.

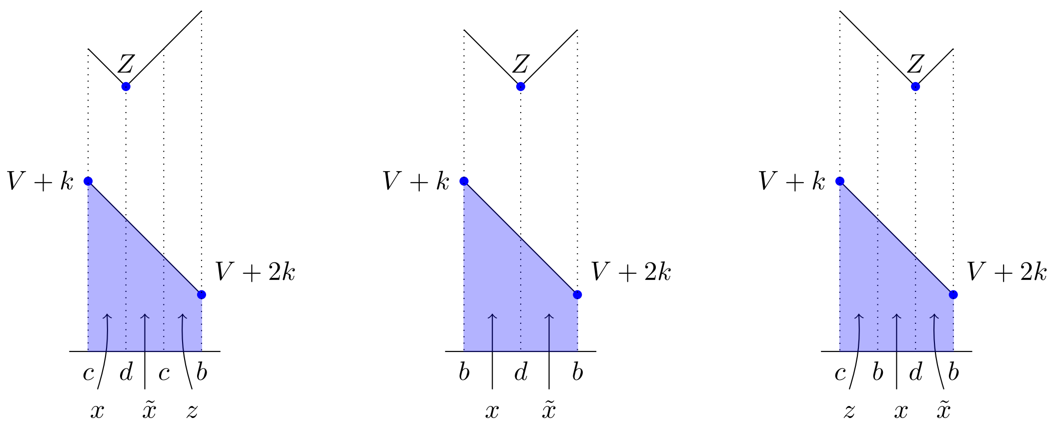
Now let Z be the next waypoint on g after Y. It is immediate that $Z{-}Y < 3k$, since otherwise, v contains the factor $cyb\tilde{y}cyc$, contrary to the assumption that v is primitive (Lemma 6 case (iv)). We consider three possibilities for the point Z, depending on where, exactly, Z is positioned in $[V{+}k, V{+}2k] = [Y{+}2k, Y{+}3k]$. The three possibilities are indicated in Fig. 4, which shows the detail of Fig. 3b in that interval. Suppose (a) that $V{+}k < Z < V{+}\frac{3}{2}k$. Then, by inspection of Fig. 4a, y must be of the form $xd\tilde{x}cz$ for some letter d and strings x and z. But we have already argued that u contains the factor

$$b\tilde{y}cyb = b(xd\tilde{x}cz)^{-1}c(xd\tilde{x}cz)b = b(\tilde{z}cxd\tilde{x})c(xd\tilde{x}cz)b$$



(a) Condition $k \le \frac{1}{3}\ell$; for illustration, $Y > V$. **(b)** Condition $\frac{1}{3}\ell < k < \frac{1}{2}\ell$; for illustration, $Y < V$.

Figure 3 The walk g has a waypoint at Y with $k = |V - Y| \ge 1$.



(a) $Z < V + \frac{3}{2}k$. **(b)** $Z = V + \frac{3}{2}k$. **(c)** $Z > V + \frac{3}{2}k$.

Figure 4 The location of Z with respect to $V + \frac{3}{2}k$ in Case (iii).

and hence the factor $cxd\tilde{x}cxd$ contrary to the assumption that u is primitive (Lemma 6 case (iv)). Suppose (b) that $Z = V + \frac{3}{2}k$. Then, by inspection of Fig. 4b, y must be of the form $xd\tilde{x}$ for some letter d and string x, and furthermore, $b = c$. But in that case u contains the factor

$$b\tilde{y}cyb = c(xd\tilde{x})^{-1}c(xd\tilde{x})c = c(xd\tilde{x})c(xd\tilde{x})c$$

and hence the factor $cxd\tilde{x}cxd$ again. Suppose (c) that $V + \frac{3}{2}k < Z < V + 2k$. Then by inspection of Fig. 4c, y must be of the form $zbxd\tilde{x}$ for some letter d and strings x and z. But we have already argued that v contains the factor

$$cyb\tilde{y}c = c(zbxd\tilde{x})b(zbxd\tilde{x})^{-1}c = c(zbxd\tilde{x})b(xd\tilde{x}b\tilde{z})c$$

and hence the factor $bxd\tilde{x}bxd$, again contrary to the assumption that u is primitive. This eliminates all possibilities for the position of Z, and thus yields the desired contradiction.

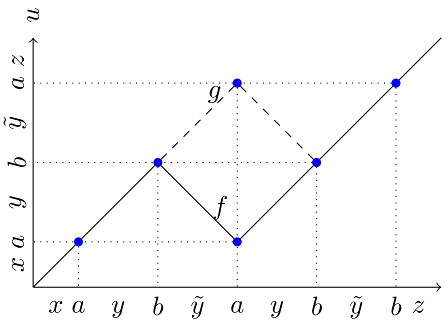


Figure 5 Distinct walks f (solid) and g (dashed and solid) on $u = xayb\tilde{y}az$ such that $u^f = u^g$.

The remaining cases, where $k > \ell/2$, or where the shortest leg is initial or final, are omitted because of space restrictions. See full version [5] for a complete proof.

4 Uniqueness of walks

In this short section, we prove Theorem 3, which states that a word is perfect if and only if it contains no non-trivial palindrome as a factor.

For the only-if direction, suppose that u contains a non-trivial palindrome. If that palindrome is odd, so that u has the form $xayb\tilde{y}az$, then the word $xayb\tilde{y}ayb\tilde{y}az$ is generated via the distinct walks f and g illustrated in Fig. 5. If the contained palindrome is even, so that u has the form $xaaz$, then the word $xaaaz$ is generated via distinct walks, one of which pauses for one step on the first a, and the other on the second.

For the converse, suppose for contradiction that u is a word of length n containing no non-trivial palindromes, for which there exist walks f and g such that $u^f = u^g$ but $f \neq g$. Let u, f and g be chosen so that $m = |u^f| = |u^g|$ is minimal. If $f(i) = f(i+1)$ for some i, we have $g(i) = g(i+1)$, since otherwise, u contains a repeated letter, and therefore a palindrome of length 2, contrary to assumption. But if both $f(i) = f(i+1)$ and $g(i) = g(i+1)$, then the functions $f' = f/[i, i+1]$ and $g' = g/[i, i+1]$ are defined, and are obviously walks, and moreover we have $u^{f'} = u^{g'}$ and $f' \neq g'$, contradicting the minimality of m. Hence, we may assume that neither f nor g is ever stationary. We claim that f and g have the same waypoints. For if i is an internal waypoint for f but not for g, we have $f(i-1) = f(i+1)$, $u[g(i-1)] = u[f(i-1)]$ and $u[g(i+1)] = u[f(i+1)]$, whence $u[g(i-1)] = u[g(i+1)]$, so that u contains an odd, non-trivial palindrome centred at $g(i)$, contrary to assumption. This establishes the claim that f and g have the same waypoints. Since u is certainly not itself a non-trivial palindrome and $f \neq g$, the walks f and g must have at least one internal waypoint between them. Now take a shortest leg of f (which must also be a shortest leg of g), say $[j, j+\ell]$. Suppose first that $[j, j+\ell]$ is an internal leg (i.e. $j < 1$ and $j + \ell < m$). To visualize the situation suppose $V = j$ and $W = j + \ell$ in Fig. 1b. Taking into account the legs either side, we see that $f(j) = f(j + 2\ell)$ and $g(j) = g(j + 2\ell)$, and moreover that $f' = f/[j, j + 2\ell]$ and $g' = g/[j, j + 2\ell]$ map $[1, m - 2\ell]$ surjectively onto $[1, n]$. Clearly, $u^{f'} = u^{g'}$. But f and g have the same waypoints over the interval $[j, j + 2\ell]$, whence $f \neq g$ implies $f' \neq g'$, contradicting the minimality of m. The cases where the shortest leg is terminal are handled similarly.

5 Words yielding the same results on distinct walks

In this section, we sketch the ideas behind the proof of Theorem 4, allowing us to characterize those primitive words which are solutions of a given equation $u^f = u^g$, for walks f and g.

Let $f' \colon [1, m] \to [1, n]$ be a walk. If $1 \leq j \leq m$, then the function $f \colon [1, m+1] \to [1, n]$ given by

$$f(i) = \begin{cases} f'(i) & \text{if } i \leq j \\ f'(i{-}1) & \text{otherwise} \end{cases}$$

is also a walk, longer by one step. We call f the *hesitation on f' at j*, as it arises by executing f' up to and including the jth step, then pausing for one step, before continuing as normal. We next proceed to define an operation of *vacillation on f'*, also producing a strictly longer walk. This operation has three forms, depending on whether it occurs at the start, in the middle, or at the end of the walk. For any k $(1 \leq k < m)$, we define the *initial vacillation on f' over* $[1, k{+}1]$ to be the walk $f \colon [1, m{+}k] \to [1, n]$ given by

$$f(i) = \begin{cases} f'(k{+}1{-}(i{-}1)) & \text{if } i \leq k+1 \\ f'(i{-}k) & \text{otherwise.} \end{cases}$$

Thus f arises by executing the first $k+1$ steps of f' in reverse order and then continuing to execute f' from the second step as normal. Likewise, we define the *final vacillation on f' over* $[m{-}k, m]$ to be the walk $f \colon [1, m{+}k] \to [1, n]$ given by

$$f(i) = \begin{cases} f'(i) & \text{if } i \leq m \\ f'(m{-}(i{-}m)) & \text{otherwise.} \end{cases}$$

Thus f arises by executing f' as normal and then repeating the k steps preceding the last in reverse order. Finally, for any j $(1 < j < m)$, and any k $(1 \leq k < j)$, we define the *internal vacillation on f' over* $[j{-}k, j]$ to be the walk $f \colon [1, m{+}2k] \to [1, n]$ given by

$$f(i) = \begin{cases} f'(i) & \text{if } i \leq j \\ f'(j{-}(i{-}j)) & \text{if } j < i \leq j+k \\ f'(i-2k) & \text{otherwise.} \end{cases}$$

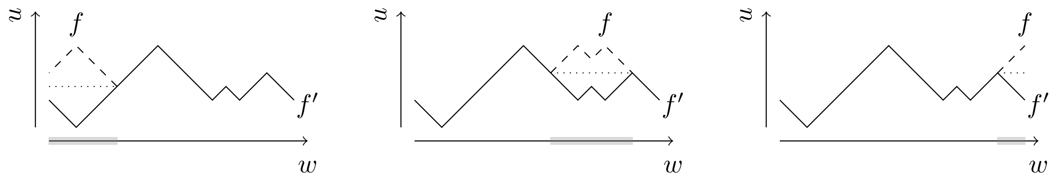
Thus f arises by executing f' up to the jth step, reversing the previous k steps back to the $(j{-}k)$th step and then continuing from the $(j-k+1)$th step as normal. A *vacillation* on f' is an initial, internal or final vacillation on f'.

Let $f' \colon [1, m] \to [1, n]$ again be a walk. We proceed to define an operation of *reflection* on f', producing a stroll (not necessarily surjective) of the same length. For any k $(1 \leq k < m)$, we take the *initial reflection on f' over* $[1, k{+}1]$ to be the function f defined on the domain $[1, m]$ by setting

$$f(i) = \begin{cases} f'(k{+}1){-}(f'(i){-}f'(k{+}1)) & \text{if } i \leq k+1 \\ f'(i) & \text{otherwise.} \end{cases}$$

Thus f arises by reflecting the segment of f' over the interval $[1, k{+}1]$ in the horizontal axis positioned at height $f'(k+1)$, and then continuing as normal (Fig. 6a). Likewise, we take the *final reflection on f' over* $[m{-}k, m]$ to be the function f defined on $[1, m]$ by setting

$$f(i) = \begin{cases} f'(m{-}k){-}(f'(i){-}f'(m{-}k)) & \text{if } i \geq m{-}k \\ f'(i) & \text{otherwise.} \end{cases}$$



(a) Initial. **(b)** Internal. **(c)** Final.

Figure 6 The stroll f (dashed and solid) is a reflection on the walk f' (solid) over I (shaded).

Thus f arises by executing f' as normal up to the $(m{-}k)$th step, and then thereafter reflecting the remaining segment of f' in the horizontal axis positioned at height $f'(m{-}k)$ (Fig. 6c). Finally, for integers j, k ($1 < j < m$, $1 \leq k \leq \min(j{-}1, m{-}j)$) such that $f'(j{-}k) = f'(j{+}k)$, the *internal reflection on* f' over $[j{-}k, j{+}k]$ is the function f defined on $[1, m]$ by setting

$$f(i) = \begin{cases} f'(j{-}k){-}(f'(i){-}f'(j{-}k)) & \text{if } j{-}k \leq i \leq j{+}k \\ f'(i) & \text{otherwise.} \end{cases}$$

Thus f arises by executing f' up to the point $j{-}k$, then reflecting the segment of f' over the interval $[j{-}k, j+k]$ in the horizontal axis positioned at height $f'(j{-}k) = f'(j{+}k)$, thereafter executing f' as normal (Fig. 6b). A *reflection* on f' is an initial, internal or final reflection on f'. As defined above, reflections can take values in the range $[-n{+}1, 2n{-}1]$; accordingly, we call a reflection *proper* if all its values are within the interval $[1, n]$, and in that case we take the resulting function to have co-domain $[1, n]$. We shall only ever be concerned with proper reflections in the sequel; and a proper reflection on a walk (more generally, on a stroll) is evidently a stroll; there is no *a priori* requirement for it to be surjective.

Reflections are of most interest in connection with walks on words containing odd palindromes. Let $f' \colon [1, m] \to [1, n]$ be a stroll, and u be a word of length n. We say that a reflection f on f' is *admissible for* u if it is either: (i) an initial reflection over $[1, k{+}1]$, and u has a palindrome of length $2k{+}1$ centred at $f'(k{+}1)$; (ii) a final reflection over $[m{-}k, m]$, and u has a palindrome of length $2k{+}1$ centred at $f'(m{-}k)$; or (iii) an internal reflection over $[j{-}k, j{+}k]$, and u has a palindrome of length $2k{+}1$ centred at $f'(j{-}k) = f'(j{+}k)$. We see by inspection of Fig. 6 that, if f is a reflection on f' admissible for u, then $u^f = u^{f'}$.

Suppose now f' and g' are walks with domain $[1, m]$ and co-domain $[1, n]$. If f and g are hesitations on f' and g', respectively, at some common point, we say that the pair of walks $\langle f, g \rangle$ is a *hesitation* on the pair $\langle f', g' \rangle$; similarly, if f and g are vacillations on f' and g' over some common interval, we say that the pair of walks $\langle f, g \rangle$ is a *vacillation* on the pair $\langle f', g' \rangle$. Evidently, if u is a word such that $u^{f'} = u^{g'}$ and $\langle f, g \rangle$ is a hesitation or vacillation on $\langle f', g' \rangle$, then $u^f = u^g$. If now f is a reflection on f' over some interval, we say that $\langle f, g' \rangle$ is a *reflection on* $\langle f', g' \rangle$, and also that $\langle g', f \rangle$ is a *reflection on* $\langle g', f' \rangle$. Evidently, if the reflection in question is (proper and) admissible for some word u such that $u^{f'} = u^{g'}$, then $u^f = u^{g'}$. Note that hesitations/vacillations on pairs of strolls are hesitations/vacillations on *both* of the strolls in question, while reflections on pairs of strolls are reflections on *either* of the strolls in question.

Now let f' and g' be walks, and suppose u is a word of length m such that $u^{f'} = u^{g'}$. We have seen that, if $\langle f, g \rangle$ is a hesitation or vacillation on $\langle f', g' \rangle$, or is a reflection on $\langle f', g' \rangle$ admissible for u, then $u^f = u^g$. The principal result of this section states that, for primitive words, this is essentially the only way in which we can arrive at distinct walks f and g such that $u^f = u^g$.

▶ **Lemma 7.** *Let u be a primitive word of length n, and let f and g be walks with domain $[1,m]$ and co-domain $[1,n]$ such that $u^f = u^g$. Then there exist sequences of walks $\{f_s\}_{s=0}^t$ and $\{g_s\}_{s=0}^t$, all having co-domain $[1,n]$, satisfying:* (i) *$f_0 = g_0$ is monotone;* (ii) *for all s $(0 \leq s < t)$, $\langle f_{s+1}, g_{s+1}\rangle$ is a hesitation on $\langle f_s, g_s\rangle$, a vacillation on $\langle f_s, g_s\rangle$, or a reflection on $\langle f_s, g_s\rangle$ admissible for u; and* (iii) *$f_t = f$ and $g_t = g$.*

Proof. Similar in character to the proof of Theorem 1. See full version [5] for details. ◀

Lemma 7 gives us everything we need for the proof of Theorem 4, which states that, for a primitive word u of length n, and walks $f, g\colon [1,m] \to [1,n]$, we have $u^f = u^g$ if and only if $\langle f(i), g(i)\rangle \in \Delta_u^*$ for all $i \in [1,m]$. Recall that Δ_u^* is the equivalence closure of Δ_u, the defect set of u.

The if-direction is almost trivial. Indeed, $\langle j,k\rangle \in \Delta_u$ certainly implies $u[j] = u[k]$, whence $\langle j,k\rangle \in \Delta_u^*$ also implies $u[j] = u[k]$. Thus, if $\langle f(i), g(i)\rangle \in \Delta_u^*$ for all $i \in [1,m]$, then $u[f(i)] = u[g(i)]$ for all $i \in [1,m]$, which is to say $u^f = u^g$.

For the only-if direction, we suppose that $u^f = u^g$. By Lemma 7, we may decompose the pair of walks $\langle f, g\rangle$ into a series $\{\langle f_s, g_s\rangle\}_{s=0}^t$ such that: (i) $f_0 = g_0$; (ii) for all s $(0 \leq s < t)$, the pair $\langle f_{s+1}, g_{s+1}\rangle$ is obtained by performing a hesitation, vacillation, or an admissible (for u) reflection on $\langle f_s, g_s\rangle$; and (iii) $\langle f_t, g_t\rangle = \langle f, g\rangle$. We establish that the following holds for all s $(0 \leq s \leq t)$:

$$\langle f_s(i), g_s(i)\rangle \in \Delta_u^* \text{ for all } i \text{ in the domain of } f_s \text{ (= the domain of } g_s). \tag{1}$$

Putting $s = t$ then secures the required condition.

We proceed by induction on s. For the base case, where $s = 0$, we have $f_0 = g_0$, and there is nothing to do. For the inductive step, we suppose (1), and show that the same holds with s replaced by $s+1$. We have three cases.

Case 1. $\langle f_{s+1}, g_{s+1}\rangle$ is obtained by a hesitation on $\langle f_s, g_s\rangle$ at j. If $i \leq j$ then $f_{s+1}(i) = f_s(i)$ and $g_{s+1}(i) = g_s(i)$; and by (1), $\langle f_s(i), g_s(i)\rangle \in \Delta_u^*$. If $i > j$ then $f_{s+1}(i) = f_s(i-1)$ and $g_{s+1}(i) = g_s(i-1)$; and by (1), $\langle f_s(i-1), g_s(i-1)\rangle \in \Delta_u^*$. Either way, $\langle f_{s+1}(i), g_{s+1}(i)\rangle \in \Delta_u^*$.

Case 2. $\langle f_{s+1}, g_{s+1}\rangle$ is a vacillation on $\langle f_s, g_s\rangle$. We consider the case of an internal vacillation over some interval over $[j-k, j]$; initial and final vacillations are handled similarly. Again, if $i \leq j$ then $f_{s+1}(i) = f_s(i)$ and $g_{s+1}(i) = g_s(i)$; and by (1), $\langle f_s(i), g_s(i)\rangle \in \Delta_u^*$. If $j < i \leq j+k$, then $f_{s+1}(i) = f_s(j-(i-j))$ and $g_{s+1}(i) = g_s(j-(i-j))$; and by (1), $\langle f_s(j-(i-j)), g_s(j-(i-j))\rangle \in \Delta_u^*$. Finally, if $i > j+k$, then $f_{s+1}(i) = f_s(i-2k)$ and $g_{s+1}(i) = g_s(i-2k)$; and by (1), $\langle f_s(i-2k), g_s(i-2k)\rangle \in \Delta_u^*$.

Case 3. $\langle f_{s+1}, g_{s+1}\rangle$ is the result of a reflection on $\langle f_s, g_s\rangle$ over some interval $[j-k, j+k]$, with the reflection in question admissible for u. By exchanging f and g if necessary, we may assume that f_{s+1} is a reflection on f_s over $[j-k, j+k]$, and $g_{s+1} = g_s$; it does not matter for the ensuing argument whether the reflection in question is internal, initial or final. If $i \notin [j-k, j+k]$, then $f_{s+1}(i) = f_s(i)$ and $g_{s+1}(i) = g_s(i)$; and by (1), $\langle f_s(i), g_s(i)\rangle \in \Delta_u^*$. So suppose $i \in [j-k, k+j]$. Since the reflection over $[j-k, j+k]$ is admissible, the factor $u[f_s(j-k), f_s(j+k)]$ is a palindrome. Moreover, from the definition of reflection, either $u[f_{s+1}(i), f_s(i)]$ or $u[f_s(i), f_{s+1}(i)]$ is a palindromic factor of u (depending on whether $f_{s+1}(i) \leq f_s(i)$ or $f_{s+1}(i) \geq f_s(i)$). That is, either $\langle f_{s+1}(i), f_s(i)\rangle \in \Delta_u$ or $\langle f_s(i), f_{s+1}(i)\rangle \in \Delta_u$. But by (1), $\langle f_s(i), g_s(i)\rangle = \langle f_s(i), g_{s+1}(i)\rangle \in \Delta_u^*$. Hence $\langle f_{s+1}(i), g_{s+1}(i)\rangle \in \Delta_u^*$, again as required. This concludes the induction, and hence the proof of the only-if direction.

▶ **Corollary 8.** *Let v_1 and v_2 be primitive words of length n. Then v_1 and v_2 satisfy the same equations $u^f = u^g$, where f and g are walks with co-domain $[1, n]$, if and only if $\Delta_{v_1} = \Delta_{v_2}$.*

Proof. The if-direction is immediate from Theorem 4. For the only-if direction, suppose v_1 and v_2 satisfy the same equations $u^f = u^g$. If v_1 contains a non-trivial palindrome of (necessarily odd) length, say, $2k + 1$ centred at i, let f and g be walks as depicted in Fig. 5, diverging at i and re-converging at $i + 2k$. Thus $v_1^f = v_1^g$ and hence $v_2^f = v_2^g$. But considering f and g over the interval $[i, i + k]$, the equation $v_2^f = v_2^g$ clearly implies that v_2 has a a palindrome of length $2k + 1$ centred at $f(i) = g(i) = i$, whence $\Delta_{v_2} \supseteq \Delta_{v_1}$. By symmetry, $\Delta_{v_1} \supseteq \Delta_{v_2}$ ◀

For a treatment of the problem of finding palindromes in words, see [2, Ch. 8].

6 Primitive generators of some morphic words

In this section, we prove Theorem 5, which states that, for each $k \geq 2$, all elements of the k-bonacci sequence $\{\alpha_n^{(k)}\}_{n\geq 1}$ from the kth onwards have the same primitive generator. In the sequel, we employ decorated versions of α, β, γ as constants denoting words.

We work with an alternative, recursive definition of the words $\alpha_n^{(k)}$. For all $k \geq 1$, let $\beta_k = \beta'_k \cdot k$, where β'_k is recursively defined by setting $\beta'_1 = \varepsilon$ and $\beta'_{k+1} = \beta'_k \cdot k \cdot \beta'_k$ for all $k \geq 1$. Now define, for any $k \geq 2$ the sequence $\{\alpha_n^{(k)}\}_{n\geq 1}$ by declaring, for all $n \geq 1$: $\alpha_n^{(k)} = \beta_n$ if $n \leq k$, and $\alpha_n^{(k)} = \alpha_{n-1}^{(k)}\alpha_{n-2}^{(k)} \cdots \alpha_{n-k}^{(k)}$ otherwise. A simple induction shows that this definition of the $\alpha_n^{(k)}$ coincides with that given in the introduction via morphisms. We remark that $|\beta_k| = 2^{k-1}$, for all $k \geq 1$.

We now define the primitive generators promised by Theorem 5. For all $k \geq 2$, let $\gamma'_k = (k-1) \cdot \beta'_{k-1}$ and $\gamma_k = \gamma'_k \cdot k$. We remark that $|\gamma_k| = 2^{k-2}+1$, for all $k \geq 2$. The following two claims are easily proved by induction.

▷ Claim 9. For all $k \geq 2$, β'_k is a palindrome over $\{1, \ldots, k-1\}$ containing exactly one occurrence of $k-1$ (in the middle); thus β_k contains exactly one occurrence of $k-1$ (at position $|\beta_k|/2$) and exactly one occurrence of k (at the end). For all $k \geq 3$, γ_k contains exactly one occurrence of each of k (at the end), $k-1$ (at the beginning) and $k-2$ (in the middle).

▷ Claim 10. For all $k \geq 2$, any position in the word γ_k is either occupied by the letter 1 or is next to a position occupied by the letter 1.

▷ Claim 11. For all $k \geq 2$, γ_k is primitive.

Proof. By induction on k. Certainly, $\gamma_2 = 12$ is primitive. For $k \geq 2$, by Claim 9, $\gamma_{k+1} = k \cdot \beta'_{k-1} \cdot (k-1) \cdot \beta'_{k-1} \cdot (k + 1)$ contains exactly one occurrence of each of $k+1$, k and $k-1$. Considering the forms given by the four cases of Lemma 6, we see that γ_{k+1} does not have a prefix or suffix which is a non-trivial palindrome, and that any occurrence of either of the patterns aa or $axb\tilde{x}axb$ must be contained in one of the embedded occurrences of β'_{k-1} and hence in γ_k. By inductive hypothesis, γ_k is primitive, and therefore does not contain either of these patterns. But then γ_{k+1} is primitive by Lemma 6. ◁

▷ Claim 12. Let $k \geq 2$. For all h $(1 \leq h \leq |\gamma'_k|)$ such that $\gamma'_k[h] = 1$, there exists a walk f such that: (i) $\beta'_k = (\gamma'_k)^f$; (ii) $f(1) = h$; and (iii) $f(|\beta'_k|) = |\gamma'_k|$.

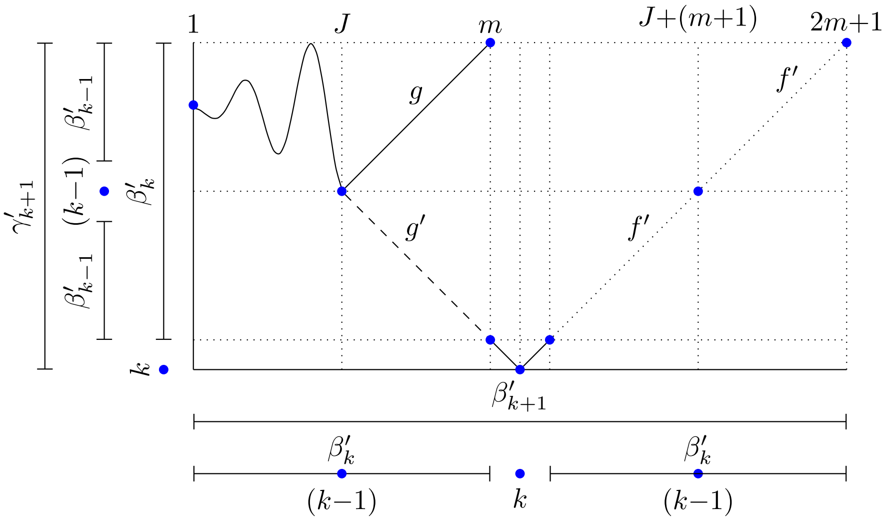


Figure 7 Proof of Claim 12: g (solid lines) is a shifted copy of a walk f on γ'_k yielding β'_k; g' (solid and dashed lines) is a final reflection on g over $[J, m]$; f' (solid, dashed and dotted lines) is a walk on $\gamma'_{k+1} = k \cdot \beta'_k \cdot (k+1)$ yielding $\beta'_{k+1} = \beta'_k \cdot k \cdot \beta'_k$.

Proof. We proceed by induction on k. For $k = 2$ and $k = 3$, the result is trivial, since $\beta'_2 = \gamma'_2 = 1$, $\beta'_3 = 121$ and $\gamma'_3 = 21$. Now suppose the claim holds for the value $k \geq 3$. For convenience, we write $m = |\beta'_k|$ and $n = |\gamma'_k| = |\beta'_{k-1}|+1$ (so $m = 2n-1$.). Remembering that $\beta'_{k+1} = \beta_k \cdot (k+1) \cdot \beta_k$, and $\gamma'_{k+1} = k \cdot \beta'_{k-1} \cdot (k-1) \cdot \beta'_{k-1} = k \cdot \beta'_{k-1} \cdot \gamma'_k$, we have $|\beta'_{k+1}| = 2m+1$ and $|\gamma'_{k+1}| = 2n$. To show that the claim also holds for the value $k+1$, pick any h' satisfying $(1 \leq h' \leq 2n)$ such that $\gamma'_{k+1}[h'] = 1$. We show that there exists a walk $f' \colon [1, 2m+1] \to [1, 2n]$ such that $\beta'_{k+1} = (\gamma'_{k+1})^{f'}$, $f'(1) = h'$, and $f'(2m+1) = 2n$.

Assume for the time being that $h' > n+1$, that is to say, h' is a position in $\gamma'_{k+1} = k \cdot \beta'_{k-1} \cdot (k-1) \cdot \beta'_{k-1}$ occupied by a 1 and lying in the *second* copy of β'_{k-1}. Then $h = h' - n$ is a position in $\gamma'_k = (k-1) \cdot \beta'_{k-1}$ occupied by a 1, so by inductive hypothesis, let $f \colon [1, m] \to [1, n]$ be a walk such that $\beta'_k = (\gamma'_k)^f$, $f(1) = h$, and $f(m) = n$. By Claim 9, β'_k contains exactly one occurrence of $k-1$ (this will be exactly in the middle), and γ'_k likewise contains exactly one occurrence of $k-1$ (this will be at the very beginning). Thus, f reaches the value 1 at just one point, namely $J = (m+1)/2$, and is otherwise strictly greater. (In fact, it is obvious that f must be a straight line from J onwards.) We first define a stroll $g \colon [1, m] \to [1, 2n]$ given by $g(i) = f(i)+n$ (Fig. 7, solid lines). Thus, $g(1) = h'$, and $g(m) = 2n$. Moreover, g reaches the value $n+1$ at just one point, namely $J = (m+1)/2$, and is otherwise strictly greater, as illustrated. Now let g' be the (final) reflection on g over the interval $[J, m]$ (Fig. 7, first solid, then dashed lines). Thus, g' is a stroll on γ'_{k+1} satisfying $g'(1) = h'$ and $g'(m) = 2$. Moreover, since $\beta'_k = \beta'_{k-1} \cdot (k-1) \cdot \beta'_{k-1}$ is a palindrome, we see by inspection that $(\gamma'_{k+1})^{g'} = (\gamma'_{k+1})^g = (\gamma'_k)^f = \beta'_k$. We now construct the desired walk $f' \colon [1, 2m+1] \to [1, 2n]$. For $i \in [1, m]$, we set $f'(i) = g'(i)$. Since $f'(m) = g'(m) = 2$, we set $f'(m+1) = 1$, and then proceed to define f' over the positions to the right, corresponding to the second copy of β'_k in the word $\beta'_{k+1} = \beta'_k \cdot k \cdot \beta'_k$. But this we can do by drawing a straight line, as shown in (Fig. 7). By inspection, f' has the required properties.

Finally, we consider the case where $h' \leq n+1$. Since $\gamma'_{k+1}[h'] = 1$ we in fact have $2 \leq h' \leq n$. And since β'_k is a palindrome, we may replace h' with the value $(n+1)+((n+1)-h)$ (i.e. reflect in the horizontal axis at height $n+1$) and construct f' as before. To re-adjust so that $f'(1)$ has the correct value, perform an initial reflection on f' over the interval $[1, J]$. ◁

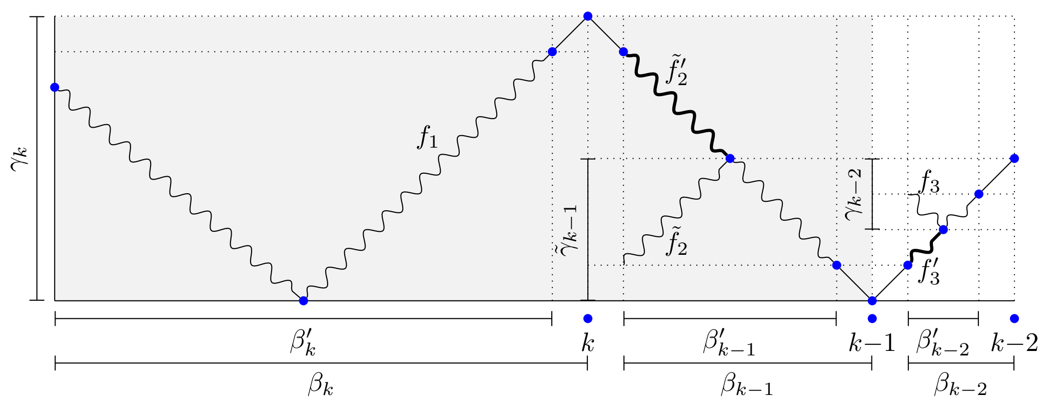


Figure 8 Proof of Claim 14 (schematic drawing): thin lines depict f_1, $\tilde{f}_2$ and f_3; thick lines denote the results $\tilde{f}_2'$ and f_3' of performing initial reflections.

▷ Claim 13. Let $k \geq 2$. For all h $(1 \leq h \leq |\gamma_k|)$ such that $\gamma_k[h] = 1$, there exists a walk f such that: (i) $\gamma_k^f = \beta_k$; (ii) $f(1) = h$; and (iii) $f(|\beta_k|) = |\gamma_k|$.

Proof. Take the walk guaranteed by Claim 12, and, noting that the final letters of β_k and γ_k are both k, extend f by setting $f(|\beta_k|) = |\gamma_k|$. ◁

▷ Claim 14. Let $k \geq 2$. For all h $(1 \leq h \leq |\gamma_n|)$ such that $\gamma_k[h] = 1$, and for all p $(1 \leq p < k)$ there exists a walk g such that: (i) $\gamma_k^f = \beta_k \beta_{k-1} \cdots \beta_{k-p}$ and (ii) $f(1) = h$.

Proof. Since $\beta_2\beta_1 = 121$ and $\gamma_2 = 12$, the claim is immediate for $k = 2$. Hence we may assume $k \geq 3$. By Claim 13, let f_1 be a walk on γ_k yielding β_k with $f_1(1) = h$ and $f_1(|\beta_k|) = |\gamma_k|$. Since f is a walk, and β_k contains only one occurrence of k, we have $f_1(|\beta_k|-1) = |\gamma_k|-1$. Set $g_1 = f_1$. By Claim 9, β_{k-2}' is a palindrome, whence $\gamma_k = (k-1) \cdot \beta_{k-1}' \cdot k = (k-1) \cdot \left(\beta_{k-2}' \cdot (k-2) \cdot \beta_{k-2}'\right) \cdot k = \tilde{\gamma}_{k-1} \cdot \beta_{k-2}' \cdot k$. Noting that the penultimate position of γ_{k-1} is always occupied by the letter 1, by Claim 13 let f_2 be a walk on γ_{k-1} yielding the word β_{k-1}, with $f_2(1) = |\gamma_{k-1}|-1$. By Claim 9, $f_2(i) = 1$ only when $i = |\beta_{k-1}|/2$, since that is the only position of β_{k-1} occupied by the letter $k-2$. It follows that the function $\tilde{f}_2$ defined by $\tilde{f}_2(i) = |\gamma_{k-1}|-(f_2(i)-1)$ is a walk on $\tilde{\gamma}_{k-1}$ yielding the word β_{k-1} with $\tilde{f}_2(1) = 2$, and achieving its maximum value $f(i) = |\gamma_{k-1}|$ only at $i = |\beta_{k-1}|/2$. Now regarding $\tilde{f}_2$ as a stroll on $\gamma_k = \tilde{\gamma}_{k-1} \cdot \beta_{k-2}' \cdot k$, let $\tilde{f}_2'$ be the initial reflection on $\tilde{f}_2$ over the interval $[1, |\beta_{k-1}|/2]$. Since $\gamma_k = (k-1) \cdot \beta_{k-1}' \cdot k$, with β_{k-1}' a palindrome, it follows that the stroll $\tilde{f}_2'$ on γ_k also yields the same result as $\tilde{f}_2$, namely β_{k-1}. Now let g_2 be the result of appending $\tilde{f}_2'$ to g_1, as shown in the shaded part of Fig. 8. (Most of the curves drawn schematically here will actually be straight lines, but no matter.) Formally, we define $g_2 : [1, |\beta_k| + |\beta_{k-1}|] \to [1, |\gamma_k|]$ to be the function:

$$g_2(i) = \begin{cases} g_1(i) & \text{if } 1 \leq i \leq |\beta_k| \\ \tilde{f}_2'(i - |\beta_k|) & \text{if } |\beta_k| < i \leq |\beta_k| + |\beta_{k-1}|. \end{cases}$$

Since $g_1(|\beta_k|) = |\gamma_k|$ and $\tilde{f}_2'(1) = |\gamma_k|-1$, we see that g_2 is indeed a walk on γ_k as shown (i.e. with no jumps), yielding $\beta_k\beta_{k-1}$. We remark that $g_2(|\beta_k| + |\beta_{k-1}|) = 1$. Notice that we needed to invert f_2 to yield $\tilde{f}_2$, so as to make the latter's reflection $\tilde{f}_2'$ join up to the end of g_1 properly.

We now repeat the above procedure, as shown in the unshaded part of Fig. 8. By Claim 13, and noting that the penultimate position of γ_{k-2} is occupied by the letter 1, let f_3 be a walk on γ_{k-2} yielding the word β_{k-2}, with $f_3(1) = |\gamma_{k-2}|-1$. By Claim 9, $f_3(i) = 1$ only when $i = |\beta_{k-2}|/2$, since that is the only position of β_{k-2} occupied by the letter $k-3$. Observing that $\gamma_{k-1} = \tilde{\gamma}_{k-2} \cdot \beta'_{k-3} \cdot k$, and hence $\tilde{\gamma}_{k-1} = k \cdot \tilde{\beta}'_{k-3} \cdot \gamma_{k-2}$, we see that, by shifting f_3 upwards by $|k \cdot \tilde{\beta}'_{k-2}|$, we can regard it as a stroll on γ_k. This (shifted) stroll reaches its minimum value $|k \cdot \tilde{\beta}'_{k-3}| + 1$ exactly once in the middle of its range. Let f'_3 be the initial reflection on of this stroll over the interval $[1, |\beta_{k-2}|/2]$. Since $\tilde{\gamma}_{k-1} = (k-1) \cdot \beta'_{k-2} \cdot (k-2)$ with β'_{k-2} a palindrome, we see by inspection that the stroll f'_3 on γ_k yields the same result as f_3, namely β_{k-2}. Now take g_3 to be the result of appending f'_3 to g_2, just as we earlier appended $\tilde{f}'_2$ to g_1. Thus, g_3 is a walk on γ_k yielding $\beta_k\beta_{k-1}\beta_{k-2}$. Notice that f_3, unlike f_2, did not need to be inverted to make its reflection f'_3 join up to the end of g_2. Evidently, this process may be continued until we obtain the desired walk g_{p+1} on γ_k yielding $\beta_k\beta_{k-1}\cdots\beta_{k-p}$, with the inversion step (producing $\tilde{f}_h$ from $\tilde{f}_h$) required only when h is even. ◁

We now prove Theorem 5, establishing by induction the following slightly stronger claim.

▷ Claim 15. Fix $k \geq 2$. For all $n \geq k$ and for all h $(1 \leq h \leq |\gamma_k|)$ such that $\gamma_n[h] = 1$, there exists a walk f such that $\alpha_n^{(k)} = \gamma_k^f$ and $f(1) = h$.

Proof. If $n = k$, then $\alpha_n^{(k)} = \beta_k$, and the result is immediate from Claim 13. If $n = k+1$, then $\alpha_n^{(k)} = \beta_k\beta_{k-1}\cdots\beta_1$, and the result is immediate from Claim 14, setting $p = k-1$.

For the inductive step we suppose $n \geq k+2$ and assume the result holds for values smaller than n. We consider first the slightly easier case where $n \geq 2k$. Set $h_1 = h$. Writing $\alpha_n^{(k)} = \alpha_{n-1}^{(k)} \cdots \alpha_{n-k}^{(k)}$, by inductive hypothesis, let g_1 be a walk such that $\alpha_{n-1}^{(k)} = \gamma_k^{g_1}$ and $g_1(1) = h_1$. Now let h'_1 be the final value of g_1, that is, $g_1(|\alpha_{n-1}^{(k)}|) = h'_1$. By Claim 10, there exists h_2 such that $|h_2 - h'_1| \leq 1$, and $\gamma_k[h_2] = 1$. Again, by inductive hypothesis, let g_2 be a walk such that $\alpha_{n-2}^{(k)} = \gamma_k^{g_2}$ and $g_2(1) = h_2$. Let h'_2 be the final value of g_2, and let h_3 be such that $|h_3 - h'_2| \leq 1$, and $\gamma_k[h_3] = 1$. Proceed in the same way, obtaining walks $g_3, \ldots, g_k$. Taking f to be the result of concatenating $g_1, g_2, g_3, \ldots, g_k$ in the obvious fashion yields the desired walk.

If $2k > n \geq k+2$, then we have $\alpha_n^{(k)} = \alpha_{n-1}^{(k)}\alpha_{n-2}^{(k)} \cdots \alpha_{k+1}^{(k)}\beta_k\beta_{k-1}\cdots\beta_{k-p}$, where $p = 2k-n$. We begin as in the previous paragraph: setting $h_1 = h$, by inductive hypothesis, let g_1 be a walk such that $\alpha_{n-1}^{(k)} = \gamma_k^{g_1}$ and $g_1(1) = h_1$. Now let h'_1 be the final value of g_1, that is, $g_1(|\alpha_{n-1}^{(k)}|) = h'_1$. By Claim 10, there exists h_2 such that $|h_2 - h'_1| \leq 1$, and $\gamma_k[h_2] = 1$. Now continue as before so as to obtain walks $g_2, g_3 \ldots$, with respective starting points $h_2, h_3, \ldots$, but stopping when we have obtained g_{k-p-1}, and the following starting point h_{k-p}. Observe that concatenating $g_1, g_2, g_3, \ldots, g_{k-p-1}$ gives a walk on γ_k which yields the word $\alpha_{n-1}^{(k)}\alpha_{n-2}^{(k)} \cdots \alpha_{k+1}^{(k)}$. By Claim 14, choose g_{k-p} to be a walk on γ_k yielding the word $\beta_k\beta_{k-1}\cdots\beta_{k-p}$ and with $g_{k-p}(1) = h_{k-p}$. Taking f to be the result of concatenating $g_1, g_2, g_3, \ldots, g_{k-p}$ establishes the claim. ◁

References

1 Bartosz Bednarczyk, Daumantas Kojelis, and Ian Pratt-Hartmann. On the Limits of Decision: the Adjacent Fragment of First-Order Logic. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111:1–111:21, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

2 M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific, Singapore and River Edge, NJ, 2002.

3 N. Pytheas Fogg. *Substitutions in Dynamics, Arithmetics and Combinatorics*. Number 1794 in Lecture Notes in Mathematics, edited by V. Berthé, S. Ferenczi, C. Mauduit, and A. Siegel. Springer Verlag, Berlin, Heidelberg, New York, 2002.

4 M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 2005.

5 Ian Pratt-Hartmann. Walking on words (v.2), 2024. `arXiv:2208.08913`.

6 Gérard Rauzy. Nombres algébriques et substitutions. *Bulletin de la Société Mathématique de France*, 110:147–178, 1982.

A Data Structure for the Maximum-Sum Segment Problem with Offsets

Yoshifumi Sakai
Graduate School of Agricultural Science, Tohoku University, Japan

Abstract

Consider a variant of the maximum-sum segment problem for a sequence X_0 of n real numbers, which asks an arbitrary contiguous subsequence of X_a that maximizes the sum of its elements for any given real number a, where X_a is the sequence obtained by subtracting a from each element in X_0. Although this problem can be solved in $O(n)$ time from scratch for any given X_0 and a, appropriate data structures for X_0 could support efficient queries of the solution for arbitrary a. We propose an $O(n \log^2 n)$-time, $O(n)$-space algorithm that takes X_0 as input and outputs such a data structure supporting $O(\log n)$-time queries.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases algorithms, sequence of real numbers, maximum-sum segment

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.26

Funding This work was supported by JSPS KAKENHI Grant Number JP23K10975.

1 Introduction

Given a sequence of real numbers, the maximum-sum segment (MSS) problem, also known as the maximum subarray problem, is to find an arbitrary segment (contiguous subsequence) of the sequence that maximizes the sum of its elements, which we call an MSS of the sequence. This problem has many applications in various industrial and academic fields such as image processing [7], pattern recognition [12], and biological sequence analysis [16]. For example, in biological sequence analysis, when the similarity between amino acids at corresponding positions in multiple amino acid sequences encoding homologous proteins is given as a score, the most highly conserved region of the sequences that is expected to play an important role [15] can be found by solving this problem [16].

The MSS problem is solvable in linear time by Kadane's algorithm, as surveyed in [2]. Due to the existence of applications in biological sequence analysis, various variants and related problems of the MSS problem have also been considered. Chen and Chao [3] designed a linear-time constructible data structure that supports constant-time queries of an MSS for any segment of the sequence. A maximal local MSS is a local MSS that is not a segment of any local MSS other than it, where a local MSS is a segment that has itself as its only MSS. Ruzzo and Tompa [13] showed that all distinct maximal local MSSs can be determined in linear time, and Sakai [14] designed a linear-time constructible data structure supporting constant-time queries of the maximal local MSS of any given segment that contains any given position. Bangtsson and Chen [1] showed that an arbitrarily given number of non-overlapping segments that maximize the sum of all their elements can be found in linear time. Yu et al. [17] considered the MSS problem where each element of the input sequence is uncertain within a specific interval and proposed a linear-time algorithm for this problem. The density of a segment is defined as the mean of all elements in the segment. Cheng et al. [4] considered the MSS problem with the condition that the density of the segment to be found is between given lower and upper bounds and showed that the problem is solvable in linearithmic time, or in linear time if we do not adopt the upper bound condition. The

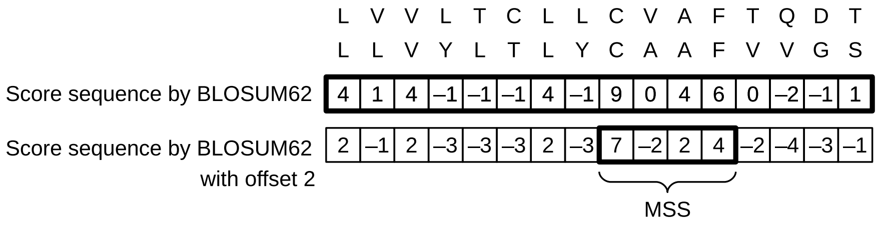

L V V L T C L L C V A F T Q D T
L L V Y L T L Y C A A F V V G S
Score sequence by BLOSUM62
4 1 4 –1 –1 –1 4 –1 9 0 4 6 0 –2 –1 1
Score sequence by BLOSUM62 with offset 2
2 –1 2 –3 –3 –3 2 –3 7 –2 2 4 –2 –4 –3 –1
MSS

Figure 1 Part of an alignment of a pair of homologous amino acid sequences, the score sequence by BLOSUM62 for it, and the same score sequence with offset 2, where thick frames represent the MSSs of the score sequences.

maximum-density segment problem [8, 10, 11] consists of finding an arbitrary segment of length between given lower and upper bounds that maximizes the density. Chung and Lu [5] showed that this problem is solvable in linear time.

The present article considers another variant of the MSS problem. Before presenting the definition, we discuss the motivation that led us to conceive this new variant. As an application of the MSS problem, consider finding a highly conserved region in a given pair of homologous amino acid sequences $u_1u_2\cdots u_n$ and $v_1v_2\cdots v_n$ with the ith amino acids u_i and v_i appearing at the corresponding positions. For any pair of amino acids u and v, let $\mathit{score}(u,v)$ be the logarithm of the ratio of the observed frequency to the expected frequency of u appearing U and v appearing V at the corresponding positions over all pairs of homologous amino acid sequences U and V. From this definition, $\mathit{score}(u,v)$ can be regarded as representing the similarity between amino acids u and v based on the likelihood of substitution as an accepted mutation. Tables designed to consist of $\mathit{score}(u,v)$ approximations are available as typical amino acid substitution matrices, including PAM matrices [6] and BLOSUMs [9]. Since the larger $\mathit{score}(u,v)$ is, the more similar amino acids u and v are, one might think that finding an MSS of $\mathit{score}(u_1,v_1)\mathit{score}(u_2,v_2)\cdots\mathit{score}(u_n,v_n)$ would yield a highly conserved region with respect to $u_1u_2\cdots u_n$ and $v_1v_2\cdots v_n$. However, the obtained MSS may be unnecessarily large due to the low threshold level for treating amino acids u and v as sufficiently similar. To resolve this undesirable situation, we can raise the low threshold level as we wish by treating the $\mathit{score}(u,v)$ value as decreased by a specific value that is set as an offset. Figure 1 shows an example of how introducing such an offset changes the MSS, in which, instead of $\mathit{score}(u,v)$s, values from BLOSUM62, one of BLOSUMs [9], are used. To obtain a highly conserved region as desired, it will be necessary to carefully adjust the offset, in some cases by more of a trial-and-error approach. It is possible to obtain the MSS by running Kadane's algorithm for each offset that is assumed to be appropriate. However, if a more efficient way to obtain the MSS for any given offset is available, this is the way to go.

The new variant of the MSS problem we consider is as follows. Let an offset-MSS data structure for a sequence X_0 of n real numbers be a data structure that supports queries of an arbitrary MSS of X_a for any real number a, where X_a denotes the sequence obtained from X_0 by subtracting a from each element. This type of query can arise when no firm meaning is evident in the value of each element in X_0 compared to 0 and only the relative differences in the values of elements are meaningful. This is because the MSS found can vary depending on the threshold level that separates positive from negative. For example, as mentioned earlier, in biological sequence analysis, by adjusting the criteria that separate whether each pair of amino acids is treated as similar or dissimilar, new regions may be identified as highly conserved in homologous amino acid sequences. In this article, we propose a straightforward $O(n)$-space offset-MSS data structure for X_0 supporting $O(\log n)$-time queries and design an $O(n\log^2 n)$-time, $O(n)$-space algorithm that constructs this data structure.

2 Preliminaries

Let n be an arbitrary positive integer and let X_0 be an arbitrary sequence of n real numbers. For any real number a, let X_a denote the sequence obtained by replacing each element x of X_0 with $x - a$. For any index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$, let $X_a(i, j)$ denote the contiguous subsequence consisting of the ith through jth elements of X_a. Note that $X_a(i, j)$ is non-empty (resp. empty), if $i \leq j$ (resp. $i = j + 1$). Let $S_a(i, j)$ denote the sum of all elements in $X_a(i, j)$, if $i \leq j$, or 0, otherwise. A *maximum-sum segment* (an *MSS*) of $X_a(i, j)$ is an arbitrary index pair (g, h) with $i \leq g \leq h + 1 \leq j + 1$ that maximizes $S_a(g, h)$. We define an *offset-MSS data structure* for X_0 as a data structure that supports queries of an arbitrary MSS of X_a for any real number a. Our aim is to design an efficient algorithm that outputs an efficient offset-MSS data structure for X_0. We assume that X_0 is given as an array of the sums $S_0(1, k)$ for all indices k with $0 \leq k \leq n$ in ascending order of k, so that $S_0(i, j)$ can be determined as $S_0(1, j) - S_0(1, i - 1)$ in $O(1)$ time.

As an efficient offset-MSS data structure for X_0, we consider a partition of the whole set of real numbers into several intervals each with a common MSS. More precisely, our goal is to design an efficient algorithm that finds a sequence consisting of $O(n)$ pairs $(\theta, (i, j))$ of a real number θ and an index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$ in descending order of θ such that for any real number a, if $(\theta, (i, j))$ is the last element with $\theta > a$ in the sequence, then (i, j) is an MSS of X_a. Let $OMSS_{X_0}$ denote an arbitrary such sequence. Apparently, $OMSS_{X_0}$ can be used as an offset-MSS data structure, which supports $O(\log n)$-time queries by performing a binary search.

Below we introduce the terminology and notations used to design our algorithm. For any real number a and any index pair (i, j) with $1 \leq i \leq j + 1 \leq n + 1$, let $X_a(i, j)$ be called *pref/suff-positive*, if both $S_a(i, k)$ and $S_a(k, j)$ are positive for any index k with $i \leq k \leq j$. Let $\alpha(i, j)$ denote the least real number such that $X_{\alpha(i,j)}(i, j)$ is not pref/suff-positive, if $i \leq j$, or ∞, otherwise. Let $\kappa(i, j)$ denote an arbitrary index k with $i \leq k \leq j$ such that at least one of $S_{\alpha(i,j)}(i, k) = 0$ or $S_{\alpha(i,j)}(k, j) = 0$, if $i \leq j$, or be undefined, otherwise. For any index pair (i, j) with $1 \leq i \leq j \leq n$, let $\delta(i, j)$ denote the real number a such that $S_a(i, j) = 0$, which is given as the *density* of $X_0(i, j)$, i.e., the mean $S_0(i, j)/(j - i + 1)$ of all elements in $X_0(i, j)$. As demonstrated in [10], it is useful to incorporate a geometric perspective when dealing with density and considering convex hulls. For any set $\mathcal{P}$ of distinct points (p, w) in the two-dimensional plane, we define the *lower (resp. upper) convex hull* of $\mathcal{P}$ to be the polygonal chain with the smallest number of points in $\mathcal{P}$ as vertices such that for any point (p, w) in $\mathcal{P}$, there exists a pair of consecutive vertices the straight line between which passes through a point (p, w') with $w' \leq w$ (resp. $w' \geq w$).

3 Algorithm constructing an offset-MSS data structure

In this section, we show that $OMSS_{X_0}$ exists and design Algorithm findOMSS as an algorithm that finds $OMSS_{X_0}$ in $O(n \log^2 n)$ time and $O(n)$ space.

Algorithm findOMSS finds $OMSS_{X_0}$ based on a technique that divides the problem of finding an MSS into two subproblems, which is presented in the following lemma.

▶ **Lemma 1.** *For any real number a and any index pair (i, j) with $1 \leq i \leq j \leq n$, if $X_a(i, j)$ is pref/suff-positive, then (i, j) is the only MSS of $X_a(i, j)$; otherwise, at least one of an arbitrary MSS of $X_a(i, \kappa(i, j) - 1)$ and an arbitrary MSS of $X_a(\kappa(i, j) + 1, j)$ is an MSS of $X_a(i, j)$.*

Table 1 Notations used in Section 3.1.

Notation	Definition
$\kappa'(i,j)$	An arbitrary index k with $i \le k \le j$ that minimizes $\delta(i,k)$
$\kappa'(i,g,h)$	An arbitrary index k with $g \le k \le h$ that minimizes $\delta(i,k)$
$H(g,h)$	The lower convex hull for all two-dimensional points $(k, S_0(1,k))$ with $g \le k \le h$
$K'(g,h)$	The sequence of all indices k with $g \le k \le h$ such that $(k, S_0(1,k))$ is a vertex of $H(g,h)$ in ascending order
$\mathcal{K}'$	The set of sequences $K'(g,h)$ for all canonical index pairs (g,h), where (g,h) is canonical if $1 \le g \le h \le n$, $h-g+1$ is a power of two, and both $g-1$ and h are divisible by $h-g+1$
$k(g,h^\star)$	The greatest index that is shared by $K'(g,h)$ and $K'(g,h^\star)$, where (g,h) is a canonical index pair with $g < h$ and $h^\star = (g+h-1)/2$
$k(g^\star,h)$	The greatest index that is shared by $K'(g,h)$ and $K'(g^\star,h)$, where (g,h) is a canonical index pair with $g < h$ and $g^\star = (g+h+1)/2$
K'	The forest of binary trees such that the set of vertices consists of all canonical index pairs (g,h) and each vertex (g,h) with $g < h$ has as children $(g,h^\star)$ with label $k(g,h^\star)$ and $(g^\star,h)$ with label $k(g^\star,h)$, which is the $O(n)$-time constructible data structure that supports $O(\log^2 n)$-time queries of $\kappa'(i,j)$ for any index pair (i,j) with $1 \le i \le j \le n$ we propose
K'	An implementation of K', which is defined as the array of arrays $\mathsf{K}'[l]$ with $0 \le l \le \lfloor \log_2 n \rfloor - 1$, where $\mathsf{K}'[l]$ consists of elements $\mathsf{K}'[l][m]$ with $1 \le m \le 2\lfloor n/2^{l+1} \rfloor$, each containing the label of the vertex $(2^l(m-1)+1, 2^l m)$ of K'

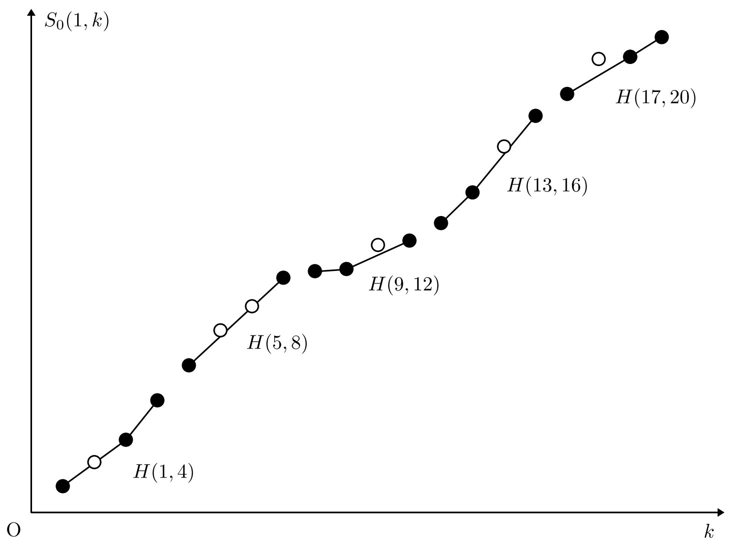
Proof. Let (g,h) be an arbitrary index pair with $i \le g \le h \le j$. If $X_a(i,j)$ is pref/suff-positive and $i < g$ (resp. $h < j$), then $S_a(i,g-1)$ is positive (resp. non-negative) and $S_a(h+1,j)$ is non-negative (resp. positive), implying that $S_a(i,j) > S_a(g,h)$. Suppose that $X_a(i,j)$ is not pref/suff-positive and $g \le \kappa(i,j) \le h$. By symmetry, it suffices to show that if $S_{\alpha(i,j)}(i,\kappa(i,j)) = 0$, then $S_a(\kappa(i,j)+1,h) \ge S_a(g,h)$. Since $S_{\alpha(i,j)}(i,g-1) \ge 0$ due to definition of $\alpha(i,j)$, $S_{\alpha(i,j)}(g,\kappa(i,j)) \le 0$. Therefore, $S_{\alpha(i,j)}(\kappa(i,j)+1,h) \ge S_{\alpha(i,j)}(g,h)$, which implies that $S_a(\kappa(i,j)+1,h) \ge S_a(g,h)$ due to $a \ge \alpha(i,j)$. ◀

Whenever applying Lemma 1, we need $\alpha(i,j)$ to investigate whether $X_a(i,j)$ is pref/suff-positive, and $\kappa(i,j)$ if it is not. To support time-efficient queries of $\alpha(i,j)$ and $\kappa(i,j)$, one might think of a lookup table as a naive data structure, which supports $O(1)$-time queries. However, it takes $O(n^2)$ time to construct it and also requires $O(n^2)$ space to store it. We design another data structure by taking a different approach to reduce preprocessing time and space requirement to $O(n)$ but manage to achieve $O(\log^2 n)$-time queries.

The remaining part of this section is organized as follows. We first propose an $O(n)$-time constructible data structure that supports $O(\log^2 n)$-time queries of $\alpha(i,j)$ and $\kappa(i,j)$ for any index pair (i,j) with $1 \le i \le j \le n$ in Section 3.1, and then design Algorithm findOMSS using this data structure in Section 3.2.

3.1 Data structure supporting queries of $\alpha(i,j)$ and $\kappa(i,j)$

The data structure we propose to support queries of $\alpha(i,j)$ and $\kappa(i,j)$ consists of two symmetric components, K' and K''. This symmetry is based on the following reduction of the problem of determining $\alpha(i,j)$ into two symmetric subproblems. Let $\kappa'(i,j)$ (resp.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
12	11	10	19	15	16	11	13	3	1	11	2	8	14	21	14	10	16	1	9	6

Figure 2 Lower convex hulls $H(1,4)$, $H(5,8)$, $H(9,12)$, $H(13,16)$, and $H(17,20)$ for a concrete example of X_0 shown at the bottom, where each point $(k, S_0(1,k))$ with $1 \le k \le 20$ is indicated by a solid bullet, if it is a vertex of the hulls, or an open bullet, otherwise.

$\kappa''(i,j)$) denote an arbitrary index k with $i \le k \le j$ that minimizes $\delta(i,k)$ (resp. $\delta(k,j)$). Hence, $\alpha(i,j)$ is equal to the minimum of $\delta(i,\kappa'(i,j))$ and $\delta(\kappa''(i,j),j)$. Furthermore, if $\delta(i,\kappa'(i,j)) = \alpha(i,j)$ (resp. $\delta(\kappa'(i,j),j) > \alpha(i,j)$), then $\kappa'(i,j)$ (resp. $\kappa''(i,j)$) satisfies the condition of $\kappa(i,j)$. Based on this reduction, if K' supports $O(\log^2 n)$-time queries of $\kappa'(i,j)$ and K'' supports $O(\log^2 n)$-time queries of $\kappa''(i,j)$, then $\alpha(i,j)$ and $\kappa(i,j)$ can be determined in $O(\log^2 n)$ time. By symmetry, we will henceforth focus only on designing K' as a data structure that can be constructed in $O(n)$ time and supports $O(\log^2 n)$-time queries of $\kappa'(i,j)$ for any index pair (i,j) with $1 \le i \le j \le n$.

This section introduces many other notations besides $\kappa'(i,j)$ and K'. Table 1 summarizes such notations.

Our strategy to achieve $O(\log^2 n)$-time queries of $\kappa'(i,j)$ is to reduce the problem of finding $\kappa'(i,j)$ to the subproblems of finding certain $O(\log n)$ candidates from which $\kappa'(i,j)$ can be found and to design a data structure that supports $O(\log n)$-time queries of the candidate. For any index pair (g,h) with $1 \le g \le h \le n$, let $K'(g,h)$ denote the sequence of all indices k with $g \le k \le h$ such that $(k, S_0(1,k))$ is a vertex of $H(g,h)$ in ascending order, where $H(g,h)$ denotes the lower convex hull for all two-dimensional points $(k, S_0(1,k))$ with $g \le k \le h$ (see Figure 2). Below is a key lemma that will serve as the foundation for our strategy.

▶ **Lemma 2.** *For any indices i, g, and h with $1 \le i \le g \le h \le n$, a binary search of $K'(g,h)$ finds an index k with $g \le k \le h$ that minimizes $\delta(i,k)$.*

Proof. For any index k with $g \le k \le h$, $\delta(i,k)$ is equal to the slope $(S_0(1,k) - S_0(1,i-1))/(k-(i-1))$ of the straight line passing through points $(i-1, S_0(1,i-1))$ and $(k, S_0(1,k))$. Thus, the lemma follows from the fact that for any index k with $g \le k \le h$ that minimizes $\delta(i,k)$, the line passing though $(i, S_0(1,i-1))$ is tangent to $H(g,h)$ at vertex $(k, S_0(1,k))$. ◀

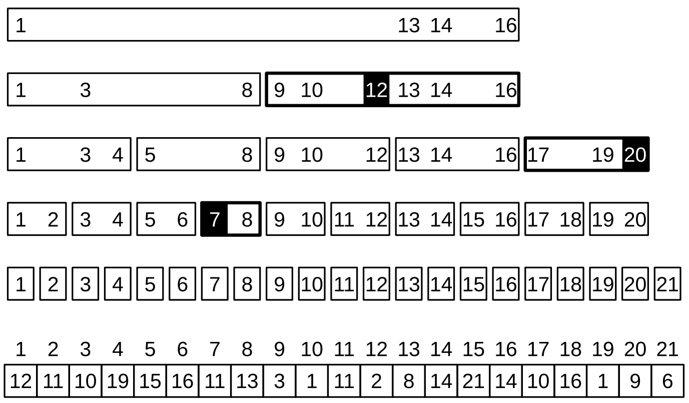
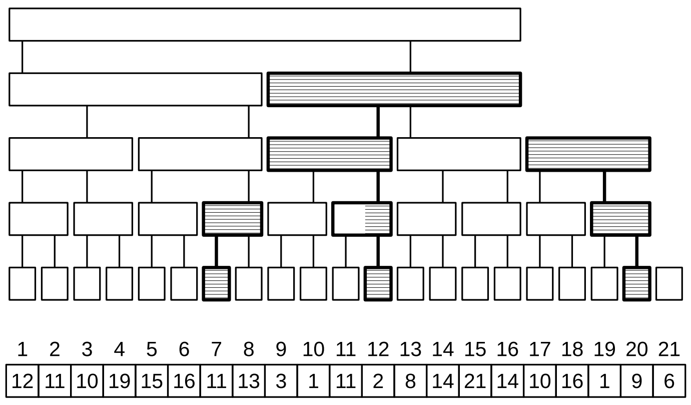


Figure 3 Set $\mathcal{K}'$ for the same X_0 as Figure 2 shown at the bottom, where all elements in $K'(g, h)$ for each canonical index pair (g, h) are presented as indices in the rectangle lying between positions g and h and, for example, the highlighted indices represent $\kappa'(7, 7, 8)$, $\kappa'(7, 9, 16)$, and $\kappa'(7, 17, 20)$, which are obtained as candidates for determining $\kappa'(7, 20)$ $(= 12)$.

A naive data structure immediately suggested by Lemma 2, which consists of sequences $K'(i, j)$ for all index pairs (i, j) with $1 \le i \le j \le n$, supports $O(\log n)$-time queries of $\kappa'(i, j)$ but consumes $O(n^3)$ space. Thus, we cannot adopt this naive data structure as is. However, by carefully choosing its particular subset, we can obtain an $O(n \log n)$-space data structure that supports $O(\log^2 n)$-time queries. This subset, which we denote by $\mathcal{K}'$, consists of sequences $K'(g, h)$ for all index pairs (g, h) with $1 \le g \le h \le n$ such that $h - g + 1$ is a power of two and both $g - 1$ and h are divisible by $h - g + 1$ (see Figure 3). We call any such index pair (g, h) *canonical*. Note that $\mathcal{K}'$ can be stored in $O(n \log n)$ space because for any power ℓ of two with $1 \le \ell \le n$, there exist at most n/ℓ canonical pairs (g, h) such that $h - g + 1 = \ell$, each having $K'(g, h)$ that can be stored in $O(\ell)$ space. The interval of indices represented by any index pair (i, j) with $1 \le i \le j \le n$ is partitioned into $O(\log n)$ intervals each represented by a canonical index pair (g, h). By applying Lemma 2 to each of such $O(\log n)$ canonical index pairs (g, h), we can obtain $O(\log n)$ candidates $\kappa'(i, g, h)$ in $O(\log^2 n)$ time, where $\kappa'(i, g, h)$ denotes an arbitrary index k with $g \le k \le h$ that minimizes $\delta(i, k)$. Any of the candidates $\kappa'(i, g, h)$ that minimizes $\delta(i, \kappa'(i, g, h))$ satisfies the condition of $\kappa'(i, j)$.

The only reason that $\mathcal{K}'$ cannot be adopted as K' is that the space required to store it is $O(n \log n)$, not $O(n)$. To resolve this issue, we define K' by removing duplicate information from $\mathcal{K}'$. We do this based on the fact that $K'(g, h)$ for any canonical index pair (g, h) with $g < h$ can recursively be represented by $K'(g, h^\star)$, $K'(g^\star, h)$, and two specific indices, where $h^\star = (g + h - 1)/2$ and $g^\star = (g + h + 1)/2$ $(= h^\star + 1)$. One of the specific indices is the greatest element of $K'(g, h^\star)$ that is shared by $K'(g, h)$ and the other is the least element of $K'(g^\star, h)$ that is shared by $K'(g, h)$. Let $k(g, h^\star)$ and $k(g^\star, h)$ denote these indices, respectively. Note that the concatenation of the prefix of $K'(g, h^\star)$ with $k(g, h^\star)$ as the last element followed by the suffix of $K'(g^\star, h)$ with $k(g^\star, h)$ as the first element constitutes $K'(g, h)$. This can be verified because $K'(g, h)$, $K'(g, h^\star)$, and $K'(g^\star, h)$ are defined to represent all vertices of the lower convex hulls $H(g, h)$, $H(g, h^\star)$, and $H(g^\star, h)$, respectively. For example, if X_0 is the same as Figure 2, then $K'(1, 8) = \langle 1, 3, 8 \rangle$ can be represented by $K'(1, 4) = \langle 1, 3, 4 \rangle$, $K'(5, 8) = \langle 5, 8 \rangle$, $k(1, 4) = 3$, and $k(5, 8) = 8$ in this manner. Thus, in our recursive representation, the only information that must be explicitly retained with respect to (g, h) to recover $K'(g, h)$ is indices $k(g, h^\star)$ and $k(g^\star, h)$.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
12	11	10	19	15	16	11	13	3	1	11	2	8	14	21	14	10	16	1	9	6

Figure 4 Forest K' for the same X_0 as Figure 2 shown at the bottom, where each rectangle lying between positions g and h indicates vertex (g, h), the position of each edge represents its label, and the shaded areas indicate the ranges between $k_{\vdash}$ and $k_{\dashv}$ when tracing paths by Algorithm determineKappa presented in Algorithm 2 to determine $\kappa'(7, 20)$.

Based on the ideas discussed above, we define K' as a forest of binary trees as follows (see also Figure 4). The set of vertices in K' consists of all canonical index pairs (g, h) with $1 \le g \le h \le n$. Each vertex (g, h) with $g < h$ has two children, $(g, h^\star)$ and $(g\star, h)$, where $h^\star = (g + h - 1)/2$ and $g^\star = (g + h + 1)/2$. Furthermore, the edge between (g, h) and $(g, h^\star)$ is labeled with $k(g, h^\star)$, the greatest index in $K'(g, h^\star)$ that is shared by $K'(g, h)$. Analogously, the edge between (g, h) and $(g^\star, h)$ is labeled with $k(g^\star, h)$, the least index in $K'(g^\star, h)$ that is shared by $K'(g, h)$. Any vertex (g, h) with $g = h$ is a leaf. Since the number of all canonical index pairs is $O(n)$, K' can be stored in $O(n)$ space, unlike the case of $\mathcal{K}'$, which requires $O(n \log n)$ space.

We implement K' as an array K' of arrays $\mathsf{K}'[l]$ with $0 \le l \le \lfloor \log_2 n \rfloor - 1$, where $\mathsf{K}'[l]$ consists of elements $\mathsf{K}'[l][m]$ with $1 \le m \le 2\lfloor n/2^{l+1} \rfloor$, each containing the label of the edge between vertex $(2^l(m-1)+1, 2^l m)$ and its parent. Algorithm 1 presents a pseudo-code of Algorithm constructK, which we propose as an algorithm that constructs K' in $O(n)$ time. The correctness of the algorithm and the execution time are shown in the following lemma.

▶ **Lemma 3.** *Algorithm* constructK *outputs* K' *in* $O(n)$ *time as an implementation of forest* K'.

Proof. For any index l with $0 \le l \le \lfloor \log_2 n \rfloor$, let $\mathcal{K}'_l$ denote the array of sequences $K'(g, h)$ for all of the $\lfloor n/2^l \rfloor$ canonical index pairs (g, h) with $1 \le g \le h \le n$ and $h - g + 1 = 2^l$ in ascending order of g. To construct K' in $O(n)$ time, Algorithm constructK initializes K_* to $\mathcal{K}'_0$ (by line 2) and updates it from $\mathcal{K}'_{l-1}$ to $\mathcal{K}'_l$ (by lines 4 through 15) for each index l from 1 to $\lfloor \log_2 n \rfloor$. In this process, when $K'(g, h^\star)$ and $K(g^\star, h)$ in $\mathcal{K}'_{l-1}$ are merged into $K'(g, h)$ in $\mathcal{K}'_l$, two labels $k(g, h^\star)$ and $k(g^\star, h)$ are also obtained, where $(g, h^\star)$ and $(g^\star, h)$ are children of (g, h). Let l and m be the indices such that $h = 2^l m \; (= 2^{l-1} \cdot 2m)$ and hence $h^\star = 2^{l-1}(2m - 1)$. The loop executed by lines 5 through 13 for l and m repeatedly deletes the last element k_{-1} of the current prefix of $K'(g, h^\star)$ stored in $\mathsf{K}_*[2m-1]$, if $(k_{-1}, S_0(1, k_{-1}))$ is not a vertex of $H(g, h)$, or the first element k_1 of the current suffix of $K'(g^\star, h)$ stored in $\mathsf{K}_*[2m]$, otherwise, until both $(k_{-1}, S_0(1, k_{-1}))$ and $(k_1, S_0(1, k_1))$ are vertices of $H(g, h)$. If the condition of line 8 holds, then $(k_{-1}, S_0(1, k_{-1}))$ is not a vertex of $H(g, h)$ because it is on or above the straight line passing through $(k_{-2}, S_0(1, k_{-2}))$ and $(k_1, S_0(1, k_1))$. The analogy holds for k_1. Furthermore, if neither of the conditions in lines 8 and 10 holds,

Algorithm 1 A pseudo-code of Algorithm constructK.

```
1  $\mathsf{K}' \leftarrow$ an array of arrays $\mathsf{K}'[l]$ with $0 \le l \le \lfloor \log_2 n \rfloor - 1$ each consisting of elements $\mathsf{K}'[l][m]$
   with $1 \le m \le 2\lfloor n/2^{l+1} \rfloor$;
2  $\mathsf{K}_* \leftarrow$ an array of $n$ elements $\mathsf{K}_*[k]$ with $1 \le k \le n$ each initialized to a bidirectional linked
   list consisting of a single element $k$;
3  foreach $l$ from 1 to $\lfloor \log_2 n \rfloor$ do
4      foreach $m$ from 1 to $\lfloor n/2^l \rfloor$ do
5          while not broken do
6              $k_{-1}, k_{-2} \leftarrow$ the last and second last elements of $\mathsf{K}_*[2m-1]$, respectively;
7              $k_1, k_2 \leftarrow$ the first and second elements of $\mathsf{K}_*[2m]$, respectively;
8              if $k_{-2}$ exists and $\delta(k_{-2}+1, k_{-1}) \ge \delta(k_{-2}+1, k_1)$ then
9                  delete $k_{-1}$ from $\mathsf{K}_*[2m-1]$
10             else if $k_2$ exists and $\delta(k_{-1}+1, k_2) \ge \delta(k_1+1, k_2)$ then
11                 delete $k_1$ from $\mathsf{K}_*[2m]$
12             else
13                 break
14         $\mathsf{K}'[l-1][2m-1] \leftarrow k_{-1}$; $\mathsf{K}'[l-1][2m] \leftarrow k_1$;
15         $\mathsf{K}_*[m] \leftarrow$ the concatenation of $\mathsf{K}_*[2m-1]$ followed by $\mathsf{K}_*[2m]$
16 output $\mathsf{K}'$ as an implementation of $K'$
```

then both $(k_{-1}, S_0(1, k_{-1}))$ and $(k_1, S_0(1, k_1))$ are vertices of $H(g, h)$. This is because the lower convex hull for the existing $(k_{-2}, S_0(1, k_{-2}))$, $(k_{-1}, S_0(1, k_{-1}))$, $(k_1, S_0(1, k_1))$, and $(k_2, S_0(1, k_2))$ has all of them as vertices. Thus, just after the loop is broken by line 13, $k_{-1} = k(g, h^\star)$ and $k_1 = k(g^\star, h)$, which are respectively stored as appropriate elements of K' by line 14. In addition, the concatenation of the eventual prefix of $K'(g, h^\star)$ followed by the eventual suffix of $K'(g^\star, h)$ constitutes $K'(g, h)$, which is stored in $\mathsf{K}_*[m]$ by line 15. Therefore, Algorithm constructK outputs K' correctly.

Each element of K_* is implemented as a bidirectional linked list and line 15 directly concatenates the lists pointed to by $\mathsf{K}_*[2m-1]$ and $\mathsf{K}_*[2m]$, respectively, and sets the resulting list to the one pointed to by $\mathsf{K}_*[m]$ in $O(1)$ time. Hence, the algorithm runs in time linear in the sum of the number of canonical index pairs and the number of indices deleted by lines 9 and 11, both of which are $O(n)$. ◀

As an algorithm that allows array K' to support $O(\log^2 n)$-time queries of $\kappa'(i, j)$, we propose Algorithm determineKappa(i, j) a pseudo-code of which is presented in Algorithm 2.

▶ **Lemma 4.** *Forest K', implemented as array K', supports $O(\log^2 n)$-time queries of $\kappa'(i, j)$ for any index pair (i, j) with $1 \le i \le j \le n$ by executing Algorithm* determineKappa(i, j).

Proof. Algorithm determineKappa(i, j) consists of two phases.

The first phase is executed by lines 1 through 7 to decompose (i, j) into a sequence C of $O(\log n)$ canonical index pairs (g, h) in a straightforward way. Obviously, this phase runs in $O(\log n)$ time.

The second phase determines indices $\kappa'(i, g, h)$ for all canonical index pairs (g, h) in C to determine $\kappa'(i, j)$ by executing lines 8 through 19. For each such (g, h), lines 10 through 16 determine $\kappa'(i, g, h)$ based on Lemma 2 without having $K'(g, h)$ explicitly. The binary search in Lemma 2 is done by tracing the path from (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$. In this tracing process, two indices $k_\vdash$ and $k_\dashv$ are maintained so that whenever an internal vertex (e, f) is visited, for any index k with $e \le k \le f$, k is an element of $K'(g, h)$ if and only if

Algorithm 2 A pseudo-code of Algorithm determineKappa(i, j).

```
C ← an empty sequence;
$\tilde{\jmath} \leftarrow 2^{\lfloor \log_2 n \rfloor}$; $\tilde{\imath} \leftarrow 2^{\lfloor \log_2 n \rfloor} + 1$;
foreach $l$ from $\lfloor \log_2 n \rfloor$ to 1 in descending order do
    if $i \le \tilde{\jmath} - 2^l + 1$ and $\tilde{\jmath} \le j$ then append $(\tilde{\jmath} - 2^l + 1, \tilde{\jmath})$ to C;
    if $i \le \tilde{\jmath} - 2^l + 1$ then $\tilde{\jmath} \leftarrow \tilde{\jmath} - 2^l$;
    if $i \le \tilde{\imath}$ and $\tilde{\imath} + 2^l - 1 \le j$ then append $(\tilde{\imath}, \tilde{\imath} + 2^l - 1)$ to C;
    if $\tilde{\imath} + 2^l - 1 \le j$ then $\tilde{\imath} \leftarrow \tilde{\imath} + 2^l$;
$\delta \leftarrow \infty$;
foreach $(g, h)$ in C do
    $l \leftarrow \log_2(h - g + 1)$; $m \leftarrow h/2^l$; $k_\vdash \leftarrow g$; $k_\dashv \leftarrow h$;
    while $l > 0$ do
        $k_\triangleleft \leftarrow \mathsf{K}'[l-1][2m-1]$; $k_\triangleright \leftarrow \mathsf{K}'[l-1][2m]$;
        if $k_\dashv < k_\triangleright$ or ($k_\vdash \le k_\triangleleft$ and $\delta(i, k_\triangleleft) \le \delta(i, k_\triangleright)$) then
            $k_\dashv \leftarrow \min(k_\dashv, k_\triangleleft)$; $l \leftarrow l - 1$; $m \leftarrow 2m - 1$
        else
            $k_\vdash \leftarrow \max(k_\vdash, k_\triangleright)$; $l \leftarrow l - 1$; $m \leftarrow 2m$
    if $\delta(i, m) < \delta$ then
        $\kappa' \leftarrow m$; $\delta \leftarrow \delta(i, m)$
output $\kappa'$ as $\kappa'(i, j)$
```

$k_\vdash \le k \le k_\dashv$ and k is an element of $K'(e, f)$. Obviously, we can maintain $k_\vdash$ and $k_\dashv$ by initializing $k_\vdash$ and $k_\dashv$ to g and h, respectively, and updating $k_\dashv$ (resp. $k_\vdash$) to the minimum (resp. maximum) of $k_\dashv$ (resp. $k_\vdash$) and $k(e, f^\star)$ (resp. $k(e^\star, f)$), if $(e, f^\star)$ (resp. $(e^\star, f)$) is chosen as the next vertex to visit after (e, f), where $f^\star = (e + f - 1)/2$ and $e^\star = (e + f + 1)/2$. To guarantee that $e \le \kappa'(i, g, h) \le f$ for any vertex (e, f) visited in the trace, the next vertex to visit after (e, f) is chosen as follows. If $k_\dashv < k(e^\star, f)$ (resp. $k(e, f^\star) < k_\vdash$), then $(e, f^\star)$ (resp. $(e^\star, f)$) is chosen, because $K'(e^\star, f)$ (resp. $K'(e, f^\star)$) shares no element with $K'(g, h)$. On the other hand, if both $k_\vdash \le k(e, f^\star)$ and $k(e^\star, f) \le k_\dashv$, then $k(e, f^\star)$ and $k(e^\star, f)$ are consecutive elements in $K'(g, h)$. Therefore, in such cases, it follows from Lemma 2 that we can choose $(e, f^\star)$, if $\delta(i, k(e, f^\star)) \le \delta(i, k(e^\star, f))$, or $(e^\star, f)$, otherwise. If we represent (e, f) by indices $l = \log_2(f - e + 1)$ and $m = f/2^l$, then $(e, f^\star)$ (resp. $(e^\star, f)$) is represented by $l - 1$ and $2m - 1$ (resp. $2m$), implying that $k(e, f^\star) = \mathsf{K}'[l-1][2m-1]$ (resp. $k(e^\star, f) = \mathsf{K}'[l-1][2m]$). Adopting this representation, lines 10 through 16 trace the path from (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$ in the manner described above, and hence the resulting m represents $\kappa'(i, g, h)$. After this trace, κ' and δ are updated by lines 17 and 18 so that δ represents the maximum value of $\delta(i, \kappa'(i, g, h))$ for $\kappa'(i, g, h)$ obtained so far and κ' represents the first $\kappa'(i, g, h)$ found that satisfies $\delta(i, \kappa'(i, g, h)) = \delta$. Thus, line 19 outputs $\kappa'(i, j)$ correctly. The execution time of this phase is $O(\log^2 n)$ because the number of elements (g, h) in C is $O(\log n)$ and the number of vertices in the path from each such (g, h) to $(\kappa'(i, g, h), \kappa'(i, g, h))$ is $O(\log n)$. ◀

We define K'' as the forest K' for the reversed X_0, which can be constructed in $O(n)$ time due to Lemma 3. Since $\kappa'(i, j)$ for the reversed X_0 represents $\kappa''((n+1) - j, (n+1) - i)$ for the original X_0 by symmetry, Lemmas 4 implies that K'' can support $O(\log^2 n)$-time queries of $\kappa''(i, j)$ for any index pair (i, j) with $1 \le i \le j \le n$. Thus, the following theorem immediately follows from Lemmas 3 and 4.

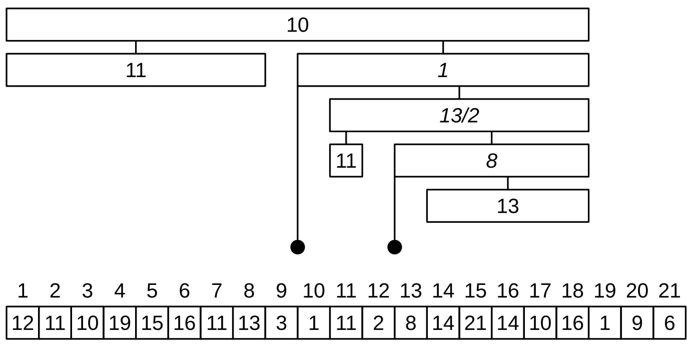


Figure 5 Tree $\tau(1,18)$ for the same X_0 as Figure 2 shown at the bottom, where each vertex (g,h) with $g \le h$ is indicated by a rectangle lying between positions g and h with $\alpha(g,h)$ as the label and each vertex (g,h) with $g = h+1$ is represented as a bullet between positions h and g.

▶ **Theorem 5.** *Forests K' and K'' can be constructed in $O(n)$ time and support $O(\log^2 n)$-time queries of $\alpha(i,j)$ and $\kappa(i,j)$ for any index pair (i,j) with $1 \le i \le j \le n$.*

3.2 Offset-MSS data structure supporting $O(\log n)$-time queries

To design Algorithm findOMSS, we consider a tree T from which a vertex (i,j) can be chosen as an MSS of X_a for any real number a. After defining T based on Lemma 1, we analyze how the vertices of T used to construct $OMSS_{X_0}$ can be chosen.

Lemma 1 gives us the MSS of X_a specifically if $a < \alpha(1,n)$, but only inductive candidates otherwise. We define T as a tree that presents explicit rather than inductive candidates, even if $a \ge \alpha(1,n)$. More precisely, T is defined as a tree such that all vertices (g,h) with $\alpha(i,j) \le a < \alpha(g,h)$ constitute the set of candidates, where (i,j) is the parent of (g,h). Our idea to realize such a tree T is to divide the problem of finding an MSS of $X_a(i,j)$ for any real number a with $a \ge \alpha(i,j)$ into the problems of finding an MSS of $X_a(g,h)$ with $\alpha(g,h) > \alpha(i,j)$ by applying Lemma 1 incrementally to define the set of children (g,h) of each internal vertex (i,j). Thus, formally, the set of children of any internal vertex (i,j) in T is defined as the set of leaves of the tree $\tau(i,j)$ introduced below. For any index pair (i,j) with $1 \le i \le j \le n$, let $\tau(i,j)$ denote the tree such that (i,j) is the root, any vertex (g,h) with $g \le h$ such that $X_{\alpha(i,j)}(g,h)$ is not pref/suff-positive (i.e., $\alpha(g,h) \le \alpha(i,j)$) has two children $(g,\kappa(g,h)-1)$ and $(\kappa(g,h)+1,h)$, and any other vertex is a leaf (see Figure 5). Although the topology of $\tau(i,j)$ is not uniquely defined in general due to the ambiguity of $\kappa(g,h)$, it is not difficult to verify that the set of leaves is unique. Let T denote the tree such that the root is $(1,n)$, any vertex (i,j) with $i \le j$ has all leaves of $\tau(i,j)$ as its children, and any other vertex is a leaf (see Figure 6). The following lemma claims that T has the property we intend.

▶ **Lemma 6.** *For any real number a, if $a < \alpha(1,n)$, then $(1,n)$ is an MSS of X_a; otherwise, any vertex (g,h) with $\alpha(i,j) \le a < \alpha(g,h)$ in T that maximizes $S_a(g,h)$ is an MSS of X_a, where (i,j) is the parent of (g,h).*

Proof. If $a < \alpha(1,n)$, then the lemma immediately follows from Lemma 1. Suppose that $a \ge \alpha(1,n)$. Consider an arbitrary series $\mathcal{C}$ of sets C of vertices in T such that

- the first set consists only of $(1,n)$,
- any set C containing at least one internal vertex (i,j) with $\alpha(i,j) \le a$ as an element is followed by the set obtained from C by replacing arbitrary such element (i,j) with all children of $\tau(i,j)$, and
- any element (g,h) in the last set satisfies that $\alpha(g,h) > a$.

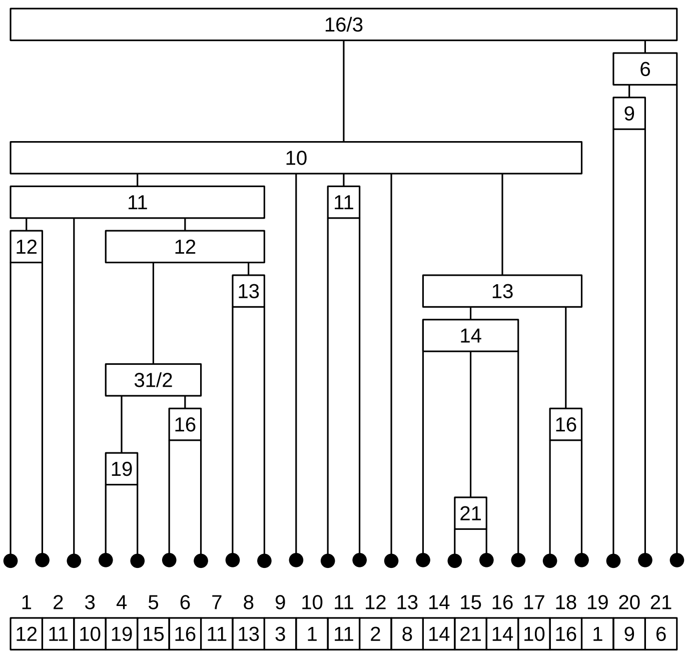


Figure 6 Tree T for the same X_0 as Figure 5 shown at the bottom, drawn in the same manner as Figure 5.

Since $\alpha(g,h) > \alpha(i,j)$ for any leaf (g,h) of $\tau(i,j)$, the last set of $\mathcal{C}$ exists and consists of all vertices (g,h) in T such that $\alpha(i,j) \le a < \alpha(g,h)$, where (i,j) is the parent of (g,h). It follows from Lemma 1 that any element (g,h) in the last set of $\mathcal{C}$ is the only MSS of $X_a(g,h)$. Thus, if the last set in $\mathcal{C}$ has an element (g,h) such that any MSS of $X_a(g,h)$ is an MSS of X_a, then the lemma holds. For any index pair (i,j) with $1 \le i \le j \le n$, if $a \ge \alpha(i,j)$, then there exists a leaf (g,h) of $\tau(i,j)$ such that any MSS of $X_a(g,h)$ is an MSS of $X_a(i,j)$. This implies by induction that any set in $\mathcal{C}$ has an element (g,h) such that any MSS of $X_a(g,h)$ is an MSS of X_a. ◀

Lemma 6 provides a set of vertices in T from which we can find an MSS of X_a. However, this candidate set varies with a. Furthermore, even if the same set is given for different real numbers a, the vertex (g,h) that is an MSS of X_a may differ from each other. For example, if we consider X_0 in Figure 6, then Lemma 6 claims that an MSS of $X_{12.4}$ and an MSS of $X_{12.6}$ can be found from the same candidates $(4,6)$, $(8,8)$, and $(14,18)$ (ignoring empty candidates such as $(1,0)$) and we can verify that $(14,18)$ is the only MSS of $X_{12.4}$ while $(4,6)$ is the only MSS of $X_{12.6}$. Instead of adopting the set of candidates suggested by Lemma 6 as is, we can consider a specific set of candidates that is not altered by a. Those candidates are introduced below. Let P denote the set of all lengths p with $0 \le p \le n$ such that T has at least one vertex (i,j) with $j-i+1=p$. For any length p in P, let w_p denote the maximum of $S_0(i,j)$ over all vertices (i,j) in T such that $j-i+1=p$ and let (i_p,j_p) denote an arbitrary such vertex (i,j) in T that achieves w_p. The following lemma claims that these vertices (i_p,j_p) can be thought of as the candidates.

▶ **Lemma 7.** *For any real number a and any length p in P that maximizes $S_a(i_p,j_p)$, (i_p,j_p) is an MSS of X_a.*

Proof. It imediately follows from Lemma 6 that any vertex (g,h) in T that maximizes $S_a(g,h)$ is an MSS of X_a. For any such vertex (g,h), (i_p,j_p) is also an MSS of X_a, where $p=h-g+1$, because $S_a(g,h) = S_0(g,h) - ap \le w_p - ap = S_0(i_p,j_p) - ap = S_a(i_p,j_p)$. ◀

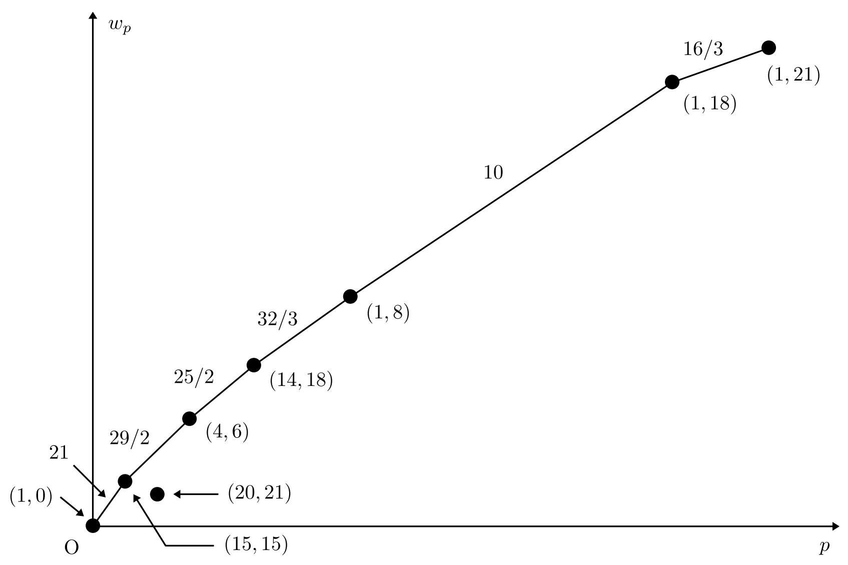


Figure 7 Two-dimensional points (p, w_p) with (i_p, j_p) as the label for all lengths p in P used to determine $OMSS_{X_0}$ as the sequence of $(\infty, (1, 0))$, $(21, (15, 15))$, $(29/2, (4, 6))$, $(25/2, (14, 18))$, $(32/3, (1, 8))$, $(10, (1, 18))$, and $(16/3, (1, 21))$ in this order, where X_0 is the same as Figure 5 and each pair of adjacent vertices (q, w_q) and (p, w_p) of H is connected by a line with its slope $(w_p - w_q)/(p - q)$ as the label.

We are now ready to define $OMSS_{X_0}$ as the offset-MSS data structure we propose. Recall that $OMSS_{X_0}$ is a sequence of $O(n)$ pairs $(\theta, (i, j))$ in descending order of θ such that for any real number a, if $(\theta, (i, j))$ is the last element with $\theta > a$, then (i, j) is an MSS of X_a. As (i, j) of each such element $(\theta, (i, j))$, we adopt (i_p, j_p) for an appropriate length p in P. To find such lengths p, we treat (p, w_p) for any length p in P as a two-dimensional point and consider the upper convex hull H for all the points (p, w_p) (see Figure 7). Note that $(0, 0)$ is a vertex of H. We define $OMSS_{X_0}$ as consisting of pairs $(\theta_p, (i_p, j_p))$ for all vertices (p, w_p) of H. As threshold θ_p, we adopt ∞, if $p = 0$, or $(w_p - w_q)/(p - q)$, otherwise, where (q, w_q) is the vertex of H that is adjacent to (p, w_p) such that $q < p$. The following theorem shows the correctness of $OMSS_{X_0}$ as an offset-MSS data structure.

▶ **Theorem 8.** *For any real number a, (i_p, j_p) is an MSS of X_a, where $(\theta_p, (i_p, j_p))$ is the last element in $OMSS_{X_0}$ such that $\theta_p > a$.*

Proof. Let q be arbitrary length in P such that $q < p$. Since (p, w_p) is a vertex of H, the slope $(w_p - w_q)/(p - q)$ of the straight line passing through (q, w_q) and (p, w_p) is greater than a. This implies that $S_a(i_q, j_q) = w_q - aq < w_p - ap = S_a(i_p, j_p)$. Analogously, for any length q in P such that $q > p$, $S_a(i_q, j_q) \leq S_a(i_p, j_p)$ because $(w_q - w_p)/(q - p) \leq a$. Thus, the theorem follows from Lemma 7. ◀

Algorithm findOMSS can be designed according to definition of $OMSS_{X_0}$. Algorithm 3 presents a pseudo-code of the algorithm, excluding the data structure proposed in Section 3.1 to support $O(\log^2 n)$-time queries of $\alpha(i, j)$ and $\kappa(i, j)$. The algorithm consists of the following two phases.

The first phase. Lines 1 through 13 determine pairs (i_p, j_p) for all lengths p in P by enumerating all internal vertices (i, j) in T. During the enumeration, element W[p] of array W for any length p with $1 \leq p \leq n$ is used to store the maximum of $S_0(i, j)$ over all vertices

Algorithm 3 A pseudo-code of Algorithm findOMSS.

```
1  W, I, J ← arrays of $n$ elements each initialized to 0;
2  T ← a stack containing a single element $(1, n)$;
3  while T is non-empty do
4      pop $(i, j)$ from T;
5      if W$[p] < w$, where $p = j - i + 1$ and $w = S_0(i, j)$, then
6          W$[p] \leftarrow w$; I$[p] \leftarrow i$; J$[p] \leftarrow j$
7      tau ← a stack containing a single element $(i, j)$;
8      while tau is non-empty do
9          pop $(g, h)$ from tau;
10         if $\alpha(g, h) \leq \alpha(i, j)$ then
11             push $(g, \kappa(g, h) - 1)$ to tau; push $(\kappa(g, h) + 1, h)$ to tau
12         else if $g \leq h$ then
13             push $(g, h)$ to T
14 H ← a sequence consisting of a single element $(0, 0)$;
15 foreach $p$ from 1 to $n$ do
16     if W$[p] > 0$ then
17         while H has more than one element and (W$[p] - w_q)/(p - q) \geq$ (W$[p] - w_r)/(p - r)$,
             where $(q, w_q)$ and $(r, w_r)$ are the last and second last elements in H, respectively, do
18             delete the last element $(q, w_q)$ from H
19         append $(p,$ W$[p])$ to H
20 OMSS ← a sequence consisting of a single element $(\infty, (1, 0))$;
21 foreach $(p, w_p)$ in H from the second element to the last in this order do
22     append $((w_p - w_q)/(p - q),$ (I$[p]$, J$[p]$)) to OMSS, where $(q, w_q)$ is the element
         immediately followed by $(p, w_p)$ in H
23 output OMSS as $OMSS_{X_0}$
```

(i, j) with $j - i + 1 = p$ in T enumerated up to the present time, if any, or 0, otherwise. In addition, elements I$[p]$ of array I and J$[p]$ of array J are used to indicate the first vertex (i, j) achieving W$[p]$. Therefore, after executing this phase, for any length p with $1 \leq p \leq n$, if p is an element of P, then W$[p]$, I$[p]$, and J$[p]$ represent w_p, i_p, and j_p, respectively; otherwise W$[p] = 0$. To enumerate all internal vertices in T, two stacks T and tau are used. Since there is no need to maintain the topology of T, T is used to store all internal vertices of T already found but whose children are not yet determined. After initializing T to a stack containing $(1, n)$ as the only element (by line 2), each such internal vertex (i, j) of T is popped from T (by line 4), treated as a new vertex found to update W, I, and J (by lines 5 and 6), and decomposed into its children (by lines 7 through 13). To decompose (i, j) into its children, tau is used to store all vertices of $\tau(i, j)$ that have found but not yet been determined to be leaves or not. After initializing tau to a stack containing (i, j) as the only element (by line 7), each element (g, h) is popped from tau (by line 9) and if (g, h) is not a leaf of $\tau(i, j)$, then its children are pushed to tau (by lines 10 and 11); otherwise, if (g, h) is an internal node of T, then (g, h) is pushed to T (by lines 12 and 13). For any internal vertex (i, j) of T and any internal vertex (g, h) of $\tau(i, j)$, there exists a distinct index k with $1 \leq k \leq n$ that corresponds to (g, h). Thus, it follows from Lemma 4 that this phase is executed in $O(n \log^2 n)$ time.

The second phase. Lines 14 through 23 determine H to construct $OMSS_{X_0}$. For any length p with $1 \leq p \leq n$, let H_p be the upper convex hull for points (q, w_p) for all lengths q in P such that $q \leq p$, so that $H_n = H$. The sequence of all vertices (q, w) of H_p in ascending order of q for each length p from 0 to n is inductively constructed as sequence H. After initializing H to a sequence consisting of $(0, w_0)$ as the only element (by line 14), for any length p in P other than 0 in ascending order, each element (q, w_q) in H that is not a vertex of H_p is removed one by one in descending order of q (by lines 17 and 18) and (p, w_p) is appended to H (by line 19). Since the sequence of all vertices (p, w) of H is eventually obtained as H, $OMSS_{X_0}$ is constructed as sequence OMSS in a straightforward manner (by lines 20 through 22). To update H from H_0 to H_n, $(p, \mathsf{W}[p])$ for any index p in P is appended to H exactly once and any such element is deleted from H at most once. Therefore, this phase is executed in $O(n)$ time.

Due to the above, together with Lemma 3, we immediately have the following theorem.

▶ **Theorem 9.** *Algorithm* findOMSS*, including the data structure supporting $O(\log^2 n)$-time queries of $\alpha(i, j)$ and $\kappa(i, j)$ proposed in Section 3.1, outputs $OMSS_{X_0}$ as an $O(n)$-space offset-MSS data structure for X_0 supporting $O(\log n)$-time queries in $O(n \log^2 n)$ time and $O(n)$ space.*

4 Conclusive remarks

The present article considered the offset maximum-sum segment problem, a variant of the maximum-sum segment problem for a sequence X_0 of n real numbers, which asks an arbitrary contiguous subsequence of X_a that maximizes the sum of its elements for any given real number a, where X_a is the sequence obtained by subtracting a from each element in X_0. An $O(n \log^2 n)$-time, $O(n)$-space algorithm that outputs a data structure supporting $O(\log n)$-time queries of the solution of the offset maximum-sum segment problem was proposed. Further improvements in query time would be unlikely. This is because the data structure output by the proposed algorithm partitions the entire set of real numbers into $O(n)$ intervals, and the offset maximum-sum problem has a distinct solution in common for all real numbers a in each interval. It remains to be clarified whether or not the upper bound on the time complexity of finding such a data structure can be improved from $O(n \log^2 n)$.

References

1 Fredrik Bengtsson and Jingsen Chen. *Computing maximum-scoring segments optimally*. Luleå tekniska universitet, 2007.

2 Jon Bentley. Programming pearls: algorithm design techniques. *Communications of the ACM*, 27(9):865–871, 1984.

3 Kuan-Yu Chen and Kun-Mao Chao. On the range maximum-sum segment query problem. *Discrete Applied Mathematics*, 155(16):2043–2052, 2007. `doi:10.1016/j.dam.2007.05.018`.

4 Chih-Huai Cheng, Hsiao-Fei Liu, and Kun-Mao Chao. Optimal algorithms for the average-constrained maximum-sum segment problem. *Information Processing Letters*, 109(3):171–174, 2009. `doi:10.1016/j.ipl.2008.09.024`.

5 Kai-Min Chung and Hsueh-I Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2005. `doi:10.1137/S0097539704440430`.

6 Margaret Dayhoff, R Schwartz, and B Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5:345–352, 1978.

7 Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Transactions on Database Systems (TODS)*, 26(2):179–213, 2001. `doi:10.1145/383891.383893`.

8 Michael H Goldwasser, Ming-Yang Kao, and Hsueh-I Lu. Fast algorithms for finding maximum-density segments of a sequence with applications to bioinformatics. In *Algorithms in Bioinformatics: Second International Workshop, WABI 2002 Rome, Italy, September 17–21, 2002 Proceedings 2*, pages 157–171. Springer, 2002. doi:10.1007/3-540-45784-4_12.

9 Steven Henikoff and Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992. doi:10.1073/pnas.89.22.10915.

10 Sung Kwon Kim. Linear-time algorithm for finding a maximum-density segment of a sequence. *Information Processing Letters*, 86(6):339–342, 2003. doi:10.1016/S0020-0190(03)00225-4.

11 Yaw-Ling Lin, Tao Jiang, and Kun-Mao Chao. Efficient algorithms for locating the length-constrained heaviest segments with applications to biomolecular sequence analysis. *Journal of Computer and System Sciences*, 65(3):570–586, 2002. doi:10.1016/S0022-0000(02)00010-7.

12 Kalyan Perumalla and Narsingh Deo. Parallel algorithms for maximum subsequence and maximum subarray. *Parallel Processing Letters*, 5(03):367–373, 1995. doi:10.1142/S0129626495000345.

13 Walter L Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *ISMB*, volume 99, pages 234–241, 1999. URL: http://www.aaai.org/Library/ISMB/1999/ismb99-027.php.

14 Yoshifumi Sakai. A maximal local maximum-sum segment data structure. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 101(9):1541–1542, 2018. doi:10.1587/transfun.E101.A.1541.

15 Nikola Stojanovic, Liliana Florea, Cathy Riemer, Deborah Gumucio, Jerry Slightom, Morris Goodman, Webb Miller, and Ross Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 1999. doi:10.1093/nar/27.19.3899.

16 Lusheng Wang and Ying Xu. Segid: Identifying interesting segments in (multiple) sequence alignments. *Bioinformatics*, 19(2):297–298, 2003. doi:10.1093/bioinformatics/19.2.297.

17 Hung-I Yu, Tien-Ching Lin, and DT Lee. Finding maximum sum segments in sequences with uncertainty. *Theoretical Computer Science*, 850:221–235, 2021. doi:10.1016/j.tcs.2020.11.005.

Finding Diverse Strings and Longest Common Subsequences in a Graph

Yuto Shida
Hokkaido University, Japan

Giulia Punzi
National Institute of Informatics, Tokyo, Japan

Yasuaki Kobayashi
Hokkaido University, Japan

Takeaki Uno
National Institute of Informatics, Tokyo, Japan

Hiroki Arimura
Hokkaido University, Japan

Abstract

In this paper, we study for the first time the *Diverse Longest Common Subsequences* (LCSs) problem under Hamming distance. Given a set of a constant number of input strings, the problem asks to decide if there exists some subset $\mathcal{X}$ of K longest common subsequences whose *diversity* is no less than a specified threshold Δ, where we consider two types of diversities of a set $\mathcal{X}$ of strings of equal length: the *Sum diversity* and the *Min diversity* defined as the sum and the minimum of the pairwise Hamming distance between any two strings in $\mathcal{X}$, respectively. We analyze the computational complexity of the respective problems with Sum- and Min-diversity measures, called the *Max-Sum* and *Max-Min Diverse LCSs*, respectively, considering both approximation algorithms and parameterized complexity. Our results are summarized as follows. When K is bounded, both problems are polynomial time solvable. In contrast, when K is unbounded, both problems become NP-hard, while Max-Sum Diverse LCSs problem admits a PTAS. Furthermore, we analyze the parameterized complexity of both problems with combinations of parameters K and r, where r is the length of the candidate strings to be selected. Importantly, all *positive results* above are proven in a more general setting, where an input is an edge-labeled directed acyclic graph (DAG) that succinctly represents a set of strings of the same length. *Negative results* are proven in the setting where an input is explicitly given as a set of strings. The latter results are equipped with an encoding such a set as the longest common subsequences of a specific input string set.

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

Keywords and phrases Sequence analysis, longest common subsequence, Hamming distance, dispersion, approximation algorithms, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.27

Related Version *Technical Report*: `https://arxiv.org/abs/2405.00131`

Funding *Giulia Punzi*: Work supported by JSPS KAKENHI Grant Number JP20H05962.
Yasuaki Kobayashi: Supported by JSPS KAKENHI Grant Numbers JP20H00595, JP23K28034, JP24H00686, and JP24H00697.
Takeaki Uno: Supported by JSPS KAKENHI Grant Number JP20H05962.
Hiroki Arimura: Supported by JSPS KAKENHI Grant Numbers JP20H00595, JP20H05963, and JST CREST Grant Number JPMJCR18K2.

Acknowledgements The authors express sincere thanks to anonymous reviewers for their valuable comments, which significantly improved the presentation and quality of this paper. The last author would like to thank Norihito Yasuda, Tesshu Hanaka, Kazuhiro Kurita, Hirotaka Ono of AFSA project, and Shinji Ito for fruitful discussions and helpful comments.

1 Introduction

The problem of finding a *longest common subsequence* (LCS) of a set of m strings, called the LCS problem, is a fundamental problem in computer science, extensively studied in theory and applications for over fifty years [8,31,33,38,41]. In application areas such as computational biology, pattern recognition, and data compression, longest common subsequences are used for consensus pattern discovery and multiple sequence alignment [25,41]. It is also common to use the length of longest common subsequence as a similarity measure between two strings. For example, Table 1 shows longest common subsequences (underlined) of the input strings $X_1 = ABABCDDEE$ and $Y_1 = ABCBAEEDD$.

Table 1 Longest common subsequences of two input strings X_1 and Y_1 over $\Sigma = \{A, B, C, D, E\}$.

ϵ, A, B, C, D, E, AA, AB, AC, AD, AE, BA, $\ldots$, CD, CE, DD, EE,
ABA, ABB, ABC, ABD, $\ldots$, CEE, $ABAD$, $ABAE$, $ABBD$, $\ldots$, $BCEE$,
$\underline{ABADD}$, $\underline{ABAEE}$, $\underline{ABBDD}$, $\underline{ABBEE}$, $\underline{ABCDD}$, $\underline{ABCEE}$

The LCS problem can be solved in polynomial time for constant $m \geqslant 2$ using dynamic programming by Irving and Fraser [33] requiring $O(n^m)$ time, where n is the maximum length of m strings. When m is unrestricted, LCS is NP-complete [38]. From the view of parameterized complexity, Bodlaender, Downey, Fellows, and Wareham [8] showed that the problem is W[t]-hard parameterized with m for all t, is W[2]-hard parameterized with the length ℓ of a longest common subsequence, and is W[1]-complete parameterized with ℓ and m. Bulteau, Jones, Niedermeier, and Tantau [9] presented a *fixed-paraemter tractable* (FPT) algorithm with different parameterization.

Recent years have seen increasing interest in efficient methods for finding a *diverse set of solutions* [5,20,27,39]. Formally, let $(\mathcal{F}, d)$ be a distance space with a set $\mathcal{F}$ of feasible solutions and a distance $d : \mathcal{F} \times \mathcal{F} \to \mathbb{R}_{\geqslant 0}$, where $d(X, Y)$ denotes the distance between two solutions $X, Y \in \mathcal{F}$. We consider two diversity measures for a subset $\mathcal{X} = \{X_1, \ldots, X_K\} \subseteq \mathcal{F}$ of solutions:

$$D_d^{\mathrm{sum}}(\mathcal{X}) := \textstyle\sum_{i<j} d(X_i, X_j), \qquad \text{(Sum diversity)}, \qquad (1)$$

$$D_d^{\mathrm{min}}(\mathcal{X}) := \min_{i<j} d(X_i, X_j), \qquad \text{(Min diversity)}. \qquad (2)$$

For $\tau \in \{\mathrm{sum}, \mathrm{min}\}$, a subset $\mathcal{X} \subseteq \mathcal{F}$ of feasible solutions is said to be Δ-*diverse* w.r.t. D_d^τ (or simply, *diverse*) if $D_d^\tau(\mathcal{X}) \geqslant \Delta$ for a given $\Delta \geqslant 0$. Generally, the Max-Sum (resp. Max-Min) Diverse Solutions problem related to a combinatorial optimization problem Π is the problem of, given an input I to Π and a nonnegative number $\Delta \geqslant 0$, deciding if there exists a subset $\mathcal{X} \subseteq \mathrm{Sol}_\Pi(I)$ of K solutions on I such that $D_d^{\mathrm{sum}}(\mathcal{X}) \geqslant \Delta$ (resp. $D_d^{\mathrm{min}}(\mathcal{X}) \geqslant \Delta$), where $\mathrm{Sol}_\Pi(I) \subseteq \mathcal{F}$ is the set of solutions on I. For many distance spaces related to combinatorial optimization problems, both problems are known to be computationally hard with unbounded K [5,6,11,18,20,27–29,34,45].

In this paper, we consider the problem of finding a diverse set of solutions for *longest common subsequences* of a set $\mathcal{S}$ of input strings under Hamming distance. The task is to select K longest common subsequences, maximizing the minimum pairwise Hamming distance among them. In general, a set of m strings of length n may have exponentially many longest common subsequences in n. Hence, efficiently finding such a diverse subset of solutions for longest common subsequences is challenging.

Let $d_H(X,Y)$ denote the Hamming distance between two strings $X, Y \in \Sigma^r$ of the equal length $r \geqslant 0$, called *r-strings*. Throughout this paper, we consider two diversity measures over sets of equi-length strings, the Sum-diversity $D^{\text{sum}}_{d_H}$ and the Min-diversity $D^{\text{min}}_{d_H}$ under the Hamming distance d_H. Let $LCS(\mathcal{S})$ denotes the *set of all longest common subsequences* of a set $\mathcal{S}$ of strings. Now, we state our first problem.

▶ **Problem 1** (Diverse LCSs with Diversity Measure $D^{\tau}_{d_H}$).
Input: Integers $K, r \geqslant 1$, and $\Delta \geqslant 0$, and a set $\mathcal{S} = \{S_1, \ldots, S_m\}$ of $m \geqslant 2$ strings over Σ of length at most r ;
Question: Is there some set $\mathcal{X} \subseteq LCS(\mathcal{S})$ of longest common subsequences of $\mathcal{S}$ such that $|\mathcal{X}| = K$ and $D^{\tau}_{d_H}(\mathcal{X}) \geqslant \Delta$?

Then, we analyze the computational complexity of Diverse LCSs from the viewpoints of approximation algorithms [43] and parameterized complexity [15, 22]. For proving positive results for the case that K is bounded, actually, we work with a more general setting in which a set of strings to select is implicitly represented by the language $L(G)$ accepted by an edge-labeled DAG G, called a Σ-DAG. This is motivated by the fact implicit within the well-known algorithm for K-LCSs by Irving and Fraser [33] that the set $LCS(\mathcal{S})$ can be succinctly represented by such a Σ-DAG (see Lemma 2). In contrast, *negative results* will be proven in the setting where an input is explicitly given as a set of strings.

Let $\tau \in \{\text{sum}, \text{min}\}$ be any diversity type. Below, we state the modified version of the problem, where an input string set is an arbitrary set of equi-length strings, no longer a set of LCSs, and it is implicitly represented by either a Σ-DAG G or the set L itself.

▶ **Problem 2** (Diverse String Set with Diversity Measure $D^{\tau}_{d_H}$).
Input: Integers K, r, and Δ, and a Σ-DAG G for a set $L(G) \subseteq \Sigma^r$ of r-strings.
Question: Decide if there exists some subset $\mathcal{X} \subseteq L(G)$ such that $|\mathcal{X}| = K$ and $D^{\tau}_{d_H}(\mathcal{X}) \geqslant \Delta$.

Main results. Let $K \geqslant 1$, $r > 0$, and $\Delta \geqslant 0$ be integers and Σ be an alphabet. The underlying distance is always Hamming distance d_H over r-strings. In Diverse String Set, we assume that an input string set $L \subseteq \Sigma^r$ of r-strings is represented by either a Σ-DAG G or the set L itself. In Diverse LCS, we assume that the number $m = |\mathcal{S}|$ of strings in an input set $\mathcal{S}$ is assumed to be constant throughout. Then, the main results of this paper are summarized as follows.

1. When K is bounded, both Max-Sum and Max-Min versions of Diverse String Set and Diverse LCSs can be solved in polynomial time using dynamic programming (DP). (see Theorem 6, Theorem 8)
2. When K is part of the input, the Max-Sum version of Diverse String Set and Diverse LCSs admit a PTAS by local search showing that the Hamming distance is a *metric of negative type*[1]. (see Theorem 13)
3. Both of the Max-Sum and Max-Min versions of Diverse String Set and Diverse LCSs are fixed-parameter tractable (FPT) when parameterized by K and r (see Theorem 15, Theorem 16). These results are shown by combining Alon, Yuster, and Zwick's *color coding technique* [1] and the DP method in Result 1 above.
4. When K is part of the input, the Max-Sum and Max-Min versions of Diverse String Set and Diverse LCSs are NP-hard for any constant $r \geqslant 3$ (Theorem 17, Corollary 20).
5. When parameterized by K, the Max-Sum and Max-Min versions of Diverse String Set and Diverse LCSs are W[1]-hard (see Theorem 18, Corollary 21).

[1] It is a finite metric satisfying a class of inequalities of negative type [16]. For definition, see Sec. 4.

Table 2 Summary of results on DIVERSE STRING SET and DIVERSE LCSs under Hamming distance, where K, r, and Δ stand for the *number, length*, and *diversity threshold* for a subset $\mathcal{X}$ of r-strings, and α: *const, param*, and *input* indicate that α is a constant, a parameter, and part of an input, respectively. A representation of an input set L is both of Σ-DAG and LCS otherwise stated.

Problem	Type	K: const	K: param	K: input
MAX-SUM DIVERSE STRING SET & LCS	Exact	Poly-Time (Theorem 8)	W[1]-hard on Σ-DAG (Theorem 18))	NP-hard on Σ-DAG if $r \geqslant 3$:const (Theorem 17)
			W[1]-hard on LCS (Corollary 21))	NP-hard on LCS (Corollary 20)
	Approx. or FPT	—	FPT if r: param (Theorem 16)	PTAS (Theorem 13)
MAX-MIN DIVERSE STRING SET & LCS	Exact	Poly-Time (Theorem 6)	W[1]-hard on Σ-DAG (Theorem 18)	NP-hard on Σ-DAG if $r \geqslant 3$:const (Theorem 17)
			W[1]-hard on LCS (Corollary 21)	NP-hard on LCS (Corollary 20)
	Approx. or FPT	—	FPT if r: param (Theorem 15)	Open

A summary of these results is presented in Table 2. We remark that the DIVERSE STRING SET problem coincides the original LCS problem when $K = 1$. It is generally believed that a W[1]-hard problem is unlikely to be FPT [17,22]. Future work includes the approximability of the *Max-Min* version of both problems for unbounded K, and extending our results to other distances and metrics over strings, e.g., *edit distance* [35,44] and *normalized edit distance* [21].

1.1 Related work

Diversity maximization for point sets in metric space and graphs has been studied since 1970s under various names in the literature [7,10,11,18,29,34,40,42] (see Ravi, Rosenkrantz, and Tayi [40] and Chandra and Halldórsson [11] for overview). There are two major versions: MAX-MIN version is known as *remote-edge*, *p-Dispersion*, and *Max-Min Facility Dispersion* [18,42,45]; MAX-SUM version is known as *remote-clique*, *Maxisum Dispersion*, and *Max-Average Facility Dispersion* [7,10,29,40]. Both problems are shown to be NP-hard with unbounded K for general distance and metrics (with triangle inequality) [18,29], while they are polynomial time solvable for 1- and 2-dimensional ℓ_2-spaces [45]. It is trivially solvable in $n^{O(k)}$ time for bounded K.

Diversity maximization in combinatorial problems. However, extending these results for finding diverse solutions to combinatorial problems is challenging [5,20]. While methods such as *random sampling*, *enumeration*, and *top-K optimization* are commonly used for increasing the diversity of solution sets in optimization, they lack theoretical guarantee of the diversity [5,6,20,27]. In this direction, Baste, Fellows, Jaffke, Masarík, de Oliveira Oliveira, Philip, and Rosamond [5,6] pioneered the study of finding diverse solutions in combinatorial problems, investigating the parameterized complexity of well-know graph problems such as *Vertex Cover* [6]. Subsequently, Hanaka, Kiyomi, Kobayashi, Kobayashi, Kurita, and Otachi [28] explored the fixed-parameter tractability of finding various *subgraphs*.

They further proposed a framework for *approximating* diverse solutions, leading to efficient approximation algorithms for diverse matchings, and diverse minimum cuts [27]. While previous work has focused on diverse solutions in graphs and set families, the complexity of finding diverse solutions in *string problems* remains unexplored. Arrighi, Fernau, de Oliveira Oliveira, and Wolf [2] conducted one of the first studies in this direction, investigating a problem of finding a diverse set of subsequence-minimal synchronizing words.

DAG-based representation for all LCSs have appeared from time to time in the literature. The LCS algorithm by Irving and Fraser [33] for more than two strings can be seen as DP on a grid DAG for LCSs. Lu and Lin's parallel algorithm [37] for LCS used a similar grid DAG. Hakata and Imai [26] presented a faster algorithm based on a DAG of *dominant matches*. Conte, Grossi, Punzi, and Uno [12] and Hirota and Sakai [30] independently proposed DAGs of maximal common subsequences of two strings for enumeration.

The relationship between Hamming distance and other metrics has been explored in string and geometric algorithms. Lipsky and Porat [36] presented linear-time reductions from String Matching problems under Hamming distance to equivalent problems under ℓ_1-metric. Gionis, Indyk, and Motwani [24] used an *isometry* (a distance preserving mapping) from an ℓ_1-metric to Hamming distance over binary strings with a polynomial increase in dimension. Cormode and Muthukrishnan [14] showed an efficient ℓ_1-embedding of edit distance allowing moves over strings into ℓ_1-metric with small distortion. Despite these advancements, existing techniques haven't been successfully applied to our problems.

2 Preliminaries

We denote by $\mathbb{Z}$, $\mathbb{N} = \{\, x \in \mathbb{Z} \mid x \geqslant 0 \,\}$, $\mathbb{R}$, and $\mathbb{R}_{\geqslant 0} = \{\, x \in \mathbb{R} \mid x \geqslant 0 \,\}$ the sets of *all integers*, *all non-negative integers*, *all real numbers*, and *all non-negative real numbers*, respectively. For any $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. Let A be any set. Then, $|A|$ denotes the *cardinality* of A. Throughout, our model of computation is the word RAM, where the space is measured in $\Theta(\log n)$-bit machine words.

Let Σ be an *alphabet* of σ symbols. For any $n \geqslant 0$, Σ^n and Σ^* denote the sets of all strings of length n and all finite strings over Σ, respectively. Let $X = a_1 \ldots a_n \in \Sigma^n$ be any string. Then, the *length* of X is denoted by $|X| = n$. For any $1 \leqslant i, j \leqslant n$, $X[i..j]$ denotes the substring $a_i \ldots a_j$ if $i \leqslant j$ and the *empty string* ε otherwise. A *string set* or a *language* is a set $L = \{X_1, \ldots, X_n\} \subseteq \Sigma^*$ of $n \geqslant 0$ strings over Σ. The *total length* of a string set L is denoted by $||L|| = \sum_{X \in L} |X|$, while the length of the longest strings in L is denoted by $\mathrm{maxlen}(L) := \max_{S \in L} |S|$. For any $r \geqslant 0$, we call any string X an *r-string* if its length is r, i.e., $X \in \Sigma^r$. Any string set L is said to be of *equi-length* if $L \subseteq \Sigma^r$ for some $r \geqslant 0$.

2.1 Σ-DAGs

A Σ*-labeled directed acyclic graph* (Σ*-DAG*, for short) is an edge-labeled directed acyclic graph (DAG) $G = (V, E, s, t)$ satisfying: (i) V is a set of vertices; (ii) $E \subseteq V \times \Sigma \times V$ is a set of labeled directed edges, where each edge $e = (v, c, w)$ in E is labeled with a symbol $c = \mathrm{lab}(e)$ taken from Σ; (iii) G has the unique *source* s and *sink* t in V such that every vertex lies on a path from s to t. We define the *size* of G as the number $\mathrm{size}(G)$ of its labeled edges. From (iii) above, G contains no unreachable nodes. For any vertex v in V, we denote the *set of its outgoing edges* by $E^+(v) = \{\, (v, c, w) \in E \mid w \in V \,\}$. Any path $P = (e_1, \ldots, e_n) \in E^n$ of length n *spells out* a string $\mathrm{str}(P) = \mathrm{lab}(e_1) \cdots \mathrm{lab}(e_n) \in \Sigma^n$ of length n, where $n \geqslant 0$. A Σ-DAG G *represents* the string set, or language, denoted $L(G) \subseteq \Sigma^*$, as the collection of all strings spelled out by its (s, t)-paths. Essentially, G is equivalent to a non-deterministic finite automaton (NFA) [32] over Σ with initial and final states s and t, and without ε-edges.

Fig. 1a shows an example of Σ-DAG representing the set of six longest common subsequences of two strings in Table 1. Sometimes, a Σ-DAG can succinctly represent a string set by its language $L(G)$. Actually, the size of G can be logarithmic in $|L(G)|$ in the best case,[2] while size(G) can be arbitrary larger than $||L(G)||$ (see Lemma 14 in Sec. 5).

▶ Remark 1. For any set L of strings over Σ, the following properties hold: (1) there exists a Σ-DAG G such that $L(G) = L$ and size$(G) \leqslant ||L||$. (2) Moreover, G can be constructed from L in $O(||L||f(\sigma))$ time, where $f(n)$ is the query time of search and insert operation on a dictionary with n elements. (3) Suppose that a Σ-DAG G represents a set of strings $L \subseteq \Sigma^*$. If $L \subseteq \Sigma^r$ for $r \geqslant 0$, then all paths from the source s to any vertex v spell out strings of the same length, say $d \leqslant r$.

Proof. (1) We can construct a *trie* T for a set L of strings over Σ, which is a deterministic finite automaton for recognizing L in the shape of a rooted trees and has at most $O(||L||)$ vertices and edges. By identifying all leaves of T to form the sink, we obtain a Σ-DAG with $||L||$ edges for L. (2) It is not hard to see that the trie T can be built in $O(||L|| \log \sigma)$ time from L. (3) In what follows, we denote the string spelled out by any path π in G by $str(\pi)$. Suppose by contradiction that G has some pair of paths π_1 and $\pi_2 \in E^*$ from s to a vertex v such that $|str(\pi_1)| - |str(\pi_2)| > 0$ (*). By assumption (iii) in the definition of a Σ-DAG, the vertex v is contained in some (s,t)-path in G. Therefore, we have some path θ that connects v to t. By concatenating π_k and θ, we have two (s,t)-paths $\tau_k = \pi_k \cdot \theta$ for all $k = 1, 2$. Then, we observe from claim (*) that $|str(\tau_1)| - |str(\tau_2)| = |str(\pi_1 \cdot \theta)| - |str(\pi_2 \cdot \theta)| = (|str(\pi_1)| + |str(\theta)|) - (|str(\pi_2)| + |str(\theta)|) = |str(\pi_1)| - |str(\pi_2)| > 0$. On the other hand, we have $L(G)$ contains both of $str(\pi_1)$ and $str(\pi_2)$ since τ_1 and τ_2 are (s,t)-paths. This means that $L(G)$ contains two strings of distinct lengths, and this contradicts that $L(G) \subseteq \Sigma^r$ for some $r \geqslant 1$. Hence, all paths from s to v have the same length. Hence, (3) is proved. ◀

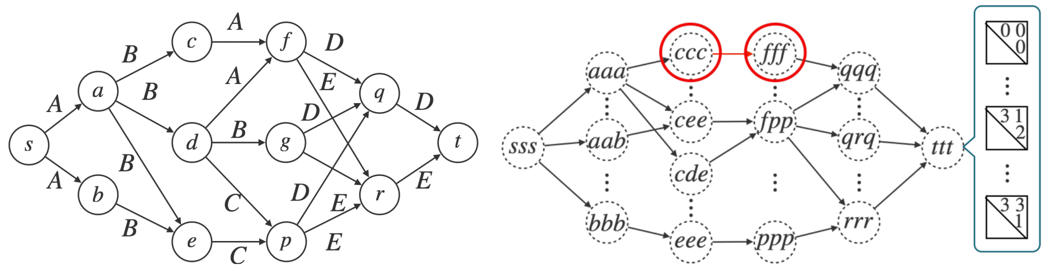
By Property (3) of Remark 1, we define the *depth* of a vertex v in G by the length depth(v) of any path P from the source s to v, called a *length-d prefix (path)*. In other words, depth$(v) = |\mathrm{str}(P)|$. Then, the vertex set V is partitioned into a collection of disjoint subsets $V_0 = \{s\} \cup \cdots \cup V_r = \{t\}$, where V_d is the subset of all vertices with depth d for all $d \in [r] \cup \{0\}$.

2.2 Longest common subsequences

A string X is a *subsequence* of another string Y if X is obtained from Y by removing some characters retaining the order. X is a *common subsequence* (CS) of any set $\mathcal{S} = \{S_1, \ldots, S_m\}$ of m strings if X is a subsequence of any member of $\mathcal{S}$. A CS of $\mathcal{S}$ is called a *longest common subsequence* (LCS) if it has the maximum length among all CSs of $\mathcal{S}$. We denote by $CS(\mathcal{S})$ and $LCS(\mathcal{S})$, respectively, the *sets of all CSs and all LCSs* of $\mathcal{S}$. Naturally, all LCSs in $LCS(\mathcal{S})$ have the same length, denoted by $0 \leqslant lcs(\mathcal{S}) \leqslant \min_{S \in \mathcal{S}} |S|$. While a string set $\mathcal{S}$ can contain exponentially many LCSs compared to the total length $||\mathcal{S}||$ of its strings, we can readily see the next lemma.

▶ **Lemma 2** (Σ-DAG for LCSs). *For any constant $m \geqslant 1$ and any set $\mathcal{S} = \{S_1, \ldots, S_m\} \subseteq \Sigma^*$ of m strings, there exists a Σ-DAG G of polynomial size in $\ell :=$ maxlen$(\mathcal{S})$ such that $L(G) = LCS(\mathcal{S})$, and G can be computed in polynomial time in ℓ.*

[2] For example, for any $r \geqslant 1$, the language $L = \{a, b\}^r$ over an alphabet $\Sigma = \{a, b\}$ consists of $|L| = 2^r$ strings, while it can be represented by a Σ-DAG with $2r$ edges.



(a) An input Σ-DAG G_1. **(b)** An example run of Algorithm 1 for $K = 3$.

Figure 1 Illustration of Algorithm 1 based on dynamic programming. In (a) a Σ-DAG G_1 represents six LCSs in Table 1. In (b), circles and arrows indicate the states of the algorithm, which are K-tuples of vertices of G_1, and transition between them, respectively. All states are associated with a set of $K \times K$-weight matrices, which are shown only for the sink *ttt* in the figure.

Proof. By Irving and Fraser's algorithm [33], we can construct a m-dimensional grid graph N in $O(\ell^m)$ time and space, where (i) the source and sink are $s = (0, \dots, 0)$ and $t = (|S_1|, \dots, |S_m|)$, respectively; (ii) edge labels are symbols from $\Sigma \cup \{\varepsilon\}$; (iii) the number of edges is $\mathrm{size}(N) = \prod_{i=1}^{m} |S_i| \leqslant O(\ell^m)$; and (iv) all of (s,t)-paths spell out $LCS(\mathcal{S})$. Then, application of the ε-removal algorithm [32] yields a Σ-DAG G in $O(|\Sigma| \cdot \mathrm{size}(N))$ time and space, where G has $O(|\Sigma| \cdot \mathrm{size}(N)) = O(|\Sigma|\ell^m)$ edges. ◀

▶ Remark 3. As a direct consequence of Lemma 2, we observe that if Max-Min (resp. Max-Sum) Diverse String Set can be solved in $f(M, K, r, \Delta)$ time and $g(M, K, r, \Delta)$ space on a given input DAG G of size $M = \mathrm{size}(G)$, then Max-Min (resp. Max-Sum) Diverse LCSs on $\mathcal{S} \subseteq \Sigma^r$ can be solvable in $t = O(|\Sigma| \cdot \ell^m + f(\ell^m, K, r, \Delta))$ time and $s = O(\ell^m + g(\ell^m, K, r, \delta))$ space, where $\ell = \mathrm{maxlen}(\mathcal{S})$, since $\mathrm{size}(G) = O(\ell^m)$.

From Remark 3, for any constant $m \geqslant 2$, there exist a polynomial time reduction from Max-Min (resp. Max-Sum) Diverse LCSs for m strings to Max-Min (resp. Max-Sum) Diverse String Set on Σ-DAGs, where the distance measure is Hamming distance.

2.3 Computational complexity

A problem with parameter κ is said to be *fixed-parameter tractable* (FPT) if there is an algorithm that solves it, whose running time on an instance x is upperbounded by $f(\kappa(x)) \cdot |x|^c$ for a computable function $f(\kappa)$ and constant $c > 0$. A many-one reduction ϕ is called an *FPT-reduction* if it can be computed in FPT and the parameter $\kappa(\phi(x))$ is upper-bounded by a computable function of $\kappa(x)$. For notions not defined here, we refer to Ausiello *et al.* [3] for *approximability* and to Flum and Grohe [22] for *parameterized complexity.*

3 Exact Algorithms for Bounded Number of Diverse Strings

In this section, we show that both of Max-Min and Max-Sum versions of Diverse String Set problems can be solved by dynamic programming in polynomial time and space in the size an input Σ-DAG and integers r and Δ for any constant K. The corresponding results for Diverse LCSs problems will immediately follow from Remark 3.

3.1 Computing Max-Min Diverse Solutions

We describe our dynamic programming algorithm for the Max-Min Diverse String Set problem. Given an Σ-DAG $G = (V, E, s, t)$ with n vertices, representing a set $L(G) \subseteq \Sigma^r$ of r-strings, we consider integers $\Delta \geqslant 0, r \geqslant 0$, and constant $K \geqslant 1$. A brute-force approach could solve the problem in $O(|L(G)|^K)$ time by enumerating all combinations of K (s,t)-paths in G and selecting a Δ-diverse solution $\mathcal{X} \subseteq L(G)$. However, this is impractical even for constant K because $|L(G)|$ can be exponential in the number of edges.

The DP-table. A straightforward method to solve the problem is enumerating all combinations of K-tuples of paths from s to t to find the best K-tuple. However, the number of such K-tuples can be exponential in r. Instead, our DP-algorithm keeps track of only *all possible patterns* of their pairwise Hamming distances. Furthermore, it is sufficient to record only Hamming distance up to a given threshold Δ. In this way, we can efficiently computes the best combination of K paths provided that the number of patterns is manageable.

Formally, we let d $(0 \leqslant d \leqslant r)$ be any integer and $\boldsymbol{P} = (P_1, \ldots, P_K) \in (E^d)^K$ be any K-tuple of length-d paths starting from the sink s and ending at some nodes. Then, we define the *pattern* of K-tuple $\boldsymbol{P}$ by the pair $\texttt{Pattern}(\boldsymbol{P}) = (\boldsymbol{w}, Z)$, where

- $\boldsymbol{w} = (w_1, \ldots, w_K) \in V_d^K$ is the K-tuple of vertices in G, called a *state*, such that for all $i \in [K]$, the i-th path P_i starts from the source s and ends at the i-th vertex w_i of $\boldsymbol{w}$.
- $Z = (Z_{i,j})_{i<j} \in [\Delta \cup \{0\}]^{K \times K}$ is an upper triangular matrix, called the *weight matrix* for $\boldsymbol{P}$. For all $1 \leqslant i < j \leqslant K$, $Z_{i,j} = \min(\Delta, d_H(\mathrm{str}(P_i), \mathrm{str}(P_j))) \in [\Delta \cup \{0\}]$ is the Hamming distance between the string labels of P_i and P_j truncated by the threshold Δ.

Our algorithm constructs as the DP-table $\texttt{Weights} = (\texttt{Weights}(\boldsymbol{w}, Z))_{\boldsymbol{w},Z}$, which is a Boolean-valued table that associates a collection of weight matrices Z to each state $\boldsymbol{w}$ such that Z belongs to the collection if and only if $\texttt{Weights}(\boldsymbol{w}, Z) = 1$. See Fig. 1 for example. Formally, we define $\texttt{Weights}$ as follows.

▶ **Definition 4.** $\texttt{Weights} : V^K \times [\Delta \cup \{0\}]^{K \times K} \to \{0, 1\}$ *is a Boolean table such that for every K-tuple of vertices $\boldsymbol{w} \in V^K$ and weight matrix $Z \in [\Delta \cup \{0\}]^{K \times K}$, $\texttt{Weights}(\boldsymbol{w}, Z) = 1$ holds if and only if $(\boldsymbol{w}, Z) = \texttt{Pattern}(\boldsymbol{P})$ holds for some $0 \leqslant d \leqslant r$ and some K-tuple $\boldsymbol{P} \in (E^d)^K$ of length-d paths from the source s to $\boldsymbol{w}$ in G.*

We estimate the size of the table $\texttt{Weights}$. Since Z takes at most $\Gamma = O(\Delta^{K^2} K^2)$ distinct values, it can be encoded in $\log \Gamma = O(K^2 \log \Delta)$ bits. Therefore, $\texttt{Weights}$ has at most $|V|^K \times \Gamma = O(\Delta^{K^2} K^2 M^K)$ entries, where $M = \mathrm{size}(G)$. Consequently, for constant K, $\texttt{Weights}$ can be stored in a multi-dimensional table of polynomial size in M and Δ supporting random access in constant expected time or $O(\log \log(|V| \cdot \Delta))$ worst-case time [13, 46].

Computation of the DP-table. We denote the K-tuples of copies of the source s and sink t by $\boldsymbol{s} := (s, \ldots, s)$ and $\boldsymbol{t} := (t, \ldots, t) \in V^K$, respectively, as the initial and final states. The *zero matrix* $\texttt{Zero} = (\texttt{Zero}_{i,j})_{i<j}$ is a special matrix where $\texttt{Zero}_{i,j} = 0$ for all $i < j$. Now, we present the recurrence for the DP-table $\texttt{Weights}$.

▶ **Lemma 5** (recurrence for $\texttt{Weights}$). *For any $0 \leqslant d \leqslant r$, any $\boldsymbol{w} \in V^K$ and any $Z = (Z_{i,j})_{i<j} \in [\Delta \cup \{0\}]^{K \times K}$, the entry $\texttt{Weights}(\boldsymbol{w}, Z) \in \{0, 1\}$ satisfies the following:*

(1) *Base case: If $\boldsymbol{w} = \boldsymbol{s}$ and $Z = \texttt{Zero}$, then $\texttt{Weights}(\boldsymbol{w}, Z) = 1$.*

(2) *Induction case: If $\boldsymbol{w} \neq \boldsymbol{s}$ and all vertices in $\boldsymbol{w}$ have the same depth d $(1 \leqslant d \leqslant r)$, namely, $\boldsymbol{w} \in V_d^K$, then $\texttt{Weights}(\boldsymbol{w}, z) = 1$ if and only if there exist*

Algorithm 1 An exact algorithm for solving MAX-MIN DIVERSE r-STRING problem. Given a Σ-DAG $G = (V, E, s, t)$ representing a set $L(G)$ of r-strings and integers $K \geqslant 1, \Delta \geqslant 0$, decide if there exists some Δ-diverse set of K r-strings in $L(G)$.

```
1 Set $\texttt{Weights}(\boldsymbol{s}, Z) := 0$ for all $Z \in [\Delta \cup \{0\}]^{K\times K}$, and set $\texttt{Weights}(\boldsymbol{s}, \texttt{Zero}) \leftarrow 1$;
2 for $d \leftarrow 1, \dots, r$ do
3     for $\boldsymbol{v} \leftarrow (v_1, \dots, v_K) \in (V_d)^K$ do
4         for $(v_1, c_1, w_1) \in E^+(v_1), \dots, (v_K, c_K, w_K) \in E^+(v_K)$ do
5             Set $\boldsymbol{w} \leftarrow (w_1, \dots, w_K)$;
6             for $U \in [\Delta \cup \{0\}]^{K\times K}$ such that $\texttt{Weights}(\boldsymbol{v}, U) = 1$ do
7                 Set $Z = (Z_{i,j})_{i<j}$ with $Z_{i,j} \leftarrow \min(\Delta, U_{i,j} + \mathbb{1}\{c_i \neq c_j\}), \forall i, j \in [K]$;
8                 Set $\texttt{Weights}(\boldsymbol{w}, Z) \leftarrow 1$ ;                    ▷ Update
9 Answer YES if $\texttt{Weights}(\boldsymbol{t}, Z) := 1$ and $D^{\min}_{d_H}(Z) \geqslant \Delta$ for some $Z$, and NO otherwise;
```

- $\boldsymbol{v} = (v_i)_{i=1}^K \in V_{d-1}^K$ *such that each* v_i *is a parent of* w_i, *i.e.*, $(v_i, c_i, w_i) \in E$, *and*
- $U = (U_{i,j})_{i<j} \in [\Delta \cup \{0\}]^{K\times K}$ *such that (i)* $\texttt{Weights}(\boldsymbol{v}, U) = 1$, *and (ii)* $Z_{i,j} = \min(\Delta, U_{i,j} + \mathbb{1}\{c_i \neq c_j\})$ *for all* $1 \leqslant i < j \leqslant K$.

(3) *Otherwise:* $\texttt{Weights}(\boldsymbol{w}, Z) = 0$.

Proof. The proof is straightforward by induction on $0 \leqslant d \leqslant r$. Thus, we omit the detail. ◀

Fig. 1b shows an example run of Algorithm 1 on a Σ-DAG G_1 in Fig. 1a representing the string set $L(G_1) = LCS(X_1, Y_1)$, where squares indicate weight matrices. From Lemma 5, we show Theorem 6 on the correctness and time complexity of Algorithm 1.

▶ **Theorem 6** (Polynomial time complexity of Max-Min Diverse String Set). *For any* $K \geqslant 1$ *and* $\Delta \geqslant 0$, *Algorithm 1 solves* MAX-MIN DIVERSE STRING SET *in* $O(\Delta^{K^2} K^2 M^K (\log |V| + \log \Delta))$ *time and space when an input string set* L *is represented by a* Σ*-DAG, where* $M = \text{size}(G)$ *is the number of edges in* G.

3.2 Computing Max-Sum Diverse Solutions

We can solve MAX-SUM DIVERSE STRING SET by modifying Algorithm 1 as follows. For computing the MAX-SUM diversity, we only need to maintain the sum $z = \sum_{i<j} d_H(\text{str}(P_i), \text{str}(P_j))$ of all pairwise Hamming distances instead of the entire $(K \times K)$-weight matrix Z.

The new table $\texttt{Weights}'$. For any $\boldsymbol{w} = (w_1, \dots, w_K)$ of the same depth $0 \leqslant d \leqslant r$ and any integer $0 \leqslant z \leqslant rK$, we define: $\texttt{Weights}'(\boldsymbol{w}, z) = 1$ if and only if there exists a K-tuple of length-d prefix paths $(P_1, \dots, P_K) \in (E^d)^K$ from s to $w_1, \dots, w_K$, respectively, such that the sum of their pairwise Hamming distances is z, namely, $z = \sum_{i<j} d_H(\text{str}(P_i), \text{str}(P_j))$.

▶ **Lemma 7** (recurrence for $\texttt{Weights}'$). *For any* $\boldsymbol{w} = (w_1, \dots, w_K) \in V^K$ *and any integer* $0 \leqslant z \leqslant rK$, *the entry* $\texttt{Weights}'(\boldsymbol{w}, z) \in \{0, 1\}$ *satisfies the following:*

(1) *Base case: If* $\boldsymbol{w} = \boldsymbol{s}$ *and* $z = 0$, *then* $\texttt{Weights}(\boldsymbol{w}, z) = 1$.

(2) *Induction case: If* $\boldsymbol{w} \neq \boldsymbol{s}$ *and all vertices in* $\boldsymbol{w}$ *have the same depth* d $(1 \leqslant d \leqslant r)$, *namely,* $\boldsymbol{w} \in V_d^K$, *then* $\texttt{Weights}(\boldsymbol{w}, z) = 1$ *if and only if there exist*

- $\boldsymbol{v} = (v_i)_{i=1}^K \in V_{d-1}^K$ *such that each* v_i *is a parent of* w_i, *i.e.*, $(v_i, c_i, w_i) \in E$, *and*
- $0 \leqslant u \leqslant rK$ *such that (i)* $\texttt{Weights}(\boldsymbol{v}, u) = 1$, *and (ii)* $z = \min(\Delta, u + \sum_{i<j} \mathbb{1}\{c_i \neq c_j\})$.

(3) *Otherwise:* $\texttt{Weights}(\boldsymbol{w}, z) = 0$.

Algorithm 2 A $(1-2/K)$-approximation algorithm for Max-Sum Diversification for a metric d of negative type on $\mathcal{X}$.

```
1 procedure LocalSearch($\mathcal{D}, K, d$);
2 $\mathcal{X} \leftarrow$ arbitrary $K$ solutions in $\mathcal{D}$;
3 for $i \leftarrow 1, \dots, \lceil \frac{K(K-1)}{(K+1)} \ln \frac{(K+2)(K-1)^2}{4} \rceil$ do
4     for $X \in \mathcal{X}$ such that $\mathcal{D} \setminus \mathcal{X} \neq \emptyset$ do
5         $Y \leftarrow \operatorname{argmax}_{Y \in \mathcal{D} \setminus \mathcal{X}} \sum_{X' \in \mathcal{X} \setminus \{X\}} d(X', Y)$;
6         $\mathcal{X} \leftarrow (\mathcal{X} \setminus \{X\}) \cup \{Y\}$;
7 Output $\mathcal{X}$;
```

From the above modification of Algorithm 1 and Lemma 7, we have Theorem 8. From this theorem, we see that the Max-Sum version of Diverse String Set can be solved faster than the Max-Min version by factor of $O(\Delta^{K-1})$.

▶ **Theorem 8** (Polynomial time complexity of Max-Sum Diverse String Set). *For any constant $K \geqslant 1$, the modified version of Algorithm 1 solves Max-Sum Diverse String Set under Hamming Distance in $O(\Delta K^2 M^K (\log |V| + \log \Delta))$ time and space, where $M = \mathrm{size}(G)$ is the number of edges in G and the input set L is represented by a Σ-DAG.*

4 Approximation Algorithm for Unbounded Number of Diverse Strings

To solve Max-Sum Diverse String Set for unbounded K, we first introduce a local search procedure, proposed by Cevallos, Eisenbrand, and Zenklusen [10], for computing approximate diverse solutions in general finite metric spaces (see [16]). Then, we explain how to apply this algorithm to our problem in the space of equi-length strings equipped with Hamming distance.

Let $\mathcal{D} = \{x_1, \dots, x_n\}$ be a finite set of $n \geqslant 1$ elements. A semi-metric is a function $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}_{\geqslant 0}$ satisfying the following conditions (i)–(iii): (i) $d(x,x) = 0, \forall x \in \mathcal{D}$; (ii) $d(x,y) = d(y,x), \forall x, y \in \mathcal{D}$; (iii) $d(x,z) \leqslant d(x,y) + d(y,z), \forall x,y,z \in \mathcal{D}$ (triangle inequalities). Consider an inequality condition, called a *negative inequality*:

$$\boldsymbol{b}^\top D\, \boldsymbol{b} := \textstyle\sum_{i<j} b_i b_j d(x_i, x_j) \leqslant 0, \qquad \forall \boldsymbol{b} = (b_1, \dots, b_n) \in \mathbb{Z}^n, \tag{3}$$

where $\boldsymbol{b}$ is a column vector and $D = (d_{ij})$ with $d_{ij} = d(x_i, x_j)$. For the vector $\boldsymbol{b}$ above, we define $\sum \boldsymbol{b} := \sum_{i=1}^n b_i$. A semi-metric d is said to be *of negative type* if it satisfies the inequalities Eq. (3) for all $\boldsymbol{b} \in \mathbb{Z}^n$ such that $\sum \boldsymbol{b} = 0$. The class $\mathbb{NEG}$ of semi-metrics of negative type satisfies the following properties.

▶ **Lemma 9** (Deza and Laurent [16]). *For the class $\mathbb{NEG}$, the following properties hold: (1) All ℓ_1-metrics over $\mathbb{R}^r$ are semi-metrics of negative type in $\mathbb{NEG}$ for any $r \geqslant 1$. (2) The class $\mathbb{NEG}$ is closed under linear combination with nonnegative coefficients in $\mathbb{R}_{\geqslant 0}$.*

In Algorithm 2, we show a local search procedure LocalSearch, proposed by Cevallos *et al.* [10], for computing a diverse solution $\mathcal{X} \subseteq \mathcal{D}$ with $|\mathcal{X}| = K$ approximately maximizing its Max-Sum diversity under a given semi-metric $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}_{\geqslant 0}$ on a finite metric space $\mathcal{D}$ of n points. The Farthest Point problem refer to the subproblem for computing Y at Line 5. When the distance d is a semi-metric of *negative type*, they showed the following theorem.

Algorithm 3 An exact algorithm for solving the MAX-SUM FARTHEST r-STRING problem. Given a Σ-DAG G for a set $L(G) \subseteq \Sigma^r$, a set $\mathcal{X} = \{X_1, \ldots, X_K\} \subseteq \Sigma^r$, and an integer $\Delta \geqslant 0$, it decides if there exists a $Y \in L(G)$ such that $D^{\text{sum}}_{d_H}(\mathcal{X}, Y) \geqslant \Delta$.

```
1 Set Weights$(s, z) := 0$ for all $z \in [\Delta]_+$, and Weights$(s, 0) := 1$;
2 for $d := 1, \ldots, r$ do
3     for $v \in V_d$ and $(v, c, w) \in E^+(v)$ do
4         for $0 \leqslant u \leqslant \Delta$ such that Weights$(v, u) := 1$ do
5             Set Weights$(w, z) := 1$ for $z := u + \sum_{i \in [K]} \mathbb{1}\{c \neq X_i[d]\}$ ;    ▷ Update
6 Answer YES if Weights$(t, \Delta) = 1$, and NO otherwise ;    ▷ Decide
```

▶ **Theorem 10** (Cevallos *et al.* [10]). *Suppose that d is a metric of negative type over $\mathcal{X}$ in which the* FARTHEST POINT *problem can be solved in polynomial time. For any $K \geqslant 1$, the procedure* LOCALSEARCH *in Algorithm 2 has approximation ratio $(1 - \frac{2}{K})$.*

We show that the Hamming distance actually has the desired property.

▶ **Lemma 11.** *For any integer $r \geqslant 1$, the Hamming distance d_H over the set Σ^r of r-strings is a semi-metric of negative type over Σ^r.*

Proof. Let $\Sigma = [\sigma]$ be any alphabet. We give an isometry ϕ (see Sec. 1.1) from the Hamming distance (Σ^r, d_H) to the ℓ_1-metric (W, d_{ℓ_1}) over a subset W of $\mathbb{R}^m$ for $m = r\sigma$. For any symbol $i \in \Sigma$, we extend ϕ by $\phi_\Sigma(i) := 0^{i-1}(0.5)0^{\sigma-i} \in \{0, 0.5\}^\sigma$ be a bitvector with 0.5 at i-th position and 0 at other bit positions such that for each $c, c' \in \Sigma$, $||\phi_\Sigma(c) - \phi_\Sigma(c')||_1 = \mathbb{1}\{c \neq c'\}$. For any r-string $S = S[1] \ldots S[r] \in \Sigma^r$, we let $\phi(S) := \phi_\Sigma(S[1]) \cdots \phi_\Sigma(S[r]) \in W$, where $W := \{0, 0.5\}^m$ and $m := r\sigma$. For any $S, S' \in \Sigma^r$, we can show $d_{\ell_1}(\phi(S), \phi(S')) = ||\phi(S)_j - \phi(S')_j||_1 = \sum_{i \in [r]} ||\phi_\Sigma(S[i]) - \phi_\Sigma(S'[i])||_1 = \sum_{i \in [r]} \mathbb{1}\{S[i] \neq S'[i]\} = d_H(S, S')$. Thus, $\phi : \Sigma^r \to W$ is an *isometry* [16] from (Σ^r, d_H) to $(\{0, 0.5\}^m, d_{\ell_1})$. By Lemma 9, ℓ_1-metric is a metric of negative type [10, 16], and thus, the lemma is proved. ◀

The remaining thing is efficiently solving the string version of the subproblem, called the FARTHEST STRING problem, that given a set $\mathcal{X}' \subseteq \Sigma^r$, asks to find the *farthest* Y from all elements in $\mathcal{X}'$ by maximizing the sum $D^{\text{sum}}_{d_H}(\mathcal{X}', Y) := \sum_{X' \in \mathcal{X}'} d_H(X', Y)$ over all elements $Y \in L(G) \setminus \mathcal{X}'$. For the class of r-strings, we show the next lemma.

▶ **Lemma 12** (FARTHEST r-STRING). *For any $K \geqslant 1$ and $\Delta \geqslant 0$, given G and $\mathcal{X}' \subseteq L(G)$, Algorithm 3 computes the farthest r-string $Y \in L(G)$ that maximizes $D^{\text{sum}}_{d_H}(\mathcal{X}', Y)$ over all r-strings in $L(G)$ in $O(K\Delta M)$ time and space, where M is the number of edges in G.*

Proof. The procedure in Algorithm 3 solves the decision version of MAX-SUM FARTHEST r-STRING. Since it is obtained from Algorithm 1 by fixing $K - 1$ paths and searching only a remaining path in G, its correctness and time complexity immediately follows from that of Theorem 6. It is easy to modify Algorithm 3 to compute such Y that maximizes $D^{\text{sum}}_{d_H}(\mathcal{X}', Y)$ by recording the parent pair (v, y) of each (w, z) and then tracing back. ◀

Combining Theorem 10, Lemma 12, and Lemma 11, we obtain the following theorem on the existence of a *polynomial time approximation scheme* (PTAS) [3] for MAX-SUM DIVERSE STRING SET on Σ-DAGs. From Theorem 13 and Remark 3, the corresponding result for MAX-SUM DIVERSE LCSs will immediately follow.

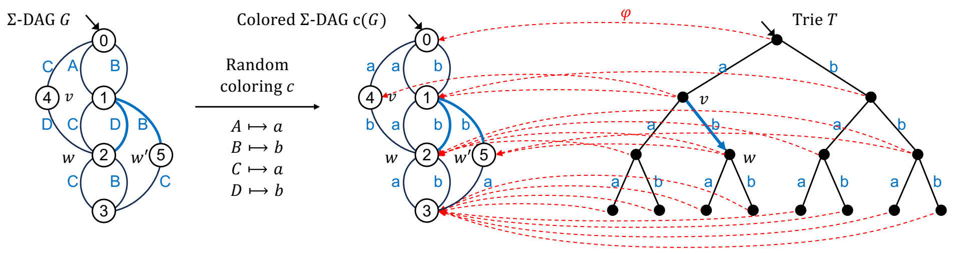


Figure 2 Illustration of the proof for Lemma 14, where dashed lines indicates a correspondence φ.

▶ **Theorem 13** (PTAS for unbounded K). *When K is part of an input,* Max-Sum Diverse String Set *problem on a Σ-DAG G admits PTAS.*

Proof. We show the theorem following the discussion of [10,27]. Let $\varepsilon > 0$ be any constant. Suppose that $\varepsilon < 2/K$ holds. Then, $K < 2/\varepsilon$, and thus, K is a constant. In this case, by Theorem 6, we can exactly solve the problem in polynomial time using Algorithm 1. Otherwise, $2/K \leqslant \varepsilon$. Then, the $(1 - 2/K)$ approximation algorithm in Algorithm 2 equipped with Algorithm 3 achieves factor $1 - \varepsilon$ since d_H is a negative type metric by Lemma 11. Hence, Max-Sum Diverse String Set admits a PTAS. This completes the proof. ◀

5 FPT Algorithms for Bounded Number and Length of Diverse Strings

In this section, we present fixed-parameter tractable (FPT) algorithms for the Max-Min and Max-Sum Diverse String Set parameterized with combinations of K and r. Recall that a problem parameterized with κ is said to be *fixed-parameter tractable* if there exists an algorithm for the problem running on an input x in time $f(\kappa(x)) \cdot |x|^c$ for some computable function $f(\kappa)$ and constant $c > 0$ [22].

For our purpose, we combine the *color-coding technique* by Alon, Yuster, and Zwick [1] and the algorithms in Sec. 3. Consider a random C-coloring $c : \Sigma \to C$ from a set C of $k \geqslant 1$ colors, which assigns a color $c(a)$ chosen from C randomly and independently to each $a \in \Sigma$. By applying this C-coloring to all each edges of an input Σ-DAG G, we obtain the C-colored DAG, called *a C-DAG*, and denote it by $c(G)$. We show a lemma on reduction of $c(G)$.

▶ **Lemma 14** (computing a reduced C-DAG in FPT). *For any set C of k colors, there exists some C-DAG H obtained by reducing $c(G)$ such that $L(H) = L(c(G))$ and $||H|| \leqslant k^r$. Furthermore, such a C-DAG H can be computed from G and C in $t_{\text{pre}} = O(k^r \cdot \text{size}(G))$ time and space.*

Proof. We show a proof sketch. Since $L(G) \subseteq \Sigma^r$, we see that the C-DAG $c(G)$ represents $L(c(G)) \subseteq C^r$ of size at most $||L(c(G))|| \leqslant k^r$. By Remark 1, there exists a C-DAG H for $L(H) = L(c(G))$ with at most k^r edges. However, it is not straightforward how to compute such a succinct H directly from G and c in $O(k^r \cdot \text{size}(G))$ time and space since $||L(G)||$ can be much larger than $k^r + \text{size}(G)$. We build a trie T for $L(H)$ top-down using breadth-first search of G from the source s by maintaining a correspondence $\varphi \subseteq V \times U$ between vertices V in G and vertices U in T (Fig. 2). Then, we identify all leaves of T to make the sink t. This runs in $O(k^r \cdot \text{size}(G))$ time and $O(k^r + \text{size}(G))$ space. ◀

Fig. 2 illustrates computation of reduced C-DAG H from an input Σ-DAG G over alphabet $\Sigma = \{A, B, C, D\}$ in Lemma 14, which shows G (left), a random coloring c on $C = \{a, b\}$, a colored C-DAG $c(G)$ (middle), and a reduced C-DAG H in the form of trie T (right). Combining Lemma 14, Theorem 6, and Alon *et al.* [1], we show the next theorem.

▶ **Theorem 15.** *When r and K are parameters, the* Max-Min Diverse String Set *on a Σ-DAG for r-strings is fixed-parameter tractable (FPT), where* $\mathrm{size}(G)$ *is an input.*

Proof. We show a sketch of the proof. We show a randomized algorithm using Alon *et al.*'s color-coding technique [1]. Let $L(G) \subseteq \Sigma^r, k = rK$, and $C = [rK]$. We assume without loss of generality that $\Delta \leqslant r$. We randomly color edges of G from C. Then, we perform two phases below.

- *Preprocessing phase*: Using the FPT-algorithm of Lemma 14, reduce the colored C-DAG $c(G)$ with $\mathrm{size}(G)$ into another C-DAG H with $L(H) = L(c(G)) \subseteq C^r$ and size bounded by $(rK)^r$. Lemma 14 shows that this requires $t_{\mathrm{pre}} = O((rK)^r \cdot \mathrm{size}(G))$ time and space.
- *Search phase*: Find a Δ-diverse subset $\mathcal{Y}$ in $L(H)$ of size $|\mathcal{Y}| = K$ from H using a modified version of Algorithm 1 in Sec. 3 (details in footnote[3]). If such $\mathcal{Y}$ exists and c is invertible, then $\mathcal{X} = c^{-1}(\mathcal{Y})$ is a Δ-diverse solution for the original problem. The search phase takes $t_{\mathrm{search}} = O(K^2\Delta^{K^2}(rK)^{rK}) =: g(K, r)$ time, where $\Delta \leqslant r$ is used.

With the probability $p = (rK)!/(rK)^{rK} \geqslant 2^{-rK}$, for $C = [rK]$, the random C-coloring yields a colorful Δ-diverse subset $\mathcal{Y} = c(\mathcal{X}) \subseteq L(H)$. Repeating the above process 2^{rK} times and derandomizing it using Alon *et al.* [1] yields an FPT algorithm with total running time $t = 2^{rK} r \log(rK)(t_{\mathrm{pre}} + t_{\mathrm{search}}) = f(K, r, \Delta) \cdot \mathrm{size}(G)$, where $f(K, r, \Delta) = O(2^{rK} r \log(rK) \cdot \{(rK)^r + g(K, r)\})$ depends only on parameters r and K. This completes the proof. ◀

Similarly, we obtain the following result for Max-Sum Diversity.

▶ **Theorem 16.** *When r and K are parameters, the Max-Sum Diverse String Set on Σ-graphs for r-strings is fixed-parameter tractable (FPT), where* $\mathrm{size}(G)$ *is part of an input.*

Proof. The proof proceeds by a similar discussion to the one in the proof of Theorem 15. The only difference is the time complexity of t_{search}. In the case of Max-Sum diversity, the search time of the modified algorithm in Theorem 8 is $t_{\mathrm{search}} = O(\Delta K^2 M^K)$, where $M = \mathrm{size}(G)$. By substituting $M \leqslant (rK)^k$ for t_{search}, we have $t_{\mathrm{search}} = g'(K, r)\Delta$, where $g'(K, r) := O(K^2(rK)^{rK})$. Since $g'(K, r)$ depends only on parameters, the claim follows. ◀

6 Hardness results

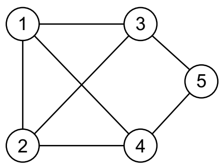
To complement the positive results in Sec. 3 and Sec. 4, we show some negative results in classic and parameterized complexity. In what follows, $\sigma = |\Sigma|$ is an alphabet size, K is the number of strings to select, r is the length of equi-length strings, and Δ is a diversity threshold. In all results below, we assume that σ are constants, and without loss of generality from Remark 1 that an input set L of r-strings is explicitly given as the set itself.

6.1 Hardness of Diverse String Set for Unbounded K

Firstly, we observe the NP-hardness of Max-Min and Max-Sum Diverse String Set holds for unbounded K even for constants $r \geqslant 3$.

▶ **Theorem 17** (NP-hardness for unbounded K). *When K is part of an input,* Max-Min *and* Max-Sum Diverse String Set *on Σ-graphs for r-strings are NP-hard even for any constant $r \geqslant 3$.*

[3] This modification of Algorithm 1 is easily done at Line 7 of Algorithm 1 by replacing the term $\mathbb{1}\{\, lab(e_i) \neq lab(e_j) \,\}$ with the term $\mathbb{1}\{\, \{c(lab(e_i)) \neq c(lab(e_j))\} \wedge \{lab(e_i) \neq lab(e_j)\} \,\}$.



	12	13	14	15	23	24	25	34	35	45
S_1	1	1	1	0	1	1	1	1	1	1
S_2	2	2	2	2	2	2	0	2	2	2
S_3	3	3	3	3	3	3	3	0	3	3
S_4	4	4	4	4	4	4	4	0	4	4
S_5	5	5	5	0	5	5	0	5	5	5

Figure 3 An example of reduction for the proof of Theorem 18 in the case of $n = 5$, consisting of an instance G of Clique, with a vertex set $V = \{1, \dots, 5\}$ and a edge set $E \subseteq \mathcal{E} = \{12, 13, \dots, 45\}$ (left), and the associated instance $F = \{S_1, \dots, S_n\}$ of Diverse r-String Set, where F contains $n = 5$ r-strings with $r = |\mathcal{E}| = 10$ (right). Shadowed cells indicate the occurrences of symbol 0.

Proof. We reduce an NP-hard problem 3DM [23] to Max-Min Diverse String Set by a trivial reduction. Recall that given an instance consists of sets $A = B = C = [n]$ for some $n \geqslant 1$ and a set family $F \subseteq [n]^3$, and 3DM asks if there exists some subset $M \subseteq F$ that is a *matching*, that is, any two vectors $X, Y \in M$ have no position $i \in [3]$ at which the corresponding symbols agree, i.e., $X[i] = Y[i]$. Then, we construct an instance of Max-Min Diverse String Set with $r = 3$ with an alphabet $\Sigma = A \cup B \cup C$, a string set $L = F \subseteq \Sigma^3$, integers $K = n$ and $\Delta = r = 3$. Obviously, this transformation is polynomial time computable. Then, it is not hard to see that for any $M \subseteq F$, M is a matching if and only if $D^{\min}_{d_H}(M) \geqslant \Delta$ holds. On the other hand, for Max-Min Diverse String Set, if we let $\Delta' = \binom{K}{2}$ then for any $M \subseteq F$, M is a matching if and only if $D^{\mathrm{sum}}_{d_H}(M) \geqslant \Delta'$ holds. Combining the above arguments, the theorem is proved. ◀

We remark that 3DM is shown to be in FPT by Fellows, Knauer, Nishimura, Ragde, Rosamond, Stege, Thilikos, and Whitesides [19]. Besides, we showed in Sec. 5 that Diverse r-String Set is FPT when parameterized with $K + r$ (Max-Sum) or $K + r + \Delta$ (Max-Min), respectively. We show that the latter problem is W[1]-hard parameterized with K.

▶ **Theorem 18** (W[1]-hardness of the string set and Σ-DAG versions for unbounded K). *When parameterized with K, Max-Min and Max-Sum Diverse String Set for a set L of r-strings are W[1]-hard whether a string set L is represented by either a string set L or a Σ-DAG for L, where r and Δ are part of an input.*

Proof. We establish the W[1]-hardness of Max-Min Diverse String Set with parameter K by reduction from Clique with parameter K. This builds on the NP-hardness of r-Set packing in Ausiello *et al.* [4] with minor modifications (see also [19]). Given a graph $G = (V, E)$ with n vertices and a parameter $K \in \mathbb{N}$, where $V = [n]$ and $E \subseteq \mathcal{E}$, we let $\mathcal{E} := \{\, \{i, j\} \mid i, j \in V, i \neq j \,\}$. We define the transformation ϕ_1 from $\langle G, K \rangle$ to $\langle \Sigma, r, F, \Delta \rangle$ and $\kappa(K) = K$ as follows. We let $\Sigma = [n] \cup \{0\}$, $r = |\mathcal{E}| = \binom{n}{2}$, and $\Delta = r$. We view each r-string S as a mapping $S : \mathcal{E} \to \Sigma$ assigning symbol $S(e) \in \Sigma$ to each unordered pair $e \in \mathcal{E}$. We construct a family $F = \{\, S_i \mid i \in V \,\}$ of r-strings such that G has a clique of K elements if and only if there exists a subset $M \subseteq F$ with (a) size $|M| \geqslant \kappa(K) = K$, and (b) diversity $d_H(S, S') \geqslant r = \Delta$ for all distinct $S, S' \in M$ (*). Each r-string S_i is defined based on the existence of the edges in E: (i) $S_i(e) = 0$ if $(i \in e) \wedge (e \notin E)$, and (ii) $S_i(e) = i$ otherwise. By definition, $d_H(S_i, S_j) \leqslant r$. We show that for any $i, j \in \mathcal{E}$, S_i and S_j have conflicts at all positions, i.e. $d_H(S_i, S_j) = r$, if and only if $\{i, j\} \in E$. See Fig. 3 for example of F. To see the correctness, suppose that $e = \{i, j\} \notin E$. Then, it follows from (i) that $S_i(e) = S_j(e) = 0$ since $(i \in e) \wedge (j \in e) \wedge e \notin E$. Conversely, if $e' = \{i, j\} \in E$, the condition (i) $S_i(e) = S_j(e) = 0$ does not hold for any $e \in \mathcal{E}$ because if $e \neq e'$, one of $(i \in e)$ and $(j \in e)$ does not hold, and if $e = e'$, $e \notin E$ does not hold. This proves the claim (*). Since ϕ_1 and κ form an FPT-reduction. The theorem is proved. ◀

In this subsection, we show the hardness results of Diverse LCSs for unbounded K in classic and parameterized settings by reducing them to those of Diverse String Set in Sec. 6.1.

▶ **Theorem 19.** *Under Hamming distance, Max-Min (resp. Max-Sum) Diverse String Set for $m \geqslant 2$ strings parameterized with K is FPT-reducible to Max-Min (resp. Max-Sum) Diverse LCSs for two string ($m = 2$) parameterized with K, where m is part of an input. Moreover, the reduction is also a polynomial time reduction from Max-Min (resp. Max-Sum) Diverse String Set to Max-Min (resp. Max-Sum) Diverse LCSs.*

We defer the proof of Theorem 19 in Sec. 6.1.1. Combining Theorem 17, Theorem 18, and Theorem 19, we have the corollaries.

▶ **Corollary 20** (NP-hardness)**.** *When K is part of an input, Max-Min and Max-Sum Diverse LCSs for two r-strings are NP-hard, where r and Δ are part of an input.*

▶ **Corollary 21** (W[1]-hardness)**.** *When parameterized with K, Max-Min and Max-Sum Diverse LCSs for two r-strings are W[1]-hard, where r and Δ are part of an input.*

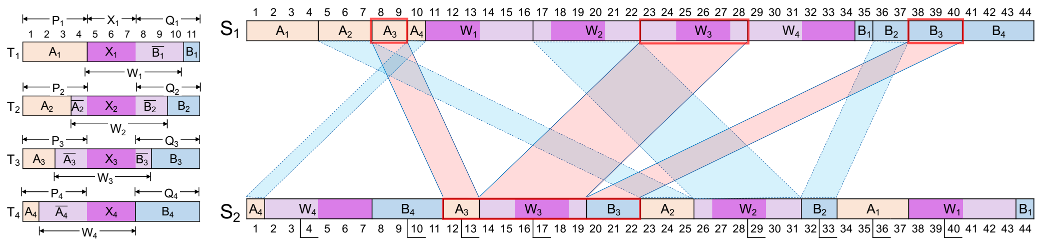
6.1.1 Proof for Theorem 19

In this subsection, we show the proof of Theorem 19, which is deferred in the previous section. Suppose that we are given an instance of Max-Sum Diverse String Set consisting of integers $K, r \geqslant 1$, $\Delta \geqslant 0$, and any set $L = \{X_i\}_{i=1}^s \subseteq \Sigma^r$ of r-strings, where $s = |L| \geqslant 2$. We let $\Xi = \{\, a_{i,j}, b_{i,j} \mid i, j \in [s] \,\}$ be a set of mutually distinct symbols, and $\Gamma = \Sigma \cup \Xi$ be a new alphabet with $\Sigma \cap \Xi = \emptyset$. We let $\mathcal{T} = \{T_i := P_i X_i Q_i\}_{i=1}^s$ be the set of s strings of length $|T_i| = r + 2s$ over Γ, where $P_i := a_{i1} \dots a_{is} \in \Gamma^s, Q_i := b_{i1} \dots b_{is} \in \Gamma^s, \forall i \in [s]$.

Now, we construct two input strings S_1 and S_2 over Γ in an instance of Max-Min Diverse LCSs so that $LCS(\mathcal{S}_1, \mathcal{S}_2) = \mathcal{T}$. For all $i \in [s]$, we factorize each strings T_i of length $(r + 2s)$ into three substrings $A_i, W_i, B_i \in \Gamma^+$, called *segments*, such that $T_i = A_i \cdot W_i \cdot B_i$ such that (i) We partition P_i into $P_i = A_i \cdot \overline{A_i}$, where $A_i := P_i[1..s - i + 1]$ is the prefix with length $s - i + 1$ and $\overline{A_i} = P_i[s - i + 2..s]$ is the suffix with length $i - 1$ of P_i. (ii) We partition Q_i into $Q_i = \overline{B_i} \cdot B_i$, where $\overline{B_i} = Q_i[1..s - i]$ is the prefix with length $s - i$ and $B_i := Q_i[s - i + 1..s]$ is the suffix with length i of Q_i. (iii) We obtain a string $W_i := \overline{A_i} \cdot X_i \cdot \overline{B_i}$ with length $r + s - 1$ from X_i by prepending and appending $\overline{A_i}$ and $\overline{B_i}$ to X_i. Let $\mathcal{A} = \{\, A_i \,\}_{i=1}^s$, $\mathcal{B} = \{\, B_i \,\}_{i=1}^s$, and $\mathcal{W} = \{\, W_i \,\}_{i=1}^s$ be the *groups* of the segments of the *same types*. See Fig. 4a for examples of $\mathcal{A}, \mathcal{B}$, and $\mathcal{W}$. Then, we define the set $\mathcal{S} = \{S_1, S_2\}$ of two input strings S_1 and S_2 of the same length $|S_1| = |S_2| = s(r + 2s)$ by:

$$\mathcal{S}_1 = \textstyle\prod_{i=1}^s A_i \cdot \prod_{i=1}^s W_i \cdot \prod_{i=1}^s B_i = (A_1 \cdots A_s) \cdot (W_1 \cdots W_s) \cdot (B_1 \cdots B_s),$$
$$\mathcal{S}_2 = \textstyle\prod_{i=s}^1 T_i = \prod_{i=s}^1 (A_i \cdot W_i \cdot B_i) = (A_s \cdot W_s \cdot B_s) \cdots (A_1 \cdot W_1 \cdot B_1). \tag{4}$$

Fig. 4b shows an example of $\mathcal{S}$ for $s = 4$. We observe the following properties of $\mathcal{S}$: (P1) S_1 and S_2 are segment-wise permutations of each other; (P2) if all segments in any group $\mathcal{Z} = \{Z_i\}_{i=1}^s \in \{\mathcal{A}, \mathcal{B}, \mathcal{W}\}$ occur one of two input strings, say S_1, in the order $Z_1, \dots, Z_s$, then they occur in the other, say S_2, in the reverse order $Z_s, \dots, Z_1$; (P3) A_i's (resp. B_i's) appear in S_2 from left to right in the order $A_s, \dots, A_1$ (resp. $B_s, \dots, B_1$); (P4) $\mathcal{A}$ and $\mathcal{B}$ satisfy $|A_1| > \cdots > |A_s|$ and $|B_1| < \cdots < |B_s|$. We associate a bipartite graph $\mathcal{B}(\mathcal{S}) = (V = V_1 \cup V_2, E)$ to $\mathcal{S}$, where (i) V_k consists of all positions in S_k for $k = 1, 2$, and (ii) $E \subseteq V_1 \times V_2$ is an edge set such that $e = (i_1, i_2) \in E$ if and only if both ends of e have the same label $S_1[i_1] = S_2[i_2] \in \Sigma$. Any sequence $M = ((i_k, j_k))_{k=1}^{\ell} \in E^{\ell}$ of ℓ edges is an *(ordered) matching* if $i_1 \neq j_1$ and $i_2 \neq j_2$, and is *non-crossing* if $(i_1 < j_1)$ and $(i_2 < j_2)$.


P1 X1 Q1
T1 A1 X1 B1 B1
W1
T2 A2 A2 X2 B2 B2
T3 A3 A3 X3 B3 B3
T4 A4 A4 X4 B4
S1 A1 A2 A3 A4 W1 W2 W3 W4 B1 B2 B3 B4
S2 A4 W4 B4 A3 W3 B3 A2 W2 B2 A1 W1 B1

(a) The set $\mathcal{T}$. **(b)** Input strings S_1 and S_2.

Figure 4 Construction of the FPT-reduction from MAX-MIN DIVERSE STRING SET to MAX-MIN DIVERSE LCS in the proof of Theorem 19, where $s = 4$. We show (a) the set $\mathcal{T}$ of s r-strings and (b) a pair of input strings S_1 and S_2. Red and blue parallelograms, respectively, indicate allowed and prohibited matchings between the copies of blocks $T_3 = A_3W_3B_3$ in S_1 and S_2.

▶ **Lemma 22.** *For any $M \subseteq V_1 \times V_2$ and any $\ell \geqslant 0$, $\mathcal{B}(\mathcal{S})$ has a non-crossing matching M of size ℓ if and only if S_1 and S_2 have a common subsequence C with length ℓ of S_1 and S_2. Moreover, the length $\ell = |M|$ is maximum if and only if $C \in LCS(S_1, S_2)$.*

Proof. If there exists a non-crossing matching $M = \{ (i_k, j_k) \mid k \in [\ell] \} \subseteq E$ of size $\ell \geqslant 0$, we can order the edges in the increasing order such that $i_{\pi(1)} < \cdots < i_{\pi(\ell)}, j_{\pi(1)} < \cdots < j_{\pi(\ell)}$ for some permutation π on $[\ell]$. Then, the string $S_1(M) := S_1[i_{\pi(1)}] \cdots S_1[i_{\pi(\ell)}] \in \Sigma^\ell$ (equivalently, $S_2(M) := S_2[j_1] \cdots S_2[j_\ell]$) forms the common subsequence associated to M. ◀

In Lemma 22, we call a non-crossing ordered matching M associated with a common subsequence C a *matching labeled with* C. We show the next lemma.

▶ **Lemma 23.** *$LCS(S_1, S_2) = \{ T_j \mid i \in [s] \}$, where $T_j = P_j \cdot X_j \cdot Q_j$ for all $j \in [s]$.*

Proof. We first observe that each segment $Z \in \Sigma^+$ in each group $\mathcal{Z}$ within $\{\mathcal{A}, \mathcal{B}, \mathcal{W}\}$ occurs exactly once in each of S_1 and S_2, respectively, as a consecutive substring. Consequently, For each Z in $\mathcal{Z}$, $\mathcal{B}(\mathcal{S})$ has exactly one non-crossing matching M_Z *labeled with Z connecting* the occurrences of Z in S_1 and S_2. From (P2), we show the next claim.

▷ Claim 24. If $\mathcal{B}(\mathcal{S})$ contains any inclusion-wise maximal non-crossing matching M_*, it connects exactly one segment Z from each of three groups $\mathcal{A}, \mathcal{B}$, and $\mathcal{W}$.

From Claim 24, we assume that a maximum (thus, inclusion-maximal) non-crossing matching M_* contains submatches labeled with segments A_i, W_j, B_k one from each group in any order, where $i, j, k \in [s]$. Then, M must contain A_i, W_j, B_k in this order, namely, $A_i \cdot W_j \cdot B_k \in CS(S_1, S_2)$ because some edges cross otherwise (see Fig. 4b). Therefore, we have that the concatenation $T_{j_*} := A_{j_*} \cdot W_{j_*} \cdot B_{j_*}$ belongs to $CS(S_1, S_2)$, and it always has a matching in $\mathcal{B}(S_1, S_2)$. From (P3) and (P4), we can show the next claim.

▷ Claim 25. If M_* is maximal and contains $A_i \cdot W_{j_*} \cdot B_k$, then $i = j_* = k$ holds.

From Claim 25, we conclude that $T_{j_*} = A_j \cdot W_j \cdot B_j$ is the all and only members of $LCS(S_1, S_2)$ for all $j \in [s]$. Since $A_j \cdot W_j \cdot B_j = P_j \cdot X_j \cdot B_j = T_j$, the lemma is proved. ◀

Using Lemma 23, we finish the proof for Theorem 19.

Proof for Theorem 19. Recall that integers $K, r \geqslant 1$, $\Delta \geqslant 0$, and a string set $L = \{X_1, \ldots, X_s\} \subseteq \Sigma^r$ of r-strings form an instance of MAX-MIN DIVERSE STRING SET. Let $\Delta' := \Delta + 2s$, $K' = \kappa(K) := K$, and $\mathcal{S} = \{S_1, S_2\} \subseteq \Gamma^*$ be the associated instance of

Max-Min Diverse LCS for two input strings. Since the parameter $\kappa(K) = K$ depends only on K, it is obvious that this transformation can be computed in FPT. We show that this forms an FPT-reduction from the former problem to the latter problem. By Lemma 23, we have the next claim.

▷ Claim 26. For any $i, j \in [s]$, $d_H(T_i, T_j) = d_H(X_i, X_j) + 2s$.

Proof of Claim 26. By Lemma 23, $LCS(S_1, S_2) = \{\, T_j \mid i \in [K] \,\}$. By construction, $T_j = P_j \cdot X_j \cdot Q_j$ and $|P_j| = |Q_j| = s$, and $d_H(P_i, P_j) = d_H(Q_i, Q_j) = s$ for any $i, j \in [s]$, $i \neq j$. Therefore, we can decompose $d_H(T_i, T_j)$ by $d_H(T_i, T_j) = d_H(P_i, P_j) + d_H(X_i, X_j) + d_H(Q_i, Q_j) = d_H(X_i, X_j) + 2s$ ◁

Suppose that $\mathcal{Y} \subseteq LCS(S_1, S_2)$ is any feasible solution such that $|\mathcal{Y}| = K'$ for Max-Sum Diverse LCSs, where $K' = K$. From Lemma 23, we can put $\mathcal{Y} = \{T_{i_j}\}_{j \in J}$ for some $J \subseteq [s]$. From Claim 26, we can see that $D^{\min}_{d_H}(\mathcal{Y}) = D^{\min}_{d_H}(\mathcal{X}) + 2s$, where $\mathcal{X} = \{X_j\}_{j \in J}$ is a solution for Max-Min Diverse String Set. Thus, $D^{\min}_{d_H}(\mathcal{X}) \geqslant \Delta$ if and only if $D^{\min}_{d_H}(\mathcal{Y}) \geqslant \Delta + 2s = \Delta'$. This shows that the transformation properly forms NP- and FPT-reductions. To obtain NP- and FPT-reductions for the Max-Sum version, we keep K and $\mathcal{S} = \{S_1, S_2\}$ in the previous proof, and modify $\Delta' := \Delta + 2s\binom{K}{2}'$, where $\binom{K}{2}' := \{(i, j) \in \binom{K}{2} \mid i \neq j\}$. From Claim 26, we have that $D^{\text{sum}}_{d_H}(\mathcal{Y}) = D^{\text{sum}}_{d_H}(\mathcal{X}) + 2s\binom{K}{2}'$, and the correctness of the reduction immediately follows. Combining the above arguments, the theorem is proved. ◀

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

2 Emmanuel Arrighi, Henning Fernau, Mateus de Oliveira Oliveira, and Petra Wolf. Synchronization and diversity of solutions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37(10), pages 11516–11524, 2023.

3 Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2012.

4 Giorgio Ausiello, Alessandro D'Atri, and Marco Protasi. Structure preserving reductions among convex optimization problems. *JCSS*, 21(1):136–153, 1980.

5 Julien Baste, Michael R. Fellows, Lars Jaffke, Tomáš Masařík, Mateus de Oliveira Oliveira, Geevarghese Philip, and Frances A Rosamond. Diversity of solutions: An exploration through the lens of fixed-parameter tractability theory. *Artificial Intelligence*, 303:103644, 2022.

6 Julien Baste, Lars Jaffke, Tomáš Masařík, Geevarghese Philip, and Günter Rote. FPT algorithms for diverse collections of hitting sets. *Algorithms*, 12(12):254, 2019.

7 Benjamin Birnbaum and Kenneth J. Goldman. An improved analysis for a greedy remote-clique algorithm using factor-revealing LPs. *Algorithmica*, 55(1):42–59, 2009.

8 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Harold T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147(1-2):31–54, 1995.

9 Laurent Bulteau, Mark Jones, Rolf Niedermeier, and Till Tantau. An FPT-algorithm for longest common subsequence parameterized by the maximum number of deletions. In *33rd Ann. Symp. on Combinatorial Pattern Matching (CPM 2022), LIPIcs*, volume 223, pages 6:1–6:11, 2022.

10 Alfonso Cevallos, Friedrich Eisenbrand, and Rico Zenklusen. An improved analysis of local search for max-sum diversification. *Math. Oper. Research*, 44(4):1494–1509, 2019.

11 Barun Chandra and Magnús M Halldórsson. Approximation algorithms for dispersion problems. *J. of Algorithms*, 38(2):438–465, 2001.

12 Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. A compact DAG for storing and searching maximal common subsequences. In *ISAAC 2023*, pages 21:1–21:15, 2023.

13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, fourth edition.* MIT Press, 2022.

14 Graham Cormode and Shan Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms (TALG)*, 3(1):1–19, 2007.

15 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

16 Michel Deza and Monique Laurent. *Geometry of Cuts and Metrics*, volume 15 of *Algorithms and Combinatorics.* Springer, 1997.

17 Rodney G. Downey and Michael R. Fellows. *Parameterized complexity.* Springer, 2012.

18 Erhan Erkut. The discrete p-dispersion problem. *European Journal of Operational Research*, 46(1):48–60, 1990.

19 Michael R. Fellows, Christian Knauer, Naomi Nishimura, Prabhakar Ragde, Frances A. Rosamond, Ulrike Stege, Dimitrios M. Thilikos, and Sue Whitesides. Faster fixed-parameter tractable algorithms for matching and packing problems. *Algorithmica*, 52:167–176, 2008.

20 Michael R. Fellows and Frances A. Rosamond. The DIVERSE X Paradigm (Open problems). In Henning Fernau, Petr Golovach, Marie-France Sagot, et al., editors, *Algorithmic enumeration: Output-sensitive, input-sensitive, parameterized, approximative (Dagstuhl Seminar 18421), Dagstuhl Reports, 8(10)*, 2019.

21 Dana Fisman, Joshua Grogin, Oded Margalit, and Gera Weiss. The normalized edit distance with uniform operation costs is a metric. In *33rd Ann. Symp. on Combinatorial Pattern Matching (CPM 2022), LIPIcs*, volume 223, pages 17:1–17:17, 2022.

22 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series).* Springer, 2006.

23 Michael R. Garey and David S. Johnson. Computers and intractability: A guide to the theory of NP-completeness, 1979.

24 Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, volume 99(6), pages 518–529, 1999.

25 Dan Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55(1):141–154, 1993.

26 Koji Hakata and Hiroshi Imai. The longest common subsequence problem for small alphabet size between many strings. In *ISAAC'92*, pages 469–478. Springer, 1992.

27 Tesshu Hanaka, Masashi Kiyomi, Yasuaki Kobayashi, Yusuke Kobayashi, Kazuhiro Kurita, and Yota Otachi. A framework to design approximation algorithms for finding diverse solutions in combinatorial problems. In *AAAI 2023*, pages 3968–3976, 2023.

28 Tesshu Hanaka, Yasuaki Kobayashi, Kazuhiro Kurita, and Yota Otachi. Finding diverse trees, paths, and more. In *AAAI 2021*, pages 3778–3786, 2021.

29 Pierre. Hansen and I.D. Moon. Dispersing facilities on a network. In *the TIMS/ORSA Joint National Meeting, Washington, D.C.* RUTCOR, Rutgers University., 1988. Presentation.

30 Miyuji Hirota and Yoshifumi Sakai. Efficient algorithms for enumerating maximal common subsequences of two strings. *CoRR*, abs/2307.10552, 2023. `arXiv:2307.10552`.

31 Daniel S Hirschberg. *Recent results on the complexity of common-subsequence problems, in Time warps, String edits, and Macromolecules*, pages 323–328. Addison-Wesley, 1983.

32 John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

33 Robert W. Irving and Campbell B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *CPM 1992*, pages 214–229. Springer, 1992.

34 Michael J. Kuby. Programming models for facility dispersion: The p-dispersion and maxisum dispersion problems. *Geographical Analysis*, 19(4):315–329, 1987.

35 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

36 Ohad Lipsky and Ely Porat. L1 pattern matching lower bound. *Information Processing Letters*, 105(4):141–143, 2008.

37 Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

38 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978.

39 Tatsuya Matsuoka and Shinji Ito. Maximization of minimum weighted Hamming distance between set pairs. In *Asian Conference on Machine Learning*, pages 895–910. PMLR, 2024.

40 Sekharipuram S. Ravi, Daniel J. Rosenkrantz, and Giri Kumar Tayi. Heuristic and special case algorithms for dispersion problems. *Operations research*, 42(2):299–310, 1994.

41 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

42 Douglas R. Shier. A min-max theorem for p-center problems on a tree. *Transportation Science*, 11(3):243–252, 1977.

43 Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2010.

44 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

45 Da-Wei Wang and Yue-Sun Kuo. A study on two geometric location problems. *Information Processing Letters*, 28(6):281–286, 1988.

46 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

Minimizing the Minimizers via Alphabet Reordering

Hilde Verbeek
CWI, Amsterdam, The Netherlands

Lorraine A.K. Ayad
Brunel University London, London, UK

Grigorios Loukides
King's College London, London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Minimizers sampling is one of the most widely-used mechanisms for sampling strings [Roberts et al., Bioinformatics 2004]. Let $S = S[1] \dots S[n]$ be a string over a totally ordered alphabet Σ. Further let $w \geq 2$ and $k \geq 1$ be two integers. The minimizer of $S[i \mathinner{.\,.} i+w+k-2]$ is the smallest position in $[i, i+w-1]$ where the lexicographically smallest length-k substring of $S[i \mathinner{.\,.} i+w+k-2]$ starts. The set of minimizers over all $i \in [1, n-w-k+2]$ is the set $\mathcal{M}_{w,k}(S)$ of the minimizers of S.

We consider the following basic problem:

Given S, w, and k, can we efficiently compute a total order on Σ that minimizes $|\mathcal{M}_{w,k}(S)|$?

We show that this is unlikely by proving that the problem is NP-hard *for any $w \geq 3$ and $k \geq 1$*. Our result provides theoretical justification as to why *there exist no exact algorithms* for minimizing the minimizers samples, while *there exists a plethora of heuristics* for the same purpose.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases sequence analysis, minimizers, alphabet reordering, feedback arc set

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.28

Related Version *Extended Version*: `https://arxiv.org/abs/2405.04052`

Funding *Hilde Verbeek*: Supported by a Constance van Eeden Fellowship.
Solon P. Pissis: Supported by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

1 Introduction

The minimizers sampling mechanism has been introduced independently by Schleimer et al. [17] and by Roberts et al. [16]. Since its inception, it has been employed ubiquitously in modern sequence analysis methods underlying some of the most widely-used tools [11, 12, 19].

Let $S = S[1] \dots S[n]$ be a string over a totally ordered alphabet Σ. Further let $w \geq 2$ and $k \geq 1$ be two integers. The minimizer of the fragment $S[i \mathinner{.\,.} i+w+k-2]$ of S is the smallest position in $[i, i+w-1]$ where the lexicographically smallest length-k substring of $S[i \mathinner{.\,.} i+w+k-2]$ starts. We then define the set $\mathcal{M}_{w,k}(S)$ of the minimizers of S as the set of the minimizers positions over all fragments $S[i \mathinner{.\,.} i+w+k-2]$, for $i \in [1, n-w-k+2]$. Every fragment $S[i \mathinner{.\,.} i+w+k-2]$ containing w length-k fragments is called a *window* of S.

▶ **Example 1.** Let $S = \texttt{aacaaacgcta}$, $w = 3$, and $k = 3$. Assuming $\texttt{a} < \texttt{c} < \texttt{g} < \texttt{t}$, we have that $\mathcal{M}_{w,k}(S) = \{1, 4, 5, 6, 7\}$. The minimizers positions are colored red: $S = \texttt{aacaaacgcta}$.

Note that by choosing the *smallest* position in $[i, i+w-1]$ where the lexicographically smallest length-k substring starts, we resolve ties in case the latter substring has multiple occurrences in a window.

It is easy to prove that minimizers samples enjoy the following three useful properties [23]:

- **Property 1 (approximately uniform sampling):** Every fragment of length at least $w+k-1$ of S has at least one representative position sampled by the mechanism.
- **Property 2 (local consistency):** Exact matches between fragments of length at least $\ell \geq w+k-1$ of S are preserved by means of having the same (relative) representative positions sampled by the mechanism.
- **Property 3 (left-to-right parsing):** The minimizer selected by any fragment of length $w+k-1$ comes at or after the minimizers positions selected by all previous windows.

Since Properties 1 to 3 hold *unconditionally*, and since the ordering of letters does not affect the correctness of algorithms using minimizers samples [6, 18, 14, 1], one would like to choose the ordering that minimizes the resulting sample as a means to improve the space occupied by the underlying data structures; contrast Example 1 to the following example.

▶ **Example 2.** Let $S = \texttt{aacaaacgcta}$, $w = 3$, and $k = 3$. Assuming $\texttt{c} < \texttt{a} < \texttt{g} < \texttt{t}$, we have that $\mathcal{M}_{w,k}(S) = \{3, 6, 7\}$. The minimizers positions are colored red: $S = \texttt{aacaaacgcta}$. In fact, this ordering is a best solution in minimizing $|\mathcal{M}_{w,k}(S)|$, together with the orderings $\texttt{c} < \texttt{g} < \texttt{t} < \texttt{a}$ and $\texttt{c} < \texttt{g} < \texttt{a} < \texttt{t}$, which both, as well, result in $|\mathcal{M}_{w,k}(S)| = 3$.

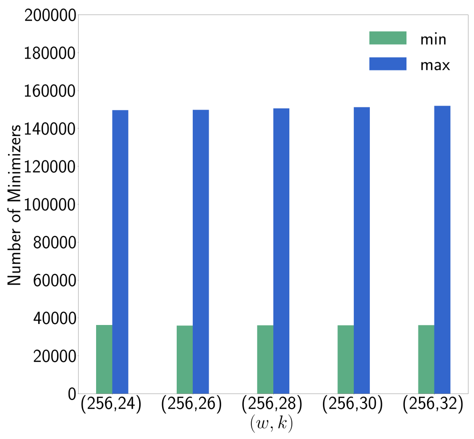
Our Problem. We next formalize the problem of computing a best such total order on Σ:

Minimizing the Minimizers
Input: A string $S \in \Sigma^n$ and two integers $w \geq 2$ and $k \geq 1$.
Output: A total order on Σ that minimizes $|\mathcal{M}_{w,k}(S)|$.

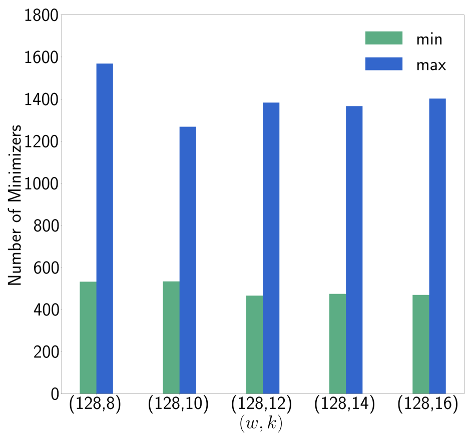
Motivation. A lot of effort has been devoted by the bioinformatics community to designing practical algorithms for minimizing the resulting minimizers sample [3, 4, 15, 21, 8, 22, 7]. Most of these approaches consider the space of all orderings on Σ^k (the set of all possible length-k strings on Σ) instead of the ones on Σ; and employ *heuristics* to choose some ordering resulting in a small sample (see Section 3 for a discussion). To illustrate the impact of reordering on the number of minimizers, we considered two real-world datasets and measured the difference in the number of minimizers between the worst and best reordering, among those we could consider in a reasonable amount of time. The first dataset we considered is the complete genome of Escherichia coli str. K-12 substr. MG1655. For selecting minimizers, we considered different orderings on Σ^k. We thus mapped every length-k substring to its lexicographic rank in $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}^k$ (assuming $\texttt{A} < \texttt{C} < \texttt{G} < \texttt{T}$) constructing a new string S over $[1, |\Sigma|^k]$. We then computed $|\mathcal{M}_{w,1}(S)|$ for different values of (w, k) and orderings on $[1, |\Sigma|^k]$. It should be clear that this corresponds to computing the size of $\mathcal{M}_{w,k}$ for the original sequence over $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$. The second dataset is the complete genome of SARS-CoV-2 OL663976.1. Figure 1 shows the min and max values of the size of the obtained minimizers samples. The results in Figure 1 clearly show the impact of alphabet reordering on $|\mathcal{M}_{w,1}(S)|$: the gap between the min and max is quite significant as in all cases we have $2\min < \max$. Note that we had to terminate the exploration of the whole space of orderings when $2\min < \max$ was achieved; hence the presented gaps are not even the largest possible.

This begs the question:

Given S, w, and k, can we efficiently compute a total order on Σ that minimizes $|\mathcal{M}_{w,k}(S)|$?



(a) Complete genome of Escherichia coli.



(b) Complete genome of SARS-CoV-2.

Figure 1 The min and max values of the size of the minimizers sample, among *some* of the possible orderings of $[1, |\Sigma|^k]$, on two real datasets using a range of (w, k) parameter values.

Our Contribution. We answer this basic question in the negative. Let us first define the decision version of Minimizing the Minimizers.

Minimizing the Minimizers (Decision)
Input: A string $S \in \Sigma^n$ and three integers $w \geq 2$, $k \geq 1$, and $\ell > 0$.
Output: Is there a total order on Σ such that $|\mathcal{M}_{w,k}(S)| \leq \ell$?

Our main contribution in this paper is the following result.

▶ **Theorem 3.** *Minimizing the Minimizers (Decision) is NP-complete if* $w \geq 3$ *and* $k \geq 1$.

Theorem 3 provides theoretical justification as to why *there exist no exact algorithms* for minimizing the minimizers samples, while *there exists a plethora of heuristics* for the same purpose. Notably, Theorem 3 almost completes the complexity landscape of the Minimizing the Minimizers problem – the only exception is the case $w = 2$ and $k \geq 1$. To cover *all practically interesting combinations* of input parameters w and k (i.e., for any $w \geq 3$ and $k \geq 1$), we design a non-trivial reduction from the feedback arc set problem [9].

The reduction we present is specifically for the case in which the size of the alphabet Σ is variable. If $|\Sigma|$ is bounded by a constant, the problem can be solved in polynomial time: one can simply iterate over the $|\Sigma|!$ permutations of the alphabet, compute the number of minimizers for each ordering in linear time [13], and output a globally best ordering.

Other Related Work. Choosing a best total order on Σ is generally not new; it has also been investigated in other contexts, e.g., for choosing a best total order for minimizing the number of runs in the Burrows-Wheeler transform [2]; for choosing a best total order for minimizing (or maximizing) the number of factors in a Lyndon factorization [5]; or for choosing a best total order for minimizing the number of bidirectional string anchors [14].

Paper Organization. Section 2 presents the proof of Theorem 3. Section 3 presents a discussion on orderings on Σ^k in light of Theorem 3. Final remarks are presented in Section 4.

2 Minimizing the Minimizers is NP-complete

We show that the MINIMIZING THE MINIMIZERS problem is NP-hard by a reduction from the well-known FEEDBACK ARC SET problem [9]. Let us first formally define the latter problem.

FEEDBACK ARC SET
Input: A directed graph $G = (V, A)$.
Output: A set $F \subseteq A$ of minimum size such that $(V, A \setminus F)$ contains no directed cycles.

We call any such $F \subseteq A$ a *feedback arc set*. The decision version of the FEEDBACK ARC SET problem is naturally defined as follows.

FEEDBACK ARC SET (DECISION)
Input: A directed graph $G = (V, A)$ and an integer $\ell' > 0$.
Output: Is there a set $F \subseteq A$ such that $(V, A \setminus F)$ contains no directed cycles and $|F| \leq \ell'$?

An equivalent way of phrasing this problem is to find an ordering on the set V of the graph's vertices, such that the number of arcs (u, v) with $u > v$ is minimal [20]. Then this is a topological ordering of the graph $(V, A \setminus F)$, and will be analogous to the alphabet ordering in the MINIMIZING THE MINIMIZERS problem; see [14] for a similar application of this idea.[1] If MINIMIZING THE MINIMIZERS is then solved on the instance constructed by our reduction, producing a total order on V, taking all arcs (u, v) with $u > v$ should produce a feedback arc set of minimum size, solving the original instance of the FEEDBACK ARC SET problem.

2.1 Overview of the Technique

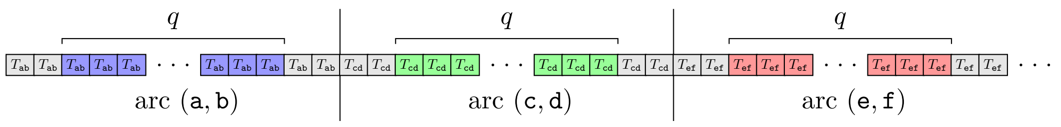
Given any instance $G = (V, A)$ of FEEDBACK ARC SET, we will construct a string S over alphabet V and of length polynomial in $|A|$. Specifically, we define string S as follows:

$$S = \prod_{(\mathtt{a},\mathtt{b}) \in A} T_{\mathtt{ab}}^{q+4},$$

where $T_{\mathtt{ab}}$ is a string consisting of the letters $\mathtt{a}$ and $\mathtt{b}$, whose length depends only on w and k, and q is an integer polynomial in $|A|$, both of which will be defined later. The product $\prod$ of some strings is defined as their concatenation, and X^q denotes q concatenations of string X starting with the empty string; e.g., if $X = \mathtt{ab}$ and $q = 4$, we have $X^q = (\mathtt{ab})^4 = \mathtt{abababab}$.

String $T_{\mathtt{ab}}$ will be designed such that each occurrence, referred to as a *block*, will contain few minimizers if $\mathtt{a} < \mathtt{b}$ in the alphabet ordering, and many minimizers if $\mathtt{b} < \mathtt{a}$, analogous to the "penalty" of removing the arc $(\mathtt{a}, \mathtt{b})$ as part of the feedback arc set. We denote by $M_{\mathtt{a}<\mathtt{b}}$ the number of minimizers starting within some occurrence of $T_{\mathtt{ab}}$ in S, provided that this $T_{\mathtt{ab}}$ is both preceded and followed by at least two other occurrences of $T_{\mathtt{ab}}$ (i.e., the middle q blocks), when $\mathtt{a} < \mathtt{b}$ in the alphabet ordering. We respectively denote by $M_{\mathtt{b}<\mathtt{a}}$ the number of minimizers starting in such a block when $\mathtt{b} < \mathtt{a}$ in the alphabet ordering. This will allow us (see Figure 2) to express the total number of minimizers in S in terms of $|F|$, the size of the feedback arc set, minus some discrepancy denoted by λ. This *discrepancy* is determined by the blocks $T_{\mathtt{ab}}$ that are not preceded or followed by two occurrences of $T_{\mathtt{ab}}$ itself; namely, those that occur near some $T_{\mathtt{cd}}$, for another arc $(\mathtt{c}, \mathtt{d})$, or those that occur near the start or the end of S.

[1] Our proof is more general and thus involved because it works for any values $w \geq 3$ and $k \geq 1$, whereas the reduction from [14] works only for some fixed parameter values.



q
q
q
T_{ab}
T_{cd}
T_{ef}
arc $(\mathtt{a}, \mathtt{b})$
arc $(\mathtt{c}, \mathtt{d})$
arc $(\mathtt{e}, \mathtt{f})$

Figure 2 Illustration of the structure of string S, with the different gadgets for different arcs in G. The highlighted blocks are the ones for which the minimizers are counted in $M_{\mathtt{a}<\mathtt{b}}$ and $M_{\mathtt{b}<\mathtt{a}}$.

Let us start by showing an upper and a lower bound on the discrepancy λ.

▶ **Lemma 4.** $|A| - 1 \leq \lambda \leq 4 \cdot |A| \cdot |T_{ab}|$ *if* $|T_{ab}| \geq \frac{1}{4}(w + k - 1)$.

Proof. We are counting the number of minimizers in q blocks of $T_{\mathtt{ab}}$, for each arc $(\mathtt{a}, \mathtt{b})$. Note that we ignore four blocks for each arc, which is $4 \cdot |A|$ blocks of length $|T_{\mathtt{ab}}|$ in total. This is $4 \cdot |A| \cdot |T_{\mathtt{ab}}|$ positions in total, which gives the upper bound on the number of disregarded minimizers. For the lower bound, note that, by hypothesis, four consecutive blocks are at least as long as a single minimizer window, meaning at least one minimizer must be missed among the four blocks surrounding the border between each pair of consecutive arcs. The lower bound follows by the fact that for $|A|$ arcs we have $|A| - 1$ such borders. ◀

Given the values $M_{\mathtt{a}<\mathtt{b}}$, $M_{\mathtt{b}<\mathtt{a}}$, and λ, we can express the total number of minimizers as a function of some feedback arc set F: if an arc $(\mathtt{a}, \mathtt{b})$ is part of the feedback arc set, this corresponds to $\mathtt{b} < \mathtt{a}$ in the alphabet ordering, so the corresponding blocks $T_{\mathtt{ab}}$ will each have $M_{\mathtt{b}<\mathtt{a}}$ minimizers, whereas if $(\mathtt{a}, \mathtt{b})$ is not in F, we have $\mathtt{a} < \mathtt{b}$ and the blocks will each have $M_{\mathtt{a}<\mathtt{b}}$ minimizers. Using these values, we can define the number of minimizers in S given some feedback arc set F as

$$\begin{aligned}\mathcal{M}_{w,k}(S, F) &= q \cdot M_{\mathtt{b}<\mathtt{a}} \cdot |F| + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot (|A| - |F|) + \lambda \\ &= q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \cdot |F| + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot |A| + \lambda. \qquad (1)\end{aligned}$$

With this in mind, we can prove the following relationship between $\mathcal{M}_{w,k}(S, F)$ and $|F|$:

▶ **Lemma 5.** *Let ℓ' be some positive integer and let $\ell = q \cdot (M_{b<a} - M_{a<b}) \cdot (\ell' + 1) + q \cdot M_{a<b} \cdot |A|$. If $M_{b<a} > M_{a<b}$, $|T_{ab}| \geq \frac{1}{4}(w + k - 1)$, and q is chosen such that $\lambda < q \cdot (M_{b<a} - M_{a<b})$, then $\mathcal{M}_{w,k}(S, F) \leq \ell$ if and only if $|F| \leq \ell'$.*

Proof. By hypothesis, $M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}$ is positive, thus, by Equation 1, $\mathcal{M}_{w,k}(S, F)$ grows linearly with $|F|$. Suppose we have a feedback arc set F with $|F| \leq \ell'$. Consider the alphabet ordering inducing F and let λ be the corresponding discrepancy for $\mathcal{M}_{w,k}(S, F)$. By hypothesis, we have $\lambda < q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}})$. Substituting the bounds on $|F|$ and λ into Equation 1 gives

$$\begin{aligned}\mathcal{M}_{w,k}(S, F) &\leq q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \cdot \ell' + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot |A| + q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \\ &= q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \cdot (\ell' + 1) + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot |A| = \ell,\end{aligned}$$

completing the proof in one direction.

For the other direction, suppose we have picked F such that $\mathcal{M}_{w,k}(S, F) \leq \ell$ and assume that $|F| \geq \ell' + 1$ towards a contradiction. Then we have the following two inequalities:

$$\begin{aligned}&\mathcal{M}_{w,k}(S, F) \leq \ell = q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \cdot (\ell' + 1) + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot |A| \\ &\mathcal{M}_{w,k}(S, F) \geq q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}}) \cdot (\ell' + 1) + q \cdot M_{\mathtt{a}<\mathtt{b}} \cdot |A| + \lambda. \quad \text{(by Equation 1)}\end{aligned}$$

By Lemma 4, for any non-trivial instance with $|A| > 1$, λ is strictly positive, meaning these inequalities are contradictory. Therefore, if $\mathcal{M}_{w,k}(S, F) \leq \ell$, it must be that $|F| \leq \ell'$. ◀

Given w and k, we must determine a string $T_{\mathtt{ab}}$ such that $M_{\mathtt{b}<\mathtt{a}} > M_{\mathtt{a}<\mathtt{b}}$ and $|T_{\mathtt{ab}}| \geq \frac{1}{4}(w+k-1)$. We then simply have to choose some q, which is polynomial in $|A|$, satisfying $\lambda < q \cdot (M_{\mathtt{b}<\mathtt{a}} - M_{\mathtt{a}<\mathtt{b}})$. At that point we will have constructed a string S for which it holds that the feedback arc set induced by the minimum set of minimizers is also a minimum feedback arc set on G, thus completing the reduction.

The following three subsections address the $T_{\mathtt{ab}}$ construction:

- Section 2.2: $w \geq k+2$ (Case A);
- Section 2.3: $w = 3$ and $k \geq 2$ (Case B);
- Section 2.4: $3 < w < k+2$ (Case C).

It should be clear that the above sections cover all the cases for $w \geq 3$ and $k \geq 1$. Section 2.5 puts everything together to complete the proof.

2.2 Case A: $w \geq k+2$

▶ **Lemma 6.** *Let $T_{ab} = ab^{w-1}$, for $w \geq k+2$. Then $M_{a<b} = 1$ and $M_{b<a} = w-k$.*

Proof. The block has length w; inspect Figure 3. Recall that, for the window starting at position i, the candidates for its minimizer are the length-k fragments starting at positions $[i, i+w-1]$. Therefore, for every window starting in a block $T_{\mathtt{ab}}$ (provided it is succeeded by another $T_{\mathtt{ab}}$), a candidate minimizer is $\mathtt{ab}^{k-1}$; so if $\mathtt{a} < \mathtt{b}$, each $T_{\mathtt{ab}}$ will contain just one minimizer. Thus we have $M_{\mathtt{a}<\mathtt{b}} = 1$.

For $\mathtt{b} < \mathtt{a}$, consider that $T_{\mathtt{ab}}$ contains $w-k$ occurrences of $\mathtt{b}^k$, and that for each window, at least one of the candidates for its minimizer is $\mathtt{b}^k$. Since there is no length-k substring that is lexicographically smaller than $\mathtt{b}^k$, each occurrence of $\mathtt{b}^k$ (and nothing else) is a minimizer, so it follows that $M_{\mathtt{b}<\mathtt{a}} = w-k$. Note that $M_{\mathtt{b}<\mathtt{a}} > M_{\mathtt{a}<\mathtt{b}}$ only if $w \geq k+2$. ◀

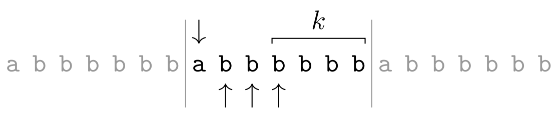


Figure 3 Illustration of 3 copies of $T_{\mathtt{ab}}$ in S for $w = 7$ and $k = 4$, along with its respective minimizers when $\mathtt{a} < \mathtt{b}$ (top) and when $\mathtt{b} < \mathtt{a}$ (bottom). It can be seen that $M_{\mathtt{a}<\mathtt{b}} = 1$ and $M_{\mathtt{b}<\mathtt{a}} = 3$.

2.3 Case B: $w = 3$ and $k \geq 2$

▶ **Lemma 7.** *Let $T_{ab} = (ab)^t bb$ with $t = \lceil \frac{w+k}{2} \rceil$, for $w = 3$ and $k \geq 2$. Then $M_{a<b} = \lfloor \frac{k}{2} \rfloor + 3$ and $M_{b<a} = \lfloor \frac{k}{2} \rfloor + 4$.*

Proof. Since $w = 3$, for every window, the minimizer is one out of three length-k fragments; inspect Figure 4. Every $\mathtt{a}$ in the block has a $\mathtt{b}$ before it. For any window starting at a position preceding an $\mathtt{a}$, two of the candidates start with a $\mathtt{b}$ and the other starts with an $\mathtt{a}$. As an example consider the window $\mathtt{babab}$ preceding an $\mathtt{a}$ in Figure 4. We have that the first and the third candidates start with a $\mathtt{b}$ and the second starts with an $\mathtt{a}$. Therefore, if $\mathtt{a} < \mathtt{b}$, the candidate starting with an $\mathtt{a}$ will be chosen and every $\mathtt{a}$ in $T_{\mathtt{ab}}$ is a minimizer. Only the window starting at the third-to-last position of the block will not consider any length-k substring starting with an $\mathtt{a}$ as its minimizer, as therein we have three $\mathtt{b}$'s occurring in a row. Since $k \geq 2$, the last $\mathtt{b}$ of the block will be chosen if $\mathtt{a} < \mathtt{b}$. Thus, $M_{\mathtt{a}<\mathtt{b}}$ counts every $\mathtt{a}$ and one $\mathtt{b}$, which gives:

$$M_{\mathtt{a}<\mathtt{b}} = t + 1 = \left\lceil \frac{w+k}{2} \right\rceil + 1 = \left\lceil \frac{3+k}{2} \right\rceil + 1 = \left\lfloor \frac{k+2}{2} \right\rfloor + 1 + 1 = \left\lfloor \frac{k}{2} \right\rfloor + 3.$$

For $M_{\mathtt{b}<\mathtt{a}}$, we apply the same logic to conclude that every `b` surrounded by `a`'s is a minimizer, which accounts for all `b`'s except the final three, which occur at positions $[2t, 2t+2]$:

- For the window starting at position $2t$, the three minimizer candidates start, respectively, with `bb`, `bb` and `ba`. Since $k \geq 2$, the first candidate $(2t)$ will be the minimizer because it is lexicographically a smallest and the leftmost ($\mathtt{b} < \mathtt{a}$).
- For the window starting at position $2t+1$, the first two candidates start, respectively, with `bb` and `ba`, and the third starts with an `a`. The first candidate $(2t+1)$ will be the minimizer, because it is lexicographically smaller ($\mathtt{b} < \mathtt{a}$).
- For the window starting at position $2t+2$, the first and third candidates start with a `b` whereas the second starts with an `a`. The third candidate starts at the second position of the next $T_{\mathtt{ab}}$-block. Since $2t > k+1$, this candidate consists of only `baba`... alternating for k letters. It is equal to the first candidate, so by tie-breaking the first candidate $(2t+2)$ is the minimizer as it is the leftmost.

Thus, every `b` in the block will be a minimizer if $\mathtt{b} < \mathtt{a}$, and we have:

$$M_{\mathtt{b}<\mathtt{a}} = t + 2 = \left\lceil \frac{3+k}{2} \right\rceil + 2 = \left\lfloor \frac{k}{2} \right\rfloor + 4.$$ ◀

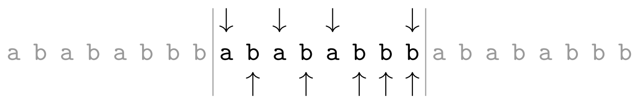


Figure 4 $T_{\mathtt{ab}}$ for $w = 3$ and $k = 3$, with its respective minimizers. The last `b` is a minimizer even when $\mathtt{a} < \mathtt{b}$, because $w = 3$. In this situation, $M_{\mathtt{a}<\mathtt{b}} = 4$ and $M_{\mathtt{b}<\mathtt{a}} = 5$.

2.4 Case C: $3 < w < k + 2$

▶ **Lemma 8.** *Let $T_{ab} = (ab)^t bb$ with $t = \lceil \frac{w+k}{2} \rceil$, for $3 < w < k+2$. Then*

- *if k is even, $M_{a<b} = \frac{k}{2} + 2 + p$ and $M_{b<a} = \frac{k}{2} + 3 + p$, where $p = (w+k) \mod 2$;*
- *if k is odd, $M_{a<b} = \lfloor \frac{k}{2} \rfloor + 3$ and $M_{b<a} = \lfloor \frac{k}{2} \rfloor + 4$.*

Proof. Every length-w fragment of the block contains at least one `a` and at least one `b`; inspect Figure 5. Because of this, only `a`'s will be minimizers if $\mathtt{a} < \mathtt{b}$ and only `b`'s if $\mathtt{b} < \mathtt{a}$ (unlike when $w = 3$, as shown in Section 2.3). We start by counting $M_{\mathtt{a}<\mathtt{b}}$. Suppose we are determining the minimizer at position i. Every candidate we consider is a string of alternating `a`'s and `b`'s (starting with an `a`), in which potentially one `a` is substituted by a `b` (if the length-k fragment contains the `bbb` at the end of the block). A lexicographically smallest length-k fragment is one in which this extra `b` appears the latest, or not at all.

First, we will consider the number of length-k fragments in which the extra `b` does not occur. For these fragments, it is the case that no other fragment in the block is lexicographically smaller when $\mathtt{a} < \mathtt{b}$, so it is automatically picked as minimizer at the position corresponding to the start of the length-k fragment. The extra `b` appears at position $2t+1$ in the block, so this applies to all length-k fragments starting with an `a` that end before position $2t+1$. That is, all `a`'s up to (and including) position $i = 2t-(k-1) = 2\lceil \frac{w+k}{2} \rceil - k + 1 = w+k+p-k+1 = w+p+1$, where $p = (w+k) \mod 2$.

Next, we consider the length-k fragments that do include the extra $\texttt{b}$. At any position past i, the smallest candidate will be the first one starting with an $\texttt{a}$, *unless* one of the candidates appears in the next $T_{\texttt{ab}}$-block, in which case the minimizer will be the first position of this next block (because this candidate does not include the extra $\texttt{b}$ and is therefore smaller than any candidate before it). Specifically, this is the case if position $|T_{\texttt{ab}}| + 1$ is one of the w candidates. Therefore, all windows starting at positions up to and including $j = |T_{\texttt{ab}}| + 1 - w = (2\lceil \frac{w+k}{2} \rceil + 2) + 1 - w = w + k + 3 + p - w = k + p + 3$ will have as their minimizer the first position with an $\texttt{a}$, meaning that all $\texttt{a}$'s up to position $j + 1$ are minimizers.

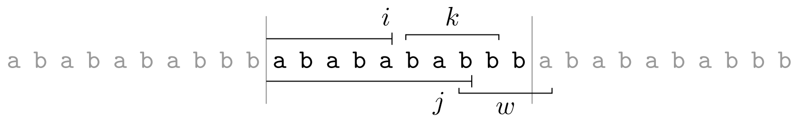


Figure 5 $T_{\texttt{ab}}$ for $w = 4$ and $k = 4$, showing the positions i and j for counting $M_{\texttt{a}<\texttt{b}}$. Position i is the final position at which the length-k fragment does not contain $\texttt{bb}$, whereas j is the final position for which the starting position of the next $T_{\texttt{ab}}$-block is not a candidate. When $\texttt{a} < \texttt{b}$, the minimizers in the block are all $\texttt{a}$'s up to position $\max\{i, j + 1\}$.

We now have that all $\texttt{a}$'s up to position $i = w+p+1$ and all $\texttt{a}$'s up to position $j+1 = k+p+4$ are minimizers. Thus we need to count the $\texttt{a}$'s up to position $\max\{w + p + 1, k + p + 4\}$. Because, by hypothesis, $w < k + 2$, this maximum is equal to $k + p + 4$. The first $k + p + 4$ letters of the block are alternating $\texttt{a}$'s and $\texttt{b}$'s, so we get

$$M_{\texttt{a}<\texttt{b}} = \left\lceil \frac{k+p+4}{2} \right\rceil = \left\lceil \frac{k+p}{2} \right\rceil + 2 = \begin{cases} \frac{k}{2} + 2 + p & \text{if } k \text{ is even;} \\ \lfloor \frac{k}{2} \rfloor + 3 & \text{if } k \text{ is odd.} \end{cases}$$

Next, we compute $M_{\texttt{b}<\texttt{a}}$. We start by showing that the final three $\texttt{b}$'s in $T_{\texttt{ab}}$ are all minimizers. There is only one length-k fragment that starts with $\texttt{bbb}$ and one that starts with $\texttt{bba}$, so the first two of these final $\texttt{b}$'s will both be minimizers for the windows that start with $\texttt{bbb}$ and $\texttt{bba}$. For the window that starts at the third $\texttt{b}$, which is position $|T_{\texttt{ab}}|$, note that the entire window does not contain $\texttt{bb}$ at all; it consists of only alternating $\texttt{b}$'s and $\texttt{a}$'s as the window has length $w + k - 1$ whereas the next occurrence of $\texttt{bb}$ is after $w + k + p$ positions. Because the window does not contain $\texttt{bb}$, none of its candidates are smaller than $\texttt{baba}\ldots$ alternating, which first appears at the start of the window. Therefore, the third $\texttt{b}$ is also a minimizer.

The rest of the minimizers consist of two sets. The first set corresponds to positions for which no candidate is smaller than $\texttt{baba}\ldots$ (alternating for k letters). These are all positions with a $\texttt{b}$, up to a certain position i (to be computed later), after which there will also be a smaller minimizer candidate, i.e., one that contains $\texttt{bb}$; inspect Figure 6. This is the second set of minimizers: ones that start with $\texttt{b}$ and contain $\texttt{bb}$ at some point. These are all positions with a $\texttt{b}$ from some position j onwards.

We start by computing j. Position j is the first position such that the length-k fragment starting at j starts with a $\texttt{b}$ and contains $\texttt{bb}$. If k is odd, the fragment ends at position $2t+2$ with $\texttt{bbb}$ as suffix; if k is even, the fragment ends at position $2t + 1$ with $\texttt{bb}$ as suffix. We have

$$j = \begin{cases} 2t + 1 - k + 1 = w + p + 2 & \text{if } k \text{ is even;} \\ 2t + 2 - k + 1 = w + p + 3 & \text{if } k \text{ is odd.} \end{cases}$$

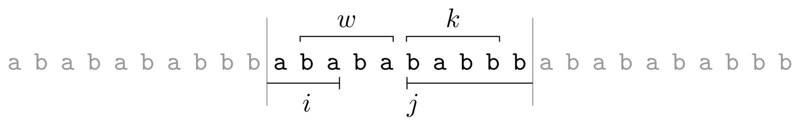


Figure 6 $T_{\texttt{ab}}$ for $w = 4$ and $k = 4$, showing the positions i and j when counting $M_{\texttt{b}<\texttt{a}}$: j is the position of the first `b` at which the corresponding length-k fragment contains `bb`; i is the last position at which j is not a candidate for its minimizer. When $\texttt{b} < \texttt{a}$, the minimizers in this block are all `b`'s up to position $i + 1$ and all `b`'s from position j onwards.

Note that $j = w + p + 2 + (k \mod 2)$. Every `b` from position j onwards is a minimizer. This includes the three `b`'s at the end of the pattern (at positions $2t$ through $2t + 2$), as well as the ones between positions j and $2t - 1$ (both inclusive). Thus we have

$$3 + \left\lfloor \frac{2t - j}{2} \right\rfloor = 3 + \left\lfloor \frac{w + k + p - (w + p + 2 + (k \mod 2))}{2} \right\rfloor$$
$$= 3 + \left\lfloor \frac{k - 2 - (k \mod 2)}{2} \right\rfloor = 2 + \left\lfloor \frac{k}{2} \right\rfloor$$

`b`'s from position j onwards.

Next, we compute i and count the number of `b`'s up to i. We take the last position for which the length-k fragment starting at j is not a candidate. This is $i = j - w$. The minimizer for the window starting at position $i + 1$ is the length-k fragment starting at j, since this is the only candidate that contains `bb`. However, if there is a `b` at position $i + 1$,[2] then $i + 1$ will still be a minimizer: when we take the minimizer for position i, the length-k fragment containing `bb` will not be a candidate so it will take the first length-k fragment starting with a `b`, which is at position $i + 1$. Therefore, we count all `b`'s that appear *up to* $i + 1$:

$$\left\lfloor \frac{i+1}{2} \right\rfloor = \left\lfloor \frac{j - w + 1}{2} \right\rfloor = \left\lfloor \frac{(w + p + 2 + (k \mod 2)) - w + 1}{2} \right\rfloor = \left\lfloor \frac{p + 3 + (k \mod 2)}{2} \right\rfloor$$
$$= 1 + \left\lfloor \frac{1 + p + (k \mod 2)}{2} \right\rfloor = \begin{cases} 1 + p & \text{if } k \text{ is even;} \\ 2 & \text{if } k \text{ is odd.} \end{cases}$$

Adding the two numbers of `b`'s together gives (inspect Figure 7):

$$M_{\texttt{b}<\texttt{a}} = 2 + \left\lfloor \frac{k}{2} \right\rfloor + \begin{cases} 1 + p & \text{if } k \text{ is even;} \\ 2 & \text{if } k \text{ is odd;} \end{cases}$$
$$= \begin{cases} \frac{k}{2} + 3 + p & \text{if } k \text{ is even;} \\ \left\lfloor \frac{k}{2} \right\rfloor + 4 & \text{if } k \text{ is odd.} \end{cases}$$

◀

2.5 Wrapping up the Reduction

Proof of Theorem 3. Minimizing the Minimizers (Decision) asks whether or not there exists some ordering on Σ such that a string $S \in \Sigma^n$ has at most ℓ minimizers for parameters w and k. Given w, k and an ordering on Σ, one can compute the number of minimizers

[2] Consider the case when $T_{\texttt{ab}} = $ `ababababab bb` with $w = 5$ and $k = 4$. For this block, we have $i = 3$ and $j = 8$. Indeed $i = j - w = 3$ and at position $i + 1 = 4$ of the block we have a `b`. Position 4 will be selected as the minimizer for the window starting at position 3.

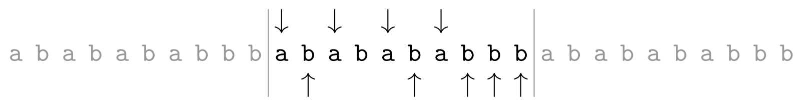


Figure 7 $T_{\texttt{ab}}$ for $w = 4$ and $k = 4$, showing its minimizers for $\texttt{a} < \texttt{b}$ (top) and $\texttt{b} < \texttt{a}$ (bottom). In this situation, $M_{\texttt{a}<\texttt{b}} = 4$ and $M_{\texttt{b}<\texttt{a}} = 5$.

for those parameters in linear time [13, Theorem 3]. Therefore, one can use an alphabet ordering as a certificate to verify a YES instance of Minimizing the Minimizers (Decision) simply by comparing the computed number of minimizers to ℓ. This proves that the Minimizing the Minimizers (Decision) problem is in NP. To prove that Minimizing the Minimizers (Decision) is NP-hard, we use a reduction from Feedback Arc Set (Decision) (see Section 2 for definition), which is a well-known NP-complete problem [9].

We are given an instance $G = (V, A)$ of Feedback Arc Set and an integer ℓ', and we are asked to check if G contains a feedback arc set with at most ℓ' arcs. We will construct an instance S of Minimizing the Minimizers (Decision), for given parameters $w \geq 3$ and $k \geq 1$, such that: the minimum number of minimizers in S, over all alphabet orderings, is at most some value ℓ if and only if G contains a feedback arc set of size at most ℓ'.

By Lemma 4, we have $\lambda \leq 4 \cdot |A| \cdot |T_{\texttt{ab}}|$. Given w and k, we must determine a string $T_{\texttt{ab}}$ such that $M_{\texttt{b}<\texttt{a}} > M_{\texttt{a}<\texttt{b}}$ and $|T_{\texttt{ab}}| \geq \frac{1}{4}(w + k - 1)$, and also choose some q satisfying $\lambda < q \cdot (M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}})$ (see Lemma 5). Let $\Sigma = V$ and let $S = \prod_{(\texttt{a},\texttt{b})\in A} T_{\texttt{ab}}^{q+4}$, with $T_{\texttt{ab}}$ and q to be determined depending on w and k.

Case A: $w \geq k + 2$. Let $T_{\texttt{ab}} = \texttt{ab}^{w-1}$, so $|T_{\texttt{ab}}| = w$. Since, by hypothesis, the maximal value of k is $w - 2$, and since $|T_{\texttt{ab}}| = w$, we have that $4|T_{\texttt{ab}}| \geq 2w - 3$. Thus, the condition on the length of $T_{\texttt{ab}}$ always holds. By Lemma 6, $M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}} = w - k - 1$. We choose $q = 4 \cdot w \cdot |A| + 1$, so that $\lambda \leq 4 \cdot |A| \cdot w < q \cdot (w - k - 1)$. Thus, $\lambda < q \cdot (M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}})$.

Case B and Case C: $w < k + 2$. Let $T_{\texttt{ab}} = (\texttt{ab})^t\texttt{bb}$ for $t = \lceil \frac{w+k}{2} \rceil$. We have $|T_{\texttt{ab}}| = 2t + 2 = 2(\lceil \frac{w+k}{2} \rceil) + 2 = w + k + p + 2$, where $p = (w + k) \mod 2$. The condition on the length of $T_{\texttt{ab}}$ always holds because $w + k + p + 2 > w + k - 1$.

- If $w = 3$, then by Lemma 7, $M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}} = \lfloor \frac{k}{2} \rfloor + 4 - (\lfloor \frac{k}{2} \rfloor + 3) = 1$.
- If $w > 3$, then by Lemma 8:
  - if k is even, $M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}} = \frac{k}{2} + 3 + p - (\frac{k}{2} + 2 + p) = 1$;
  - if k is odd, $M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}} = \lfloor \frac{k}{2} \rfloor + 4 - (\lfloor \frac{k}{2} \rfloor + 3) = 1$.

In any case, $M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}} = 1$. We choose $q = 4 \cdot |A| \cdot (w + k + 3) + 1$, so that $\lambda \leq 4 \cdot |A| \cdot (w + k + p + 2) < q$. Thus, $\lambda < q \cdot (M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}})$.

Finally, we set $\ell = q \cdot (M_{\texttt{b}<\texttt{a}} - M_{\texttt{a}<\texttt{b}}) \cdot (\ell' + 1) + q \cdot M_{\texttt{a}<\texttt{b}} \cdot |A|$. By Lemma 5, we have that $\mathcal{M}_{w,k}(S, F) \leq \ell$ if and only if $|F| \leq \ell'$; in other words, G contains a feedback arc set of size at most ℓ' if and only if S has an alphabet ordering with at most ℓ minimizers.

Hence we have shown that (G, ℓ') is a YES instance of Minimizing the Minimizers (Decision) if and only if (S, ℓ) is a YES instance of Feedback Arc Set (Decision). Moreover, the length of S is $(q + 4) \cdot |A| \cdot |T_{\texttt{ab}}|$, with $T_{\texttt{ab}}$ being of polynomial length, so the reduction can be performed in polynomial time. The existence of a polynomial-time reduction from Feedback Arc Set (Decision) to Minimizing the Minimizers (Decision) proves our claim: Minimizing the Minimizers (Decision) is NP-complete if $w \geq 3$ and $k \geq 1$. ◀

3 Considering the Orderings on Σ^k

Most of the existing approaches for minimizing the minimizers samples consider the space of all orderings on Σ^k instead of the ones on Σ. Such an approach has the advantage of an easy and efficient construction of the sample by using a rolling hash function $h : \Sigma^k \rightarrow \mathbb{N}$, such as the popular Karp-Rabin fingerprints [10]; this results in a random ordering on Σ^k that usually performs well in practice [23]. Let us denote by Minimizing the Minimizers ($\leq \Sigma^k$) the version of Minimizing the Minimizers that seeks to minimize $|\mathcal{M}_{w,k}(S)|$ by choosing a best ordering on Σ^k (instead of a best ordering on Σ). It is easy to see that any algorithm solving Minimizing the Minimizers solves also Minimizing the Minimizers ($\leq \Sigma^k$) with a polynomial number of extra steps: We use an arbitrary ranking function rank from *the set* of length-k substrings of S to $[1, n-k+1]$. We construct the string S' such that $S'[i] = \mathsf{rank}(S[i\,..\,i+k-1])$, for each $i \in [1, n-k+1]$. Let Σ' be the set of all letters in S'. It should be clear that $|\Sigma'| \leq n$ because S has no more than n substrings of length k. We then solve the Minimizing the Minimizers problem with input $\Sigma := \Sigma'$, $S := S'$, $w := w$, and $k := 1$. It is then easy to verify that an optimal solution to Minimizing the Minimizers for this instance implies an optimal solution to Minimizing the Minimizers ($\leq \Sigma^k$) for the original instance. We thus conclude that Minimizing the Minimizers is at least as hard as Minimizing the Minimizers ($\leq \Sigma^k$); they are clearly equivalent for $k = 1$.

▶ **Example 9.** Let $S =$ `aacaaacgcta`, $w = 3$, and $k = 3$. We construct the string $S' =$ `235124687` over $\Sigma' = [1, 8]$ and solve Minimizing the Minimizers with $w = 3$, $k = 1$, and $\Sigma = \Sigma'$. Assuming `1` $<$ `3` $<$ `5` $<$ `6` $<$ `2` $<$ `4` $<$ `7` $<$ `8`, $\mathcal{M}_{3,1}(S') = \mathcal{M}_{3,3}(S) = \{2, 4, 7\}$. The minimizers positions are colored red: $S' =$ `235124687`. This is one of many best orderings.

Another advantage of Minimizing the Minimizers ($\leq \Sigma^k$) is that a best ordering on Σ^k is at least as good as a best ordering on Σ at minimizing the resulting sample. Indeed this is because every ordering on Σ implies an ordering on Σ^k but not the reverse.

Unfortunately, Minimizing the Minimizers ($\leq \Sigma^k$) comes with a major disadvantage. Suppose we had an algorithm solving Minimizing the Minimizers ($\leq \Sigma^k$) (either exactly or with a good approximation ratio or heuristically) and applied it to a string S of length n, with parameters w and k. Now, in order to compare a query string Q to S, the first step would be to compute the minimizers of Q, but to ensure local consistency (Property 2), we would need access to the ordering output by the hypothetical algorithm. The size of the ordering is $\mathcal{O}(\min(|\Sigma|^k, n))$ and storing this defeats the purpose of creating a sketch for S. This is when it might be more appropriate to use Minimizing the Minimizers instead.

Since Minimizing the Minimizers is NP-hard for $w \geq 3$ and $k = 1$, Minimizing the Minimizers ($\leq \Sigma^1$) is NP-hard for $w \geq 3$; hence the following corollary of Theorem 3.

▶ **Corollary 10.** *Minimizing the Minimizers ($\leq \Sigma^1$) is NP-hard if $w \geq 3$.*

4 Final Remarks

The most immediate open questions are:

- Is Minimizing the Minimizers NP-hard for $w = 2$ and $k \geq 1$?
- Is Minimizing the Minimizers ($\leq \Sigma^k$) NP-hard for $k > 1$?

References

1 Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns: Anchors are all you need. *Proc. VLDB Endow.*, 16(9):2117–2131, 2023. doi:10.14778/3598581.3598586.

2 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ESA.2020.15.

3 Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. *J. Comput. Biol.*, 22(5):336–352, 2015. doi:10.1089/CMB.2014.0160.

4 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal *k*-mer counting. *Bioinform.*, 31(10):1569–1576, 2015. doi:10.1093/BIOINFORMATICS/BTV022.

5 Daniel Gibney and Sharma V. Thankachan. Finding an optimal alphabet ordering for Lyndon factorization is hard. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 35:1–35:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.STACS.2021.35.

6 Szymon Grabowski and Marcin Raniszewski. Sampled suffix array with minimizers. *Softw. Pract. Exp.*, 47(11):1755–1771, 2017. doi:10.1002/SPE.2481.

7 Minh Hoang, Hongyu Zheng, and Carl Kingsford. Differentiable learning of sequence-specific minimizer schemes with DeepMinimizer. *J. Comput. Biol.*, 29(12):1288–1304, 2022. doi:10.1089/CMB.2022.0275.

8 Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian Walenz, Sergey Koren, and Adam M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinform.*, 36(Supplement-1):i111–i118, 2020. doi:10.1093/BIOINFORMATICS/BTAA435.

9 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

10 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/RD.312.0249.

11 Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinform.*, 32(14):2103–2110, 2016. doi:10.1093/BIOINFORMATICS/BTW152.

12 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinform.*, 34(18):3094–3100, 2018. doi:10.1093/BIOINFORMATICS/BTY191.

13 Grigorios Loukides and Solon P. Pissis. Bidirectional string anchors: A new string sampling mechanism. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 64:1–64:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ESA.2021.64.

14 Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. Bidirectional string anchors for improved text indexing and top-k similarity search. *IEEE Trans. Knowl. Data Eng.*, 35(11):11093–11111, 2023. doi:10.1109/TKDE.2022.3231780.

15 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Compact universal k-mer hitting sets. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2016. doi:10.1007/978-3-319-43681-4_21.

16 Michael Roberts, Wayne B. Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinform.*, 20(18):3363–3369, 2004. doi:10.1093/bioinformatics/bth408.

17 Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. Winnowing: Local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003. doi:10.1145/872757.872770.

18 Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Space-efficient representation of genomic k-mer count tables. *Algorithms Mol. Biol.*, 17(1):5, 2022. doi:10.1186/S13015-022-00212-0.

19 Derrick E. Wood and Steven L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):R46, 2014.

20 Daniel H. Younger. Minimum feedback arc sets for a directed graph. *IEEE Transactions on Circuit Theory*, 10(2):238–245, 1963. doi:10.1109/TCT.1963.1082116.

21 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinform.*, 36(Supplement-1):i119–i127, 2020. doi:10.1093/BIOINFORMATICS/BTAA472.

22 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Sequence-specific minimizers via polar sets. *Bioinform.*, 37(Supplement):187–195, 2021. doi:10.1093/BIOINFORMATICS/BTAB313.

23 Hongyu Zheng, Guillaume Marçais, and Carl Kingsford. Creating and using minimizer sketches in computational genomics. *J. Comput. Biol.*, 30(12):1251–1276, 2023. doi:10.1089/CMB.2023.0094.